Abductive reasoning modulo theories and an application to program verification

Yanis Sellami

▶ **To cite this version:**

Yanis Sellami. Abductive reasoning modulo theories and an application to program verification. Data Structures and Algorithms [cs.DS]. Université Grenoble Alpes [2020-..], 2020. English. NNT : 2020GRALM018 . tel-02990726

HAL Id: tel-02990726

https://theses.hal.science/tel-02990726v1

Submitted on 5 Nov 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Yanis SELLAMI

Thèse dirigée par **Nicolas PELTIER**
et codirigée par **Mnacho ECHENIM**, Université Grenoble Alpes

préparée au sein du **Laboratoire Laboratoire d'Informatique de Grenoble**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Raisonnement abductif modulo des théories et application à la vérification de programmes

Abductive reasoning modulo theories and an application to program verification

Thèse soutenue publiquement le **23 juin 2020**,
devant le jury composé de :

Monsieur NICOLAS PELTIER
CHARGE DE RECHERCHE HDR, CNRS DELEGATION ALPES, Directeur de thèse
Monsieur PASCAL FONTAINE
PROFESSEUR DES UNIVERSITES, UNIVERSITE DE LIEGE - BELGIQUE, Rapporteur
Monsieur LUCA VIGANO
PROFESSEUR DES UNIVERSITES, KING'S COLLEGE DE LONDRES, Rapporteur
Monsieur DAVID MONNIAUX
DIRECTEUR DE RECHERCHE, CNRS DELEGATION ALPES, Président
Madame MIHAELA SIGHIREANU
MAITRE DE CONFERENCES HDR, UNIVERSITE PARIS 7, Examinatrice
Monsieur MNACHO ECHENIM
MAITRE DE CONFERENCES HDR, GRENOBLE INP, Co-directeur de thèse

Abstract

This thesis introduces a generic method to compute the *prime implicates* of a logical formula, *i.e.*, the most general clausal consequences of a given logical formula, in any given decidable theory. The principle used is to recursively add to an initially empty set, literals taken from a preselected set of hypotheses until it can be proven that the disjunction of these literals is a consequence of the formula under consideration. Proofs of the termination, correctness and completeness of this algorithm are provided, which ensures that clauses computed by the method are indeed implicates and that all the prime implicates that can be constructed from the initial set are indeed returned. This algorithm is then applied in the context of program verification, in which it is used to generate loop invariants that permit to verify assertions on programs. The Abdulot framework, a C++ implementation of the system, is also introduced. Technical considerations required for the design of such systems are detailed, such as their insertion within a well-furnished ecosystem of provers and SMT-solvers. This implemented framework will be used to perform an experimental evaluation of the method and will show its practical relevance, as it can be used to solve a large scope of problems.

This thesis also presents introductory work on an implicant minimizer and applies it in the context of separation logic. The method described in this part could also be used to perform bi-abduction in separation logic, with an algorithm that is described but has not been implemented.

Résumé

Le travail présenté dans cette thèse introduit une méthode générique pour calculer les *impliqués premiers* d'une formule logique donnée dans une théorie décidable. Il s'agit intuitivement des conséquences clausales les plus générales de cette formule modulo cette théorie. Cette méthode fonctionne en ajoutant récursivement à un ensemble initialement vide des littéraux extraits d'un ensemble d'hypothèses présélectionnées, et ce jusqu'à ce qu'elle puisse démontrer que la disjonction des littéraux de cet ensemble est une conséquence de la formule sur laquelle elle travaille. On démontrera la terminaison, la correction et la complétude de cet algorithme. Cela confirmera qu'il calcule effectivement des impliqués de la formule de départ et qu'il retourne tous les impliqués premiers qui peuvent être construits à partir de l'ensemble d'hypothèses de départ. On construira ensuite à partir de cette méthode un algorithme l'appliquant à la génération d'invariants de boucles de programmes que l'on cherche à vérifier. On présentera également une implémentation C++ de ces deux méthodes, regroupées dans une infrastructure logicielle baptisée Abdulot. Cette implémentation sera utilisée pour évaluer expérimentalement les deux méthodes. On découvrira à cette occasion qu'elles sont capables de générer des solutions pour une large classe de problèmes.

Cette thèse présente également un travail introductif sur la minimisation de modèles en logique de séparation et son implémentation. La méthode décrite pourrait également être utilisée pour résoudre des instances de bi-abduction en logique de séparation, à l'aide d'un algorithme qui sera décrit mais pas implémenté.

Résumé détaillé

Cette section a pour but de proposer un résumé en français, chapitre par chapitre, de ce manuscrit. Les travaux qui y sont présentés concernent l'étude et le développement de méthodes pour le raisonnement abductif modulo des théories et une de ses applications : la vérification de programmes. Bien qu'épuré des détails théoriques et techniques nécessaires pour établir les résultats scientifiques présentés, il pourra fournir au lecteur peu familier avec la langue anglaise un aperçu du contenu et de la portée de ces travaux.

Chapitre 1 — Contexte et définitions

Le premier chapitre présente les définitions nécessaires à la compréhension des travaux présentés dans ce document. Nous commençons par y rappeler les notions habituelles de logique du premier ordre, en particulier celles d'interprétation, de modèle et de conséquence logique. Nous y présentons ensuite les notions plus spécifiques à l'abduction et à la méthode étudiée dans cette thèse pour automatiser ce type de raisonnement : la génération d'impliqués premiers. En effet, si le problème d'abduction consiste à chercher des formules logiques (impliquants) qui permettent de déduire les propriétés que nous considérons, il est possible de résoudre le problème par son dual en cherchant les impliqués de la négation de ces propriétés. Nous introduisons ensuite des définitions et propriétés pertinentes pour gérer le problème de redondance des impliqués. Ces définitions sont utiles pour éviter la génération d'impliqués que l'on pourrait déduire d'autres plus généraux et déjà connus. Le chapitre sert également à préciser le contexte dans lequel ces travaux s'appliquent et les définitions qui y sont associées. Dès lors qu'il existe une procédure pour tester la satisfaisabilité des formules logiques modulo des théories, il sera possible d'appliquer les algorithmes présentés dans cette thèse. Nous introduisons enfin les démonstrateurs modulo des théories [78, 51, 10], outils dont nous nous servirons pour effectuer lesdits tests de satisfaisabilité.

Chapitre 2 — État de l'art

Le chapitre 2 présente brièvement l'état de l'art des deux domaines de recherche sur lesquels cette thèse repose : le raisonnement abductif automatisé par la génération d'impliqués premiers et la génération d'invariants de boucles de programmes.

Pour la génération d'impliqués premiers, qui consiste à calculer les conséquences clausales minimales d'une formule logique donnée, il est de coutume de rechercher des procédures complètes. De telles procédures retournent un ensemble de clauses minimales de sorte que toute clause conséquence de la formule de départ soit aussi la conséquence de

l'une des clauses retournées. Nous décrivons dans ce chapitre les différentes méthodes qui ont été développées pour effectuer cette tâche, d'abord en logique propositionnelle, puis dans des logiques plus expressives. Nous y parlons en particulier des méthodes ZRes-Tison [171, 163] et PRIMER [141] qui, à l'heure actuelle, font partie des méthodes les plus efficaces pour générer des impliqués premiers en logique propositionnelle. En ce qui concerne les logiques plus expressives, nous détaillons la méthode SOL [96, 127], qui cible la logique du premier ordre, et l'algorithme CSP [173] pour la logique equationnelle.

Nous présentons ensuite un bref état de l'art des méthodes pour générer automatiquement des invariants de boucles de programme. Ce problème, déjà très étudié, consiste à obtenir pour chaque boucle d'un programme des formules logiques vérifiant les deux propriétés suivantes : 1. la formule est vraie dès lors que le programme entame l'exécution de la boucle et 2. lorsque la formule est vérifiée au début de la boucle, elle l'est toujours à la fin de l'exécution du code que la boucle contient. L'objectif de cette partie n'est pas de donner au lecteur une vision exhaustive des travaux de recherche sur la génération d'invariants de boucles, mais plutôt de lui donner un aperçu des alternatives efficaces de la littérature. Nous présentons ainsi certaines méthodes couramment utilisées pour résoudre les problèmes de ce type telles que l'interprétation abstraite [42, 44], les méthodes à base de patrons [72], les méthodes abductives [57] et les méthodes dynamiques [137, 130].

Partie I — Une infrastructure générique pour le raisonnement abductif modulo des théories

Chapitre 3 — Un générateur d'impliqués premiers efficace modulo des théories

Le chapitre 3 nous sert à introduire et à présenter un nouvel algorithme pour générer des impliqués premiers modulo des théories, et ce indépendamment de la théorie considérée, dès lors qu'il est possible de tester la satisfaisabilité des formules dans cette théorie. On y présente en premier lieu un algorithme naïf, nommé ImpGen, à partir duquel on obtiendra ensuite une version plus efficace. Ce premier algorithme construit des impliqués à partir d'un ensemble initial de littéraux que l'on nomme *littéraux abducibles*. Il explore ensuite des conjonctions de ces littéraux jusqu'à construire une hypothèse dont l'adjonction à la formule initiale est insatisfaisable. La négation d'une telle hypothèse étant nécessairement une conséquence logique de la formule de départ, l'algorithme peut ainsi retourner un impliqué de cette formule. Cette première version fonctionne en explorant toutes les conjonctions possibles. Elle définit et applique ensuite des méthodes de gestion de redondance, ce qui lui permet de retourner tous les impliqués premiers (non-redondants) qu'il est possible de construire à partir des littéraux abducibles qui lui auront été fournis.

L'énumération systématique de toutes les conjonctions possibles n'étant pas efficace, nous proposons des améliorations pour réduire l'espace de recherche et rendre l'algorithme suffisamment efficace pour être utilisable en pratique. Parmi ces améliorations, nous nous intéressons d'abord à l'utilisation d'un ordre sur les littéraux. Grâce à celui-ci, nous pouvons éviter de reconstruire plusieurs fois le même impliqué en ajoutant ses littéraux

dans des ordres différents. Nous détaillons aussi l'utilisation que nous pouvons faire des modèles qu'il est possible de récupérer lors des tests de satisfaisabilité. Nous montrons que grâce à cela, nous pouvons réduire le nombre de littéraux à tester lorsque certains d'entre eux sont vrais dans ces modèles. Nous présentons alors un algorithme final, nommé ImpGen-PiD, intégrant l'ensemble de ces ajustements.

Nous démontrons alors que, dès lors que le problème de la satisfaisabilité d'une formule est décidable pour la théorie dans laquelle nous travaillons et que l'ensemble des littéraux abducibles est fini, cet algorithme termine. Nous démontrons également que l'algorithme de départ et celui obtenu après l'ajout des améliorations sont à la fois corrects, c'est-à-dire qu'ils génèrent uniquement des impliqués de leur formule de départ, et complets, c'est-à-dire qu'ils génèrent tous les impliqués premiers de leur formule d'entrée, dès lors qu'il peuvent être construits à partir des littéraux abducibles fournis. Dans le cas général, la terminaison de l'algorithme n'est pas garantie. Il est toutefois possible de forcer cette terminaison en sacrifiant la complétude. Cela peut être réalisé en imposant un temps maximal aux tests de satisfaisabilité et en considérant que tout test dépassant ce temps ne dénotait pas la présence d'un impliqué.

Nous terminons le chapitre en étudiant ce qu'il est possible de changer dans ce nouvel algorithme pour améliorer ses performances dans une théorie particulière, la plus simple, en logique propositionnelle. Cela passe par l'intégration plus en profondeur de l'algorithme dans une procédure classique vérifiant la satisfaisabilité des formules en logique propositionnelle (CDCL [47, 178]). Grâce aux conflits détectés lors de l'application des règles de propagation unitaire de cette procédure, nous pouvons détecter et ajouter des impliqués supplémentaires en amont du test qui les aurait détectés. Par ailleurs, l'exploitation des informations calculées par ces mêmes règles nous permettent de minimiser les impliqués ainsi obtenus en utilisant les raisons pour lesquelles les littéraux en conflit ont été choisis par la procédure de décision. L'algorithme obtenu spécifiquement pour cette logique est également présenté.

Chapitre 4 — Stockage et réutilisation des impliqués modulo des théories

Dans le chapitre 4, nous nous intéressons au stockage des impliqués générés par l'algorithme du chapitre précédent. Nous nous concentrons en particulier sur deux objectifs. D'une part, nous cherchons à obtenir une structure qui réduit au maximum la complexité des opérations de stockage d'impliqués, notamment grâce une gestion de la redondance par des opérations de suppression des conséquences logiques. D'autre part, nous cherchons une structure qui nous permet de réutiliser efficacement les impliqués déjà engendrés pour éviter des calculs inutiles à l'algorithme principal.

Pour ce faire, nous introduisons une structure ordonnée sur les littéraux que nous appelons $\mathcal{A}$-arbre-préfixe, et dont chaque chemin de la racine à une feuille sera interprété comme un impliqué. Nous détaillons ensuite les opérations classiques sur cette structure permettant l'insertion et la suppression d'impliqués. Nous détaillons également un algorithme permettant de vérifier s'il existe dans l'$\mathcal{A}$-arbre-préfixe une conséquence logique d'une formule donnée et un autre algorithme permettant de vérifier si une formule donnée est la conséquence logique d'un des impliqués de l'$\mathcal{A}$-arbre-préfixe. Ces deux derniers

algorithmes réalisant des appels à un SMT-solver pour vérifier les propriétés d'implications, nous préciserons que le nombre de tels tests est de l'ordre de la profondeur de l'$\mathcal{A}$-arbre-préfixe pour le premier algorithme et de celle de la taille de l'$\mathcal{A}$-arbre-préfixe pour le second.

Nous achevons le chapitre en donnant une version adaptée de l'algorithme ImpGen-PiD utilisant nos $\mathcal{A}$-arbre-préfixes pour stocker et réutiliser les impliqués générés.

Partie II — Une infrastructure générique pour utiliser le raisonnement abductif dans un cadre de vérification

Chapitre 5 — Un modèle abstrait pour utiliser un générateur d'impliqués premiers en vérification

Ce chapitre présente une infrastructure générale permettant l'utilisation d'algorithmes d'abduction pour la résolution de problèmes de vérification. Ce travail se concentre sur la résolution des problèmes pour lesquels il est possible d'obtenir des conditions de vérification, formules logiques qui, lorsqu'elles sont valides, assurent la correction du problème. Nous introduisons ainsi une représentation des problèmes de vérification définie par un ensemble de formules dites « guides » et par un ensemble de « générateurs », fonctions associant aux formules guides un ensemble de conditions de vérification. Le problème est alors de trouver un ensemble de formules guides à partir desquelles les générateurs ne produisent que des conditions de vérification valides.

À partir de cette représentation, nous construisons un algorithme basé sur les travaux de Dillig *et al.* [57], nommé Ilinva, pour résoudre ces problèmes de vérification. Recevant en entrée la représentation d'un problème dont certains générateurs produisent des conditions de vérification non-valides, l'algorithme renforce successivement les formules guides de la représentation jusqu'à ce que le problème soit résolu. Pour ce faire, il commence par sélectionner un générateur dont l'image n'est pas une formule valide. Il appelle ensuite la procédure d'abduction développée en partie I sur cette formule pour obtenir des hypothèses supplémentaires qui garantissent sa validité. Il choisit alors une de ces solutions et la traduit dans le contexte du programme de telle sorte que son ajout à l'une des formules guides du problème garantisse que l'image du générateur considéré à cette étape soit désormais une formule valide. L'existence d'une telle traduction devra être garantie par la définition des générateurs de conditions de vérification. L'algorithme renforce alors la formule guide choisie avec cette traduction puis poursuit son travail sur les autres conditions de vérification. Si besoin, il peut également revenir sur certaines de ses décisions (choix de solution, choix de formule guide, choix de générateur) pour essayer les autres possibilités. Un tel retour arrière est nécessaire lorsque des choix précédents ont conduit à un programme dont certaines conditions de vérification ne peuvent plus être renforcées pour être valide. Il termine lorsqu'il obtient une représentation valide ou échoue après avoir essayé toutes les possibilités.

Nous concluons alors le chapitre en donnant une preuve que si l'algorithme n'échoue pas, son résultat est effectivement une séquence de formules guides pouvant remplacer

celles du problème de départ et de sorte que tous les générateurs du problème ainsi construit ne retournent que des formules valides.

Chapitre 6 — Une application du modèle pour générer des invariants de boucles de programmes

Dans le chapitre 6, nous définissons une spécialisation de l'algorithme ILINVA que nous avons introduit au chapitre précédent. Nous commençons par proposer une représentation standardisée des programmes. Nous définissons notamment les constructions que l'on autorise, la manière dont nous y représentons les invariants de boucle (potentiels) ainsi que la manière nous y ajoutons des assertions : formules logiques supposées vraies à l'endroit où elles apparaissent dans un programme. Nous rappelons ensuite les méthodes usuelles pour générer des conditions permettant d'assurer que des assertions sont effectivement vérifiées et que des invariants de boucle potentiels sont effectivement des invariants de boucle.

Nous détaillons ensuite la manière dont nous utilisons le cadre du chapitre 5 pour représenter ces programmes. Nous utilisons les formules guides pour représenter les invariants de boucle potentiels et les générateurs de conditions de vérification pour représenter la sémantique du programme et obtenir les conditions de vérification assurant la validité de ses assertions et de ses invariants. Nous obtenons ainsi une représentation qui nous permet d'utiliser l'algorithme ILINVA pour mettre à jour ces invariants potentiels jusqu'à ce qu'ils soient de vrais invariants et qu'ils permettent de démontrer que les assertions du programme sont vérifiées.

Nous poursuivons en détaillant les différents ajustements nécessaires pour rendre l'utilisation de l'algorithme ILINVA sur cette représentation utilisable en pratique. En particulier, nous expliquons quelles heuristiques nous utilisons pour choisir les générateurs, les impliqués et les invariants de boucle potentiels. Nous développons également une méthode pour générer les littéraux abducibles nécessaires à l'algorithme IMPGEN-PID pour résoudre nos problèmes d'abduction. En extrayant du programme les variables et fonctions qu'il utilise, nous construisons des littéraux correspondant à leur traduction dans la logique utilisée par les conditions de vérification que l'on obtient des générateurs et que l'on utilise pour l'abduction. Nous précisons enfin les paramètres avec lesquels nous configurons le générateur d'impliqués pour en obtenir les meilleurs performances possibles dans cette application.

Le chapitre conclut en détaillant les différences avec les travaux dont l'algorithme ILINVA est inspiré (Dillig *et al.* [57]), ainsi que les limites théoriques de la méthode présentée dans cette partie, notamment le fait que certaines de nos heuristiques de sélection nous empêchent de résoudre certains problèmes.

Partie III — Implémentation et évaluation

La troisième partie de la thèse est consacrée à la description et à l'évaluation d'une infrastructure logicielle générique dans laquelle nous avons implémenté les algorithmes présentés dans les deux parties précédentes. À l'aide d'un système d'interfaces avec des

outils externes, nous pouvons tirer parti de la généricité de ces algorithmes pour obtenir des outils pratiques conservant cette généricité. En branchant cette infrastructure à des démonstrateurs automatiques et à la plateforme Why3 [21, 74, 75] pour travailler sur les programmes, nous obtenons des exécutables sur lesquels nous effectuons des tests de performance.

Chapitre 7 — Une infrastructure logicielle générique

Nous utilisons le chapitre 7 pour présenter le développement d'une infrastructure logicielle permettant de tirer partie de la généricité théorique des algorithmes ImpGen-PiD de la partie I et Ilinva de la partie II.

Nous commençons par présenter une méthode pour brancher nos outils avec des démonstrateurs automatiques que nous traitons comme boites noires. Nous proposons pour cela des modèles d'interface capables de construire des tests de satisfaisabilité puis d'invoquer un démonstrateur pour obtenir l'état de la formule.

Le chapitre présente ensuite une infrastructure pour l'implémentation du générateur d'impliqués premiers ImpGen-PiD. Cette infrastructure est constituée de trois composants : un exécutant l'algorithme, un second travaillant sur le choix des littéraux pour construire l'hypothèse servant à obtenir les impliqués, et enfin une interface avec un démonstrateur. Nous détaillons alors la manière dont le composant qui construit les conjonctions de littéraux implémente l'ensemble des aménagements réduisant l'espace de recherche proposés au chapitre 3. Nous appelons l'outil obtenu GPiD. Nous introduisons ensuite un format de fichier pour représenter les littéraux utilisés pour l'abduction et nous détaillons sa syntaxe à l'aide d'exemples.

Nous détaillons également dans ce chapitre la manière dont nous implémentons l'algorithme Ilinva, toujours de manière générique, à l'aide d'une interface qui aura pour but de représenter les problèmes traités. Cette interface s'occupe en particulier de transmettre au composant exécutant l'algorithme les conditions de vérification non-valides du problème qu'elle représente, ainsi que de choisir les formules guides qui devront être modifiées par les solutions retournées par GPiD. L'interface sert enfin à implémenter les techniques de traduction nécessaires pour transformer les conditions de vérification du programme en formules traitables pour la procédure d'abduction et, inversement, les impliqués obtenus en formules syntaxiquement valides dans les assertions du programme. Nous appelons l'outil obtenu Ilinva.

Chapitre 8 — Des interfaces pour brancher l'infrastructures à des outils concrets

Le chapitre 8 nous permet de détailler l'implémentation d'interfaces concrètes. Nous y expliquons les adaptation nécessaires pour brancher de vrais démonstrateurs ou vérificateurs de programmes dans notre infrastructure.

Nous détaillons ainsi la création d'interfaces pour le démonstrateur de logique propositionnelle MiniSAT [70], les démonstrateurs modulo des théories Alt-Ergo [39], CVC4 [7], VeriT [24] et Z3 [50], ainsi que les raisons justifiant la création d'une interface plus

générale, pouvant être utilisée comme sur-couche à n'importe quel démonstrateur capable de traiter des problèmes exprimés au format SMT-LIB [9].

Nous présentons ensuite l'implémentation d'une interface de représentation de programme pour ILINVA qui utilise la plateforme de raisonnement déductif pour programmes WHY3 [21, 74, 75]. Cette plateforme, déjà capable de générer les conditions de vérifications dont nous avons besoin, nous permet de traiter la plupart du travail sur les programmes considérés en boite grise à l'extérieur de notre infrastructure. En représentant nos programmes dans le format de fichier défini par WHY3 (WHYML), nous pouvons obtenir une interface pratique pour la génération d'invariants de boucles dans ces programmes, moyennant quelques ajustements sur la configuration de GPID. Nous détaillons également la manière dont nous extrayons les littéraux pour l'abduction des programmes WHYML, en ajoutant dans nos fichiers de littéraux des informations supplémentaires sur le contexte des variables du programme. Ces informations serviront notamment à retraduire correctement la syntaxe des impliqués de la logique du démonstrateur à la logique de WHYML.

Le chapitre se conclut par des remarques sur la documentation, la compilation et la distribution de l'infrastructure, disponible publiquement sur la plateforme GITHUB [160].

Chapitre 9 — Évaluation expérimentale de GPID et ILINVA

Le dernier chapitre de la partie III est consacré à l'évaluation expérimentale des outils présentés dans les deux chapitres précédents. Après avoir présenté l'infrastructure technique sur laquelle les expériences ont été effectuées, nous étudions les résultats obtenus pour le générateur d'impliqués premiers et le générateur d'invariants.

Nous commençons par l'évaluation de GPID en logique propositionnelle, où nous observerons que sur de nombreux exemples tirés de dépôts standards (not. SAT-LIB [94]), notre outils est capable de générer soit tous les impliqués premiers en quelques secondes, soit au moins quelques impliquées en moins d'une minute. Nous observons également que, comme attendu, l'algorithme se comporte d'autant mieux que les problèmes qu'il traite contiennent moins de littéraux d'abduction ou possèdent peu d'impliqués. En comparant notre outil aux meilleurs concurrents disponibles : ZRES [163] et PRIMER [141], nous constatons que le temps de calcul observé est du même ordre de grandeur que celui de ces outils dès lors que les problèmes considérés possèdent beaucoup de contraintes. Pour les autres problèmes, notre algorithme met généralement plus de temps que les deux autres pour retourner des impliqués.

Nous poursuivons ensuite l'évaluation de GPID en lui faisant traiter des problèmes avec égalités entre variables et des problèmes d'arithmétique linéaire sur les entiers, tous tirés des bibliothèques de problèmes SMT-LIB [9]. Nous observons que, pour tous ces exemples, GPID est capable de générer des impliqués, bien que la génération de tous les impliqués soit en général impossible au regard de la taille importante de cet ensemble. Pour les problèmes equationnels, nous comparons également les performances de notre générateur à celles de CSP [173]. Nous observons que nous sommes systématiquement capable de générer des impliqués plus rapidement que CSP sur les exemples étudiés.

La deuxième partie du chapitre est consacrée à l'évaluation de notre générateur d'invariants de boucles ILINVA. Après avoir construit un ensemble de programmes à partir de problèmes de génération d'invariants habituels et des bibliothèques d'exemples de WHY3, nous testons notre algorithme afin de vérifier s'il est capable de générer des invariants satisfaisants. Nous observons que, même si le temps de calcul est généralement élevé, notre outil est capable de générer des solutions pour des exemples dans diverses théories, dont certaines n'étaient pas connues pour être dans le domaine des outils précédents. Nous nous intéressons également aux limites mises en évidence par les résultats. Nous relevons notamment le fait que le nombre de littéraux envoyés aux problèmes d'abduction a un impact significatif sur la performance du système global et que, d'une manière générale, davantage de travaux pour réduire l'espace de recherche du générateur d'impliqués, peut-être aux dépends de sa complétude, seront nécessaires pour obtenir de meilleures performances.

Partie IV — Un algorithme de minimisation de modèles pour le raisonnement abductif en logique de séparation

La dernière partie de cette thèse est consacrée à l'élaboration d'un algorithme alternatif pour obtenir des impliquants d'une formule logique. Cet algorithme consiste à utiliser un modèle de la formule pour construire un impliquant initial. Ensuite, grâce à une décomposition de cet impliquant en sous-formules et à des règles d'affaiblissement prédéfinies, nous affaiblissant les sous-formules de cet impliquant tant que la formule obtenue reste un impliquant de celle de départ.

Chapitre 10 — Un algorithme générique de minimisation d'implicants

Le chapitre 10 présente l'algorithme de minimisation d'impliquant évoqué ci-dessus. Notre algorithme se base sur une procédure de minimisation de modèles en logique propositionnelle [26]. Cette méthode peut être vue comme la minimisation d'un impliquant d'une formule propositionnelle obtenu en prenant la conjonction de tous les littéraux valides dans un modèle. Elle retire ensuite des littéraux de cet impliquant tant que ces suppressions ne brisent pas son caractère d'impliquant. Cette suppression est réalisée en découpant l'ensemble des littéraux en deux dès lors qu'ils ne peuvent pas tous être retirés, puis en appliquant récursivement la méthode sur ces deux parties en considérant l'autre comme étant fixe. L'algorithme retourne alors un impliquant plus faible que le premier.

L'algorithme que nous présentons étend cette méthode pour qu'elle soit capable de traiter des théories dans lesquelles l'affaiblissement d'un impliquant ne résume pas à la suppression de certains de ses littéraux, mais peut consister à les remplacer successivement par des formules moins fortes. Il permet aussi d'étendre la décomposition en sous parties à plus de deux. Pour faire cela, nous définissons des séquences de formules, qui représentent

les affaiblissements successifs des sous-formules de l'impliquant initial. À l'aide de règles de généralisation, prédéfinies pour le contexte des formules traitées, l'algorithme ajoute à ces séquences les généralisations associées aux formules qu'elles représentent, et ne considère ainsi que la dernière formule de chaque séquence comme partie de l'impliquant potentiel représenté. Grâce à des opérateurs permettant de reconstruire une séquence minimale dont la représentation implique toujours la formule de départ, nous obtenons une structure sur laquelle appliquer l'algorithme pour affaiblir au maximum et sur plusieurs niveaux les sous-formules de l'impliquant initial.

Nous poursuivons ensuite en proposant deux spécialisations de cet algorithme (ce qui consiste à définir les règles de généralisation et les formules considérées) pour obtenir une version équivalent à l'algorithme en logique propositionnelle dont nous nous sommes inspirés [26] et une version permettant de minimiser des formules en logique de séparation [151]. Nous terminons le chapitre en proposant un algorithme simple qui utilise cette procédure de minimisation pour résoudre des problèmes d'abduction en logique de séparation (dans lesquels la conjonctions est remplacée par la conjonction séparée).

Chapitre 11 — Implémentation et évaluation de l'algorithme et de son utilité pour la logique de séparation

Le chapitre 11 présente l'implémentation et l'évaluation de l'algorithme de minimisation et de ses spécialisations présentées dans le chapitre 10. Nous commençons par y décrire une spécialisation simple de l'algorithme qui a pour but de simplifier son évaluation (en en réduisant le temps de calcul). Nous détaillons ensuite les considérations qui doivent être étudiées pour l'implémentation de l'algorithme de minimisation et nous présentons comment cela a été fait dans l'infrastructure Abdulot. Une interface utilisant la procédure de décision vérifiant la satisfaisabilité des formules en logique de séparation proposée par le démonstrateur CVC4 [7] a également été implémentée. Les détails de cette implémentation sont présentés à la suite de celle de l'algorithme de minimisation.

Nous évaluons ensuite l'algorithme en comptant le nombre de tests de satisfaisabilité (nécessaires pour vérifier si une formule est bien un impliquant) qu'il exécute, et le comparant au nombre de tests effectués par un algorithme de minimisation naïf. Les expériences montrent que l'algorithme proposé est particulièrement efficace pour minimiser des impliquants lorsque le nombre de composants que contiennent ces derniers est important ou que ces composants peuvent être généralisés de nombreuses fois (plus de cinq). Il apparait également que cet algorithme est efficace lorsqu'une part importante des composants de l'impliquant initial peuvent être généralisés. Dans les autres cas, les tests de satisfaisabilité provoqués par la décomposition en partitions échouent et provoquent ainsi un surcoût par rapport à l'algorithme naïf. Nous évaluons également le nombre de tests de satisfaisabilité et le temps de calcul nécessaire à la minimisation en pratique de formules simples en logique de séparation, puis vérifions s'il est possible d'utiliser cette méthode en pratique pour résoudre quelques problèmes d'abduction. Les résultats montrent que tant que les formules ne contiennent ni quantificateur ni implication séparée, l'algorithme est capable de minimiser les impliquants obtenus à partir des modèles du démonstrateur

modulo des théories. En revanche, lorsque les formules contiennent des quantificateurs ou implications séparées, le démonstrateur modulo des théories n'est plus en mesure de résoudre les problèmes qui lui sont envoyés.

Conclusion

Les contributions principales de ce travail de thèse sont la conception et l'implémentation de deux systèmes génériques. L'un automatisant un raisonnement abductif en générant des impliqués premiers de formules logiques. L'autre utilisant le premier pour générer des invariants de boucles de programmes.

Les résultats expérimentaux montrent l'intérêt pratique de ces travaux pour générer des solutions à des problèmes qui n'avaient pas de solution automatisée connue. Ils ouvrent également la voie à des travaux de recherche s'intéressant à la limitation de leurs espaces de recherche et à la génération des littéraux abducibles.

Annexe A — Grammaire des fichiers de littéraux abducibles utilisés par Abdulot

Cette annexe présente la grammaire complète de la syntaxe des fichiers de littéraux pour la procédure d'abduction.

Annexe B — Exemple de condition de vérification générée par Why3

Cette annexe présente des conditions de vérification complètes générées par Why3 pour les exemples détaillés dans ce mémoire.

Annexe C — Manuel utilisateur de l'infrastructure logicielle Abdulot

Cette annexe présente un court manuel utilisateur à utiliser pour installer et démarrer l'infrastructure logicielle Abdulot, comportant GPiD, Ilinva, et plusieurs outils associés.

Introduction

One day, one person may make the following hypothesis: "If I am an animal, then I can cross a bridge". Another day, a smarter person will assess the opposite, build a rope bridge between two buildings and have a carp try to cross it.

This ability to produce conjectures and then try to decide whether they are true or false is a structural foundation of mathematics and scientific reasoning [140]. On one end, people devise hypotheses based on either their intuition or observations and, on the other end, they devise theoretical or experimental techniques to either prove or provide evidence for these hypotheses.

Both hypothesis assessment and verification have been done more or less successfully by human beings through history. However, with the development of computer science, it became possible to search for automated methods to perform these bases of mathematical reasoning.

Let us first talk about the problem of automatically deciding whether a given conjecture holds. In computer science, this has been studied under the name of automated theorem proving [153]. Being a little more formal, we first need to decide on a common language to express facts as well as on a logic to determine whether these facts are true or not. Now if we define a theory by a set of axioms, or facts we admit hold, the goal of such methods is to automatically check if given properties can be deduced from them. Typically, if we can define a theory in which we can express what an animal is, what a bridge is and what crossing is, then, we may be interested in obtaining an automatic system to prove or disprove conjectures expressed in it, such as whether an animal can cross a bridge or not. Different approaches have been tried to produce such systems. Some are designed to imitate the process humans use to prove conjectures, by defining deduction rules and letting an algorithm apply them until it reaches the expected conclusion [4, 91, 128]. Others rewrite the problems in a form that is more adapted to the use of computers and design alternative techniques that would be very fastidious for humans to apply but can be naturally processed by computers [47, 178]. This has produced an ecosystem of tools we call automated theorem provers, in which each tool is usually adapted for solving problems in a specific logic and theory (such as propositional logic [70], first order logic [159, 111, 125], modal logic [12, 11, 133], *etc*).

When the conjectures we want to prove are expressed in theories that are too hard for such systems to reason on, one could also use another class of system that are called proof assistants [79]. Such systems may be able to produce a valid logical proof of the conjectures with the help of some additional input from their users. They may also make use of automated theorem provers to produce proofs of parts of their reasoning [19]. For instance, there could exist a bridge crossing simulation that, given the model of an animal and the model of a bridge, can provide a proof detailing why the animal can or

cannot cross the bridge successfully. The user of such a system would then be tasked with providing the simulation with interesting animal and bridge models or abstractions in order to prove some given property on this bridge crossing experiment, such as what family of animals is always able to cross bridges. Again, a large collection of tools has been produced, in which each tool is well-adapted/oriented for handling a particular type of problem or formalism (such as program proofs [21], mathematical or logic proofs [40, 132], *etc*).

Concerning the ability to produce conjectures from a set of observations, various methods have also been explored. The method we take an interest in for this work is a logical formalism called abduction [101]. The idea behind it is, given some logical axioms and a set of facts unexplained by them, to look for additional conjectures that, when added to the initial axioms, allow us to explain those facts. For instance, even though they were shown evidence that not all animals can cross any bridge, our first person may be interested in finding additional restrictions that would make this possible, that is, additional conjectures stating among others the fact that a carp cannot cross a rope bridge. Examples of such conjectures would typically be restrictions on the shape of the animals or on the type of the bridge. Obviously, we do not want these additional conjectures to be too weak (typically by choosing the unexplained facts themselves, such as "if I am an animal that can cross a bridge, then I can cross a bridge"), or contradictory (such as "if I am an animal that is not an animal, then I can cross a bridge"). What we actually look for is a set of conjectures as generic as possible, consistent, and if possible minimal.

Many techniques have been tried to automatically perform abductive reasoning, some of which will be presented in Chapter 2. In this thesis, we devise a new method for it that is theory-generic and, as such, can be applied to solve abduction problems as long as there exists an algorithm to check if a given additional candidate permits to prove the initial goal. Our technique uses an initial set of abducible literals, that is, a set of elementary hypotheses we have good reasons to think may be part of satisfying additional candidates. The method then efficiently builds logical conjunctions of these literals to produce such additional candidates. The theorem prover is used to check whether these hypotheses permit to entail de desired property. If we go back to our illustration and assume that we do own some sort of animal-bridge checker that, given an animal and a bridge, can tell us whether or not the animal can cross the bridge, our algorithm would then build from an initial set of animal parts and bridge parts possible solutions for the problem. It will then check these possible solutions using the animal-bridge checker and return only those that work. As our algorithm returns minimal solutions, we will obtain the set of minimal restrictions on our animals and bridges that could allow them to successfully perform this crossing experiment.

Still, a picky person may wonder how we can be certain that the program we use to check if an animal can cross a bridge is not broken, giving results that are not actually consistent due to some implementation mistakes or design incompleteness. Thankfully, theorem proving can be naturally applied to the verification of systems [177], in particular for computer software, hardware structures [84], *etc*. Indeed, with the development of computer programs and their applications to many critical tasks (such as traffic lights, automated pilot, banking, *etc*), people got interested in obtaining evidence that these programs were indeed executing the tasks they were designed to accomplish. This has first

been done extensively by running tests on the programs, but this method does not ensure any property out of the conditions of these tests, which can have tragic consequences.

In order to obtain such proofs, researchers have designed so-called static analysis verification techniques. Such techniques use the programs themselves along with a model of their behaviour to try and prove the expected properties. Here also, it is possible that a given technique fails to return a proof for a given program property. For those cases, researchers have designed methods to find additional preconditions on the inputs of a program that could be added to ensure that its properties are verified. Again, a brief overview of some of these techniques will be presented in Chapter 2. In this thesis, we use a generic algorithm developed in [57] that uses abduction to solve this second problem. To prove a property on a program, such as the fact that a condition is always satisfied at the end of its execution, it is usually necessary to establish intermediate properties within the program, to show that those are also verified, that they are preserved by the loops of the programs or by the functions it invokes. By starting from an initially empty set of additional conjectures, we try to prove the properties expected from the handled program and, if this fails, we use abduction to produce the additional conjectures allowing the system to make this proof. Thanks to the genericity of the abduction algorithm we devised, we will be able to use this algorithm to target the generation of preconditions in a large range of problems.

This covers the scope of this thesis: the design of an automated theory-generic method of abduction and its application to program verification. It presents both the theoretical frameworks and their implementation and shows that they permit to generate solutions in a large range of theories.

Outline and publications

This thesis is organized as follows. After recalling the theoretical notions and terminology we use (Chapter 1) and presenting the related and state-of-the-art automated techniques used to solve the issues we introduced (Chapter 2), we will present our techniques for abductive reasoning (Part I) and for program verification (Part II, extending the work of Dillig *et al.* [57]). These two works have been published and presented in international peer-reviewed committee computer science conferences: the abduction algorithm at the International Joint Conference on Automated Reasoning (IJCAR), 2018, Oxford, U.K. [65] and the theory-generic abduction-based loop invariant generator at the Frontiers of Combining Systems (FroCoS) conference, 2019, London, U.K. [66]. These two theoretical framework descriptions will be followed in Part III by the details on how we implemented our approaches in practice. We will also perform an experimental evaluation of the resulting tools. Finally, Part IV will present a closely related implicant minimization algorithm extending a propositional logic model minimizer [26]. This technique will use the models we can recover from the satisfiability tests we use to prove that a given set of literals is an implicant of a formula (such as the ones we used in Part I), construct an implicant from this model and refine this implicant by decomposing it into several components that will be weakened independently. This implicant minimizer will additionally be used to perform abduction in separation logic [151].

Contents

List of Algorithms and Interfaces

List of Figures

List of Tables

Aknowledgements

First, I would like to thank all the members of my PhD committee for the time they invested in reading this thesis, for the comments the provided and for remaining available to reschedule the defense, test the video-conference system and attend this unusual defense in those times of covid-19 pandemic.

I also want to thank my two advisors, Nicolas and Mnacho, for the wonderful job they have done supervising me, providing help and directions when needed and without whom I couldn't have hoped to achieve the results I present here. I can also thank the other members of the CAPP team for welcoming me and hosting me during this work.

A word to the friends I spent these years with: Nicolas, Valentin, Remy, Clément, Cyril, with whom I had the joy to share both interesting and uninteresting discussions as well as the tasteful meals of the university restaurants.

Finally, I have to thank my family for the support they haven given me throughout my thesis.

Chapter 1

Context and general definitions

This chapter presents the theoretical background that is common to all the parts of this work. We start by recalling usual notions of logic, then pursue with the definitions of the core concepts this work is built upon. In the last section, we will also recall definitions required by the use of theories as external dependencies.

1.1 Usual notions of logics

The notions and notations used within this work for propositional logic and first order logic are mostly standard. We present the exact formalism we use in this thesis and highlight some required differences with standard definitions, as introduced in, *e.g.*, [34].

Definition 1. A *sorted signature* Σ is a set of function symbols and types (or sorts), as well as a function σ assigning to every function in its domain a tuple of sorts the first element of which defines the *return type* of the function's value and the other elements define the type of its *parameters*. We assume that a signature Σ always contains a boolean sort $\mathbb{B}$. If the return type of a function is this boolean sort, we will call this function a *predicate*. For clarity reasons, we will assume that the denomination *function* in this context will not refer to predicates. We will call *arity* of a function or predicate symbol the number of its parameters. If the arity of a function or predicate is equal to zero, we will call it a *constant*. We also assume that a countably infinite set of *variables* $\mathcal{V}$ of every sort is given.

In all the following, will assume given a fixed sorted signature Σ and a set of variables $\mathcal{V}$ on its sorts.

Notation 2. Given a variable $v \in \mathcal{V}$, we will denote by $sort(v)$ the sort of the variable v.

Definition 3. We define the *terms* on Σ recursively as follows:

- Any constant of Σ is a term.
- Any variable of $\mathcal{V}$ for any sort is a term.
- Given a function or predicate f in Σ such that $\sigma(f) = (s_0, s_1, \dots s_n)$ and $t_1, \dots, t_n$ such that n is the arity of f and such that for any $i \in [\![1, n]\!]$, t_i is either a constant

of sort s_i, a variable of sort s_i or a term whose outermost symbol has the return type s_i, then $f(t_1, \ldots, t_n)$ is a term.

Remark 4. Note that, contrarily to the usual practice, we allow function or predicate symbols with boolean parameters. This choice has been made to ease the use of generic provers and fits in with the SMT-LIB standard.

Remark 5. For simplicity reasons, we will assume that Σ contains an equality predicate for all the sorts it contains. We will denote these predicates using their usual infix notation $\simeq$, independently of the sort they apply to.

Definition 6. We call *atoms* the terms of the sort $\mathbb{B}$. We will denote by $\mathcal{K}$ the set of *atoms* that can be built on Σ and $\mathcal{V}$.

Definition 7. We define the *set of logical formulas on* Σ, denoted by $\mathfrak{S}$, inductively as follows:

- $\top$ and $\bot$ are elements of $\mathfrak{S}$.
- Any atom of $\mathcal{K}$ is an element of $\mathfrak{S}$.
- If $\phi \in \mathfrak{S}$, then its negation $\neg\phi$ is in $\mathfrak{S}$.
- For any two formulas $\phi, \psi \in \mathfrak{S}$, their conjunction $\phi \wedge \psi$ and their disjunction $\phi \vee \psi$ are also formulas in $\mathfrak{S}$.
- Given a formula $\phi \in \mathfrak{S}$ and a variable $v \in \mathcal{V}$ the quantified expressions $\forall v.\phi$ and $\exists v.\phi$ are formulas in $\mathfrak{S}$.

Definition 8. We call *literal* either an atom of $\mathcal{K}$ or its negation. We will denote by $\mathcal{L}$ the set of all literals that can be built on Σ and $\mathcal{V}$.

Notation 9. For any two formulas $\phi, \psi \in \mathfrak{S}$, we will use the usual notations $\phi \Rightarrow \psi$ for $\neg\phi \vee \psi$ and $\phi \Leftrightarrow \psi$ for $(\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)$.

Definition 10. Given a formula $\phi \in \mathfrak{S}$, we define $\overline{\phi}$ the *complement of* ϕ inductively as follows:

- $\overline{\top} = \bot$.
- $\overline{\bot} = \top$.
- If ϕ is an atom of $\mathcal{L}$, then $\overline{\phi} = \neg\psi$.
- If $\phi = \neg\psi$ with $\psi \in \mathfrak{S}$, then $\overline{\phi} = \psi$.
- If $\phi = \psi_1 \wedge \psi_2$ with $\psi_1, \psi_2 \in \mathfrak{S}$, then $\overline{\phi} = \overline{\psi_1} \vee \overline{\psi_2}$.
- If $\phi = \psi_1 \vee \psi_2$ with $\psi_1, \psi_2 \in \mathfrak{S}$, then $\overline{\phi} = \overline{\psi_1} \wedge \overline{\psi_2}$.
- If $\phi = \forall v.\psi$ with v a variable in Σ and $\psi \in \mathfrak{S}$, then $\overline{\phi} = \exists v.\overline{\psi}$.
- If $\phi = \exists v.\psi$ with v a variable in Σ and $\psi \in \mathfrak{S}$, then $\overline{\phi} = \forall v.\overline{\psi}$.

Example 11. In order to illustrate the notions introduced in this chapter, we will consider a signature Σ_0 containing the function symbols a, b, c of arity 0 and return type $\mathbb{N}$, the usual unary minus and sum functions $-, +$ over integers, and the usual equality predicate $\simeq$. In this context, expressions such as $a \simeq b$, $\neg(b \simeq c)$, $a + b \simeq -c$ are elements of $\mathcal{L}_0$. Likewise, expressions such as $a \simeq b \wedge \neg(b \simeq c)$, $\neg(a \simeq b) \vee c \simeq c$, $\neg(a \simeq c) \wedge a \simeq b$ are elements of $\mathfrak{S}_0$.

We now briefly recall the usual notions of interpretation, model and satisfiability. Though we will mostly work with total interpretation and models in this thesis, partial models will be required in Section 3.4. This explains why we give a definition for partial interpretations and not exclusively for total interpretations.

Definition 12. Given a signature Σ, an *interpretation* of Σ is a tuple $I : (I^s, I^f)$ such that:

1. I^s is a total function associating to each sort s in Σ a non-empty set of possible values denoted by $I^s(s)$. We also impose that, for any two sorts s_0, s_1 in Σ, if $s_0 \neq s_1$ then $I^s(s_0) \cap I^s(s_1) = \emptyset$.
2. $I^s(\mathbb{B}) = \{\top, \bot\}$.
3. I^f is a partial function associating to each function symbol $f \in \Sigma$, in the domain of σ and such that $\sigma(f) = (s_0, s_1, \ldots, s_n)$, $I^f(f)$ a function from $I^s(s_1) \times \ldots \times I^s(s_n)$ to $I^s(s_0)$.

Definition 13. Given a set of variables of $\mathcal{V}$ and an interpretation I, we call *interpretation of the variables*, and denote by I^v, a partial function associating to any variable v of the sort s in its domain a value in $I^s(s)$. Given an interpretation of the variables I^v, a variable v and $x \in I^s(sort(v))$, we will denote by $I^v_{v \mapsto x}$ the function defined on $\textsc{Dom}(I^v) \cup \{v\}$, such that

$$\forall v' \in \textsc{Dom}(I^v) \setminus \{v\}, I^v_{v \mapsto x}(v') = I^v(v)$$

and such that

$$I^v_{v \mapsto x}(v) = x.$$

Definition 14. Given a term t on Σ, an interpretation I and an interpretation of the variables of $\mathcal{V}$ I^v, the *evaluation of the term* t, denoted by $[\![t]\!]_{I,I^v}$, is defined inductively as follows:

- If t is a constant, then $[\![t]\!]_{I,I^v}$ is the image of $I^f(t)$ if $t \in \textsc{Dom}(I^f)$ and is undefined otherwise.
- If t is a variable, then $[\![t]\!]_{I,I^v} = I^v(t)$ if $t \in \textsc{Dom}(I^v)$ and is undefined otherwise.
- If $t = f(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms, then $[\![t]\!]_{I,I^v}$ is undefined if the evaluation of at least one of the terms $t_1, \ldots, t_n$ is undefined or if f is not in the domain of I^f. Otherwise, $[\![t]\!]_{I,I^v} = I^f(t)([\![t_1]\!]_{I,I^v}, \ldots, [\![t_n]\!]_{I,I^v})$.

Definition 15. Given a formula ϕ, an interpretation I of Σ and an interpretation of the variables I^v, we call *evaluation of* ϕ *in* I, I^v and denote by $[\![\phi]\!]_{I,I^v}$ the value in $\{\top, \bot, \iota\}$ defined inductively as follows:

- $[\![\top]\!]_{I,I^v} = \top$.
- $[\![\bot]\!]_{I,I^v} = \bot$.
- If ϕ is an atom, then $[\![\phi]\!]_{I,I^v}$ is equal to the evaluation of the term ϕ if it exists. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.
- If $\phi = \neg\psi$ then $[\![\phi]\!]_{I,I^v} = \top$ if and only if $[\![\psi]\!]_{I,I^v} = \bot$ and $[\![\phi]\!]_{I,I^v} = \bot$ if and only if $[\![\psi]\!]_{I,I^v} = \top$. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.
- If $\phi = \psi_1 \wedge \psi_2$ then $[\![\phi]\!]_{I,I^v} = \top$ if and only if both $[\![\psi_1]\!]_{I,I^v} = \top$ and $[\![\psi_2]\!]_{I,I^v} = \top$. If either $[\![\psi_1]\!]_{I,I^v} = \bot$ or $[\![\psi_2]\!]_{I,I^v} = \bot$, then $[\![\phi]\!]_{I,I^v} = \bot$. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.
- If $\phi = \psi_1 \vee \psi_2$ then $[\![\phi]\!]_{I,I^v} = \top$ if and only if either $[\![\psi_1]\!]_{I,I^v} = \top$ or $[\![\psi_2]\!]_{I,I^v} = \top$. If both $[\![\psi_1]\!]_{I,I^v} = \bot$ and $[\![\psi_2]\!]_{I,I^v} = \bot$, then $[\![\phi]\!]_{I,I^v} = \bot$. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.
- If $\phi = \forall v.\psi$ with v a variable in Σ then $[\![\phi]\!]_{I,I^v} = \top$ if and only if, for any element x in $I^s(sort(v))$ we have $[\![\psi]\!]_{I,I^v_{v\mapsto x}} = \top$. If there exists an element x of $I^s(sort(v))$ such that $[\![\psi]\!]_{I,I^v_{v\mapsto x}} = \bot$, then $[\![\phi]\!]_{I,I^v} = \bot$. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.
- If $\phi = \exists v.\psi$ with v a variable in Σ then $[\![\phi]\!]_{I,I^v} = \top$ if and only if there exists an x element of $I^s(sort(v))$ such that $[\![\psi]\!]_{I,I^v_{v\mapsto x}} = \top$. If for any element x of $I^s(sort(v))$ we have $[\![\psi]\!]_{I,I^v_{v\mapsto x}} = \bot$, then $[\![\phi]\!]_{I,I^v} = \bot$. Otherwise, $[\![\phi]\!]_{I,I^v} = \iota$.

Given a closed formula ϕ and an interpretation I, we call *evaluation of ϕ in I* and denote by $[\![\phi]\!]_I$ the value $[\![\phi]\!]_{I,\emptyset}$.

Definition 16. We say that an interpretation I is *total* if I^f is a total function. Otherwise, we say that I is a *partial interpretation.*

Proposition 17. *Let I be a total interpretation. For any formula ϕ with no free variables, $[\![\phi]\!]_I \in \{\top, \bot\}$.*

In all the following, unless we explicitly mention the use of partial interpretations, we will only consider closed formulas and total interpretations.

Example 18. Assume we interpret the functions and predicates of Example 11 with their usual definition on integers. If we take an interpretation I that maps, a and c to 0 and b to 1, then we will have $[\![a \simeq c]\!]_I = \top$, $[\![a + b \simeq c]\!]_I = \bot$, $[\![a \simeq c \wedge \neg(a \simeq b)]\!]_I = \top$.

Remark 19. We will assume that for any interpretation I and for any sort s of Σ, we have for the equality predicate $\simeq$ of s that $I^f(\simeq) = (t_l, t_r) \mapsto t_l = t_r$.

Definition 20. Given a formula ϕ and an interpretation I, we say that I is a *model* of ϕ if $[\![\phi]\!]_I = \top$. In this case, we will write $I \models \phi$.

If we review Example 18, we can say that the interpretation I is a model of $a \simeq c \wedge \neg(a \simeq b)$.

Definition 21. Let ϕ_1 and ϕ_2 be two formulas. We say that ϕ_1 *entails* ϕ_2, and write $\phi_1 \models \phi_2$, if all the total models of ϕ_1 are also models of ϕ_2. In this case, we will also say that ϕ_2 is a *logical consequence* of ϕ_1.

Definition 22. If there exists a model of a formula ϕ, we say that ϕ is *satisfiable*. If no such interpretation exists, we say that ϕ is *unsatisfiable*.

Proposition 23. *A formula ϕ is unsatisfiable if and only if $\phi \models \bot$.*

Notation 24. As a consequence of proposition 23, we will use the notation $\phi \not\models \bot$ to denote that a formula ϕ is satisfiable and $\phi \models \bot$ to denote that ϕ is unsatisfiable.

Typically, $a \simeq c \wedge \neg(a \simeq b)$ is a satisfiable formula, as we exhibited a model of it. The formula $a \simeq b \wedge \neg(a \simeq b)$ however, does not admit any model, as an interpretation I' such that $[\![a \simeq b]\!]_{I'} = \top$ will impose that $[\![\neg(a \simeq b)]\!]_{I'} = \bot$ and conversely. This formula is thus unsatisfiable.

Definition 25. Given a signature Σ and the set $\mathfrak{S}$ of logical formulas we can build on it, we call *theory* a subset $\mathcal{T} \subseteq \mathfrak{S}$ such that any logical consequence of $\mathcal{T}$ belongs to $\mathcal{T}$.

We will say that an interpretation I is *a model of the theory* $\mathcal{T}$ if $I \models \mathcal{T}$.

Definition 26. Let $\mathcal{T}$ be a theory. Given a formula ϕ and an interpretation I such that $I \models \mathcal{T}$, we say that I is a *$\mathcal{T}$-model* of ϕ, and write $I \models_{\mathcal{T}} \phi$, if $[\![\phi]\!]_I = \top$.

If such an interpretation exists for a given formula ϕ, we will say that this formula is *$\mathcal{T}$-satisfiable*. If no such interpretation exists, we will say that ϕ is *$\mathcal{T}$-unsatisfiable*

When all the total $\mathcal{T}$-models of a formula ϕ are also models of another formula ψ, we will write $\phi \models_{\mathcal{T}} \psi$ and say that ϕ *entails* ψ *in* $\mathcal{T}$ or that ψ is a *$\mathcal{T}$-consequence* of ϕ.

Proposition 27. *Given a theory $\mathcal{T}$, a formula ϕ is $\mathcal{T}$-unsatisfiable if and only if $\phi \models_{\mathcal{T}} \bot$.*

Similarly to what we did in Notation 24, we may thus write $\phi \not\models_{\mathcal{T}} \bot$ to denote that ϕ is $\mathcal{T}$-satisfiable and $\phi \models_{\mathcal{T}} \bot$ to denote that it is $\mathcal{T}$-unsatisfiable.

Usual examples of theories include the theory of linear arithmetic [64, 83] or the theory of arrays [33]. The main goal of this work is to define systems that are theory-independent. This means that we do not want to design algorithms that will use particular properties of one given theory but rather that will remain generic until instantiated with external systems that deal with the specific theory.

In all the following, and unless specified otherwise, we will assume that we are given a specific but undetermined theory $\mathcal{T}$. We will also assume the existence of a correct procedure to check whether a given formula is $\mathcal{T}$-satisfiable or not, that is, for which it is guaranteed that when it gives an answer, this answer correctly describes the status of the formula. Even though it is clear that it is better to work with decidable procedures, we do not impose the decidability of the satisfiability problem in general. We will later propose solutions to handle semi-decidable procedures at the expense of some completeness properties for the algorithms we will present.

Definition 28. We call *tautology* any formula that is satisfied by every total interpretation. Given a theory $\mathcal{T}$, we call *$\mathcal{T}$-tautology* any formula that is satisfied by every model of $\mathcal{T}$.

Notation 29. In all the following, we will identify sets of formulas with the conjunction of their elements. That is, given a finite set of formulas $\Phi = \{\phi_1, \dots \phi_n\}$ and any other formula ψ, the notation $\psi \models \Phi$ will be equivalent to $\psi \models \bigwedge_{i \in [\![1,n]\!]} \phi_i$, the formula $\psi \wedge \Phi$ to $\psi \wedge \bigwedge_{i \in [\![1,n]\!]} \phi_i$, *etc.*

Within the context of Example 11, we will write indistinctively $\{a \simeq b, \neg(c \simeq b)\}$ and $a \simeq b \wedge \neg(c \simeq b)$.

Definition 30. We define *clauses* as possibly empty disjunctions of literals with no repetition. If a clause contains only one literal, we will call it is a *unit clause*. If a clause does not contain any literal, we will call it the *empty clause*.

A consequence of Definition 30 is that, for two clauses C and D, if $C \vee D$ is also a clause, then C and D share no literal.

For clarity reasons, we will also identify unit clauses with the literal they contain.

In this work, we will mostly study the generation of logical consequences that can be represented as clauses.

Notation 31. We will denote the *empty clause* by $\bot$. This is justified by the fact that the empty clause has no model.

Notation 32. As a consequence of Notation 29, for a given clause $C = l_1 \vee \ldots \vee l_n$ we will denote by $\overline{C}$ the set $\{\overline{l_1}, \ldots, \overline{l_n}\}$.

Notation 33. Given a set of clauses Φ, we will denote by $\mathrm{Lits}(\Phi)$ the set of literals contained in at least one clause of Φ.

Proposition 34. *Let l be a literal and let C, D be clauses. The following statements hold:*

1. $l \vee C \models_{\mathcal{T}} D$ *iff* $l \models_{\mathcal{T}} D$ *and* $C \models_{\mathcal{T}} D$.

2. $C \models_{\mathcal{T}} l \vee D$ *iff* $C \wedge \bar{l} \models_{\mathcal{T}} D$.

Proof. Let l be a literal and let C, D be clauses.

1. Let us assume that $l \vee C \models_{\mathcal{T}} D$. The definition of the disjunction gives us both that $l \models_{\mathcal{T}} l \vee C$ and $C \models_{\mathcal{T}} l \vee C$. This means that all models l are models $l \vee C$, thus models of D. Thus we have that $l \models_{\mathcal{T}} D$. Similarly we have that $C \models_{\mathcal{T}} D$.

   Conversely, if we assume that both $l \models_{\mathcal{T}} D$ and $C \models_{\mathcal{T}} D$, as any model of $l \vee C$ if necessarily a model of l and/or a model of C, we have that $l \vee C \models_{\mathcal{T}} D$.

2. Let us assume that $C \models_{\mathcal{T}} l \vee D$. Let us take a model of $C \wedge \bar{l}$. The definition of conjunctions ensures that this model is both a model of C and of $\bar{l}$. The first point gives us that it is model of $l \vee D$ and the second that it is not a model of l, hence that it is necessarily a model of D. Conversely, if we assume that $C \wedge \bar{l} \models_{\mathcal{T}} D$, we can split the models of C into those that also satisfy $\bar{l}$ (in which case the hypothesis gives us that they satisfy D, thus also $l \vee D$), and those that do not, and thus necessarily satisfy l, thus $l \vee D$.

□

1.2 Implicates and implicants

The main line of contribution of the work presented here is the computation and usage of logical consequences of a given formula (the implicates of this formula). We will restrict this work to the computation of logical consequences in the form of clauses, and define the corresponding notions accordingly:

Definition 35. We say that a clause C is an *implicate* of a formula ψ if $\psi \models C$. Similarly, we say that C is a $\mathcal{T}$*-implicate* of ψ if $\psi \models_{\mathcal{T}} C$.

We also define the dual notion of *implicants*: a set of literals Q is an *implicant* of ψ if $Q \models \psi$ and a $\mathcal{T}$*-implicant* of ψ if $Q \models_{\mathcal{T}} \psi$.

Remark 36. The definitions of implicates and implicants could be extended to include formulas that cannot be represented by a conjunction of clauses. As we will not compute formulas of this form in this work, we will stick to this definition to clarify what we compute.

It is easy to see that the implicates of a formula can be numerous and redundant. Let us take, for instance, the formula $a \simeq b \vee b \simeq c$ in the context of Example 11. If interpreted within the theory of linear arithmetic, this formula admits many implicates (such as $\neg(a \simeq c) \vee a \simeq b$, $a \simeq b \vee a \simeq c \vee b \simeq c$, $a \simeq b \vee a \simeq c \vee b \simeq c$ – in a different order – or $a \simeq b \vee b \simeq c$ – the formula itself) which could all be derived from the formula itself. This is why we are interested in a minimality property for the set of implicates of a formula. To obtain a set of implicates only composed of formulas that carry mutually distinct information from the others, we define the concept of prime implicates.

Definition 37. An implicate C of a formula ψ is a *prime implicate* of ψ if for every implicate D of ψ, $D \models C$ only if $C \models D$. Similarly, C is a *prime* $\mathcal{T}$*-implicate* of ψ if for every $\mathcal{T}$-implicate D of ψ, $D \models_{\mathcal{T}} C$ only if $C \models_{\mathcal{T}} D$.

Similarly to what we did in Definition 35, we will also consider the dual notion of *prime implicant* and *prime* $\mathcal{T}$*-implicant*: an implicant Q of a formula ψ is a *prime implicant* of ψ if for every implicant D of ψ, $D \models Q$ only if $Q \models D$, and it is a *prime* $\mathcal{T}$*-implicant* of ψ if for every $\mathcal{T}$-implicant D of ψ, $D \models_{\mathcal{T}} Q$ only if $Q \models_{\mathcal{T}} D$.

As previously explained, we will mostly consider sets of formulas (typically sets of implicates). This means that will we need a way to express whether a formula is already a consequence that could be derived from an element in the set. The following definitions serve this purpose.

Definition 38. Given a set of formulas Φ and a formula ϕ, we say that Φ *strongly entails* ϕ, and write $\Phi \Vdash \phi$, if there is a formula $\psi \in \Phi$ such that $\psi \models \phi$. Identically, we will say that Φ *strongly entails* ϕ *in* $\mathcal{T}$, and write $\Phi \Vdash_{\mathcal{T}} \phi$, if there is a formula $\psi \in \Phi$ such that $\psi \models_{\mathcal{T}} \phi$.

Definition 39. Given two sets of formulas Φ and Ψ, we say that Φ *strongly entails* Ψ, and write $\Phi \Vdash \Psi$, if Φ strongly entails all the formulas in Ψ. Identically, we will say that Φ *strongly entails* Ψ *in* $\mathcal{T}$, and denote $\Phi \Vdash_{\mathcal{T}} \Psi$, if and only if $\forall \psi \in \Psi, \Phi \Vdash_{\mathcal{T}} \psi$.

Definition 40. We will say that two sets of formulas Φ_1 and Φ_2 are *identical modulo $\mathcal{T}$-equivalence* if they verify both $\Phi_1 \Vdash_{\mathcal{T}} \Phi_2$ and $\Phi_2 \Vdash_{\mathcal{T}} \Phi_1$. Such a property will be written $\Phi_1 \sim_{\mathcal{T}} \Phi_2$.

Such sets carry the same information about the consequences of the formulas they represent.

The following definitions are aimed at defining more precisely the problem we tackle, and a first requirement for that is the definition of a framework to reduce a set of implicates into a minimal equivalent one.

Notation 41. In all the following, we will assume the existence of a total well-founded order $\prec$ on clauses, that agrees with inclusion. This means that for any two clauses C and D, we will have $C \prec D$ whenever $C \subsetneq D$.

Example 42. Let us consider the clauses $C_0 : a \simeq b$, $C_1 : a \simeq b \vee c \simeq d$, $C_2 : c \simeq d$. As we have $C_0 \subset C_1$ and $C_2 \subset C_1$, we will necessarily have $C_0 \prec C_1$ and $C_2 \prec C_1$. The clauses C_0 and C_2 can then be ordered arbitrarily, *e.g.*, $C_0 \prec C_2$.

Definition 43. Given a set of clauses Φ, we denote by $\textsc{SubMin}(\Phi)$, and call *entailment minimal equivalent of* Φ, the set obtained by deleting from Φ all clauses D such that one of the following hold:

1. D is a tautology.
2. There exists $C \in \Phi$ such that C entails D and either C is not an implicate of D or $C \prec D$.

Similarly, given a theory $\mathcal{T}$, we call *subsumption minimal equivalent in $\mathcal{T}$*, and denote by $\textsc{SubMin}_{\mathcal{T}}(\Phi)$, the set obtained by deleting from Φ all clauses D such that either

1. D is a $\mathcal{T}$-tautology.
2. There exists $C \in \Phi$ such that C entails D in $\mathcal{T}$ and either C is not a $\mathcal{T}$-implicate of D or $C \prec D$.

Example 44. If we consider the set of clauses $\Psi = \{a \simeq b, a \simeq b \vee c \simeq d, c \simeq d, a \simeq a, d \simeq c\}$, with $\simeq$ interpreted with the usual definition of the equality, we first notice that the clause $a \simeq a$ is a $\mathcal{T}$-tautology. As we saw in the previous example, $a \simeq b \vee c \simeq d$ is a logical consequence of both $a \simeq b$ and $c \simeq d$. We also have both $c \simeq d \models_{\mathcal{T}} d \simeq c$ and $d \simeq c \models_{\mathcal{T}} c \simeq d$. Let us assume that we have $c \simeq d \prec d \simeq c$. In this case, we will have $\textsc{SubMin}_{\mathcal{T}}(\Psi) = \{a \simeq b, c \simeq d\}$ as we removed the $\mathcal{T}$-tautology, the weaker logical consequence $a \simeq b \vee c \simeq d$ and kept only the smallest of the two equivalent clauses $c \simeq d$ and $d \simeq c$.

Proposition 45. *Given a set of clauses Φ containing at least one formula that is not a $\mathcal{T}$-tautology,* $\textsc{SubMin}(\Phi) \sim_{\mathcal{T}} \Phi$.

Proof. $\textsc{SubMin}(\Phi) \subset \Phi$ by definition. This gives us that $\Phi \Vdash_{\mathcal{T}} \textsc{SubMin}(\Phi)$ as any formula entails itself. Now let us take a formula $D \in \Phi$. If $D \in \textsc{SubMin}(\Phi)$, then it entails itself and we have found a satisfying entailing formula. If it is not the case, then by Definition 43, it is either a $\mathcal{T}$-tautology (in which case it is entailed by any formula

of SubMin(Φ)) or it verifies the second point of the definition. In this case, there exists $C \in \Phi$ entailing D that is either not entailed by D or smaller than D with respect to $\prec$. If $C \in$ SubMin(Φ), we have found our entailing clause. If not, we can recursively apply the same definition to C. We are certain to find one clause of SubMin(Φ) because $\prec$ is well-founded, This gives us that SubMin(Φ) $\Vdash_{\mathcal{T}} \Phi$ □

Definition 46. In the following, we will also consider the notion of *abducible literals*, that is, a finite subset of $\mathcal{L}$ upon which we want to obtain implicates or implicants of a given formula.

Given a set of abducible literals $\mathcal{A}$, we will say that an implicate I of a formula ϕ is an $\mathcal{A}$-*implicate* of ϕ if $\overline{I} \subseteq \mathcal{A}$. Similarly, we will say that an implicant I of ϕ is an $\mathcal{A}$-*implicant* of ϕ if $I \subseteq \mathcal{A}$, that a $\mathcal{T}$-implicate I of ϕ is a $(\mathcal{T}, \mathcal{A})$-*implicate* of ϕ if $\overline{I} \subseteq \mathcal{A}$, and that a $\mathcal{T}$-implicant I of ϕ is a $(\mathcal{T}, \mathcal{A})$-*implicant* of ϕ if $I \subseteq \mathcal{A}$.

Intuitively, the set of abducible literals will contains the literals that are considered interesting in the context of the formula we are studying. Typically, if the formula is obtained from the logical representation of a program, it could exclude the literals containing symbols not corresponding to any inputs available to the programmer.

Example 47. Let us consider the sample program presented in Algorithm 1.1 on two integer inputs x and y. And let us consider the problem of computing a sufficient precondition for the program not to perform a division by zero at Line 4. An obvious precondition for this to be verified would be that $\delta \neq 0$. However, it is clear that, in the context, the user has no way to act directly on the value of δ, as it is a variable only used by the program internally. This means that the precondition we are actually interested in is expressed in the form of $x + y \neq 0$ (which in this case is strictly equivalent). This type of issue can be solved by using this concept of abducible literals. Indeed, in this context composed of the three integer constants x, y, δ, will we define a set of abducible literals such that it does not contain any literal referencing δ. This will permit to prune the logical solutions containing it and to generate only the ones that are relevant in the context.

It is clear that the notion of abducible literals puts an additional burden on the user of any method using it, as he will also be tasked with defining this set and forwarding it to the system. We will see (Chapter 6.4) however that we can design generic methods relevant to the problems we tackle that generate sets of abducible literals that, even if they contain some non-relevant elements, allow to obtain an acceptable solution for the problems we aim to solve.

Algorithm 1.1: SP0

```
1 input x Int;
2 input y Int;
3 let δ ← x + y;
4 return x/δ ;
```

We now define the notations we will use to represent the sets of implicates we will generate.

Notation 48. For any given theory $\mathcal{T}$, set of formulas Φ and set of abducible literals $\mathcal{A}$, we will denote by $\mathcal{I}_{\mathcal{T},\mathcal{A}}(\Phi)$ the set consisting of all the $(\mathcal{T}, \mathcal{A})$-implicates of Φ, and by $\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi)$ the one consisting all the prime $(\mathcal{T}, \mathcal{A})$-implicates of Φ.

In a context where we do not consider any theory, we will also use the notation $\mathcal{I}_{\mathcal{A}}(\Phi)$ for the set of $\mathcal{A}$-implicates of Φ, as well as $\mathcal{I}^*_{\mathcal{A}}(\Phi)$ for its prime $\mathcal{A}$-implicates. Likewise, we will allow the notations $\mathcal{I}_{\mathcal{T}}(\Phi)$ and $\mathcal{I}^*_{\mathcal{T}}(\Phi)$ when a theory but no abducible literal set is considered.

Proposition 49. *For any given theory $\mathcal{T}$, set of formulas Φ and set of abducible literals $\mathcal{A}$, we have $\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi) \sim_{\mathcal{T}} \textsc{SubMin}_{\mathcal{T}}(\mathcal{I}_{\mathcal{T},\mathcal{A}}(\Phi))$.*

Proof. This is a direct consequence of Proposition 45 and of the fact that $\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi) \sim_{\mathcal{T}} \mathcal{I}_{\mathcal{T},\mathcal{A}}(\Phi)$. □

1.3 Theories as external dependencies

As explained in both this chapter's introduction and content, one of the major advantages of the work presented in this thesis is its independence on the concrete theory it handles. Users can express the problems they want to solve in the theory they deem appropriate then plug our systems with tools that can handle the theories they chose. This genericity suggests that we can abstract the usage of the theory-dependent elements used within the methods we provide. This will be done through the framework we present in this section.

Definition 50. Given a signature Σ and a theory $\mathcal{T}$, we call *satisfiability problem in* $\mathcal{T}$ the problem of deciding whether a given formula admits a model in $\mathcal{T}$, that is if there exists a model of $\mathcal{T}$ that is also a model of this formula.

In the work presented in this thesis, all the tasks delegated to theory-related operations will be in the form of satisfiability tests. This means that as long as one can provide an algorithm to decide such satisfiability queries in a theory, it will be possible to use the methods presented in this work within this theory. We formalize this fact with the following definition:

Definition 51. By a slight abuse of words, we will say that a given theory $\mathcal{T}$ is *decidable* if there exists an algorithm that is able to decide for any formula ϕ expressible within $\mathcal{T}$ if it admits a model in $\mathcal{T}$ or not. If no such algorithm exists, we shall say that $\mathcal{T}$ is *undecidable*.

In what follows, when we talk about theories and genericity modulo theories, we restrict ourselves to decidable theories, because the queries we require an answer for are satisfiability tests. We point out here that more general properties could be derived as, technically, we should not require full decidability for the theories we use, but only that of the fragment we use. We will also see that, in practice (see Chapters 6.4 and 8), we can in some cases tweak our algorithms to bypass some satisfiability tests whose results cannot be computed but still reach an acceptable conclusion for the problem we tackled. However, within the theoretical presentation of those methods, we will stick to decidable theories.

We will call *decision procedure* any algorithm that decides a satisfiability problem in at least one theory. The theory will be detailed when necessary and a decision procedure for a given theory $\mathcal{T}$ will be called a $\mathcal{T}$-decision procedure. A technical system for computationally deciding such queries is called an SMT-*solver* [78, 10, 51, 125]. Such systems usually implement several $\mathcal{T}$-decision procedures.

We will assume that a decision procedure will always return an element of $\mathfrak{B} = \{\texttt{Sat}, \texttt{Unsat}, \texttt{Unknown}\}$. For a formula ϕ and a theory $\mathcal{T}$, we will assume that the decision procedure ensures the following:

1. If it returns `Sat`, then ϕ is $\mathcal{T}$-satisfiable.

2. If it returns `Unsat`, then ϕ is $\mathcal{T}$-unsatisfiable.

If $\mathcal{T}$ is a decidable theory, we will assume that any decision procedure handling this theory will never return `Unknown`. Even though, in practice, an SMT-solver may still return `Unknown`.

When `Sat` is returned, the decision procedure (and the SMT-solver) may additionally return a model of the formula it computed. If it is the case, we assume that we can retrieve this model.

In all the following, every time we test a satisfiability problem, we will assume that it is handled by a decision procedure handling the underlying theory. We assume that such queries will thus be handled by an SMT-solver in any practical system implementing the algorithms we will define in Chapters 3 to 6.

An SMT-solver is *incremental* if it offers a way to update only a part of the formula it was trying to determine the satisfiability of. Intuitively, we expect that if the SMT-solver previously obtained a result for an initial formula, the computation for the updated one will be more efficient than if it was restarting from scratch.

Such solvers are particularly adapted to algorithms that compute their solutions by incrementally building hypotheses and test them against a given formula until reaching the desired conclusion. As we will see in the following chapters, the algorithms we design in this work comply with this framework, which makes the use of incremental SMT-solver to decide the satisfiability tests they contain particularly relevant.

This consideration concludes this chapter of global definitions to which the readers can continuously refer when browsing this document. The readers can also read the Index of definitions and symbols at the end of this document. They will be redirected from there to the corresponding definition within this chapter or any of the following ones.

Chapter 2

State of the art

This chapter aims at presenting the context the research of this thesis was carried out in. The terminology used in this chapter complies with the one introduced in Chapter 1 and any additional notion specific to this chapter will be specified when it is used. We start by presenting the related methods for the abduction technique we consider in this work: the generation of the prime implicates of logical formulas. Prime implicate generation, and abduction in general, is used in a wide range of applications and in a large range of logical frameworks. We can note in particular its use in error diagnosis [56] and program synthesis as well as its recent applications to description logic [107, 53, 63] and separation logic [35, 61]. In this thesis we consider an application of abductive reasoning to generate (property-directed) loop invariants of programs, that is, formulas associated to a program loop that should be and remain true whenever the loop is executed (see also Definition 130). Thus, we will also briefly introduce alternatives and state-of-the-art algorithms for performing loop invariant generation.

2.1 Prime implicate computation

We start by presenting in this section an overview of the various methods existing for performing abduction by computing prime implicates, that is, minimal clausal consequences of a given formula (see also Definitions 35 and 37). Prime implicate computation is a natural method to perform abduction. Indeed, finding the missing conjunctive hypotheses M allowing a formula ϕ to entail another formula ψ (*i.e.* ensuring that $\phi \wedge M \models \psi$) can be done simply by computing the implicates of $\phi \wedge \neg\psi$. The negation of any such implicate is a conjunctive hypothesis ensuring such an entailment.

Example 52. Let us consider the following logical formula over the four boolean variables p, q, r, s:

$$\begin{aligned}\phi = &(p \vee \neg q \vee r)\\ \wedge &(q \vee \neg r \vee \neg s)\\ \wedge &(\neg p \vee \neg q \vee s)\end{aligned}$$

If one wants to find the minimal set for inclusion of minimal consequences of ϕ expressed in the form of disjunctions of boolean constants or their negation, they will find that the three disjunctions (made explicit by the lines of the formula description) belong to this

set. Moreover, the clause $C = \neg q \vee r \vee s$ is another consequence that was not originally part of the formula. Any consequence of ϕ will be a logical consequence of one of these four disjunctive formulas.

2.1.1 Early prime implicate computation in propositional logic

The concept of prime implicate for propositional logic was introduced by Quine [142] at the beginning of the second half of the twentieth century. Since then, work has been done to develop techniques to automatically compute the prime implicates of a propositional formula. The complexity of this computation has been found to be DP-hard [109, 113, 71, 16]. This ensures that any general algorithm automatically computing the prime implicates of propositional formulas will have at least an exponential complexity (unless P = NP).

For almost twenty years, the tools tackling this problem were built upon a specific representation of propositional formulas: the *minterm* representation [23]. This consists in substituting the formula by the set of its models, usually represented as sequences of boolean values to explicitly exhibit the value of each propositional variable in a given model.

Example 53. Let us consider once again the formula ϕ of Example 52. This formula admits ten models:

$$p = \top, q = \top, r = \bot, s = \top \tag{2.1}$$
$$p = \top, q = \top, r = \top, s = \top \tag{2.2}$$
$$p = \top, q = \bot, r = \bot, s = \bot \tag{2.3}$$
$$p = \top, q = \bot, r = \top, s = \bot \tag{2.4}$$
$$p = \top, q = \bot, r = \bot, s = \top \tag{2.5}$$
$$p = \bot, q = \top, r = \top, s = \top \tag{2.6}$$
$$p = \bot, q = \top, r = \top, s = \bot \tag{2.7}$$
$$p = \bot, q = \bot, r = \top, s = \bot \tag{2.8}$$
$$p = \bot, q = \bot, r = \bot, s = \bot \tag{2.9}$$
$$p = \bot, q = \bot, r = \bot, s = \top \tag{2.10}$$

By always representing the variables in the order p, q, r, s and interpreting 0 as $\bot$ and 1 as $\top$, we can represent ϕ using the minterm representation by the set $\{(1,1,0,1), (1,1,1,1), (1,0,0,0), (1,0,1,0), (1,0,0,1), (0,1,1,1), (0,1,1,0), (0,0,1,0), (0,0,0,0), (0,0,0,1)\}$.

This representation is space consuming (exponential *w.r.t.* to the size of the initial formula), which is one of the main reasons why, even though polynomial algorithms were found to compute the prime implicates of formulas previously rewritten using minterms [167], those were never able to perform efficiently on even reasonably sized problems. We let the reader refer to more detailed surveys (*e.g.* [148]) for additional information on these early works.

Around 1970, people started working on alternative approaches for computing prime implicates, which were based directly on the formula. These methods are usually arranged in two categories: those based on the resolution rule and those based on a decomposition of the formula, though other approaches not fitting in any of these classes also exist.

2.1.2 Resolution-based prime implicate computation in propositional logic

Prime implicate computation methods fitting in this category are built upon the *resolution calculus* [4] which consists of the following rule:

Definition 54. Given two clauses (disjunctions of boolean variables or their negation, see also 30) $C = l \vee \chi$ and $D = \neg l \vee \rho$, the *(propositional) resolution rule* is

$$\frac{C \quad D}{\chi \vee \rho}$$

The clause that is generated is called a *resolvent*.

As this rule is defined for clauses, these methods will require their input formula to be in conjunctive normal form (CNF). They will then select two clauses, produce their resolvent and remove both the tautologies and the clauses subsumed by another within the updated clause set. This will be performed until a fixed-point in found for the set of clauses, this fixed-point being the expected set of prime implicates of the formula.

Example 55. Let us consider once again the formula ϕ of Example 52. The clausal form of this formula consists of three clauses $\{p \vee \neg q \vee r, q \vee \neg r \vee \neg s, \neg p \vee \neg q \vee s\}$. Applying the resolution rule on these clauses will, most of the time, result in a valid clause (that is, satisfied in all interpretations or a tautology). For instance, applying the resolution rule on the clauses $p \vee \neg q \vee r$ and $q \vee \neg r \vee \neg s$ on the variable q generates the resolvent $p \vee r \vee \neg r \vee \neg s$ which is a valid clause. One can get the missing implicate of this set by applying the resolution rule on $p \vee \neg q \vee r$ and $\neg p \vee \neg q \vee s$ for the variable p. Once all the tautologies have been removed, and as no smaller clauses can be generated, the set will be composed of all the prime implicates of ϕ.

An improvement of this algorithm introduced by Tison [171] and refined as the incremental technique IPIA [106] makes use of an order on the variables. By following this order and applying all the possible resolutions on a given variable at the same time, it is able to limit as much as possible the number of redundant resolution steps, as it avoids the need to reapply the resolution rule on one of the previously handled variables. It is also important to consider the problem of selecting on which clauses the resolution rule should be applied, as a smarter choice has a good chance to reduce the total number of steps of the algorithm. Such techniques have been implemented in yet another incremental refinement of Tison's method called PIGLET [99], where the authors have chosen to prioritize the selection of clauses sharing literals, producing smaller resolvents that are more likely to be minimal. This technique was found to be more efficient than IPIA.

Another line of work that has been investigated over IPIA and Tison's method is related to the efficiency of clausal subsumption tests. By using tries [77] to store the clauses (thus represented as ordered sets of literals), one can perform the subsumption tests directly within the storage structure which improves the general performance significantly. This was implemented in a method called CLTMS [49] which incrementally constructs the trie of all the prime implicants for a formula. The method permitted to solve non-trivial prime implicate generation problems for the first time.

Refining this work on the efficient storage of prime implicates, Simon and del Val proposed a last algorithm based on Tison's method [163] called ZRes or ZRes-Tison. In

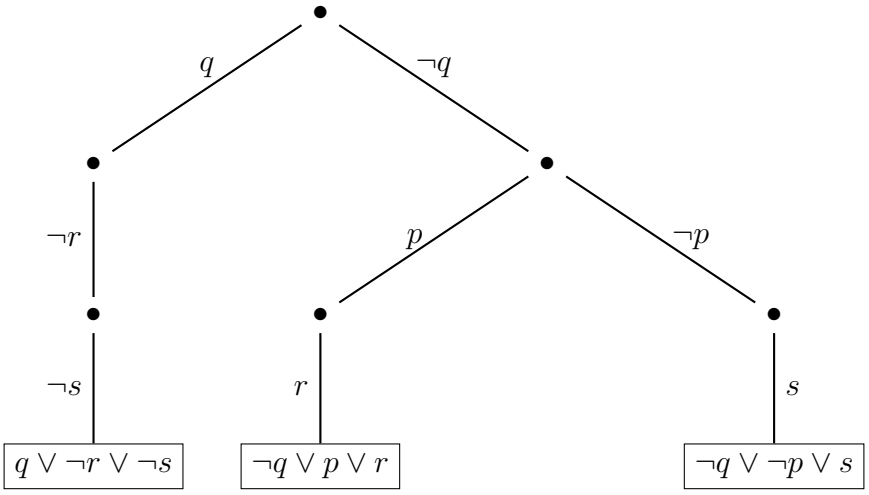


Figure 2.1: Trie representation of the formula ϕ from Example 52; literal order $q > \neg q > p > \neg p > r > \neg r > s > \neg s$

ZRes-Tison, clauses are represented using Zero-suppressed binary decision diagrams (or ZBDDs) [122], in which they appear as paths between the root and a 1-labeled terminal node (see Figure 2.2). Moreover, by applying multi-resolution (a variant of the resolution calculus) directly on the ZBDD, ZRes-Tison can perform all the resolutions on a given variable at the same time, thus making the algorithm independent of the number of actual resolution steps. All those considerations allow ZRes-Tison to produce ZBDDs containing all the prime implicates of a set of clauses very efficiently.

2.1.3 Decomposition-based prime implicate computation in propositional logic

The other main class of prime implicate generators (those not built upon inference rules) is usually referred to as decomposition-based. Algorithms is this category tend to decompose their input problems into smaller formulas, clauses, conjunction of literals, *etc*, recursively compute a result on these smaller part and finally merge these computed results. Depending on the type of decomposition they perform, they may accept more various forms of input problems than the one we described in the previous subsection (such as disjunctive normal forms (DNF), negative normal forms (NNF)). One of the oldest decomposition-based prime implicate generation method is called the Tree Method [164]. It works by constructing the set of prime implicates by a depth-first search in an ordered semantic tree with edges labelled by literals and nodes labelled by DNF formulas. This technique has also been improved by using *set enumeration trees*, a generalization of the semantic trees used in the Tree Method [155].

The Matrix Method [100] is another approach similar to the Tree Method but browsing the tree in breadth instead and labelling nodes based on conjunctions of literals instead of the literals themselves. Jackson and Pais additionally compared the two approaches and showed that their method (Matrix Method) performed better.

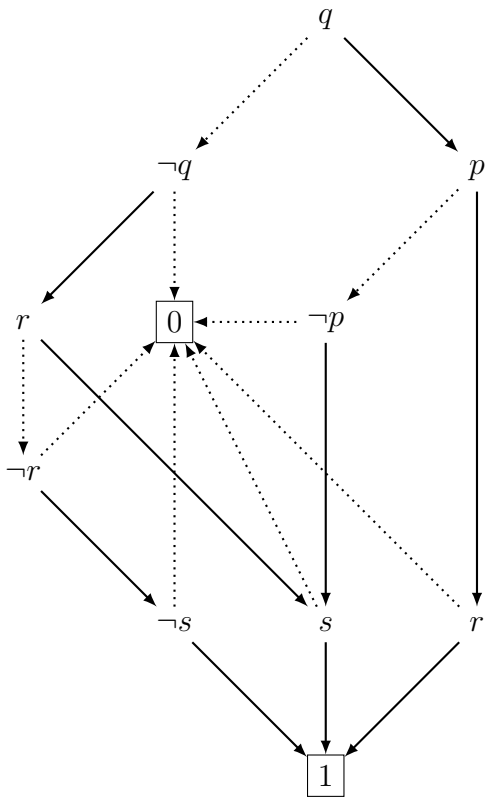


Figure 2.2: ZBDD representation of the prime implicates of the formula ϕ from Example 52; literal order $q > \neg q > p > \neg p > r > \neg r > s > \neg s$

By requiring their input formula to be expressed in a specific form before being processed (CNF, DNF, NNF), the prime implicate generators previously introduced (including those based on resolution) can be impractical to use in some contexts [129, 144, 143]. GEN-PI [129] is an incremental algorithm proposed to solve this issue. By computing closure operations, it is able to compute all the prime implicates of inputs expressed as conjunctions of DNF formulas. Ngair showed that this algorithm is as efficient as CLTMS in general but exhibits much better performances on formulas one cannot easily represent in normal form. An alternative approach extending the Matrix Method for NNF formulas [143, 144] by looking for paths in a previously simplified graph representation of the NNF formula has also been tried, but it was found to require numerous subsumption tests.

A tentative to reduce the need for logical operation by choosing an adequate representation of the input and results was proposed by Coudert and Madre in 1992. They combined the use of Binary decision diagrams (BDD) [1] to store clauses and a meta-product representation of the variables of an initial DNF formula. This resulted in an algorithm only depending on BDD operations to compute the prime implicates, which put into light the need for a variable ordering leading to a BDD as compact as possible. Most such orderings are heuristic-based (see *e.g.* [116]).

Manquinho *et al.* [116] later compared Coudert and Madre's method with a more efficient strategy, introduced by Errico *et al.* [73], and making use of integer linear programming (ILP) [158]. This technique computes the prime implicates of a DNF formula

by converting it into an ILP problem by representing literals with integer variables and constructing constraints that correspond to the formulas of the problem. It is also able to compute the prime implicates one at a time and can be hinted on the literals the next generated implicate should be comprised of. The other major advantage of the use of integer linear programming is that it gives us access to all the techniques already developed for solving such problems. The prime implicate generator exposed in [116] made use of the search algorithm GRASP [162, 131], inspired from the propositional satisfiability testing algorithm DPLL [48, 47], and specifically optimized for its use in prime implicate generation.

Coudert and Madre's technique [115] has been reused beside ILP to produce many other modern decomposition-based prime implicate generators. Examples include the algorithm by Henocque [88] creating three cases for the variables of DNF input formulas in a DPLL fashion and Matusiewicz *et al.*'s method using prime implicate tries [119, 120] for CNF formulas. By using BDDs in the first case and prime implicate tries in the latter to store the prime implicates efficiently, these techniques managed to tackle problems with up to tens of quintillions of prime implicates. The refinement of Matusiewicz *et al.*'s method [120] was also shown to perform twice as fast as resolution-based methods.

2.1.4 Other approaches for prime implicate computation in propositional logic

More recently, alternative approaches have emerged for generation of prime implicates. For instance, Bittencourt *et al.* applied the use of quantum notation [17] on DNF formulas, which annotates each literal with the identifiers of each conjunction of literals it belongs to and then selects the smallest set of quanta covering all terms to obtain the prime implicates. Alternative approaches also include the work of Ribeiro *et al.* [152] which learns transition rules and uses specialization to obtain the prime implicants of a formula.

Finally, the most efficient method to generate prime implicates in propositional logic up to this date is built upon works from Jabbour *et al.* [98] and Palopoli *et al.* [135] and reduces the problem of computing prime implicates to a model building problem by transforming the formula. It is thus possible to obtain the prime implicates of the initial formula by performing a model enumeration of the new one [141].

Example 56. Consider one last time the formula ϕ of Example 52. The method of Previti *et al.* generates the equisatisfiable formula $\psi \cup D$ where $\psi = (l_p \vee l_{\neg q} \vee l_r) \wedge (l_q \vee l_{\neg r} \vee l_{\neg s}) \wedge (l_{\neg p} \vee l_{\neg q} \vee l_s)$ and $D = (\neg l_p \vee \neg l_{\neg p}) \wedge (\neg l_q \vee \neg l_{\neg q}) \wedge (\neg l_r \vee \neg l_{\neg r}) \wedge (\neg l_s \vee \neg l_{\neg s})$.

The method then generates formulas encoding the fact that when a variable is evaluated to true, then the clause it appears in does not require its literals to be true any longer. In our case, this gives us the following formulas:

$$(\neg l_p \vee \neg l_{\neg q}) \wedge (\neg l_p \vee \neg l_r) \tag{2.11}$$
$$(\neg l_{\neg q} \vee \neg l_p) \wedge (\neg l_{\neg q} \vee \neg l_r) \tag{2.12}$$
$$(\neg l_r \vee \neg l_{\neg q}) \wedge (\neg l_r \vee \neg l_p) \tag{2.13}$$
$$(\neg l_q \vee \neg l_{\neg r}) \wedge (\neg l_q \vee \neg l_{\neg s}) \tag{2.14}$$
$$(\neg l_{\neg r} \vee \neg l_q) \wedge (\neg l_{\neg r} \vee \neg l_{\neg s}) \tag{2.15}$$
$$(\neg l_{\neg s} \vee \neg l_q) \wedge (\neg l_{\neg s} \vee \neg l_{\neg r}) \tag{2.16}$$
$$(\neg l_{\neg p} \vee \neg l_{\neg q}) \wedge (\neg l_{\neg p} \vee \neg l_q) \tag{2.17}$$
$$(\neg l_{\neg q} \vee \neg l_{\neg p}) \wedge (\neg l_{\neg q} \vee \neg l_s) \tag{2.18}$$
$$(\neg l_s \vee \neg l_{\neg p}) \wedge (\neg l_s \vee \neg l_{\neg q}) \tag{2.19}$$

From this onwards, the algorithm looks for the models of the union of these formulas with $\psi \cup D$ to find the implicants or the implicates of ϕ.

This method has been implemented in a tool called PRIMER and found to perform better than all the previously introduced methods.

If the reader is interested in a more complete survey of the methods and algorithms presented in this chapter for the computation of prime implicates of propositional formulas, it can refer to, *e.g.*, [118]. .

2.1.5 Prime implicate computation out of propositional logic

To extend prime implicate generation out of propositional logic, various lines of work have been explored [96, 117, 108, 36, 147, 149, 15, 174, 145, 58, 146, 68, 173, 69], mainly focusing on restricted subsets of logical interest of the theory their consider. This first required to extend the notion of prime implicate to first-order logic, which was done by Marquis in 1991 [117], which also permitted to identify related issues that were not present in propositional logic, such as the minimality of the set of implicates or the undecidability of their computation. These additional difficulties led to the subsequent definition of classes of first-order logic formulas for which the computation of prime implicates can be performed, such as the clauses of bounded length and of bounded depth. Most methods devised to compute prime implicates in first-order logic use either first-order resolution [117] or the sequent calculus [36]. The greater complexity of the problem is probably a cause to the fact that only few prime implicate generators for first-order logic can display reasonable performance. One of them is the SOL calculus [96], for full first-order logic, based on tableau resolution for theorem refutation [95]. After building a tableau for the formula, it uses a skip rule to stop developing branches of the tableau without refuting them, letting the leaves of the skipped branches form an implicate of the formula. This method has been implemented in a JAVA program called SOLAR [127]. By using a modification method [31, 97], the tool can also work natively with equational formula [170].

Another method of practical use for the computation of prime implicates in subsets of first-order logic has been devised by Dillig *et al.* [58]. It can be used for any formula expressed in a first-order theory for which both a quantifier elimination procedure and a decision procedure for satisfiability exist, and has been implemented within the `MISTRAL`

SMT-solver [59, 56], using a branch and bound algorithm. This method is, however, not complete, and only computes the implicates of the formula that can be expressed as the negation of minimum partial variable assignments falsifying the formulas.

Prime implicate generation methods targetting other specific logics have also been developed. Examples of such techniques include the work of Knill *et al.* [108] which analyses unification failures to generate prime implicates for first order logic or equational logic formulas, the work of Tran *et al.* [174] using the superposition calculus [3] to generate positive and unit prime implicates and the work of Sofronie-Stokkermans [165, 166] relying on external first-order satisfiability checkers to generate application-targetted constraints. It was shown however [108, 67] that the superposition calculus is not complete out of the scope of positive unit prime implicates. Furthermore, no result simplification procedure is known to simplify the constraints generated in [165, 166].

To bypass these limitations, Tourret *et al.* devised an alternative approach using instantiated equality axioms [68, 69] built over the work of de Kleer *et al.* [49]. This approach is based is based on constrained calculus defined by the usual inference rules of the superposition calculus together with additional rules to dynamically assert new abducible hypotheses on demand into the search space. The asserted hypotheses are attached to clauses as constraints and, when an empty clause is generated, the negation of these hypotheses yields an implicate. They shown that their approach, targetting the generation of prime implicates in equational logic, offered better results than previously mentioned approaches.

Other approaches experimenting the generation of prime implicate in other logics such as modal logics [18, 15, 146], where decomposition algorithms similar to those used in propositional logic have been devised.

2.2 Loop invariant generation

We present in this section a brief overview of some known practical techniques to generate loop invariants of programs. As a lot of work has been done on this problem, we do not try to provide the reader with an exhaustive introduction to the many existing approaches. More complete descriptions of method-specific techniques can be found in *e.g.* [62, 124, 125]. Rather, we present a small introduction that should help the readers familiarize themselves with the loop invariant generation issue and get some pointers on the efficient techniques that have been proposed to tackle it.

Example 57. Let us consider the very simple program presented in Sample Program 2.1, for which we would have to obtain a proof of the proposition it contains at its last line ($y \geq 1$). It is clear that, in this case, the expected property cannot be derived from the negation of the loop condition. This means that the proof will require an invariant for the loop from which we would be able to derive the asserted formula. A satisfying invariant for this program would be, *e.g.*, $y \geq 1 \wedge x \geq 1$, in which the first part ensures the validity of the assertion and the second part is necessary to ensure that this property is preserved by the loop (see also Example 131). The problem considered here is the one of automatically generating such an invariant.

The work on loop invariant generation derived from early works on automated program verification [91, 105]. Those led to many different approaches that are usually

Sample Program 2.1: *LoopCheck*1()

```
1 let x ← 1;
2 let y ← 1;
3 while random({⊤, ⊥}) do
4 |   invariant [?];
5 |   let _x ← x;
6 |   let _y ← y;
7 |   x ← _x + _y;
8 |   y ← _x + _y;
9 assert y ≥ 1;
```

classified as either *static* or *dynamic*.

2.2.1 Static loop invariant generation

Static approaches for loop invariant generation consist in building an abstract representation of the considered programs from which we can derive interesting properties. The first historical approaches for loop invariant generation fall in this category (see *e.g.* [105]). One of the most successful approach for static property inference is the *Abstract interpretation* framework [42, 44], which performs a symbolic execution of the program over an abstract domain. It then computes over-estimations of possible memory states after an unbounded number of loop iterations and removes from these sets the ones it finds contradict other known program constraints. This process is then repeated until a fixed point is reached, which gives us the invariant exhibited by the method on the selected abstract domain.

Example 58. Let us consider the Sample Program 2.1. Now let us consider a symbolic execution of this program over the abstract domain of linear inequalities, that is, associating to each program state a set intervals for each variable, representing its execution. After the initial assignments of Lines 1 and 2, the program state would typically be $\{1 \leq x \leq 1, 1 \leq y \leq 1\}$. To obtain an invariant from this symbolic execution, an abstract interpretation framework will start by executing the content of the loop one time on its abstract domain. As both variables are increased by the value of the other, this would give us, at the end of Line 8, the set $\{2 \leq x \leq 2, 2 \leq y \leq 2\}$. Now, the program will detect that some of the states represented by the set are unreachable due to the condition of the loop (as the loop may never be entered due to its condition). Removing these unreachable states from the intervals we obtained would give us the updated set $\{1 \leq x \leq 2, 1 \leq y \leq 2\}$. By executing the code of the loop again and repeating the same state exclusion process (and, as required in this case, by using a widening operation to obtain the limit of the execution states), the framework will generate the fixed point representing the set of abstracted states reachable at the end of loop on the domain of integer intervals: $\{1 \leq x \leq +\infty, 1 \leq y \leq +\infty\}$. This set gives us an invariant for the considered loop ($x \geq 1 \wedge y \geq 1$) and additionally permits to prove the assertion Line 9.

The technique has been refined from the initial work of Cousot and Cousot to obtain efficient abstract domains [123] and handle other modern program structures [87]. Though

it may be difficult to find an abstract domain in which the generated invariants are precise enough [121], the abstract interpretation framework was successfully applied to develop tools of practical use for program verification [43, 89].

Other successful static approaches for loop invariant generation use decision procedures over specific domains to compute exact estimations of possible memory states after the loop for some specific templated invariant shapes [72, 169, 102]. The initial property can also be iteratively reinforced until it is found to be inductive for a given transition relation [28, 92]. It is also possible to abstract the states of a program using predicates and then use model checking techniques to automatically generate such predicates, among which are loop invariants [82, 5]. Researchers have also investigated the use of other theoretical framework to obtain invariants from symbolic execution such as the use of SMT-solvers [37] or quantifier-elimination based abduction [57]. In the latter case, authors derived from an initial oracle-found invariant necessary consequences after loop executions through Hoare logic [91]. When such consequences did not permit to ensure pre-existing safety-property within the program, they used an quantifier elimination-based algorithm to infer missing requirements within these invariants.

Example 59. Consider once more Sample Program 2.1. Here the method consists in starting by replacing the placeholder invariant Line 4 by a propagating invariant for the considered loop, *e.g.*, $\top$. Using Hoare Logic, one can derive from this initial placeholder invariant a formula that should hold if the assertion Line 9 is verified. For this program and the current placeholder invariant, this condition will be $(\top \Rightarrow y \geq 1)$. As it is clear that this formula does not hold (take for instance $y = 0$), the algorithm will try to strengthen the invariant by finding an abductive strengthening in the form of a placeholder ψ for the formula $(true \wedge \psi \Rightarrow y \geq 1)$. Many solutions for this problem exists but let us assume that we always obtain a good solution for the sake of keeping this example small. Using a quantifier-elimination based algorithm for Presburger arithmetic [86] to produce this solution, we will obtain that for instance, $\psi = y \geq 1$ works as intended. The algorithm will then replace the place holder invariant Line 4 by the new computed one ($true \wedge y \geq 1$, or simply $y \geq 1$) and proceed with Hoare Logic to ensure that new placeholder invariant is a true invariant (it is preserved by the content of the loop). The formula derived to ensure this fact will in this case be $(y \geq 1 \Rightarrow (x + y \geq 1))$. Once again, this formula is not verified (take for instance $y = 1$ and $x = -8000$), which will lead the algorithm to generate a new abduction problem for this condition: $(y \geq 1 \wedge \psi \Rightarrow (x + y \geq 1))$. Applying the quantifier-elimination based abduction algorithm again will return a satisfying solution such as $x \geq 1$, whose resulting invariant $y \geq 1 \wedge x \geq 1$ will finally be proven to both propagate and ensure the safety of the assertion Line 9.

In general, static analysis techniques rely on theoretical results over the mathematical domain they build their inference procedures on [134]. The mathematical theories they usually target and for which they have been found to produce invariants efficiently include numeric domains [26, 27, 29, 102], linear inequality [44, 38], polynomial invariants [154, 156], octagons [123], arrays [30, 25, 110, 80], *etc.* More recently, methods to combine these theories or extend the methods to work with higher-level properties have been investigated. Such techniques were developed for the combination of linear arithmetic and uninterpreted function symbols [14] by reducing the invariant inference to the verification of a sequence of arithmetic, for first-order logic [112] by extracting interpolants from

local or split proofs using symbol-eliminating inferences, for universal invariants [103] by extending the work of Bradley *et al.* on incremental invariant reinforcement, and for parametric systems [138] again by iteratively strengthening based on a symbol-elimination procedure.

2.2.2 Dynamic loop invariant generation

Another class of techniques to generate loop invariants use direct executions of the program to infer the invariants, and are thus referred to as *dynamic*. These methods are briefly presented in this section but stay mainly out of the scope this work.

Early works on dynamic loop invariant generation [72] showed the practical efficiency of executing the program over many possible inputs to detect in an initial set of possible invariants those that happen to be violated by these executions. Refinements of this technique [137, 130, 45] lead to efficient dynamic loop invariant generators. Though such techniques do not provide formal proofs that the invariants they generate are indeed correct, *a posteriori* evidence [176] has shown that they are able to produce true invariants most of the time.

2.3 Summary

We have seen in this chapter various techniques to computes prime implicates of logical formulas in propositional logic and beyond. We saw that most of the techniques proposed required their input formulas to be expressed in a specific format (such as CNF, DNF, NNF) and that, beyond propositional logic, the existing methods were either specialized for a very specific subclass of logical formulas or unable to perform very efficiently. In this thesis, we will present a novel approach for prime implicate generation that solves these two issues by relying on satisfiability tests modulo theories.

We also briefly presented the issue of loop invariant generation of programs and some of the approaches that can be used to solved it. We also explained how Dillig *et al.* used an abduction algorithm to generate loop invariants for numeric programs. In this thesis, we will extend their work by replacing their quantifier-based abduction algorithm by our approach to prime implicate generation. We will then adapt their algorithm to fit our needs and produce a loop invariant generator independent of the theory.

Part I

A generic framework for abduction modulo theories

Chapter 3

Efficient prime implicate generation modulo theories

The core contribution of this work is the design of a generic, in the sense of theory-independent, algorithm to compute prime implicates of a formula. More formally, given a decidable theory $\mathcal{T}$, a finite set of $\mathcal{T}$-satisfiable abducible literals $\mathcal{A} \subseteq \mathcal{L}$ and a set of formulas Φ, we will construct an algorithm that is able to generate all the prime $(\mathcal{T}, \mathcal{A})$-implicates of Φ (modulo $\mathcal{T}$-equivalence).

The algorithm we present is based on the fact that a clause C is a $(\mathcal{T}, \mathcal{A})$-implicate of a set of clauses Φ if and only if $\overline{C} \subseteq \mathcal{A}$ and $\Phi \cup \overline{C} \models_{\mathcal{T}} \bot$. This means that we can find the $(\mathcal{T}, \mathcal{A})$-implicates of Φ by simply enumerating the subsets of $\mathcal{A}$ and keeping those whose union with Φ is $\mathcal{T}$-unsatisfiable.

3.1 An intuitive decomposition algorithm for prime implicate generation

We start by presenting in this section a naive but intuitive algorithm to perform this task. It consists in a systematic enumeration of the subsets of $\mathcal{A}$ performed by adding one of its elements to a *candidate implicate* or *hypothesis* until the union of this candidate with the set Φ is unsatisfiable. Each of these sets is the complement of an implicate of Φ. Then, after finding an implicate, we can backtrack to the last literal added and pursue the exploration until the whole set of abducible literals $\mathcal{A}$ has been explored. Clearly this is not an efficient algorithm. For instance, we are certain that the same non-unit $(\mathcal{T}, \mathcal{A})$-implicate will be generated several times, depending on the order in which the literals are selected. However, it serves as a basis algorithm to present the core concept of the method and as a didactic framework for its applications in the following chapters. It will be improved in the following sections of the chapter to obtain a more efficient algorithm that can be used in practice.

We start by exposing formally the reasons that allow us to construct $(\mathcal{T}, \mathcal{A})$-implicates iteratively by decomposition:

Definition 60. Let Φ be a set of formulas. Let M, A be sets of literals such that

$M \cup A \subseteq \mathcal{A}$. We define

$$\mathfrak{I}_{M,A}(\Phi) = \left\{C \in \mathcal{I}_{\mathcal{T},\mathcal{A}}(\Phi) \;\middle|\; \exists Q \subseteq A,\, C = \overline{M} \vee \overline{Q}\right\},$$
$$\mathfrak{I}^*_{M,A}(\Phi) = \textsc{SubMin}_{\mathcal{T}}(\mathfrak{I}_{M,A}(\Phi)).$$

Intuitively, a clause $\overline{M} \vee \overline{Q}$ thus belongs to $\mathfrak{I}_{M,A}(\Phi)$ if and only if $\overline{Q}$ is a $(\mathcal{T}, A)$-implicate of $\Phi \cup M$. This justifies the fact that we can handle a single candidate $(\mathcal{T}, \mathcal{A})$-implicate to which we successively add relevant literals of $\mathcal{A}$ until we obtain an actual $(\mathcal{T}, \mathcal{A})$-implicate. We also have to justify that we can stop adding literals to this candidate implicate and backtrack when it is an actual implicate.

Proposition 61. *Let Φ be a set of formulas, M and A be sets of literals such that $M \cup A \subseteq \mathcal{A}$. If M is $\mathcal{T}$-satisfiable, then $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable if and only if $\mathfrak{I}^*_{M,A}(\Phi) = \{\overline{M}\}$. If M is $\mathcal{T}$-unsatisfiable, then $\mathfrak{I}^*_{M,A}(\Phi) = \emptyset$.*

Proof. If $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable then $\Phi \models_{\mathcal{T}} \overline{M}$ and $\overline{M} \in \mathcal{I}_{\mathcal{T},A}(\Phi)$, thus $\overline{M} \in \mathfrak{I}_{M,A}(\Phi)$ (by letting $Q = \emptyset$ in Definition 60). By definition, for any clause $C \in \mathfrak{I}_{M,A}(\Phi)$, we have $\overline{M} \subseteq C$; thus $\overline{M} \preceq C$ and $\overline{M} \models_{\mathcal{T}} C$. Since $\overline{M}$ is not a $\mathcal{T}$-tautology by hypothesis, we deduce that $\textsc{SubMin}_{\mathcal{T}}(\mathfrak{I}^*_{M,A}(\Phi)) = \{\overline{M}\}$. Conversely, if $\mathfrak{I}^*_{M,A}(\Phi) = \{\overline{M}\}$ then $\overline{M} \in \mathcal{I}_{\mathcal{T},A}(\Phi)$ by definition, hence $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable.

If M is $\mathcal{T}$-unsatisfiable, then any clause containing $\overline{M}$ is a $\mathcal{T}$-tautology. Consequently, all clauses in $\mathfrak{I}_{M,A}(\Phi)$ are $\mathcal{T}$-tautologies, and $\mathfrak{I}^*_{M,A}(\Phi)$ is empty. □

Proposition 61 justifies both that we can stop adding literals when our candidate $(\mathcal{T}, \mathcal{A})$-implicate M is a $(\mathcal{T}, \mathcal{A})$-implicate of the initial formula (as it will not be possible to construct implicates that are not consequences of $\overline{M}$ by adding more literals to it), and that we can stop adding literals to it when the candidate $(\mathcal{T}, \mathcal{A})$-implicate is by itself a tautology.

Proposition 62. *Let Φ be a set of formulas and $M \subseteq \mathcal{A}$. If $\Phi \cup M \sim_{\mathcal{T}} \Psi \cup M$, then $\mathfrak{I}_{M,A}(\Phi) = \mathfrak{I}_{M,A}(\Psi)$.*

Proof. Assume that $C \in \mathfrak{I}_{M,A}(\Phi)$. Then $C = \overline{M} \vee \overline{Q}$, with $Q \subseteq A$ and $C \in \mathcal{I}_{\mathcal{T},A}(\Phi)$. Since $C \in \mathcal{I}_{\mathcal{T},A}(\Phi)$ we have $\Phi \models C$, hence $\Phi \cup M \cup Q \models_{\mathcal{T}} \bot$. Since $\Phi \cup M \sim_{\mathcal{T}} \Psi \cup M$ we deduce that $\Psi \cup M \cup Q \models_{\mathcal{T}} \bot$, hence $C \in \mathcal{I}_{\mathcal{T},A}(\Psi)$, and therefore $C \in \mathfrak{I}_{M,A}(\Psi)$. Consequently, $\mathfrak{I}_{M,A}(\Phi) \subseteq \mathfrak{I}_{M,A}(\Psi)$. By symmetry, we deduce that $\mathfrak{I}_{M,A}(\Phi) = \mathfrak{I}_{M,A}(\Psi)$. □

Propositions 61 and 62 prove that the exploration schema that was described in the introduction allows us to generate a set of prime $(\mathcal{T}, A)$-implicates entailing the set of $(\mathcal{T}, A)$-implicates of the initial formula. Still, it is clear that some very obvious and minimal improvements can be made in the choice of the literals we add to a candidate $(\mathcal{T}, A)$-implicate. Typically, it is clear that it is useless to add a new hypothesis l into M both if $M \cup \{l\}$ is $\mathcal{T}$-unsatisfiable (because the obtained $\mathcal{T}$-implicate would be a $\mathcal{T}$-tautology), or if this set is equivalent to M (because the $\mathcal{T}$-implicate would not be minimal). This motivates the following definition:

Definition 63. Let Φ be a set of formulas and let M, A be two sets of literals. We denote by $\texttt{fix}(M, A, \Phi)$ a set obtained by deleting from A some literals l such that either $M \cup \Phi \models_{\mathcal{T}} l$ or $\Phi \models_{\mathcal{T}} \bar{l}$.

This definition permits to reduce the number of abducible hypotheses to try, and thus the search space of the algorithm. Still, we do not assume that all the literals l satisfying the condition above are to be deleted because, in practice, such literals may be hard to detect. Hence, we prefer to state the properties of our algorithm without relying on this assumption. To ensure termination, we will assume, however, that no element from M is in $\texttt{fix}(A, M, \Phi)$.

Proposition 64. *Given a set of formulas Φ, a set of abducible literals $\mathcal{A}$, and a set of literals M, we have $\mathfrak{I}^*_{M,\texttt{fix}(\mathcal{A},M,\Phi)}(\Phi) \subseteq \mathfrak{I}^*_{M,\mathcal{A}}(\Phi)$.*

Proof. Let $I \in \mathfrak{I}^*_{M,\texttt{fix}(\mathcal{A},M,\Phi)}(\Phi)$. As $\texttt{fix}(\mathcal{A}, M, \Phi) \subseteq \mathcal{A}$, we have that $I \in \mathfrak{I}_{M,\mathcal{A}}(\Phi)$. As $I \in \mathfrak{I}^*_{M,\texttt{fix}(\mathcal{A},M,\Phi)}(\Phi)$, we are certain that I is not a $\mathcal{T}$-tautology. Now let us consider a clause $C \in \mathfrak{I}_{M,\mathcal{A}}(\Phi)$ such that $C \models_{\mathcal{T}} I$. Necessarily C contains at least one literal $l \in \mathcal{A} \setminus \texttt{fix}(\mathcal{A}, M, \Phi)$. This means that either $M \cup \Phi \models_{\mathcal{T}} l$ or that $\Phi \models_{\mathcal{T}} \bar{l}$, from which we can deduce from the form of C that $I \models C$. □

At this point, we have described the core elements of an algorithm to compute a set of $(\mathcal{T}, A)$-implicates for a set of formulas Φ, and such that no prime $(\mathcal{T}, A)$-implicate is missed. What misses from this point to the goal we have set is to reduce this set of $(\mathcal{T}, A)$-implicates we obtained in order to keep the most general ones only. In order to do this, we devise a method to build the set of $(\mathcal{T}, A)$-implicates incrementally. Proposition 65 provides a method to inductively enlarge the set of $(\mathcal{T}, A)$-implicates by incrementally adding new elements to it. If Proposition 61 represented the base cases of our algorithm, Proposition 65 presents the inductive case to guarantee its completeness.

Proposition 65. *Consider a set of formulas Φ and two sets of literals M, A such that $\mathfrak{I}^*_{M,A}(\Phi) \neq \{\overline{M}\}$. The following equalities hold:*

1. $\mathfrak{I}^*_{M,A}(\Phi) = \textsc{SubMin}_{\mathcal{T}}(\bigcup_{l \in A} \mathfrak{I}^*_{M \cup \{l\},A}(\Phi))$.
2. $\mathfrak{I}^*_{M,A}(\Phi) = \mathfrak{I}^*_{M,\texttt{fix}(A,M,\Phi)}(\Phi)$.

Proof.

1. It suffices to prove that $\mathfrak{I}_{M,A}(\Phi) = \bigcup_{l \in A \setminus M} \mathfrak{I}_{M \cup \{l\},A}(\Phi))$. Let $C \in \mathfrak{I}_{M,A}(\Phi)$. By hypothesis, C is of the form $\overline{M} \vee \overline{Q}$, where $Q \subseteq A$ and $M \cap Q = \emptyset$. Since $\mathfrak{I}^*_{M,A}(\Phi) \neq \{\overline{M}\}$, necessarily $C \neq \overline{M}$ and $Q \neq \emptyset$. Let $l \in Q$ and $m = \bar{l}$. We have $C = \overline{M} \vee m \vee \overline{Q'}$, with $Q' = Q \setminus \{l\}$, and since $l \in A$, $C \in \mathfrak{I}_{M \cup \{l\},A}(\Phi)$. Conversely, if $C \in \mathfrak{I}_{M \cup \{l\},A}(\Phi)$ with $l \in A$, then $C \in \mathcal{I}_{M,A}(\Phi)$ and $C = \overline{M} \vee \bar{l} \vee Q$, for some $Q \subseteq A$, so that $C \in \mathfrak{I}_{M,A}(\Phi)$.

2. Since $\texttt{fix}(A, M, \Phi) \subseteq A$, we have by proposition 64 that $\mathfrak{I}^*_{M,\texttt{fix}(A,M,\Phi)}(\Phi) \subseteq \mathfrak{I}^*_{M,A}(\Phi)$. Now let l be a literal in A such that either $M \cup \Phi \models l$ or $M \models \bar{l}$. If $C \in \textsc{SubMin}_{\mathcal{T}}(\mathfrak{I}_{M,A}(\Phi))$ is of the form $\overline{M} \vee \overline{Q}$, then Q cannot contain l: indeed, in the former case $\bar{l}$ could be removed from $\Phi \cup \overline{C}$ while preserving equivalence, hence the implicate would not be minimal, and in the later case C would be a $\mathcal{T}$-tautology. □

The results above lead to a basic algorithm for generating $(\mathcal{T}, A)$-implicates which is described in Algorithm 3.1.

Algorithm 3.1: ImpGen (Φ, M, A)

```
1 if M ⊨_T ⊥ then
2     return ∅ ;
3 if Φ ∪ M ⊨ ⊥ then
4     return {M̄} ;
5 let B = fix(A, M, Φ) ;
6 foreach l ∈ B do
7     let P_l = ImpGen(Φ, M ∪ {l}, B) ;
8 return SubMin_T(⋃_{l∈B} P_l) ;
```

Lemma 66. $\mathfrak{I}^*_{M,A}(\Phi) = \textsc{ImpGen}(\Phi, M, A)$.

Proof. The result is proved by a straightforward induction on $\textsc{Card}(\mathcal{A}\backslash M)$. As explained above, the algorithm works by adding literals from A as hypotheses until a contradiction can be derived. The **return** statement at Line 4 avoids enumerating the subsets that strictly contain M once it is known that $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable. The **return** statement at Line 2 similarly avoids the generation of inconsistent $(\mathcal{T}, A)$-implicates. Thus by Proposition 61, $\mathfrak{I}^*_{M,A}(\Phi) = \emptyset$ if M is $\mathcal{T}$-unsatisfiable, and $\mathfrak{I}^*_{M,A}(\Phi) = \{\overline{M}\}$ if M is $\mathcal{T}$-satisfiable and $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable.

Otherwise, after reducing the set of abducible literals according to Definition 63 (Line 5), we generate the $(\mathcal{T}, A)$-implicates we can derive from all the possible adding to our hypothesis still available (Line 7). by Proposition 65, we have $\mathfrak{I}^*_{M,A}(\Phi) = \mathfrak{I}^*_{M,\texttt{fix}(A,M,\Phi)}(\Phi) = \mathfrak{I}^*_{M,B}(\Phi)$. By the induction hypothesis, for each $l \in A$, $P_l = \mathfrak{I}^*_{M\cup\{l\},B}(\Phi)$, and by Proposition 65, $P_l = \mathfrak{I}^*_{M\cup\{l\},A}(\Phi)$; we deduce that $\mathfrak{I}^*_{M,A}(\Phi) = \textsc{SubMin}_{\mathcal{T}}(\bigcup_{l\in A} P_l)$. Note that at each recursive call, a new element is added to M, since $\texttt{fix}(A, M, \Phi)$ is assumed not to contain any element from M. □

This leads to the following result:

Theorem 67. *If Φ is a set of formulas then $\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi) = \textsc{ImpGen}(\Phi, \emptyset, \mathcal{A})$.*

We obtain a very simple algorithm that is able to compute all the prime $(\mathcal{T}, A)$-implicates of a set of formulas in any decidable theory. However, as announced in the introduction, this algorithm is clearly not efficient, and thus not usable in practice. Though it is easy to design simple improvements on this algorithm, we are going to push this further and dedicate Section 3.2 to the design of an efficient algorithm based on this one.

Remark 68. As mentioned in the introduction of the chapter, we have assumed that all the satisfiability tests we query in our algorithm are decidable. This is obviously a result that will depend on the theory in which the formulas we handle are expressed. In practice, as long as the SMT-solver used is correct, Algorithm 3.1 will also be correct (in the sense that all the clauses it will generate will indeed be $(\mathcal{T}, A)$-implicates of Φ). The Algorithm will be complete, and thus return a set of clauses entailing the set of prime $(\mathcal{T}, A)$-implicates of Φ, only if the SMT-solver is also complete (never returns `Unknown`). Finally, if the SMT-solver may not terminate, Algorithm 3.1 may not either, though this limitation will be discussed in Chapters 6.4 and 8.

3.2 Improving the prime implicates generator

In this section, we present several improvements that can be made to the previous algorithm to prevent as many redundant or uninteresting (as in not permitting to obtain an implicate) recursive call to be made. This results in a significant decrease of the size of the exploration tree (as supported by experiments in Chapter 9). The final algorithm is presented as Algorithm 4.1, and will conclude this part of the work.

3.2.1 Reducing the number of candidates

Let us start with the example of our introduction chapter, where $l_1 \vee l_2$ is a prime $(\mathcal{T}, \mathcal{A})$-implicate of a set Φ, but is generated twice by the ImpGen algorithm. Such redundant calls are quite straightforward to avoid by ensuring that every invocation of the algorithm corresponds to a distinct set of literals M. This can be done by fixing an ordering $<$ among literals in $\mathcal{A}$, and by assuming that hypotheses are always added in increasing order with respect to $<$. Typically, in our case, assuming that $l_1 < l_2$, the order will prevent the generation of this implicate in the form $l_2 < l_1$, as at the moment l_2 has been added as an hypothesis in our candidate $(\mathcal{T}, \mathcal{A})$-implicate, the algorithm will not be allowed to add l_1 after it.

Another interesting method for restricting the set of candidate hypotheses is to exploit information that can be extracted from the satisfiability tests made by the algorithm. Indeed, along with the result of a satisfiability test occurring at line 1 of Algorithm 3.1, if $\Phi \cup M$ is satisfiable, it is likely that we can also recover a model of $\Phi \cup M$. Let us assume that $\Phi \cup \{l_1\}$ is satisfiable, and that we can recover the corresponding model. Then if the model of this set also satisfies l_2, we are already certain that $\Phi \cup \{l_1\} \cup \{l_2\}$ is also satisfiable (as the same model ensures this fact). Thus, it is not necessary to consider l_2 as a hypothesis. In particular, if a model of $\Phi \cup \{l_1\}$ validates all the literals in A at Line 1 of Algorithm 3.1, then $\mathfrak{I}^*_{M,A}(\Phi)$ is necessarily empty and no literal should be selected. We can thus take advantage of the existence of a model of $\Phi \cup M$ in order to guide the choice of the next literals in A. We will refer to this refinement as a *semantic guidance* on the choices of the literals.

Observe, however, that this refinement interferes with the previous one based on the order $<$. Indeed, non-minimal hypotheses will have to be considered if all the smaller hypotheses are dismissed because they are true in the model.

Example 69. To illustrate this conflict between literal ordering and semantic guidance using models, let us consider the following example. Assume that we have a set of abducible literals $A = \{a, b, \overline{a}, \overline{b}\}$ such that $a < b < \overline{b} < \overline{a}$, and that we are looking for the prime $(\mathcal{T}, A)$-implicates of $\Phi = \{a \vee b\}$. Now let us choose to prune the set of abducible literals using our order on literals (such that we do not consider smaller literals) and semantic guidance using models when we can recover them from the satisfiability tests. Let us execute our $(\mathcal{T}, A)$-implicate generation algorithm starting with a hypothesis set $M = \emptyset$.

We start the execution by verifying the satisfiability of Φ. As it is satisfiable, we will assume that we recover a model from the satisfiability test, such as $\mathcal{M} = a \wedge \overline{b}$. In this case, we will prune from the set of abducible literals the literals the model satisfies, which will result in $A = \{b, \overline{a}\}$.

At this step, we have to choose which literal we want to add to our hypothesis. We follow the rule we set at the beginning of the example and take $M = \{b\}$, then call the algorithm again. From this point, we will not be able to derive any $(\mathcal{T}, A)$-implicate of Φ, because all the satisfiable hypotheses we can obtain by adding literals from A to $\{b\}$ would not be in contradiction with $a \vee b$. This means that, after trying those possibilities, the algorithm will backtrack to the initial level where it will try the other possible addition to the empty hypothesis set: $M = \{\overline{a}\}$.

From here, one would expect that it could derive the $(\mathcal{T}, A)$-implicate of Φ, but as $\overline{b}$ (which is the literal required to generate the $(\mathcal{T}, A)$-implicate) happens to be smaller than $\overline{a}$ in our example, it will never be added as a hypothesis during this stage. And because it was pruned from A at the previous level, it was never added as a hypothesis there either. Thus the algorithm will terminate without generating the $(\mathcal{T}, A)$-implicate we wanted to obtain, which is a shame.

Notice that this problem only occurs due to the joint usage of literal ordering and semantic guidance. This exhibits the fact that if we want to use them simultaneously, we will need to tweak the algorithm to prevent such $(\mathcal{T}, A)$-implicates to be missed.

These two improvement ideas and the consideration detailed in Example 69 justify the following formalization of the principles. In what follows, we consider a total ordering $<$ on the elements of $\mathcal{A}$, not necessarily related to the ordering $\prec$ on clauses.

Definition 70. For $A \subseteq \mathcal{A}$ and $l \in A$, we define $A[l] \stackrel{\text{def}}{=} \{l' \in A \mid l < l'\}$. If I is a set of literals then we denote by $A^I[l]$ the set $\left\{l' \in A \,\middle|\, l' < l \wedge \overline{l'} \notin I\right\} \cup A[l]$.

Example 71. Assume that $A = \{p_i, \neg p_i \mid i = 1, \ldots, 6\}$ and that for all $i, j \in [\![1, 6]\!]$, for all literals $l \in \{p_i, \neg p_i\}$ and $l' \in \{p_j, \neg p_j\}$, $l < l'$ if and only if either $i < j$ or ($i = j$, $l = p_i$ and $l' = \neg p_i$). Then $A[p_4] = \{\neg p_4, p_5, \neg p_5, p_6, \neg p_6\}$.

If $I = \{p_1, \neg p_2\}$, then $A^I[p_4] = \{p_1, \neg p_2, p_3, \neg p_3, \neg p_4, p_5, \neg p_5, p_6, \neg p_6\}$.

Intuitively, the definition $A[l]$ is an implementation of the literal ordering pruning strategy (based on a given literal l). It defines which literals can be kept as possible hypotheses for the enclosed sublevels such that fewer redundant calls are made. The definition $A^I[l]$ aims to repair the issue that was discussed in Example 69. It defines the set of literals that should be considered when some have been removed by the use of a model at the previous level. These literals are actually the ones that were consequences of the model but happened to be smaller than the hypothesis added to the candidate $(\mathcal{T}, A)$-implicate for the current level.

We now have to define how we handle the models we may obtain from the satisfiability tests in our framework. This is done through the following definitions:

Definition 72. Let Φ be a set of formulas. A set of literals I is Φ-*compatible with respect to* $\mathcal{A}$ (or simply Φ-*compatible*) if every prime $(\mathcal{T}, \mathcal{A})$-implicate of Φ contains a literal in I.

Intuitively, a Φ-compatible set I consists of literals l such that $\overline{l}$ will be allowed to be added as a hypothesis to generate $(\mathcal{T}, \mathcal{A})$-implicates of Φ (see Lemma 78 below). The set I can always be defined by taking the negations of all the abducible literals from $\mathcal{A}$. In this case, all literals will remain possible hypotheses. It is possible, however, to restrict the size of I when a model of Φ is known, as evidenced by the following proposition:

Proposition 73. *If Φ is a set of clauses and $\mathcal{J}$ is a model of Φ, then the set $I \stackrel{def}{=} \{l \in \mathcal{L} \mid \mathcal{J} \models l\}$ is Φ-compatible.*

Proof. Let M be a set of literals such that $\overline{M}$ is a prime $(\mathcal{T}, \mathcal{A})$-implicate of Φ, and assume that for all $l \in M$, $\bar{l} \notin I$, *i.e.*, that for all $l \in M$, $\mathcal{J} \not\models \bar{l}$. Then $\mathcal{J} \models l$ holds for every $l \in M$, hence $\mathcal{J} \models \Phi \cup M$ and $\overline{M}$ cannot be a $(\mathcal{T}, \mathcal{A})$-implicate of Φ. □

The goal here will thus be to use the models we recover from the satisfiability tests to build Φ-compatible sets that are as small as possible. Note that the condition of having a model of Φ was not added to Definition 72 because in practice, such a model cannot always be constructed efficiently, or may not be available. Still, this completes the modification of the framework for using both literal orderings and models.

3.2.2 Reusing the $(\mathcal{T}, A)$-implicates

The next update we will discuss is based on some considerations on how we can reuse the logical consequences we generate during the execution of the algorithm. In this section, we only work with sets of formulas in negative normal form. The results could be extended to other formulas by reasoning on the polarity of their atoms (number of negations applied to them). One thing we can do is to use the implicates we have generated to simplify the formula we are computing, in the hope to detect additional consequences without the need of a satisfiability test. We can also expect an SMT-solver to perform faster on a simplified formula. The easiest consequences to reuse are the unit clauses of the problem. This motivates the handling of the unit consequences we can derive from our initial set of formulas Φ; it can pay off if we update Φ with them when they are derived.

Definition 74. Let Φ be a set of formulas and $M \subseteq \mathcal{A}$. We denote by $\mathfrak{U}_{\mathcal{T},\Phi,M}$ the set of unit clauses logically entailed by $\Phi \cup M$ modulo $\mathcal{T}$, *i.e.*,

$$\mathfrak{U}_{\mathcal{T},\Phi,M} \stackrel{\text{def}}{=} \{l \in \mathcal{L} \mid \Phi \cup M \models_{\mathcal{T}} l\}$$

If Φ is a set of clauses, given a set U such that $M \subseteq U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$, we denote by $\mathfrak{U}_{\mathcal{T},M}[U](\Phi)$ a formula obtained from Φ by replacing *some* (arbitrarily chosen) literals l' by $\bot$ if $U \models_{\mathcal{T}} \overline{l'}$ and by $\top$ if $M \models_{\mathcal{T}} l'$.

Note that U is not necessarily identical to $\mathfrak{U}_{\mathcal{T},\Phi,M}$, because in practice the latter set is hard to generate. Similarly we do not assume that all literals l' are replaced in Definition 74 since testing logical entailment may be costly. Thus we prefer to give a very general and flexible formulation of the definition so that upcoming results hold for any possible choice of U and l'. Lemma 76 shows that the $(\mathcal{T}, \mathcal{A})$-implicates of a set of clauses Φ and those of $\mathfrak{U}_{\mathcal{T},M}[U](\Phi)$ are identical. In the case where Φ is not a set of clauses, we will assume that this property holds.

Proposition 75. *Let Φ be a set of formulas and $M \subseteq \mathcal{A}$. Consider a set of literals U such that $M \subseteq U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$. We have:*

$$\Phi \cup M \sim_{\mathcal{T}} \mathfrak{U}_{\mathcal{T},M}[U](\Phi) \cup M$$

Proof. We have $\Phi \cup M \models \mathfrak{U}_{\mathcal{T},M}[U](\Phi)$, since $\Phi \cup M \models_{\mathcal{T}} U$, and $U \models (l' \Rightarrow \bot)$ if $U \models \overline{l'}$. Conversely, it is clear that $\mathfrak{U}_{\mathcal{T},M}[U](\Phi) \cup M \models \Phi \cup M$. □

Lemma 76. *Let Φ be a set of formulas and $M \subseteq \mathcal{A}$. Consider a set of literals U such that $M \subseteq U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$. Then:*

$$\mathfrak{I}_{M,A}(\Phi) = \mathfrak{I}_{M,A}(\mathfrak{U}_{\mathcal{T},M}[U](\Phi))$$

Proof. This is an immediate consequence of Proposition 75. □

We are now left to implement the use of those newly acquired unit consequences within our algorithm.

Definition 77. Let U, M, A be sets of literals. We define:

$$\mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi) \stackrel{\text{def}}{=} \left\{ \overline{M} \vee \bar{l} \,\middle|\, l \in A \wedge \bar{l} \in U \right\}$$

The lemma below can be viewed as a refinement of Proposition 65. It is based on the previous results, according to which, when adding a new hypothesis l, it is possible to remove from the set of abducible literals A every literal that is strictly smaller than l, provided its complementary is in I (because we can always assume that the smallest available hypothesis is considered first). This is why the recursive call is on $A^I[l]$ instead of $\mathcal{A}$. Note also that due to the interference between semantic guidance and the ordering $<$, the smaller the set I is, the larger $A^I[l]$ will be.

Lemma 78. *Assume that $\Phi \cup M$ is $\mathcal{T}$-satisfiable and let I be an $(\Phi \cup M)$-compatible set of literals. Let U be a set of literals such that $M \subseteq U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$. We have the following:*

$$\mathfrak{I}^*_{M,A}(\Phi) \;=\; \textsc{SubMin}_{\mathcal{T}}\left(\mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi) \cup \bigcup_{l \in A, \bar{l} \in I} \mathfrak{I}^*_{M \cup \{l\}, A^I[l]}(\Phi) \right)$$

Proof. First note that $\mathfrak{I}^*_{M,A}(\Phi) \neq \{\overline{M}\}$, since $\Phi \cup M$ is $\mathcal{T}$-satisfiable. We first prove that $\mathfrak{I}^*_{M,A}(\Phi) \subseteq \mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi) \cup \bigcup_{l \in A, \bar{l} \in I} \mathfrak{I}^*_{M \cup \{l\}, A^I[l]}(\Phi)$. Let $C \in \mathfrak{I}^*_{M,A}(\Phi)$. By hypothesis, C is of the form $\overline{M} \vee \overline{Q}$, where $\emptyset \neq Q \subseteq A$. Since I is $(\Phi \cup M)$-compatible, Q necessarily contains a literal $l \in A$ such that $\bar{l} \in I$. Assume that l is the smallest literal in Q satisfying this property. We distinguish the following cases.

- Assume that Q contains a literal l' such that $\overline{l'} \in U$. In this case, since $U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$, $\Phi \cup M \models_{\mathcal{T}} \overline{l'}$. Since $Q \subseteq A$, we also have $l' \in A$, and since $\mathfrak{I}^*_{M,A}(\Phi) \neq \{\overline{M}\}$, we deduce that $\overline{M} \vee \overline{l'} \in \mathfrak{I}^*_{M,A}(\Phi)$. Since $\overline{M} \vee \overline{l'} \models_{\mathcal{T}} C$ and $C \in \mathfrak{I}^*_{M,A}(\Phi)$, C must be smaller or equal to $\overline{M} \vee \overline{l'}$, which is possible only if $C = \overline{M} \vee \overline{l'}$. We deduce that $C \in \mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi)$.

- Otherwise, we show that $Q \setminus \{l\} \subseteq A^I[l]$. By Definition 70, we have $A[l] = \{l' \in A \mid l < l'\}$ and $A^I[l] = \{l' \in A \mid l' < l \wedge l' \notin l\} \cup A[l]$. Let $l' \in Q$, with $l' \neq l$. If $l' > l$ then $l' \in A[l] \subseteq A^I[l]$. If $l' \not> l$, then since $>$ is total and $l \neq l'$, necessarily $l > l'$. Since l is the smallest literal in Q such that $\bar{l} \in I$, we deduce that $\overline{l'} \notin I$. Thus $l' < l$ and $\overline{l'} \notin I$, which entails that $l' \in A^I[l]$. Consequently, $Q \setminus \{l\} \subseteq A^I[l]$. Since $C = \overline{M \cup \{l\}} \vee \overline{Q \setminus \{l\}}$, this entails that $C \in \mathfrak{I}^*_{M \cup \{l\}, A^I[l]}(\Phi)$.

We now prove that $\mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi) \cup \bigcup_{l\in B,\bar{l}\in I} \mathfrak{I}^*_{M\cup\{l\},B^I[l]}(\Phi) \subseteq \mathfrak{I}_{M,A}(\Phi)$.

Let $C \in \mathfrak{U}^*_{\mathcal{T},M,A,U}(\Phi)$. By definition, C is of the form $\overline{M} \cup l$ with $l \in \overline{A} \cap U$. Since $U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$, we deduce that $\Phi \cup M \models_{\mathcal{T}} l$, *i.e.*, that $\Phi \models_{\mathcal{T}} \overline{M} \vee l$. Since $\bar{l} \in A$, this entails that $\overline{M} \vee l \in \mathfrak{I}_{M,A}(\Phi)$, hence $C \in \mathfrak{I}_{M,A}(\Phi)$.

Let $C \in \mathfrak{I}^*_{M\cup\{l\},A^I[l]}(\Phi)$ with $l \in A$, $\bar{l} \in I$. By definition, $C = \overline{M} \vee \bar{l} \vee \overline{Q}$, with $Q \subseteq A^I[l]$ and $C \in \mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi)$. But $A^I[l] \subseteq A$ by definition, thus $Q \cup \{l\} \subseteq A$ and $C = M \vee (\overline{Q} \vee \bar{l}) \in \mathfrak{I}_{M,A}(\Phi)$.

□

We restrict the usage of this solution to the $\mathcal{T}$-implicates of size one because using larger ones could be numerous and thus increase the complexity of the related computation.

3.2.3 External restrictions to prune more candidates

Another change we can do to our algorithm targets a more user-oriented feature. As the set of $(\mathcal{T}, A)$-implicate can be very large, one may not necessarily want to generate them all. Pruning the $(\mathcal{T}, A)$-implicates that are not of use after their generation is the easiest way to obtain the result we expect, but it may also be possible to use those external restrictions directly within the generation algorithm. This has at least two advantages. First, it removes the necessity to prune the first result set, which reduces the total computation time. Second, it may be used to detect that some hypothesis M would only lead to $(\mathcal{T}, A)$-implicates violating the restrictions. Such discoveries can be used to prevent those literals from being added to the candidate $(\mathcal{T}, A)$-implicate and reduce the computation time once more. In order to implement that into our framework, we parametrize our algorithm by a predicate $\mathcal{P}$. This predicate will be used to filter the implicates that are generated according to the choices of the user. We also assume it has the following property:

Definition 79. A predicate $\mathcal{P}$ on sets of literals is $\subseteq$*-closed* if for all sets of literals A such that $\mathcal{P}(A)$ holds, if $B \subseteq A$ then $\mathcal{P}(B)$ also holds.

Examples of $\subseteq$-closed predicates include cardinality constraints: $\mathcal{P}_k \stackrel{\text{def}}{=} \lambda A.\ \textsc{Card}(A) \leq k$, where $k \in \mathbb{N}$, or implicant constraints: $\mathcal{P}_\phi \stackrel{\text{def}}{=} \lambda A.\ \phi \models_{\mathcal{T}} A$, where ϕ is a formula. Note that $\subseteq$-closed predicates can safely be combined by the conjunction and disjunction operators.

An important feature of $\subseteq$-closed predicates is that implicates verifying such predicates can be generated on the fly without any post-processing step, as evidenced below:

Example 80. To understand the use of such predicates better, let us consider the following example: Assume that a user wants to generate the $(\mathcal{T}, A)$-implicates of a formula ϕ that are not logical consequences of another formula ψ. If he were to use our algorithm to generate all the prime implicates of the formula, then retain only those satisfying this constraint, ImpGen would have made a lot work constructing hypotheses that are necessarily consequences of ψ, typically when adding more literal to a hypothesis that

was already a consequence of ψ while not being an implicate of ϕ. This also means that a lot of additional satisfiability tests are performed. This work can typically take a lot of time as the deeper the algorithm searches, the larger the exploration tree becomes (which is a trivial consequence of the structure of the algorithm). Moreover, the user would have to check, at the end of the computation, all the implicates that were generated to remove from them those that are a consequence of ψ. If he were to use the version with a filter predicate however, in this case $\mathcal{P} : \mathcal{V} \mapsto \psi \not\models_{\mathcal{T}} \overline{\mathcal{V}}$, he could recover the same result while preventing the algorithm from executing all the unnecessary computation. This, as we will see in Chapter 9, can have a significant impact on the interest of our method.

3.3 An updated version of the prime implicate generation algorithm

We now take some time to formally insert the previously described changes that have been made to our initial Algorithm 3.1. The resulting algorithm is presented in Algorithm 3.2. The general structure remains identical. We add at Line 1 the restriction to the hypotheses that derive from the use of a filtering predicate that is given as a parameter of the algorithm. It removes from the exploration all the branches that do not satisfy $\mathcal{P}$. Lines 5 to 6 handle the update of the initial formula using the unit consequences that could be derived at the time of the computation. The set of literals created at Line 8 will typically be constructed from a model recovered from the satisfiability test performed Line 3, as explained in Proposition 73. It is then used to prune the abducible literals we try to add to our main hypothesis M in the branching loop Line 9; we point out that the order on abducible literals is effectively used by the construction of the new abducible set sent as a parameter to the recursive call Line 10. This same construction also handles the correction necessary to use this method alongside the semantic guidance, as detailed in previous sections. The minimal set of $(\mathcal{T}, A)$-implicate we can obtain from all the recursive calls is finally computed and returned at Line 11.

Algorithm 3.2: ImpGen-PiD $(\Phi, M, A, \mathcal{P})$

1 **if** $M \models \bot$ **or** $\neg\mathcal{P}(M)$ **then**
2 $\quad$ **return** $\emptyset$;
3 **if** $\Phi \cup M \models \bot$ **then**
4 $\quad$ **return** $\{\overline{M}\}$;
5 **let** $U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$ such that $M \subseteq U$;
6 **let** $\Phi = \mathfrak{U}_{\mathcal{T},M}[U](\Phi)$;
7 **let** $A = \texttt{fix}(M, A, \Phi)$;
8 **let** I be an $(\Phi \cup M)$-compatible set of literals ;
9 **foreach** $l \in A$ *such that* $\bar{l} \in I$ **do**
10 $\quad$ **let** $P_l = \text{ImpGen-PiD}(\Phi, M \cup \{l\}, A^I[l], \mathcal{P})$;
11 **return** $\text{SubMin}_{\mathcal{T}}(\mathfrak{U}^*_{M,A,U}(\Phi) \cup \bigcup_{l \in A} P_l)$;

This leaves us with the task of proving that this modified algorithm still performs the

way we expect it to.

Lemma 81. *If $\mathcal{P}$ is $\subseteq$-closed then* ImpGen-PiD$(\Phi, M, A, \mathcal{P}) = \mathfrak{I}^*_{M,A}(\Phi) \cap \left\{\overline{A} \,\middle|\, A \in \mathcal{P}\right\}$.

Proof. If one of M or $\Phi \cup M$ is $\mathcal{T}$-unsatisfiable, or $\mathcal{P}(M)$ does not hold, then the result follows from Proposition 61 and Definition 79. Otherwise the result is proved by induction on Card$(\mathcal{A} \setminus M)$, using Proposition 65 and Lemmata 78 and 76. □

Theorem 82. *If $\mathcal{P}$ is $\subseteq$-closed then $\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi) \cap \left\{\overline{A} \,\middle|\, A \in \mathcal{P}\right\}$ =* ImpGen-PiD$(\Phi, \emptyset, \mathcal{A}, \mathcal{P})$.

Proof. This is a direct consequence of Lemma 81 and Definition 60 □

3.4 Improvements for performing abduction in propositional logic

We present in this section some restrictions to the ImpGen-PiD Algorithm that can be used to improve its efficiency when generating prime implicates of propositional formulas [161]. It is important to note that the modifications presented in this section only apply in the context of propositional logic as they use hypotheses verified in this context only to preserve the completeness of the algorithm. Their use is thus not applicable in the general context.

It is possible to represent any logical formula in clausal form. In propositional logic, this can be done for instance by using the following transformation.

Definition 83. Let ϕ be a propositional logic formula. We define the *Conjunctive normal form transformation of* ϕ, and denote $cnf(\phi)$, the formula inductively defined as follows:

- $cnf(\top) = \top$.
- $cnf(\bot) = \bot$.
- If ϕ is an atom, then $cnf(\phi) = \phi$.
- If $\phi = \neg\psi$ then $cnf(\phi) = \phi$.
- If $\phi = \psi_1 \wedge \psi_2$ then $cnf(\phi) = cnf(\psi_1) \wedge cnf(\psi_2)$.
- If $\phi = \psi_1 \vee \psi_2$ then $cnf(\phi) = \bigwedge_{C_i \in cnf(\psi_1), D_j \in cnf(\psi_2)} (C_i \vee D_j)$.

Building on this result, we will assume in this section that the sets of formulas of which we want to generate the prime implicates of are sets of clauses.

Remark 84. Note that the transformation of Definition 83 may, due to the last rule, construct a new formula that is exponential in the size of ϕ. To solve this problem, there exists structure-preserving transformations [139, 2] that construct clauses linear in the size of ϕ by adding new variables. Such transformations could also be used here as long as one does not consider these added variables when providing abducible literals to the prime implicates generation algorithm.

Definition 85. Given a propositional formula ϕ that admits a (possibly partial) model $\mathcal{M}$, we say that $\mathcal{M}$ is a *minimal model* of ϕ if, for any model $\mathcal{J}$ of ϕ, either $\mathcal{J} = \mathcal{M}$ or $\mathcal{J}$ coincides with $\mathcal{M}$ on all the atoms interpreted in the latter and there exists an atom in ϕ that has an interpretation in $\mathcal{J}$ but not in $\mathcal{M}$.

We can easily see that, when we use models recovered from the satisfiability tests to obtain sets of literals reducing the size of the search space in the ImpGen-PiD Algorithm that, if we can minimize the model we use, we will also reduce the corresponding set of literals and thus the search space. As we believe that the size of the search space is one of the major factors that could degrade the performance of the prime implicate generator, we will try to minimize the models we recover from the satisfiability tests when handling propositional logic formulas. A method to perform this minimization is described in Chapter 10. However, before we can minimize our models, we have to ensure that the use of partial models in the ImpGen-PiD Algorithm preserves its properties.

Proposition 86. *If Φ is a set of propositional clauses and $\mathcal{J}$ is a partial model of Φ, then the set $I \stackrel{def}{=} \{l \in \mathcal{L} \mid \mathcal{J} \models l\}$ is Φ-compatible (see Definition 72).*

Proof. Let M be a set of literals such that $\overline{M}$ is a prime $\mathcal{A}$-implicate of Φ, and assume that for all $l \in M$, $\bar{l} \notin I$, *i.e.*, that for all $l \in M$, $\mathcal{J} \not\models \bar{l}$. Then for every $l \in M$, either $\mathcal{J} \models l$, in which case $\mathcal{J} \models \Phi \cup M$ and $\overline{M}$ cannot be an $\mathcal{A}$-implicate of Φ, or there exists a model $\mathcal{J}'$ of Φ such that $\mathcal{J}' \models \bar{l}$, in which case $\overline{M}$ cannot be an $\mathcal{A}$-implicate of Φ either. □

Moreover, it is possible to unfold the behaviour of the SAT-solver (especially the unit propagation) within the execution of the ImpGen-PiD algorithm. Typically, instead of performing a complete satisfiability test when the algorithm adds a new literal to its set of hypotheses, we can delay this costly operation by unit-propagating this literal instead. This means that we will update the set of formulas we consider by deleting the clauses in which we detect that this selected literal appears and removing its complement from the clauses where we detect that it appears. For efficiency reasons, these operations are usually not performed in a systematic way in modern SAT-solver. Instead the program merely watches two literals in each clause which is sufficient to detect when new unit clauses are produced (which can be in turn unit propagated) [126]. This detection is usually performed by watching two literals for each clause until the corresponding variable is assigned by the SAT-solver. This permits to reduce the number of abducible candidates for the next recursive call of the algorithm.

Similarly, unit propagation can be used to detect beforehand possible conflicts with previously selected literals. In this case, we can immediately deduce that the current hypothesis in an implicate of the initial formula.

Definition 87. Given a set of clauses Φ, the set of *literals deduced by unit propagation from* Φ, which we will denote by $\mathfrak{U}^*(\Phi)$, and the *explanation of a literal l in* Φ are defined inductively as follows:

- If $l \in \Phi$ is a unit clause, then $l \in \mathfrak{U}^*(\Phi)$ and $\mathfrak{E}^*_\Phi(l) \stackrel{\text{def}}{=} \bot$.
- If $C \in \Phi$ is of the form $D \vee l$ and $\overline{D} \subseteq \mathfrak{U}^*(\Phi)$, then $l \in \mathfrak{U}^*(\Phi)$ and $\mathfrak{E}^*_\Phi(l) \stackrel{\text{def}}{=} C$ (if several such clauses exist then one is chosen arbitrarily).

With the help of Definition 87, we describe the unfolding of unit propagation in Algorithm 3.3 and the adapted version of the ImpGen-PiD algorithm for the generation of prime implicates in propositional logic in Algorithm 3.4. In both algorithms, the set of prime implicates Π we construct is handled by reference. The calls to UpdateImplicates and UpdateImplicatesLocal are assumed to update this set of implicates according to the information they obtained from their location within the algorithms and by the unfolding of the unit propagation of the literals. They proceed by applying the following definitions:

Definition 88. Given a set of clauses Φ, three sets of literals M, U, A and a set of clauses Π, UpdateImplicates(Φ, M, U, A, Π) is a strong entailment minimal equivalent of the union of Π with a set of A-implicates of $\Phi \wedge \overline{M}$ denoted by Π' and ensuring that $\Pi \cup \Pi' \models \left\{\overline{M} \vee \bar{l} \,\middle|\, l \in A \wedge \bar{l} \in U\right\}$.

Remark 89. As the choice of Π' is arbitrary, Definition 88 does not ensure the unicity of UpdateImplicates(Φ, M, U, A, Π).

In the ImpGen-PiD Algorithm, when checking whether $\Phi \cup M \models \bot$ or not at Line 3, we can (if the test succeeds) conclude that $\overline{M}$ is an A-implicate of Φ. This single implicate is a good choice for the set Π' of Definition 88 because the satisfiability test it results from is already mandatory in the main algorithm.

Definition 90. Given a set of clauses Φ, two sets of literals M, A and a set of clauses Π, UpdateImplicatesLocal(Φ, M, A, Π) is the strong entailment minimal equivalent of the union of Π with a set of A-implicates of $\Phi \wedge \overline{M}$ denoted by Π' and ensuring that $\Pi \cup \Pi' \models \overline{M}$.

Definition 90 leaves us with the problem of finding good such subsets, as this is the key point of this algorithmic improvement for propositional logic. Once again, we can reuse the fact that we unfold unit propagation to our advantage and construct a method of building such subsets that uses the reasons why literals are unit propagated.

Algorithm 3.3: UnitUpdate$(\Phi, M, A, \text{ref } \Pi)$

```
1 let Π_I ⊆ Π;
2 let U = 𝔘*(M ∪ Π_I ∪ Φ);
3 repeat
4 |   UpdateImplicates(Φ, M, U, A, Π);
5 |   Π_I = Π'_I ⊆ Π;
6 |   U = U ∪ 𝔘*(M ∪ Π_I ∪ Φ);
7 until U is no longer updated;
8 return U;
```

Definition 91. Given a set of clauses, a hypothesis M and a set of abducible literals A, we define the *A-reason of a literal l in the addition of M to Φ* and denote by $\mathcal{R}_{A,\Phi,M}(l)$ the following set:

- If $l \in \Phi$ then $\mathcal{R}_{A,\Phi,M}(l) \stackrel{\text{def}}{=} \{\emptyset\}$.

Algorithm 3.4: ImpGen-Propositional(Φ, M, A, ref Π)

let $U = \text{UnitUpdate}(\Phi, M, A, \Pi)$;
let $\Phi_U = \{C \in \Phi \mid U \models C\}$;
let $\Phi'_U = \left\{C \setminus \overline{U} \;\middle|\; C \in \Phi \setminus \Phi_U\right\}$;
let $\mathfrak{A} = A \cap \left\{l \in \mathcal{L} \;\middle|\; \bar{l} \in \text{Lits}(\Phi'_U) \wedge \{l, \bar{l}\} \cap U = \emptyset\right\}$;
if $\bot \in \Phi'_U$ **or** $\mathfrak{A} = \emptyset \wedge \Phi'_U \models \bot$ **then**
    UpdateImplicatesLocal(Φ, M, A, Π) ;
    return ;
let $I = \{l \mid \mathcal{M} \models l\}$ if a (partial) model $\mathcal{M}$ of $\Phi \cup M$ is found, $I = \mathcal{L}$ otherwise.;
ImpGen-Propositional($\Phi, M \cup \{\max(\mathfrak{A})\}, \mathfrak{A}^I[\max(\mathfrak{A})], \Pi$) ;
if $P \not\models \overline{M}$ **then**
    return ;
$U = \text{UnitUpdate}(\Phi, M, A, \Pi)$;
$\mathfrak{A} = \{l_i\}_{i<j \Rightarrow l_i < l_j}$;
foreach $l_i \in \left\{l \in \mathfrak{A} \;\middle|\; \bar{l} \in I \wedge l \neq \max(\mathfrak{A})\right\}$ *in decreasing order* **do**
    if $\Pi \not\models \overline{M} \vee \overline{l_i}$ **then**
        ImpGen-Propositional($\Phi, M \cup \{l_i\}, \mathfrak{A}^I[l_i], \Pi$) ;

- If $l \in M \setminus \Phi$ then $\mathcal{R}_{A,\Phi,M}(l) \stackrel{\text{def}}{=} \{\{l\}\}$.
- Otherwise, if $\mathfrak{E}^*_{\Phi \cup M}(l) = l_1 \vee \ldots \vee l_n \vee l$, then

$$\mathcal{R}_{A,\Phi,M}(l) \stackrel{\text{def}}{=} \left\{ \bigcup_{i=1}^{n} P_i \;\middle|\; \forall i = 1, \ldots, n, P_i \in \mathcal{R}_{A,\Phi,M}(\overline{l_i}) \right\} \bigcup_{\text{if } l \in \mathcal{A}} \{\{l\}\}$$

.

Proposition 92. *Let $l \in \mathfrak{U}^*(\Phi \cup M)$. If $P \in \mathcal{R}_{A,\Phi,M}(l)$, then $P \subseteq (A \cup M) \cap \mathfrak{U}^*(\Phi \cup M)$ and $P \cup \Phi \models l$.*

Proof. This is proved by induction. The result is obvious if $l \in \Phi$ or $l \in M$. Otherwise, let $D \vee l = \mathfrak{E}^*_{\Phi \cup M}(l)$, where $D = l_1 \vee \ldots \vee l_n$, so that $P = \bigcup_{i=1}^n P_i$ and $P_i \in \mathcal{R}_{A,\Phi,M}(\overline{l_i})$. By hypothesis, for all $i = 1, \ldots, n$, $\Phi \cup P_i \models \overline{l_i}$. Hence, $\Phi \cup P \models \Phi \cup \overline{D} \models l$. The fact that $P \subseteq (A \cup M) \cap \mathfrak{U}^*(\Phi \cup M)$ is similarly proved by induction. □

Proposition 93. *For all $l \in \mathfrak{U}^*(\Phi \cup M)$, there exists a $P \in \mathcal{R}_{A,\Phi,M}(l)$ such that $P \subseteq M$.*

Proof. The proof is by induction on the length of the derivation of l from $\Phi \cup M$. If $l \in \Phi$ then $\mathcal{R}_{A,\Phi,M}(l) = \{\emptyset\}$ and $\emptyset \subseteq M$. If $l \in M \setminus \Phi$ then $\mathcal{R}_{A,\Phi,M}(l) = \{\{l\}\}$ and $\{l\} \subseteq M$. Otherwise, necessarily $\mathfrak{E}^*_{\Phi \cup M}(l) = l_1 \vee \cdots \vee l_n \vee l$ where, for all $i \in [\![1, n]\!]$, $l_i \in \mathfrak{U}^*(\Phi \cup M)$ and the derivation yielding l_i is strictly lower than that of l. Thus by the induction hypothesis $\mathcal{R}_{A,\Phi,M}(l_i)$ contains a set $P_i \subseteq M$ and $\mathcal{R}_{A,\Phi,M}(l_i)$ contains the set $\bigcup_{i=1}^n P_i \subseteq M$. □

Notation 94. We will denote by $\rho_M(l)$ an (arbitrary) element $P \in \mathcal{R}_{A,\Phi,M}(l)$ such that $P \subseteq M$.

Definition 95. Given a set of clauses Φ and a set of literals M, we let $\mathfrak{H}_A(\Phi, M)$ be defined by

$$\mathfrak{H}_A(\Phi, M) \stackrel{\text{def}}{=} \left\{ \overline{P} \vee l \,\middle|\, l \in \mathfrak{U}^*(\Phi \cup M) \cap A \wedge P \in \mathcal{R}_{A,\Phi,M}(l) \right\}$$

Lemma 96. *If $U = \mathfrak{U}^*(\Phi \cup M)$, then we have*

$$\mathfrak{H}_A(\Phi, M) \subseteq \left\{ C \,\middle|\, \overline{C} \subseteq U \cap (M \cup A) \wedge \Phi \cup \overline{C} \models \bot \right\}$$

(where C denotes a clause), and also

$$\mathfrak{H}_A(\Phi, M) \Vdash \left\{ \overline{M} \vee \bar{l} \,\middle|\, l \in A \wedge \bar{l} \in U \right\}$$

.

Proof. Let $\overline{P} \vee l \in \mathfrak{H}_A(\Phi, M)$, where $l \in U \cap A$. By Proposition 92, $P \subseteq (A \cup M) \cap U$ and $\Phi \cup P \cup \{\bar{l}\} \models \bot$, which proves that $\overline{P} \vee l \in \left\{ \overline{C} \,\middle|\, C \subseteq U \cap (M \cup A) \wedge \Phi \cup C \models \bot \right\}$.

By definition, every clause in $\left\{ \overline{M} \vee \bar{l} \,\middle|\, l \in A \wedge \bar{l} \in U \right\}$ is of the form $\overline{M} \vee l'$ where $l' \in U \cap A$. By Proposition 93, there is a $P \in \mathcal{R}_{A,\Phi,M}(l')$ such that $P \subseteq M$, which proves that $\mathfrak{H}_A(\Phi, M) \Vdash \overline{M} \vee l'$. □

Lemma 97. *Let l be a literal such that $\{l, \bar{l}\} \subseteq \mathfrak{U}^*(\Phi \cup M)$ and let $D = \rho_M(l) \cup \rho_M(\bar{l})$. Then $\overline{D} \Vdash \overline{M}$ and $\Phi \models \overline{D}$.*

Proof. By Proposition 92 we have $\Phi \cup D \models \{l, \bar{l}\} \models \bot$. □

The A-implicates of Φ that can be added to the set of already generated implicates can therefore be taken from $\mathfrak{H}_A(\Phi, M)$. Any subset of the latter that contains $\rho_M(l)$ for all $l \in \mathfrak{U}^*(\Phi \cup M)$ can be safely added while maintaining the correctness of the algorithm.

Chapter 4

Storing $(\mathcal{T}, A)$-implicates efficiently

The number of implicates of a given formula may be huge, hence it is essential in practice to have appropriate data structures to store them in a compact way. Moreover, as stored implicates can be used to prune exploration branches (typically by detecting that all the $(\mathcal{T}, A)$-implicates it can generate will be a consequence of a stored one), it is also critical to possess efficient algorithms to check that a newly generated $(\mathcal{T}, A)$-implicate I is not redundant (this is usually called *forward subsumption modulo* $\mathcal{T}$), and if so, to delete all the already generated implicates that are less general than I (usually called *backward subsumption modulo* $\mathcal{T}$), before inserting I into the structure for stored implicates.

4.1 A structure for storing the $(\mathcal{T}, A)$-implicates

In this section, we devise a trie-like data structure to perform these tasks. We will also explain how we can make use of this data structure to improve the efficiency of Algorithm 3.2.

In order to comply with the genericity restrictions of our main algorithm, once again, the definition and usage of the structure we define only rely on the existence of a decision procedure for testing satisfiability in $\mathcal{T}$. We will use this procedure to perform the implication verifications required for storing the implicates.

Definition 98. Let $<_s$ be an order on the literals in $\mathcal{A}$, possibly, but not necessarily, equal to the order $<$ used for literal ordering in the implicate generation algorithm. An $\mathcal{A}$-*trie* is inductively defined as $\bot$ (the *empty leaf*) or a possibly empty set of pairs $\{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, where $l_1, \ldots, l_n$ are pairwise distinct literals in $\mathcal{A}$ and $\tau_i, i \in [\![1, n]\!]$ is an $\mathcal{A}$-trie only containing literals that are strictly greater than l_i (according to the order $<_s$). An $\mathcal{A}$-trie is associated with a set of $\mathcal{A}$-clauses inductively defined as follows:

$$\begin{array}{rcl} \mathcal{S}(\bot) & \stackrel{\text{def}}{=} & \{\bot\}, \\ \mathcal{S}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}) & \stackrel{\text{def}}{=} & \bigcup_{i=1}^{n} \{l_i \vee C \mid C \in \mathcal{S}(\tau_i)\}. \end{array}$$

Note in particular that $\mathcal{S}(\emptyset) = \emptyset$. Intuitively an $\mathcal{A}$-trie may be seen as a tree in which the edges are labeled by literals and the leaves are labeled by $\emptyset$ or $\bot$, and represents a set of clauses corresponding to paths from the root to $\bot$.

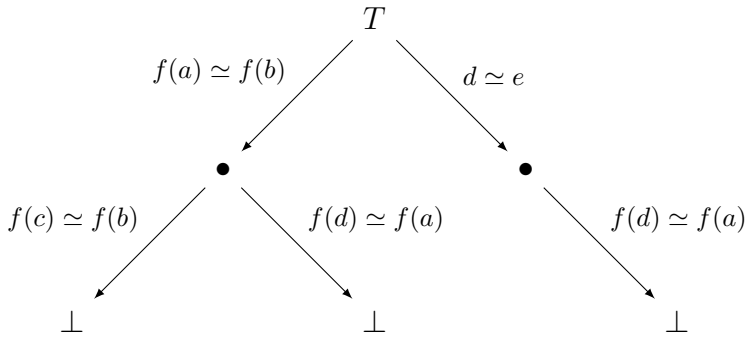


Figure 4.1: A representation of an $\mathcal{A}$-trie

Example 99. We represent an $\mathcal{A}$-trie in Figure 4.1. Assuming we work in the theory of equality with uninterpreted function, this $\mathcal{A}$-trie contains three empty leaves and thus three paths from its root T to a leaf. This $\mathcal{A}$-trie thus represents the set of clauses $\{f(a) \simeq f(b) \vee f(c) \simeq f(b), f(a) \simeq f(b) \vee f(d) \simeq f(a), d \simeq e \vee f(d) \simeq f(a)\}$.

4.2 Algorithms to efficiently insert and reuse the $(\mathcal{T}, A)$-implicates

We now have to define the operations we need on this structure. We start by presenting the straightforward algorithm permitting the insertion of a clause in an $\mathcal{A}$-trie.

Definition 100. Let C be a clause and let τ be an $\mathcal{A}$-trie. $\mathbf{insert}(\tau, C)$ denotes the $\mathcal{A}$-trie defined inductively as follows:

- If $C = \bot$ then $\mathbf{insert}(\tau, C) = \bot$
- If $C = l_0 \vee l_1 \vee \ldots l_n$ with $l_0 <_s l_1 <_s \ldots <_s l_n$ then
  - If there exists a pair $(l', \tau') \in \tau$ verifying $l' = l_0$ then
    $$\mathbf{insert}(\tau, C) = (\tau \setminus \{(l', \tau')\}) \cup \{(l', \mathbf{insert}(\tau', l_1 \vee \ldots l_n))\}$$
  - Otherwise, $\mathbf{insert}(\tau, C) = \tau \cup \{(l_0, \mathbf{insert}(\emptyset, l_1 \vee \ldots l_n))\}$

Proposition 101. *Given a clause C and an $\mathcal{A}$-trie, $\mathcal{S}(\mathbf{insert}(\tau, C)) \models\!\models \mathcal{S}(\tau) \cup \{C\}$.*

Proof. This is ensured inductively. For any clause C, $\{C\} \models\!\models \{C\}$. Moreover, for $C = l_0 \vee D$ with D a clause, if $\mathcal{S}(\mathbf{insert}(\emptyset, D)) \models\!\models \mathcal{S}(\emptyset) \cup \{D\}$ then

$$\mathcal{S}(\tau \cup \{(l_0, \mathbf{insert}(\emptyset, D))\}) \models\!\models \mathcal{S}(\tau) \cup \{C\}$$

and if $\mathcal{S}(\mathbf{insert}(\tau', D)) \models\!\models \mathcal{S}(\tau') \cup \{D\}$ then

$$\mathcal{S}((\tau \setminus \{(l', \tau')\}) \cup \{(l', \mathbf{insert}(\tau', D))\}) \models\!\models \mathcal{S}(\tau) \cup \{C\}.$$

□

One can remark that this algorithm does not perform any subsumption verification. Indeed, we expect the subsumption verifications and modifications to be done before any insertion into the $\mathcal{A}$-trie. Before inserting a given clause, we will first verify that it is not already entailed by another one in the storage, then remove from the $\mathcal{A}$-trie any clause that would be less general than it is, and then only insert it into the $\mathcal{A}$-trie. We now describe these two subsumption algorithms.

Definition 102. We introduce the following simplification rule (which may be applied at any depth inside a tree, not only at the root level):

$$\textbf{del}: \quad \tau \cup \{l : \emptyset\} \quad \to \quad \tau$$

Informally, the rule deletes all leaves labeled by $\emptyset$ except for the root. It may be applied recursively, for instance $\{l : \{l_1 : \emptyset, \dots, l_n : \emptyset\}\} \to_{\textbf{del}}^{n+1} \emptyset$. Termination is immediate since the size of the tree is strictly decreasing. This rule will be used to clean $\mathcal{A}$-tries after some path deletion operations have been performed in it (see below).

Proposition 103. *If $\tau \to_{\textbf{del}} \tau'$ then τ' is an $\mathcal{A}$-trie and $\mathcal{S}(\tau) = \mathcal{S}(\tau')$.*

The following lemma provides a simple algorithm to check whether a clause is a logical consequence modulo $\mathcal{T}$ of some clause in $\mathcal{S}(\tau)$ (forward subsumption). The algorithm proceeds by induction on the $\mathcal{A}$-trie.

Lemma 104. *Let C be a clause and let τ be an $\mathcal{A}$-trie. We have $\mathcal{S}(\tau) \Vdash C$ if and only if one of the following conditions hold:*

- $\tau = \bot$.
- $\tau = \{l_1 : \tau_1, \dots, l_n : \tau_n\}$ *and there exists $i \in [\![1, n]\!]$ such that $l_i \models_{\mathcal{T}} C$ and $\mathcal{S}(\tau_i) \Vdash C$.*

Proof. If $\tau = \bot$ then $\mathcal{S}(\tau) = \{\bot\} \Vdash C$ hence the equivalence holds. Otherwise, let $\tau = \{l_1 : \tau_1, \dots, l_n : \tau_n\}$. By definition, $\mathcal{S}(\tau) \Vdash C$ holds if and only if there exists a clause $D \in \mathcal{S}(\tau)$ such that $D \models_{\mathcal{T}} C$. Since $\mathcal{S}(\{l_1 : \tau_1, \dots, l_n : \tau_n\}) = \bigcup_{i=1}^{n} \{l_i \vee E \mid E \in \mathcal{S}(\tau_i)\}$, the previous property holds if and only if there exists $i \in [\![1, n]\!]$ and $E \in \mathcal{S}(\tau_i)$ such that $l_i \vee E \models_{\mathcal{T}} C$, *i.e.*, such that $l_i \models_{\mathcal{T}} C$ and $E \models_{\mathcal{T}} C$ by Proposition 34. By definition, $\exists E\, (E \models_{\mathcal{T}} C \wedge E \in \mathcal{S}(\tau_i))$ if and only if $(\mathcal{S}(\tau_i) \Vdash C)$. Furthermore, $l_i \Vdash C$ holds if and only if $\overline{C} \cup \{l_i\}$ is $\mathcal{T}$-unsatisfiable, hence the result. □

Example 105. Let us consider again the $\mathcal{A}$-trie we presented in Example 99. Now let us assume that we want to check if $\mathcal{S}(T) \Vdash_{\mathcal{T}} D = f(d) \simeq f(e) \vee f(d) \simeq f(a)$ with the algorithm of Lemma 104. The execution is detailed in Figure 4.2 and described below. It starts by verifying if a clause entailing D can be found in the left part of T. To do this, it checks whether $f(a) \simeq f(b) \models_{\mathcal{T}} D$ (second point of Lemma 104). As this is not the case, it proceeds to the second possible literal it can find from the root of T : $d \simeq e$. This time, we have $d \simeq e \models_{\mathcal{T}} D$. The algorithm will then look at the subtrie below this literal and continue to check if they subsumes the clause. As we have both that $f(d) \simeq f(a) \models_{\mathcal{T}} D$ and that $\bot$ is a subtrie below this literal, we can conclude that D is entailed by $\mathcal{S}(T)$, and that the clause showing this is $d \simeq e \vee f(d) \simeq f(a)$.

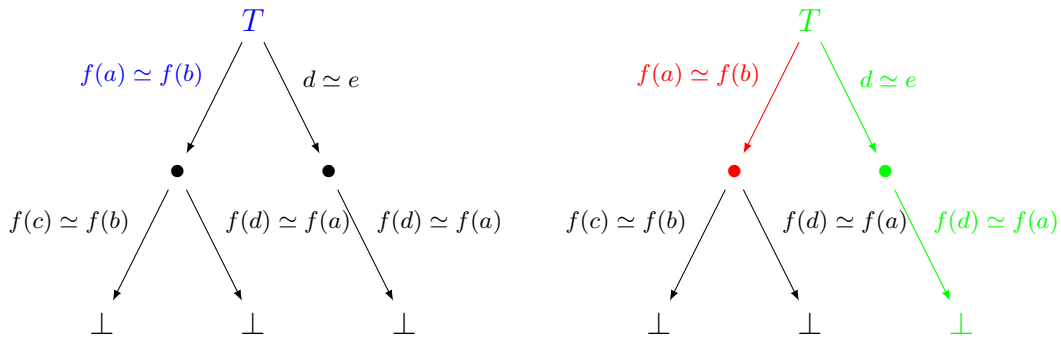


Figure 4.2: Example of forward subsumption

Before inserting any clause into an $\mathcal{A}$-trie, we will thus use this algorithm to check if the insertion is useful. If it is, we will then use the following definition, which provides an algorithm to remove, in a given $\mathcal{A}$-trie, all branches corresponding to clauses that are logical consequences of a given formula modulo $\mathcal{T}$ (backward subsumption).

Definition 106. Let ϕ be a formula and let τ be an $\mathcal{A}$-trie. $\mathbf{clear}(\tau, \phi)$ denotes the $\mathcal{A}$-trie defined as follows:

- If ϕ is $\mathcal{T}$-unsatisfiable, then $\mathbf{clear}(\tau, \phi) \stackrel{\text{def}}{=} \emptyset$.
- If ϕ is $\mathcal{T}$-satisfiable, then:
  - $\mathbf{clear}(\bot, \phi) \stackrel{\text{def}}{=} \bot$,
  - $\mathbf{clear}(\{l_1 : \tau_1, \ldots, l_n : \tau_n\}, \phi) \stackrel{\text{def}}{=} \bigcup_{i=1}^{n} \{l_i : \mathbf{clear}(\tau_i, \phi \wedge \overline{l_i})\}$.

Intuitively, starting with some clause ϕ, the algorithm incrementally adds to a temporary set literals $\overline{l_1}, \ldots, \overline{l_n}$ occurring in the clauses $D = l_1 \vee \cdots \vee l_n \in \mathcal{S}(\tau)$ and invokes the SMT-solver after each addition. If a contradiction is found then this means that $\phi \models_{\mathcal{T}} D$, hence the branch corresponding to D can be removed. The advantage of using this within the $\mathcal{A}$-trie is that the calls are shared among all common prefixes. Of course, this algorithm is even more interesting if the SMT-solver is able to perform efficient incremental satisfiability testing. The following Lemma provides a proof that the function effectively removes from an $\mathcal{A}$-trie the clauses entailed by its parameter.

Lemma 107. *Let ϕ be a formula and let τ be an $\mathcal{A}$-trie. Then $\mathbf{clear}(\tau, \phi)$ is an $\mathcal{A}$-trie, and $\mathcal{S}(\mathbf{clear}(\tau, \phi)) = \{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\}$.*

Proof. The proof is by induction on τ. We also prove that all literals in $\mathbf{clear}(\tau, \phi)$ also occur in τ. We distinguish several cases.

- If ϕ is $\mathcal{T}$-unsatisfiable, then $\phi \models_{\mathcal{T}} C$, for every $C \in \mathcal{S}(\tau)$. Consequently,

  $$\{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\} = \emptyset.$$

  Furthermore, $\mathbf{clear}(\tau, \phi) = \emptyset$, thus $\mathbf{clear}(\tau, \phi)$ is an $\mathcal{A}$-trie and $\mathcal{S}(\mathbf{clear}(\tau, \phi)) = \emptyset$. Therefore, the result holds. Note that $\mathbf{clear}(\tau, \phi)$ contains no literal at all.

- If ϕ is $\mathcal{T}$-satisfiable and $\tau = \bot$, then $\mathcal{S}(\tau) = \{\bot\}$ and $\phi \not\models \bot$, thus
$$\{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\} = \{\bot\}.$$
Furthermore, $\mathbf{clear}(\tau, \phi) = \bot$, thus $\mathbf{clear}(\tau, \phi)$ is an $\mathcal{A}$-trie and the equality holds. As in the previous case, $\mathbf{clear}(\tau, \phi)$ contains no literal.

- Now assume ϕ is $\mathcal{T}$-satisfiable and $\tau = \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, where $l_1, \ldots, l_n$ are pairwise distinct, and all literals in τ_i are strictly greater than l_i. Then
$$\mathcal{S}(\{l_1 \colon \tau_1, \ldots, l_n \colon \tau_n\}) = \bigcup_{i=1}^{n} \{l_i \vee D \mid D \in \mathcal{S}(\tau_i)\}.$$
hence
$$\{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\} = \bigcup_{i=1}^{n} \{l_i \vee D \mid D \in \mathcal{S}(\tau_i) \text{ and } \phi \not\models_{\mathcal{T}} l_i \vee D\}.$$
By Proposition 34, we deduce that
$$\{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\} = \bigcup_{i=1}^{n} \left\{l_i \vee D \;\middle|\; D \in \mathcal{S}(\tau_i) \text{ and } \phi \wedge \overline{l_i} \not\models_{\mathcal{T}} D\right\}.$$
By the induction hypothesis, $\mathbf{clear}(\tau_i, \phi \wedge \overline{l_i})$ is an $\mathcal{A}$-trie and
$$\mathcal{S}(\mathbf{clear}(\tau_i, \phi \wedge \overline{l_i})) = \left\{D \in \mathcal{S}(\tau_i) \;\middle|\; \phi \wedge \overline{l_i} \not\models D\right\}$$
thus:
$$\{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\} = \bigcup_{i=1}^{n} \left\{l_i \vee D \;\middle|\; D \in \mathcal{S}(\mathbf{clear}(\tau_i, \phi \wedge \overline{l_i}))\right\}.$$
By definition, $\mathbf{clear}(\tau, \phi) = \bigcup_{i=1}^{n} \{l_i : \mathbf{clear}(\tau_i, \phi \wedge \bar{l})\}$. The $l_1, \ldots, l_n$ are pairwise distinct and all literals in $\mathcal{S}(\tau_i)$ also occur in τ_i, thus are greater than l_i. Hence this entails that $\mathbf{clear}(\tau, \phi)$ is an $\mathcal{A}$-trie, and
$$\mathcal{S}(\mathbf{clear}(\tau, \phi)) = \bigcup_{i=1}^{n} \left\{l_i \wedge D \;\middle|\; D \in \mathcal{S}(\mathbf{clear}(\tau_i, \phi \wedge \bar{l}))\right\} = \{C \in \mathcal{S}(\tau) \mid \phi \not\models_{\mathcal{T}} C\}.$$
By definition, the literals in $\mathbf{clear}(\tau, \phi)$ occur either in $l_1, \ldots, l_n$ or $\mathbf{clear}(\tau_i, \phi \wedge \bar{l})$ (hence τ_i), thus must occur in τ.

□

Example 108. Let us consider one last time the $\mathcal{A}$-trie of Example 99. This time, let us consider that we are about to add the unit clause $C = a \simeq b$ to T. Before that, we will have to remove all the clauses contained in T that are logical consequences of C. We describe this execution of the algorithm from Definition 106. It is also represented in Figure 4.3. It starts at the root of T, by adding the complement of the first literal of the node $(f(a) \simeq f(b))$ to C. As we can immediately derive a contradiction from this, the algorithm will remove this whole subtrie. It then pursues its exploration of T with the other literal it finds at the root: $d \simeq e$. No contradiction can be found with this second literal, nor when we add the one in the related subtrie later. This means that we will keep this part of the $\mathcal{A}$-trie.

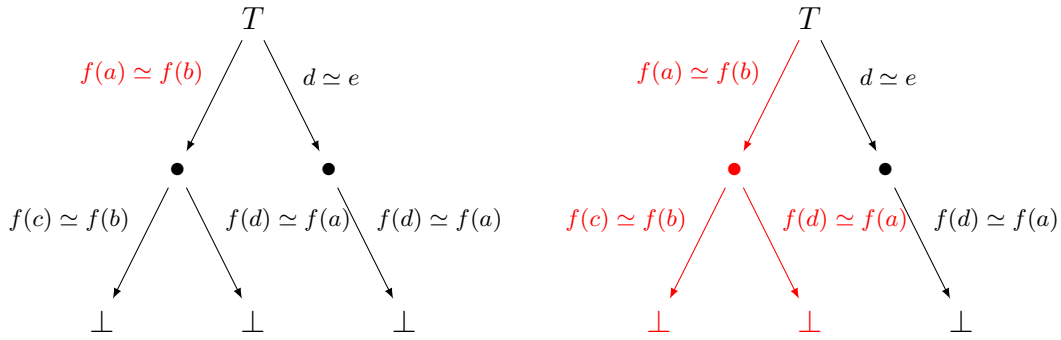


Figure 4.3: Example of backward subsumption

Remark 109. The $\mathcal{A}$-tries may be represented as directed acyclic graphs instead of trees. In this case, it is clear that the complexity, when defined as the number of satisfiability tests of forward subsumption (as defined in Lemma 104) is of the same order as the size of the directed acyclic graph, since the recursive calls only depend on the considered subtree. For backward subsumption (see Definition 106), the situation is different since the recursive calls have an additional parameter that is the formula ϕ, which depends on the path in the $\mathcal{A}$-trie. The maximal number of satisfiability tests is therefore equal to the size of the underlying tree, and not that of the directed acyclic graph.

4.3 An updated algorithm including storage management

We provide in Algorithm 4.1 an updated version of ImpGen-PiD that uses $\mathcal{A}$-tries to store the generated implicates.

Algorithm 4.1: ImpGen-Storage $(\Phi, M, A, \tau, \mathcal{P})$

if $M \models \bot$ **or** $\neg\mathcal{P}(M)$ **or** $\mathcal{S}(\tau) \models_{\mathcal{T}} \overline{M}$ **then**
  return τ ;
if $\Phi \cup M \models \bot$ **then**
  return **insert**(**clear**$(\tau, \overline{M}), \overline{M})$;
let $U \subseteq \mathfrak{U}_{\mathcal{T},\Phi,M}$ such that $M \subseteq U$;
let $\Phi = \mathfrak{U}_{\mathcal{T},M}[U](\Phi)$;
let $A = \texttt{fix}(M, A, \Phi)$;
let I be an $(\Phi \cup M)$-compatible set of literals ;
foreach $l \in A$ *such that* $\bar{l} \in I$ **do**
  let $\tau =$ ImpGen-Storage$(\Phi, M \cup \{l\}, A^I[l], \tau, \mathcal{P})$;
return τ ;

Theorem 110. *If $\mathcal{P}$ is $\subseteq$-closed then*

$$\mathcal{I}^*_{\mathcal{T},\mathcal{A}}(\Phi) \cap \left\{ \overline{A} \,\middle|\, A \in \mathcal{P} \right\} = \textsc{ImpGen-Storage}(\Phi, \emptyset, \mathcal{A}, \bot, \mathcal{P}).$$

This concludes the presentation of the efficient generic algorithm to compute the prime $(\mathcal{T}, A)$-implicates of a set of formulas Φ. We have considered various improvements to reduce the search space of the algorithm, provided a storage structure that permits to store and work efficiently with the $(\mathcal{T}, A)$-implicates we have already generated, and added a user-tunable filter feature that can help pruning the search space even further if we can design external restrictions on the form of the desired implicates. It is clear that more improvements could be made to this algorithm, this will discussed in Chapter 11.6 after we have performed an extensive evaluation of the version we designed in this chapter (see Chapter 9 for this). Still, this generic implicate generation algorithm will be at the centre of an application to program loop invariants generation we will develop in Chapters 5 and 6. Finally, the next chapter provides some interesting information on methods that can be used to adapt this algorithm for the specific case of prime implicate generation in propositional logic.

Part II

Using abduction in verification

Chapter 5

An abstract theoretical framework

In part I, we devised an algorithm that, given a set of formulas Φ expressed in a decidable theory $\mathcal{T}$ and a set of abducible literals $\mathcal{A}$, computes all the prime $(\mathcal{T}, \mathcal{A})$-implicates of Φ. Now let us consider, for instance, a program, with a set of preconditions (a set of logical formulas we assume to hold at the beginning of the program), and for which we want to ensure some postconditions (a set of logical formulas we want to prove hold after the execution of the program). A standard method to perform this type of verification is to generate from the program, its preconditions and its postconditions, a set of verification conditions that are valid if and only if the postconditions of the program hold. When the program is only composed of a sequence of instructions with no loops or recursive calls, it is usually easy to generate these verification conditions and to decide whether they are valid or not. As long as the verification conditions are expressed in a simple enough theory, we can decide whether or not they are verified, typically by tasking an SMT-solver to answer this question. When no such restrictions are considered however, this problem is undecidable. In the case where the programs under consideration contain loops, it is possible to use loop invariants to extend this verification framework. Intuitively, a loop invariant is a formula that holds when the program first enters the loop and is preserved by the instructions contained in this loop. By using these loop invariants as preconditions of loop-free sequences of instructions, it becomes possible to check whether the postconditions of programs containing loops are verified. This, however, requires that the loop invariants are also verified, that is, that they hold at the moment the program enters the loop they refer to and they are preserved by the content of this loop. A similar approach can be devised for recursive functions. This leaves us with the problem of generating loop invariants for the programs we consider.

Given a program in which not all the verification conditions are valid, we want to update the preconditions or find loop invariants to ensure that all the verification conditions are satisfied.

Example 111. Let us consider the sample program presented in Algorithm 5.1. It turns out that the postcondition of this program is not verified. Take for instance an array $T = [-1, 0, 0]$, $a = 1$ and $b = 1$. After executing Algorithm 5.1, we would have $T = [-1, 0, -1]$ and thus $T[b+1] < T[b]$, which contradicts the postcondition. Such instances of T, a and b could be obtained from an SMT-solver. If we want to obtain an additional precondition ensuring the postcondition holds however, we are not interested in trivial solutions such as $\bot$ or too restrictive solutions such as $a = 0 \wedge b = 1$. The kind of "interesting conditions"

Algorithm 5.1: `ArrayExample`(Array[Int] T, Int a, Int b)

1 **requires** $T[a] > 0$;
2 **requires** $\forall x, y \in [\![a, b]\!].x \leq y \Rightarrow T[x] \leq T[y]$;
3 **let** $T[b+1] = T[b-1] + T[b]$;
4 **ensures** $\forall x, y \in [\![a, b+1]\!].x \leq y \Rightarrow T[x] \leq T[y]$;

we look for are, for example $a \neq b$ or $T[b-1] \geq 0$.

In this chapter, we present an algorithm that performs this task of finding meaningful additional preconditions to ensure that a set of derived verification conditions is satisfied. In [57], Dillig *et al.* proposed a similar approach to generate property-directed loop invariants of numeric programs, and restricted to theories admitting quantifier elimination. In their work, given a program decorated with loop invariant candidates, preconditions and postconditions, they derive verification conditions ensuring when valid that the considered candidates are indeed loop invariants and that they are strong enough to deduce the postconditions. If not all these verification conditions are valid, they use an abduction algorithm based on quantifier elimination to strengthen their candidate loop invariants until they are able to obtain valid verification conditions only.

Here we reuse the techniques they introduced in [57] and combine them with the more general abductive reasoning procedure introduced in Chapter 3. We thus obtain a completely generic approach for computing either loop invariants or preconditions of programs. In order to cope with the huge search space related to this genericity, we will also instantiate and adapt this algorithm specifically for loop invariant generation in Chapter 6, notably by introducing strategies to determine which verification conditions should be considered first and which loop invariants should be strengthened.

5.1 A framework for describing verification problems

Definition 112. We define a *verification structure* P as a tuple (Φ, Υ) where $\Phi = (\varphi_1, \ldots, \varphi_n)$ is a sequence of logical formulas called *guide formulas*, and where $\Upsilon = (\upsilon_1, \ldots, \upsilon_m)$ is a sequence of functions from $\mathfrak{S}^n$ to $\mathfrak{S}$ called *verification conditions generators*.

We will say that a verification structure $P = (\Phi, \Upsilon)$ is *satisfied* if $\models \bigwedge_{i \in [\![1,m]\!]} \upsilon_i(\Phi)$. If this is the case, we will write $\models P$

Intuitively, verification structures are a generic representation in which we can encode our programs. The preconditions and loop invariants we consider will be encoded into guide formulas and the content of the programs (as well the postconditions we want to prove on them) will be encoded into verification condition generators. The fact that we can have several guide formulas will be useful in the context of loop invariant generation, where we can encode a loop invariant by a guide formula.

Notation 113. Given a verification structure $P = (\Phi, \Upsilon)$, we will denote its i^{th} guide formula by $\text{Guide}_i(P)$ and its j^{th} verification condition generator by $\text{VCGen}_j(P)$. We will also denote by $\text{Guides}(P)$ the sequence Φ and by $\text{VCGen}(P)$ the sequence Υ.

Definition 114. Given a verification structure $P = (\Phi, \Upsilon)$, we will call *verification conditions of* P the set of formulas

$$VC(P) = \{\text{VCGen}_i(P)(\Phi) \mid 1 \leq i \leq \textsc{Card}(\Upsilon)\}.$$

We will call i^{th} verification condition of P the formula

$$VC_i(P) = \text{VCGen}_i(P)(\Phi).$$

Example 115. Let us consider the sample program of Example 111 (Algorithm 5.1). We can express the problem of whether the program verifies its postcondition within the framework of Definition 112. Consider first the sequence $\Phi_{5.1} = ((\forall x, y \in [\![a, b]\!].x \leq y \Rightarrow T[x] \leq T[y]) \wedge T[a] > 0)$. Now consider the verification condition generator $\upsilon_{5.1} : \phi \mapsto \neg\phi \vee (\forall x, y \in [\![a, b+1]\!].x \leq y \Rightarrow T[x] \leq T[y])$. If we consider the verification structure $P_{5.1} = (\Phi_{5.1}, (\upsilon_{5.1}))$, we obtain a verification structure that represents the problem of verifying whether Algorithm 5.1 verifies all its postconditions. As this is not the case, our goal will be to update the guide formula of $P_{5.1}$ until it represents a set of preconditions that solves this problem.

Notation 116. Given a verification structure $P = ((\varphi_1, \ldots, \varphi_{i-1}, \varphi_i, \varphi_{i+1}, \ldots, \varphi_n), \Upsilon)$ and a formula ϕ, we will denote by $P[i \longleftarrow \phi]$ the verification structure in which we have replaced the i^{th} guide formula by ϕ, that is, $((\varphi_1, \ldots, \varphi_{i-1}, \phi, \varphi_{i+1}, \ldots, \varphi_n), \Upsilon)$.

Similarly, given a verification structure $P = ((\varphi_1, \ldots, \varphi_n), \Upsilon)$ and a sequence of formulas $\Phi = (\phi_1, \ldots, \phi_n)$ such that $\textsc{Card}(\Phi) = \textsc{Card}(\text{Guides}(P))$, we will denote by $P[* \longleftarrow \Phi]$ the verification structure in which all the guide formulas have been replaced by the formulas at the same index in Φ, that is, $((\phi_1, \ldots, \phi_n), \Upsilon)$.

Definition 117. We will say that two verification structures $P_1 = (\Phi_1, \Upsilon_1)$ and $P_2 = (\Phi_2, \Upsilon_2)$ are *similar* if they only differ by their guide formulas, that is, if $\Upsilon_1 = \Upsilon_2$. In this case, we will write $P_1 \sim P_2$.

The problem we wish to solve is thus, given a verification structure P that is not satisfied, to find a satisfied verification structure P' that is similar to P. Typically, in Example 115, the verification structure $P'_{5.1} = P_{5.1}[1 \longleftarrow \text{Guide}_1(P_{5.1}) \wedge a \neq b]$ is a solution to this problem.

As mentioned in the introduction, we want to search for this similar verification structure by using an abduction algorithm to compute strengthenings of the guide formulas we already have. In order to do that, we are going to select in our verification structure a verification condition that is not valid, and use an abduction algorithm for the theory it is expressed in to generate relevant additional formulas to make it valid. Once such a formula is recovered, we will have to select one of the guide formulas of the verification structure to strengthen. What we need to ensure is that, when the guide formula is strengthened, the verification condition generated within the new program by the same verification condition generator will be valid. We will succeed if we can find a sequence of strengthenings to update our initial verification structure with and satisfy it.

Notation 118. Let $P = (\Phi, \Upsilon)$ be a non-satisfied verification structure. We will denote by $\text{pending}(P)$ the set of indices of verification condition generators returning non-valid verification conditions in P. More formally, we have

$$\text{pending}(P) = \{j \mid \not\models \text{VCGen}_j(P)(\text{Guides}(P))\}.$$

Definition 119. Consider a verification condition generator υ and an index i such that υ has at least i parameters. We say that υ *depends on its* i^{th} *parameter* if we have $\exists(\phi_1, \ldots, \phi_{i-1}, \phi_{i+1}, \phi_n) \in \mathfrak{S}^{n-1}.\exists\psi_1, \psi_2 \in \mathfrak{S}^2.\upsilon(\phi_1, \ldots, \phi_{i-1}, \psi_1, \phi_{i+1}, \ldots, \phi_n) \neq \upsilon(\phi_1, \ldots, \phi_{i-1}, \psi_2, \phi_{i+1}, \ldots, \phi_n)$.

Given a verification structure $P = (\Phi, \Upsilon)$ and an index i such that the verification condition generators of P have at least i parameters, we will denote by $\mathfrak{D}^i(P)$ the set of verification condition generators of P that depend on their i^{th} parameter.

Definition 120. We say that a verification structure P_0 is *a strict update* of another verification structure P_1, and write $P_0 \prec P_1$, if $P_0 \sim P_1$ and there exists an index i and a formula ϕ such that $P_0 = P_1[i \longleftarrow \phi]$, $\text{Guide}_i(P_1) \not\models \phi$ and $\phi \models \text{Guide}_i(P_1)$.

We say that a verification structure P_0 is *an update* of another verification structure P_1, and write $P_0 \prec^+ P_1$, if either P_0 is a strict update of P_1 or there exists a non-empty finite sequence $(Q_0, \ldots, Q_n)$ such that P_0 is a strict update of Q_0, Q_n is a strict update of P_1, and for any $i \in [\![0, n[\![$, Q_i is a strict update of Q_{i+1}.

Definition 121. We say that a verification structure P is *inconsistent* if it is not satisfied and there exists no satisfied update of P.

Verifying that a verification structure is inconsistent is undecidable in the general case. However, it is still possible, in some particular cases, to detect that a given verification structure in inconsistent.

Notation 122. We assume given a correct but incomplete procedure that always terminates and that may detect that a verification structure is inconsistent. If this procedure detects that a given verification structure P is inconsistent, we will write $P \longrightarrow \mathfrak{fails}$.

Remark 123. Note that $P \longrightarrow \mathfrak{fails}$ entails that P is inconsistent but that the converse does not hold.

Definition 124. Let $P = (\Phi, \Upsilon)$ be a verification structure, υ be a verification condition generator of P and let $\phi = \upsilon(\Phi)$. Let i be the index of a guide formula in Φ. We call *verification update inverter on* i, and denote by $\upsilon^{-1}|_i$ any partial function such that for any formula I, one of the following is verified:

- $\upsilon^{-1}|_i(I)$ does not exist.
- We have $\upsilon(\text{Guides}(P[i \longleftarrow \text{Guide}_i(P) \wedge \upsilon^{-1}|_i(I)]) = I \wedge \phi$.

Definition 124 will be used to translate the solution of the abduction problems into corresponding updates for the verification structure. It is clear that if we want the algorithm to work, we will need verification update inverters whose domain contains the abduction solutions we generate on our verification conditions.

5.2 Using implicate generation to solve abstract verification problems

We now have all the background required to present our strengthening algorithm. It is shown in Algorithm 5.2.

Algorithm 5.2: ILINVA(P)

```
1  if (⊨ P) then
2      return P ;
3  if (P ⟶ fails) then
4      return failure ;
5  let j ∈ pending(P) ;
6  let i such that VCGen_j(P) ∈ 𝔇^i(P) ;
7  let Ξ ← ILINVAABDUCE(P, j, i) ;
8  foreach ξ ∈ Ξ do
9      let P_ξ ← P[i ⟵ Guide_i(P) ∧ ξ] ;
10     P_ξ ← ILINVA(P_ξ) ;
11     if P_ξ ≠ failure then
12         return P_ξ ;
13 return failure;
```

Note that we delegate the computation of the possible strengthenings to an abduction algorithm Line 7. This abduction algorithm is presented in Algorithm 5.3. It starts by recovering the verification condition it should perform abduction on (Line 1). Assuming it can obtain a relevant set of abducible literals (we will detail methods to do that in Chapters 6.4 and 8) and a filtering predicate if necessary, it will then invoke the IMPGEN-PID algorithm to compute the prime implicates of the formula (Line 4). Once obtained, we can update this initial result set of implicants, typically by constructing disjunctions of the ones we recovered from the call to IMPGEN-PID. We present this Line 5 in the form of a call to a procedure UPDATEIMPLICANTS left unspecified. It is assumed to update the initial set of implicants by either removing some it deemed irrelevant or by adding new ones it is able to construct (such as disjunctions of the previous ones). Finally, as what we currently have is a set of implicants for the formula obtained by applying a verification condition generator, we need to translate these implicants into valid strengthenings for the selected guide formula. This is what we do Line 6. We will generally assume that we have selected our abducible literals such that we can ensure that we have $\Xi \subset \text{DOM}(\upsilon^{-1}|_i)$.

Algorithm 5.3: ILINVAABDUCE(P, j, i)

```
1  let φ ⟵ VCGen_j(P)(Guides(P)) ;
2  let 𝒜 a set of abducible literals for φ ;
3  let 𝒫 a literal set predicate filter for φ ;
4  let Ξ ⟵ {Ī | I ∈ IMPGEN-PID(φ̄, ∅, 𝒜, 𝒫)} ;
5  UPDATEIMPLICANTS(Ξ) ;
6  return {υ^{-1}|_i(ξ) | ξ ∈ Ξ ∩ DOM(υ^{-1}|_i)} ;
```

Theorem 125. *Let P be a verification structure. If* ILINVA(P) *terminates and if we have* ILINVA(P) $\neq$ `failure`*, then* ILINVA(P) $\sim P$ *and* ILINVA(P) *is satisfied.*

Proof. The proof is by induction on the recursive calls. It is clear that $\textsc{Ilinva}(P) \sim P$ because the algorithm only modifies guide formulas. By construction (Line 1 of Algorithm 5.2), the verification structure returned must be satisfied. Similarly, when returning at Line 4 or at Line 13, the result is always a failure by construction. By induction, we have that any verification structure returned at Line 12 is necessarily satisfied. □

Theorem 126. *Let P be a verification structure and let $\mathcal{T}$ be a decidable theory. Let $\mathfrak{A}$ be a finite set of literals expressible in $\mathcal{T}$. Assume that the verification conditions of P and updates of P are in $\mathcal{T}$. Assume also that for every such verification condition, the associated set of abducible literals $\mathcal{A}$ (the one we recover at Line 2 of Algorithm 5.3) is such that $\mathcal{A} \subseteq \mathfrak{A}$, and such that the conjunction of any conjunction of its literals with the verification condition is in $\mathcal{T}$. Then $\textsc{Ilinva}(P)$ terminates.*

Proof. Termination can be derived from the fact that there are only finitely many possible guide formulas built on $\mathfrak{A}$. At each recursive call, the verification structure is strictly strengthened, as one of the guide formulas is strictly updated and the other ones are left unchanged. Hence, the multiset of guide formulas is strictly decreasing, according to the multiset extension of the update relation. □

Remark 127. We point out that, even though Theorem 126 assumes that $\mathcal{T}$ is decidable, the result holds as long as we can decide the satisfiability of the verification conditions we generate. If this is not the case, we will see in Chapter 7 that we can tweak the computation of satisfiability tests to ensure termination, although we may miss some possible solutions when doing so.

This concludes the presentation of the generic Ilinva algorithm to satisfy verification structures by strengthening them. It can be used as is to generate, *e.g.*, additional preconditions for the verification of programs (see again Example 115). In the next chapter, we will present an instantiation of the verification structures that will allow us to use this algorithm to generate property-directed loop invariants of programs.

Chapter 6

An instance of the framework to generate loop invariants of programs

In Chapter 5, we devised an algorithm to strengthen unsatisfied verification structures until they are satisfied. We saw (through Example 115), that we could use it to generate additional preconditions for the verification of programs. In the present chapter, we are going to develop an instance of verification structures that allow to use the ILINVA algorithm for the generation of property-directed loop invariants. The problem tackled is thus similar to that of [57] but the genericity of the abduction algorithm used within ILINVA (the one we presented in Chapter 3.2) allows for the generation of loop invariants in theories that are out of the scope of [57]. A more detailed comparison with this approach will be proposed in Section 6.4.

6.1 Representing the programs

In order to define our problem formally, we will consider the class of programs we detail in Definition 128. It is clear that all the work in this chapter could be adapted to other definitions of programs, as long as we can obtain the associated verification condition generators we detail later. We will also assume, to stick to the framework of Chapter 5 and for clarity reasons, that the logical formulas in which we express, within the programs, properties on its memory (such as preconditions, postconditions, invariants) are expressed within the same signature and theory as the ones the abduction algorithm will handle. If this were not the case, one could tweak the definition of the verification conditions generators and their inverters to perform the required translation.

Definition 128. We define the programs we consider by induction as follows :

$$
\begin{aligned}
P ::= & \texttt{empty} \\
& \mid I; P \\
I ::= & \langle \texttt{base-instruction} \rangle \\
& \mid \texttt{assume } [\phi] \\
& \mid \texttt{assert } [\phi] \\
& \mid \texttt{if } C \texttt{ then } P \texttt{ else } P \texttt{ end} \\
& \mid \texttt{while } C \texttt{ do } [\phi] \ P \texttt{ done}
\end{aligned}
$$

where C is a condition on the state of the memory, ϕ is a formula and I is an instruction.

Notation 129. We will denote by $P_1 \cdot P_2$ the concatenation of two programs P_1 and P_2. That is, the program containing all the instructions of P_1 followed by all the instructions of P_2.

The semantics of program sequences and conditionals are standard. Instructions are executed in the order in which they appear in the program, within a conditional, the first subprogram is executed if and only if the condition is satisfied and the second subprogram if and only if the condition is not. The semantics for assumptions (`assume` $[\phi]$) corresponds to formulas that are taken as hypotheses, and thus assumed to hold where they appear within a program. They are typically used to specify preconditions. Assertions (`assert` $[\phi]$) correspond to formulas that are to be proved. We say informally that a program is verified if all the assertions it contains hold in all executions of the programs that fulfills the assumptions. Base instructions are left unspecified, they shall depend on the target language and application domain; they may include, for instance, assignments and pointer redirections. The semantics of the loop instruction is also standard: the subprogram is executed if and only if the initial condition is verified when the program attains the loop instruction and, when its execution is complete, it jumps back to the initial condition of loop and executes the subprogram again as long as the condition is verified. It jumps after the loop whenever its condition is not verified any longer.

Definition 130. Given a loop instruction `while` C `do` $[\phi]$ P `done`, we call the formula ϕ a *candidate loop invariant.*

Given a loop instruction `while` C `do` $[\phi]$ P `done`, we will say that its candidate loop invariant is a *loop invariant* if it holds every time the condition C is tested during any execution of the program.

Example 131. Let us consider the sample program presented in Algorithm 6.1, expressed within the framework of Definition 128.

This algorithm is standard for computing, given two integers n, m such that $m \geq 0$ (expressed within an assumption Line 1), the quantity n^m. The postcondition at Line 8 is meant to guarantee this is indeed the case. It contains base instructions at Lines 2, 3, 6, 7 and 9, and a loop at Line 4. The current candidate loop invariant of the loop is $\top$ (Line 4), we will see later that it does not permit to deduce the assertion of the program. We will delegate the generation of verification conditions and the satisfiability tests required to check their validity to external tools. The method we design is an

Algorithm 6.1: $SampleExp(n, m)$

```
1 assume [m ≥ 0] ;
2 let p ⟵ 1 ;
3 let w ⟵ 0 ;
4 while w < m do
5 |  [⊤]
6 |  p ⟵ p × n ;
7 |  w ⟵ w + 1 ;
8 assert [p = n^m] ;
9 return p ;
```

algorithm that updates the candidate loop invariant to allow those tools to produce a proof of the program's assertions.

Definition 132. Given a program P, we define $rmLoops(P)$ by the program P in which we have removed all the loop instructions.

Notation 133. In the following, we will sometime represent sequences of integers as separated by a dot (such as $1.2.0 = \langle 1, 2, 0\rangle$). We will also extend this notation for sets of sequences, which means that for any integer n and any two sets of sequences $\mathcal{L}_1, \mathcal{L}_2$ we will write $n.\mathcal{L}_1 = \{n.n' \mid n' \in \mathcal{L}_1\}$, $\mathcal{L}_1.n = \{n'.n \mid n' \in \mathcal{L}_1\}$ and also $\mathcal{L}_1.\mathcal{L}_2 = \{n_1.n_2 \mid n_1 \in \mathcal{L}_1, n_2 \in \mathcal{L}_2\}$.

As we want to express the problem of property-directed loop invariant generation in the framework of Definition 112, it is fairly intuitive to use the candidate loop invariants as guide formulas. What we need now is a generic method to construct verification conditions generators that can generate conditions ensuring that all the properties we want our programs to have hold. In the technical system (see Chapter 8), this will be delegated to an external tool, for which we will only require that the validity of the generated verification conditions guarantees all the properties of the program hold. For self-completeness, we shall now briefly review a well-known algorithm for generating verification conditions. It is based on calculi for computing weakest preconditions and strongest postconditions of logical formulas.

The first element we need is a formal way to locate a specific instruction within a program. This is the purpose of the following definition:

Definition 134. We define a *location* within a program or an instruction. We define $locs(P)$ the *set of locations within a program* P inductively as follows:

$$\begin{aligned}
\texttt{empty} &\mapsto \{0\} \\
I; P_1 &\mapsto \{0\} \cup 0.locs(I) \cup \{(i+1).\ell \mid i.\ell \in locs(P_1)\} \\
\langle \texttt{base-instruction} \rangle &\mapsto \emptyset \\
\texttt{assume } [\phi] &\mapsto \emptyset \\
\texttt{assert } [\phi] &\mapsto \emptyset \\
\texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} &\mapsto 1.locs(P_1) \cup 2.locs(P_2) \\
\texttt{while } C \texttt{ do } [\phi] \ P_1 \texttt{ done} &\mapsto 1.locs(P_1)
\end{aligned}$$

Definition 135. In the following, we will identify an instruction within a program by the location preceding it. For a given program P and a given location ℓ within P, we will denote the instruction appearing directly after ℓ in P by $P|_\ell$ and define it inductively as follows:

$$
\begin{aligned}
I; P_1|_0 &\mapsto I \\
I; P_1|_{0.\ell} &\mapsto I|_\ell \\
I; P_1|_{(i+1).\ell} &\mapsto P_1|_{i.\ell} \text{ if } P_1 \neq \texttt{empty} \\
\texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}|_{1.\ell} &\mapsto P_1|_\ell \text{ if } P_1 \neq \texttt{empty} \\
\texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end}|_{2.\ell} &\mapsto P_2|_\ell \text{ if } P_2 \neq \texttt{empty} \\
\texttt{while } C \texttt{ do } [\phi] \ P_1 \texttt{ done}|_{1.\ell} &\mapsto P_1|_\ell \text{ if } P_1 \neq \texttt{empty}
\end{aligned}
$$

Remark 136. Note that, given a program P, $P|$ is a partial function, since locations denoting the end of a sequence do not correspond to any instruction.

Notation 137. In all the following, we will order program locations as we would lexicographically order sequences of integers. For two given locations ℓ_1, ℓ_2, we will denote $\ell_1 < \ell_2$ if ℓ_1 is lower than ℓ_2, and $\ell_1 \leq \ell_2$ if it is lower or equal.

Definition 138. Given a program P and two locations ℓ, ℓ' in P, we define $P_{[\ell:\ell']}$ the *subprogram of P between ℓ and ℓ'* inductively as follows:

$$
\begin{array}{rcll}
P_{[\ell:\ell]} & = & \texttt{empty} & \\
P_{[\ell:\ell'.(i+1)]} & = & P_{[\ell:\ell'.i]} \cdot P|_{\ell'.i} & \text{if } \ell \leq \ell'.i \\
P_{[\ell:\ell'.0]} & = & P_{[\ell:\ell']} & \text{if } \ell \leq \ell' \\
P_{[\ell.i.\ell':\ell.(i+1)]} & = & P_{[\ell.i.\ell':\ell.i.m]} & m = \max(\{j \mid \ell.i.j \in \mathit{locs}(P)\})
\end{array}
$$

6.2 A standard method for verifying assertions within programs

In all the following, we will associate to a given program P a verification structure (Φ, Υ), where Φ are the candidate loop invariants of the program, in the order they appear in (according to Notation 137). Υ will be a set of verification condition generators generating all the verification conditions for the program at hand. We describe below how we construct this set from a source program.

First, we distinguish three types of verification conditions for a program:

1. The *assertion conditions*, which check that the assertion formulas hold at the corresponding locations in the program. These conditions may also include so-called property verifiers to ensure usual verifications not explicitly expressed within the program (such as divisions by zero, bounds of array accesses, *etc*). Given a program P, we will denote those by $\mathfrak{V}_{assert}(P)$.

2. The *propagation conditions*, which check that the candidate loop invariants within the program do propagate. Given a program P, we will denote those by $\mathfrak{V}_{ind}(P)$ and, given a loop $\texttt{L} \in P$, we will denote the propagation conditions for $\texttt{L}$ by $\mathfrak{V}^{\texttt{L}}_{ind}(P)$.

3. The *loop preconditions*, which check that the candidate loop invariants hold when the program first enters the loop they appear in. Given a program P, we will denote those by $\mathfrak{V}_{init}(P)$ and, given a loop $\texttt{L} \in P$, we will denote the propagation conditions for $\texttt{L}$ by $\mathfrak{V}^{\texttt{L}}_{init}(P)$.

Given a verification structure P representing a program, we assume that $VC(P) = \mathfrak{V}_{assert}(P) \cup \mathfrak{V}_{ind}(P) \cup \mathfrak{V}_{init}(P)$. Moreover, we expect that the pairwise intersections of $\mathfrak{V}_{assert}(P)$, $\mathfrak{V}_{ind}(P)$ and $\mathfrak{V}_{init}(P)$ are all empty.

A classical way to generate such verification conditions from a program is to use a weakest precondition calculus and/or a strongest postcondition calculus, such as the ones presented in Definitions 139 and 140.

Definition 139. We define the *weakest precondition of the formula ψ relatively to the program P*, and denote by $wp(\psi, P)$, the formula inductively defined as follows:

$$\begin{aligned}
\psi, \texttt{empty} &\mapsto \psi \\
\psi, I; P_1 &\mapsto wp(wp(\psi, P_1), I) \\
\psi, \langle \texttt{base-instruction} \rangle &\mapsto \psi' \text{depending on the specific instruction} \\
\psi, \texttt{assume } [\phi] &\mapsto \phi \Rightarrow \psi \\
\psi, \texttt{assert } [\phi] &\mapsto \phi \wedge \psi \\
\psi, \texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} &\mapsto C \Rightarrow wp(\psi, P_1) \wedge \neg C \Rightarrow wp(\psi, P_2) \\
\psi, \texttt{while } C \texttt{ do } [\phi] \ P_1 \texttt{ done} &\mapsto \phi \wedge \forall x.(\phi \Rightarrow wp(\psi, P_1)) \wedge \forall x.(\phi \wedge \neg C \Rightarrow \psi)
\end{aligned}$$

Definition 140. Similarly, we define a *strongest postcondition of the formula ψ relatively to the program P* and denote by $sp(\phi, P)$, the formula inductively defined by:

$$\begin{aligned}
\psi, \texttt{empty} &\mapsto \psi \\
\psi, I; P_1 &\mapsto sp(sp(\psi, I), P_1) \\
\psi, \langle \texttt{base-instruction} \rangle &\mapsto \psi' \text{depending on the specific instruction} \\
\psi, \texttt{assume } [\phi] &\mapsto \phi \wedge \psi \\
\psi, \texttt{assert } [\phi] &\mapsto \psi \\
\psi, \texttt{if } C \texttt{ then } P_1 \texttt{ else } P_2 \texttt{ end} &\mapsto sp(\psi \wedge C, P_1) \vee sp(\psi \wedge \neg C, P_2) \\
\psi, \texttt{while } C \texttt{ do } [\phi] \ P_1 \texttt{ done} &\mapsto \phi \wedge \neg C
\end{aligned}$$

Remark 141. The reader can observe that, contrarily to the weakest precondition calculus of Definition 139, the strongest postcondition calculus of Definition 140 does not contain the loop verification conditions.

Remark 142. For simplicity reasons, we assumed that C is expressed in the same logic as the one we use to express the formula within the program (such as loop invariants). In practice, it may not be true and, in this case, one should start by translating those conditions into the logic the formulas are expressed in beforehand.

6.3 A loop invariant generation framework

Given a program, we can now define a verification structure to represent the problem we tackle.

Definition 143. Given a program P, we define (Φ, Υ) the *verification structure representing* P as follows.

- Φ is the sequence of candidate loop invariants in P, in their order of appearance.
- For any location $\ell \in P$ such that $P|_\ell$ is of the form `assert` $[\phi]$,

$$v^\ell_{assert} : \varphi_1, \ldots, \varphi_n \mapsto sp(\top, P[* \longleftarrow \varphi_1, \ldots, \varphi_n]_{[0:\ell]}) \Rightarrow \phi$$

  belongs to Υ. It is the verification condition generator ensuring that the assertion at location ℓ is verified. Remark that this verification condition generator depends on all the candidate loop invariant locations of $P_{[0:\ell]}$.
- For any location $\ell \in P$ such that $P|_\ell$ is of the form `while` C `do` $[\phi]$ P' `done`,

$$v^\ell_{init} : \varphi_1, \ldots, \varphi_n \mapsto sp(\top, P[* \longleftarrow \varphi_1, \ldots, \varphi_n]_{[0:\ell]}) \Rightarrow \varphi_i$$

  belongs to Υ, where φ_i is such that $P[* \longleftarrow \varphi_1, \ldots, \varphi_n]|_\ell$ is the instruction

$$\texttt{while } C \texttt{ do } [\varphi_i] \ P'^* \texttt{ done}.$$

  It is the verification condition generator ensuring that the candidate loop invariant of the loop at location ℓ is verified whenever the loop is entered. Note that this verification condition generator depends on all the candidate loop invariants of $P_{[0:\ell]}$ as well as on the candidate loop invariant of $P|_\ell$.
- For any location $\ell \in P$ such that $P|_\ell$ is of the form `while` C `do` $[\phi]$ P' `done`,

$$v^\ell_{ind} : \varphi_1, \ldots, \varphi_n \mapsto \varphi_i \Rightarrow wp(\varphi_i, rmLoops(P'^*))$$

  belongs to Υ, where P'^* and φ_i are such that $P[* \longleftarrow \varphi_1, \ldots, \varphi_n]|_\ell$ is the instruction

$$\texttt{while } C \texttt{ do } [\varphi_i] \ P'^* \texttt{ done}.$$

  It is the verification condition generator ensuring that the candidate loop invariant of the loop at location ℓ propagates. Note that this verification condition generator depends on all the candidate loop invariants of P' as well as on the candidate loop invariant of $P|_\ell$.
- Υ does not contain any other verification condition generator.

Definition 144. Given a verification structure (Φ, Υ) representing a program P as in Definition 143, we define verification update inverters for its verification condition generators as follows:

- Given the verification condition generator v^ℓ_{assert} of an assertion occurring in the program at the location ℓ, and given ℓ_i the location of a loop it depends on (assuming this loop is the i^{th} loop of the program), we define the associated verification update inverter ${v^\ell_{assert}}^{-1}|_i$ as follows:

$${v^\ell_{assert}}^{-1}|_i : I \mapsto wp(I, rmLoops(P_{[\ell_i:\ell]}))$$

- Given the initialization verification condition generator υ^{ℓ}_{init} of a loop occurring in the program at the location ℓ, and given ℓ_i the location of a loop it depends on (assuming this loop is the i^{th} loop of the program), we define the associated verification update inverter ${\upsilon^{\ell}_{init}}^{-1}|_i$ as follows:

$$ {\upsilon^{\ell}_{init}}^{-1}|_i : I \mapsto wp(I, rmLoops(P_{[\ell_i:\ell]})) $$

- Given the propagation verification condition generator υ^{ℓ}_{ind} of a loop occurring in the program at the location ℓ, and given ℓ_i the location of a loop it depends on (assuming this loop is the i^{th} loop of the program), we define the associated verification update inverter ${\upsilon^{\ell}_{ind}}^{-1}|_i$ as follows:

$$ {\upsilon^{\ell}_{ind}}^{-1}|_i : I \mapsto I $$

This gives us a verification structure able to represent a program. We can use the ILINVA algorithm presented in Chapter 5 to try and compute satisfying loop invariants for it.

Example 145. Let us consider the Sample Program 6.2. Using the definitions we pre-

Sample Program 6.2: *LoopCheck2*(y_0)

```
1 let x ← −50;
2 let y ← y_0;
3 while x < 0 do
4 |   invariant [ψ] ;
5 |   x ← x + y;
6 |   y ← y + 1;
7 assert y ≥ 0;
```

sented in this section, we would obtain for this program the following verification condition generators:

- $\upsilon^{2}_{init} : \phi \mapsto x = -50 \wedge y = y_0 \Rightarrow \phi$
- $\upsilon^{2}_{ind} : \phi \mapsto \phi \Rightarrow wp(wp(\phi, x \leftarrow x + y), y \leftarrow y + 1)$
- $\upsilon^{6}_{assert} : \phi \mapsto x = -50 \wedge y = y_0 \wedge \neg x < 0 \wedge \phi \Rightarrow y \geq 0$

Similarly, the associated verification update inverters will be defined as follows:

- ${\upsilon^{2}_{init}}^{-1}|_2 : I \mapsto I$
- ${\upsilon^{2}_{ind}}^{-1}|_2 : I \mapsto I$
- ${\upsilon^{6}_{assert}}^{-1}|_2 : I \mapsto I$

Lemma 146. *Consider a verification structure P representing a program in the sense of Definitions 128, 143 and 144. If P is satisfied then we have both that*

1. *the assertions of the program are verified and*

2. *all the loop invariants it contains are true when first checked and propagate.*

Proof. This is a consequence of the Definition 143. The fact that the verification structure is satisfied ensures that all its verification conditions are valid. The assertion verification conditions ensure that the assertions of the program are verified and the other ensure the validity of the loop invariants. □

Corollary 147. *Let P be a verification structure representing a program in the sense of Definitions 128 and 143, to which we associate the verification update inverters as defined in Definition 144. If* ILINVA(P) *terminates and if we have* ILINVA$(P) \neq$ `failure`*, then all the verification conditions of the program are valid. This means that all the assertions it contains hold and that all its candidate loop invariants are actual loop invariants.*

Proof. This is a direct consequence of Theorems 125, 126 and Lemma 146, applying the result we obtained for the general algorithm in this particular case. □

Even though this algorithm works in theory, it still contains a lot of imprecision for someone who would like to implement it. In particular, we did not detail how we select the abducible literals required for using IMPGEN-PID as the abduction algorithm within ILINVA. Likewise, we did not explain how we can obtain interesting filtering predicates for abduction from the program. Finally, we will see that additional considerations can be made regarding the use of IMPGEN-PID. Those issues will be discussed in Section 6.4. The readers interested in the practical implementation of the system can refer to Chapters 7 and 8. In order to use the ILINVA algorithm to generate loop invariants of programs in practice, we will delegate the verification condition generation tasks to the program verification platform WHY3 [75, 21]. It performs roughly the tasks let to the verification condition generator as defined in the present chapter, even though these conditions are tweaked to handle a larger class of programs and with additional considerations on, *e.g.*, the simplification of the verification condition generated. Still, it gives us the guarantee we expect on the program it handles

6.4 Parametrization of the algorithm

In Chapters 5 and 6, we devised an algorithm using the IMPGEN-PID abduction algorithm to compute loop invariants of programs. In this part, we describe the additional parametrization required to implement such a system. In particular, we discuss how we generate abducible literals for the IMPGEN-ILINVA abduction algorithm (see Line 2 of Algorithm 5.3), the order we use to test the solution it returns, and the method we use to select an unsatisfied verification condition to perform abduction on, and which candidate loop invariant to strengthen with the solutions.

6.4.1 Selecting verification conditions and candidate loop invariants

The first nondeterministic part of the ILINVA algorithm we need to discuss is the selection of the verification condition to strengthen, as well as the candidate loop invariants we

will strengthen using the solutions of the abduction algorithm (see Lines 5 and 6 of Algorithm 5.2). As explained in Section 6.2, we decompose the verification conditions of our program into three subsets: the set $\mathfrak{V}_{init}(P)$ of verification conditions ensuring that all the candidate loop invariants hold when the program first enters the loop they belong to, the set $\mathfrak{V}_{ind}(P)$ of verification conditions ensuring that all the candidate loop invariants propagate, and the set $\mathfrak{V}_{assert}(P)$ of verification conditions ensuring that we can prove the assertions of the program.

For simplicity reasons, we made the choice to never perform abduction on the verification conditions of $\mathfrak{V}_{init}(P)$. We will assume, even though this is not always true, that the verification structure associated with a program for which the set of verification conditions related to the loop invariants initializations contains at least one unsatisfied verification condition is inconsistent. This will permit to define a filtering predicate for the ImpGen-PiD algorithm that can be used if we do not wish to compute the disjunctions of the abduction solutions. Indeed, refusing to add a candidate to the current hypothesis during the execution of the ImpGen-PiD algorithm if the result happens not to satisfy one of the loop initializations verification conditions will allow us to remove exploration branches we are certain would only produce such candidate loop invariants.

We make the choice of always considering the verification conditions ensuring the propagation of the candidate loop invariants first. This means that whenever the set $\mathfrak{V}_{ind}(P)$ contains at least one non-valid formula, the verification condition we send to the abduction algorithm will be in it. We do this in order to ensure that the candidate invariants we strengthened when trying to prove the program's assertions are not too strong to propagate. This prevents the algorithm from exploring strengthening possibilities that would indeed prove other assertions within the program but would never allow us to obtain a complete proof of the program due to this failure to propagate.

This leaves us with the problem of deciding, within a set $\mathfrak{V}_{ind}(P)$ of verification conditions related to the propagation of a loop invariant, which one we should select. We decided to consider the verification conditions in the reverse order of the loops in the program. When we have at least two candidate loop invariants that do not propagate within the program, we will consider the location of the loops they belong to, and select the verification condition corresponding the maximum of those locations. Intuitively, this is the condition that is the closest to the loop we have to prove the propagation of, and thus the one the modification of which would introduce the least amount of data in the system.

We now have to select the candidate loop invariant we will try to strengthen with the strengthenings that the abduction algorithm will return. In the most simple case (that is, with no nested loops), the candidate loop invariant of a loop will always be the guide formula associated with the verification condition related to the propagation of itself. When the loop contains other loops, it is clear that the verification condition for the propagation of its candidate loop invariants is based on the value of the nested loops invariants. More work should be done to design relevant heuristics allowing to try to strengthen the invariants of those nested loops to obtain a propagation proof for the main loop. In this work however, we will stick to always selecting the candidate loop invariant of the loop from which the verification conditions was derived.

Concerning the selections of the verification condition for proving the assertions of program, we consider them in the order they appear in within the program. This means

that whenever the set $\mathfrak{V}_{assert}(P)$ contains more than one unsatisfied verification condition, we will always select the one derived from the assertion with a minimal location. We now have to select which candidate loop invariant to strengthen using the solutions of the abduction algorithm. It is easy to see that the verification condition of an assertion does not depend on the candidate loop invariants appearing after it within the program. Those will thus never be considered. This leaves us with the candidate loop invariants preceding the assertion. We make the choice of trying all these possible strengthening locations in their reverse order within the program, unless they belong to a loop nested in the path leading to the assertion. Obviously, if we can show that the verification condition did not depend on a given candidate loop invariant, we will not try to strengthen it with the abduction solution of the verification of the assertion.

Example 148. Let us consider the Sample Program 6.3. As this program contains two

Sample Program 6.3: *MultiLoopCheck*()

```
1  let x ← 0;
2  let y ← 0;
3  let n ← 0;
4  while random({⊤, ⊥)} do
5  |  invariant [?];
6  |  x ← x + 1;
7  |  y ← y + 1;
8  while x ≠ n do
9  |  invariant [?];
10 |  x ← x − 1;
11 |  y ← y − 1;
12 assert x = y;
13 assert y = n;
```

loops and two assertions (which are, in this case, equivalent), we will have to decide which ones we consider first. Applying the rules we suggested in the present section, we will first try to find a strengthening allowing to prove the assertion at Line 12, reinforcing first the invariant at Line 9, then the one at Line 5 if nothing is found. If we reach a state in which one of the candidate loop invariants does not propagate, we will immediately try to strengthen this candidate with one of the non-valid verification condition built from this propagation requirement. If none of the candidates propagate, we will start by reinforcing the one at Line 9.

6.4.2 Selecting abducible literals

We can now tackle the construction of the abducible literals for the ImpGen-PiD algorithm. This will be done using the following method, of which all steps will be detailed below. We start by extracting from the program a set of *core abducible elements*. Then, for each verification condition forwarded to the abduction algorithm, we will remove from this initial set of core abducible elements the ones that are unrelated to the current

verification condition (*e.g.* those containing variables undefined in the context of the verification condition). The remaining set will be forwarded to the ImpGen-PiD algorithm alongside the unsatisfied verification condition.

Using the program, it is clear that we can obtain relevant abducible literals from the identifiers (variables, functions, *etc*) it contains. This is what we use to build our set of core abducible elements. We will consider all the literals built upon the set of symbols occurring in the program up to the maximal depth of the terms occurring in the program. We can also allow the use of additional functions or predicates defined by the user to help the verification of the program. In this case, those will also be used to generate the set of core abducible elements.

Example 149. Going back to the algorithm of Example 131 (Algorithm 6.1), we see that it contains four variable identifiers (n, m, p, w), two constant integers $(0, 1)$, three binary functions $(\times, +, \square^{\square})$. and three binary predicates $(\geq, <, =)$. All the functions of the algorithm are only applied to variables, which is also the case of the two comparison predicates. The equality predicate will not be applied for two parameter that are results of the power function. This means that our set of core abducible elements will contain all the possible comparisons between constants and variable identifiers, as well as the equalities between variables, constants, and in one side results of the application of the power function.

As such, the set of core abducible elements contains elements expressed within the framework of the program, that do not relate to any of the verification conditions. Those cannot be forwarded to the abduction algorithm in this state, which is why we now present a transformation method to adapt this set when given a verification condition. Given a verification condition, we first remove from the set of core abducible elements the elements that are unrelated to the verification condition. Those are the elements containing variables or functions not appearing in the context of the verification condition, or undefined at the location of the candidate loop invariant. When this is done, we use the same method as the one we used for the translation of the verification condition (see again Definition 143) to obtain the corresponding logical formula of the core abducibles at the location of the verification condition we consider. This is the set of abducible literals we will forward to the ImpGen-PiD algorithm.

It is clear that additional work should be done to obtain smaller and more relevant sets of abducible literals. Indeed, even if we removed all the elements clearly unrelated to the verification conditions we process, most of the resulting literals are not particularly relevant for the proof, because they were built using only low-level information and do not encode any higher-level information on the program. We also refer the reader to Chapter 9 for the effects this has on the practical efficiency of the algorithm.

6.4.3 Parametrizing the ImpGen-PiD algorithm and the generation of abduction solutions

Even though it is stated that in Algorithm 5.3 we recover all the solutions from ImpGen-PiD before using them to strengthen our candidate loop invariants, it is natural to assume that any implementation of the algorithm will generate these solutions one by one, at the moment a new candidate is required. Such an implementation will guarantee that the

only solutions that are generated are the ones that will actually be tested as candidate loop invariant strengthenings. This suggests that it will be important to generate the best possible strengthenings in the least amount of time.

The first element to consider to obtain this result is the use of the ImpGen-PiD algorithm. This algorithm explores the hypotheses it can construct from its set of abducible literals depth first. In particular, this implies that when this set is large, it may spend a lot of time exploring and trying hypothesis within a branch where all the hypotheses will end up being satisfiable before trying other possible solutions that may be more relevant. We will see in Chapter 9 that this behaviour can severely degrade the performance of the system. Moreover, as the Ilinva algorithm already uses a recursive method to strengthen the same candidate several times if necessary, it appears that we are not necessarily interested in obtaining deep (*i.e.*, containing many literals) solutions, but rather in obtaining simpler ones, sufficient enough to obtain a proof. These considerations suggest that it may be useful to tweak the ImpGen-PiD algorithm slightly to simulate a breadth-first search. The method described below is recent work, and it is clear that more work should be done to devise a more integrated and efficient breadth-first version of the ImpGen-PiD algorithm.

We proceed using an iterative deepening strategy, that is, by calling the ImpGen-PiD algorithm multiple time with additional size filtering predicates. Given a maximal limitation size n, we will call successively the ImpGen-PiD algorithm n times with a filtering predicate limiting the size of the solutions to the current number of times the algorithm was called. We will also call the algorithm without those restrictions one last time, to be certain not to miss any solution. This ensures that we will generate small solutions first, even though it also implies that we are doing redundant work.

The second thing we have to consider is the behaviour of the UpdateImplicants method in Algorithm 5.3. The way we decide to use it within this work is to construct disjunctions of the previously obtained solutions. Assuming that no strengthening leading to a complete proof of the program was found using the implicants returned by the ImpGen-PiD algorithm, it is possible to construct weaker implicants, by building disjunctions of those first ones. Still, it is clear that the number of such implicants can be extremely large, even if we limit the number of disjunctions in a given implicant.

6.4.4 Differences with the work of Dillig *et al.* [57]

The algorithm proposed by Dillig *et al.* in [57] is similar to ours and also uses abduction to strengthen candidate invariants so that verification conditions are satisfied. The algorithms differ by the way the verification conditions and abductive hypotheses are proceeded. In our approach, the conditions always propagate forward from an invariant to another along execution paths, and we eagerly ensure that all the loop invariants are inductive. In their work, Dillig *et al.* allow for a more general selection structure, which may result in a larger search space, though they also additionally use a strongest postcondition calculus to instantiate their candidate loop invariants, which helps counterbalancing this effect. Another difference is that we use a completely different technique to perform abductive reasoning: the one in [57] is based on model construction and quantifier elimination for Presburger arithmetic, whereas our approach uses a generic

algorithm, assuming only the existence of a decision procedure for the underlying theory. This permits to generate invariants expressed in theories that are out of the scope of [57].

6.4.5 Limitations

After all these considerations, we observe that the ILINVA algorithm used for the generation of loop invariants of programs still suffers some limitations. The first issue we notice comes from the construction of the set of abducible literals. As it is extracted strictly from the elements of the program (or additional user-provided elements). This means that if the candidate loop invariants can only be expressed using more complex functions or predicates, or expressed more clearly using those, we will not be able to find such solutions. Secondly, as mentioned before, the way we handle the loop initialization verification conditions may prevent us from generating a complete proof of a program if it can only be obtained by strengthening candidate loop invariants of other loops afterwards. Thirdly, the tweak we operate for the use of the IMPGEN-PID algorithm makes it perform some redundant work, notably redundant satisfiability tests, which is notably costly. We also note that we generate many abduction solutions that can appear as irrelevant to a human user. Finally, the algorithm uses successive strengthenings of candidate invariants to generate the proof that the verification conditions of the system are valid. This implies that if the program has no unsatisfied verification condition that can be verified by strengthening one and only one candidate loop invariant, then we will not be able to generate a solution for proving all its verification condition, even though such a solution may exist.

We will see in Chapter 9 that those limitations strongly influence the efficiency of the algorithm. However, we will also see that the fairly simple heuristics we chose still permit to generate solution for a wide range of programs, although not always in reasonable time. This suggest that this solution is promising and that additional work should be initiated to tackle the issues we mentioned.

Part III

Implementation and evaluation of the abduction algorithm and of its application

Chapter 7

An infrastructure that transcribes the genericity of the algorithms

In Part I, we designed an algorithm to compute the prime $(\mathcal{T}, \mathcal{A})$-implicates of a formula and, in Part II, we used this algorithm to design a generic loop invariant generator. In order to evaluate and test these two algorithms, we present in this part a C++ framework implementing both. This system is called ABDULOT for abduction modulo theories. This chapter describes the design, development and basic usage of the system. We will discuss in Chapter 8 the details related to the implementation of concrete interfaces for SMT-solvers and program provers, and we will evaluate the framework experimentally in Chapter 9. For a detailed user manual, the reader is referred to Appendix C.

As described in Chapter 3, we have designed an abduction algorithm that works modulo theories. It is independent of the theory, as long as we can provide a decision procedure for testing the satisfiability of formulas in this theory. This genericity being the main goal of this work, we implemented a system that is also generic. This means that it does not depend on any specific satisfiability decision algorithm, but rather exposes a generic interface for formula satisfiability checkers in which we will plug any SMT-solver that can handle the problems we consider.

Similarly, the strengthening algorithm is not inherently dependent on the tool it uses to detect strengthening locations and ensure the global verification of the program. This is why the system will also be generic with respect to this tool, and will communicate with it via a standardized interface.

7.1 Generic interfaces for SMT-solvers

We start by describing the interface we use to communicate with SMT-solvers in our system. We recall that we assume an SMT-solver to always return an element of $\mathfrak{B} = \{\texttt{Sat}, \texttt{Unsat}, \texttt{Unknown}\}$, and possibly a model when it returns `Sat`. In practice, it is possible that the SMT-solver, being a technical system, fails more significantly when trying to compute a formula and ends up returning an error. We will handle those errors as if they were `Unknown` results, as it is the only information we can extract from them. It is also possible for the SMT-solver not to terminate on a given formula. To ensure the termination of the main algorithm in those cases, it is possible to restrict the computation time of any SMT-solver query with a timeout. In this case, and if the SMT-solver has not

returned any answer before the timeout ends, we will also interpret it as the SMT-solver returning `Unknown`.

The issue that arises when we allow the SMT-solver to return `Unknown` is that the ImpGen-PiD algorithm is not conceived to handle such answers. This justifies the following assumption:

In all the following, we will assume that it is possible to project the answers in $\mathfrak{B}$ given by SMT-solvers into $\{\texttt{Sat}, \texttt{Unsat}\}$. We will use such projections to change the value of `Unknown` results to either `Sat` or `Unsat` depending on our needs. The most natural solution being to interpret `Unknown` answers as if they were `Sat` answers. Indeed, the ImpGen-PiD algorithm searches for implicates by deriving contradictions between the initial formula and a set of hypotheses it is constructing. If the SMT-solver tasked with the verification of those contradictions is unable to answer, we cannot ensure that the hypothesis it tried is a contradiction. In this case, rather that stopping the algorithm, we will handle it as if it was not a contradiction. It is clear that, in this case, we may miss actual implicates of the initial formulas. This choice ensures, however, that the algorithm is correct at the expense of completeness.

Now that the issue of `Unknown` results has been handled, we can tackle the presentation of the generic SMT-solver interface we designed in order to make the system independent of the SMT-solver. If one wants to plug an SMT-solver to our abduction algorithm, in order to generate implicates in a theory of his or her choice for instance, the only work that will be required is to write the corresponding wrapper to make the use of the SMT-solver match the interface. We present this interface in Interface 7.1.

Interface 7.1: SMT-solver Interface Template

```
1  interface Constraint contains
2      str : void ⟶ string ;
3  interface Model contains
4      implies : Constraint ⟶ {⊤, ⊥} ;
5  interface Solver contains
6      getModel : void ⟶ Model (only if the last check call returned Sat);
7      addConstraint : Constraint ⟶ void ;
8      push : void ⟶ void ;
9      pop : void ⟶ void ;
10     check : void ⟶ 𝔅 ;
```

An SMT-solver interface contains a constraint wrapper, which represents the constraints that can be added to the set of formulas the SMT-solver should handle. We require that such constraints can be converted to C++ strings in order to print them when generating implicates or related results (such as implicants). We also expect the interface to wrap its models (when they exist) to allow the system to check whether a given constraint is validated by this model. The actual interface for the SMT-solver contains the usual methods to add constraints to the formula set, `push` and `pop` functions to stack constraints on multiple levels (necessary for solving incremental problems without recreating a new formula from scratch for each test) and a method to check the satisfiability of the constraints, and recover a model if possible. An additional class (usually

called a manager) can also be sent to the solver via the interface if necessary, though this has not been explicitly expressed in Interface 7.1. Such classes are usually required by SMT-solver APIs to allocate, construct and manage the formulas.

One may also have to devise a specific problem loader, which will handle the loading of a problem file and its conversion into constraints in the format handled by the interface defined above. The generic interface for problem loaders is described in Interface 7.2. It contains code to load problem files for the solver used and will perform a step-by-step constraint forwarding. This means that it works by being asked whether it contains at least one more constraint before being called to return this constraint.

Interface 7.2: SMT-solver Problem Loader Template

```
1 interface ProblemLoader contains
2 |  load : (filename : string), (opts : string) ⟶ void ;
3 |  hasConstraint : void ⟶ {⊤, ⊥} ;
4 |  nextConstraint : void ⟶ Constraint (only valid if the last access to the
  |    interface was a call to hasConstraint that returned ⊤);
```

Similarly, one can also devise a specific abducible literal loader. As we presented it in Chapter 3, the ImpGen-PiD algorithm computes $(\mathcal{T}, \mathcal{A})$-implicates of a formula, that is, the $\mathcal{T}$-implicates it can construct from the literals contained in $\mathcal{A}$. This means that, in the implementation, it is necessary to compute the set $\mathcal{A}$ that is fed to to the ImpGen-PiD algorithm. This is done by adding a component to the system that will recover information about those abducible literals and will generate constraints (in the sense of Interface 7.1) corresponding to the actual literals that can be selected to build hypotheses during the generation of $(\mathcal{T}, \mathcal{A})$-implicates by the algorithm. The generic interface for abducible literal generators is described in Interface 7.3. It provides a method to load abducible literals from a file (technically in any format, even if we provide a specific one to represent them — see also Section 7.4). It should also provide a method for automatically generating abducible literals from the problem. Finally, after loading or generating abducible literals, the interface is able to return the total number of literals it obtained and to return them in an array. The finite set of abducible literals is then stored in memory and the literals referred by their index in the corresponding table. This table will be accessed when we need to use the value of the literals (such as when performing satisfiability tests). We will additionally assume that the order we should use on abducible literals is the one induced by their indices in this context. If the literals have been obtained by loading an input file, this order will correspond to the one they appeared in within the file. If the literals have been generated from the problem, then this order will be set by the generators; It is generally induced by the order in which the literals were found within the problem but can be set differently by modifying the generator.

The Abdulot framework comes with already implemented interfaces for Alt-Ergo [39] (via its SMT-lib interface only), CVC4 [7] (via SMT-lib [9] and via its C++ API), MiniSAT [70] (using its Dimacs [60] format and its C++ API), VeriT [24] (via its SMT-lib interface) and Z3 [50] (via SMT-lib and via its C++ API). The framework also provides a generic interface for SMT-lib-compliant [9] solvers, which can be easily

Interface 7.3: SMT-solver Literal Generator Template

```
1 interface LiteralGenerator contains
2 |  load : (filename : string) ⟶ void ;
3 |  generate : (generatorId : string) ⟶ void ;
4 |  count : void ⟶ ℕ ;
5 |  getMapper : void ⟶ {cid ∈ ℕ ↦ Constraint} ;
```

parametrized with the executable of the SMT-solver. All of the interfaces provided are used to generate test executables for the implementation of both the ImpGen-PiD algorithm and the Ilinva algorithm.

7.2 Implementation of ImpGen-PiD

In this section, we will describe the implementation of the ImpGen-PiD algorithm. This implementation will be called GPiD and is schematically represented in Figure 7.1. It is decomposed into three components: a hypothesis engine, with the task of selecting which abducible literals should be added to the hypothesis constructed to contradict the original problem, the interface of the SMT-solver, that will be used to test the satisfiability of the queries and conclude whether any hypothesis it is given is an implicate of the source problem or not, and an algorithmic engine (called *implicate generator* in Figure 7.1), which will manage the other two and handle the global execution of the ImpGen-PiD algorithm. All those components are packaged in a wrapping interface that is available within the library and also compiled as executables.

We consider the GPiD implicate generation system as a process taking as input some problem data, exporting implicates and possibly communicating with external hypothesis checkers. The inputs it requires are a set of logical formulas (compatible with the theories handled by the SMT-solver it was configured with) and a method to generate the abducible literals it should work with (see 7.4). We offer two methods to inject those in the system: one using files as well as a command line interface to tell the executable in which files these inputs can be found; and another using the C++ API directly. The input data is then split to send the formula to the implicate generator and the abducible literals to the hypothesis engine, where they will be ordered according to the order they were sent in, and then selected (by following the refinements we presented in Chapters 3 and 4) in order to build possible implicates. Similarly, the source problem is forwarded to the formula manager of the implicate generator to configure the tasks that will be delegated to the SMT-solver for finding its implicates. The source formula can also be used by the implicate generator to implement Definition 63 and prune some of the abducible literals stored within the hypothesis engine when they are logical consequences of the conjunction of the input formula and of the current hypothesis.

The ImpGen-PiD algorithm is executed by the implicate manager. It requests the hypothesis engine to select an abducible literal to add to its current hypothesis, forges a task with its formula manager and forwards it to the SMT-solver. Depending on the answer, it can forward a model to the hypothesis engine to prune future abducible candidates, or notify any external tool using it that it has found an implicate. If possible, the

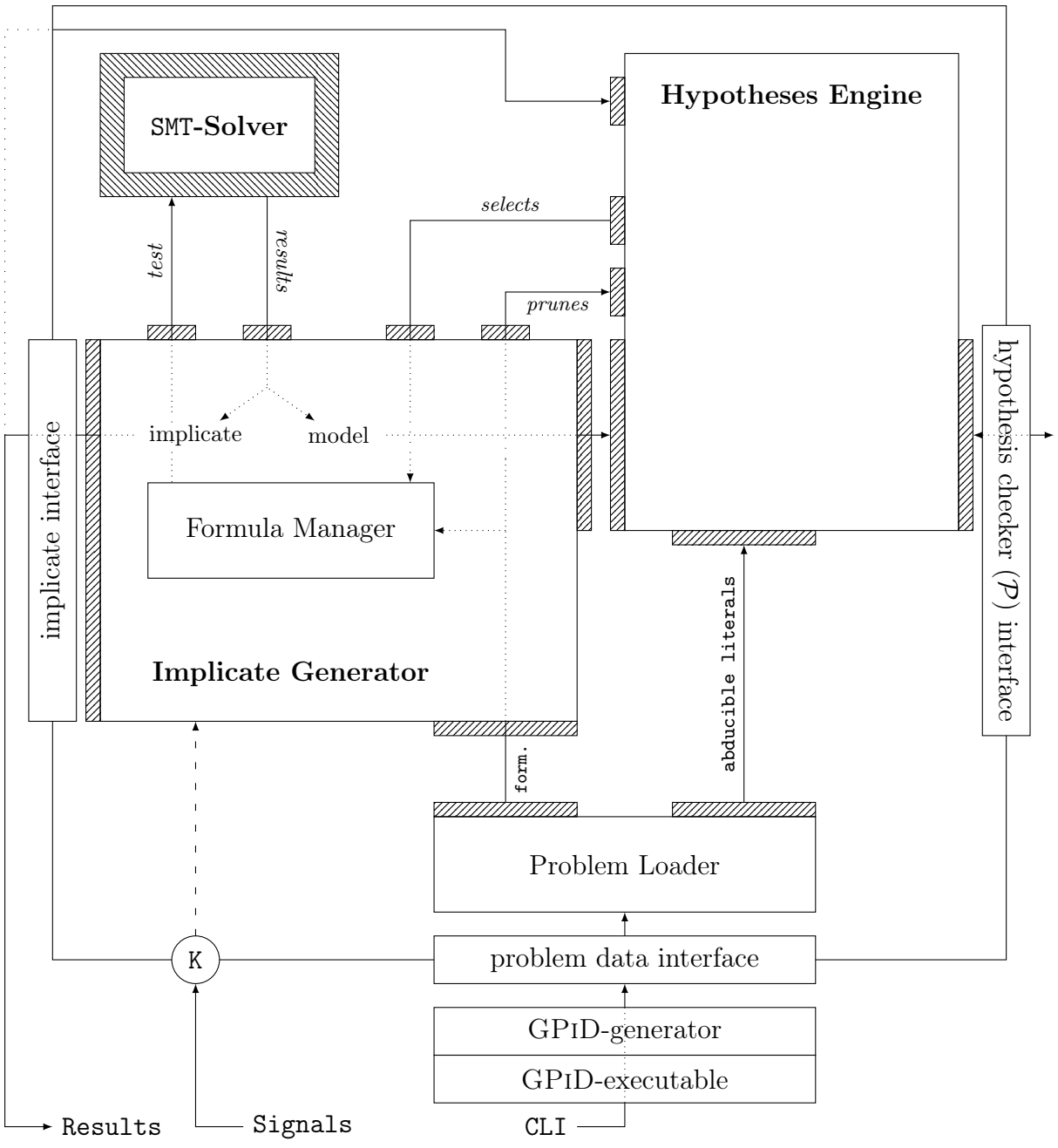


Figure 7.1: Structure of GPiD within the Abdulot framework

hypothesis engine will also store this implicate. This will also allow it to prune exploration branches (see Algorithm 4.1). Similarly, the hypothesis engine can communicate with any available hypothesis checker, which are external components implementing user-defined filtering predicates (see Definition 79 and the use of filtering predicates in Algorithm 3.2), in order to prune abducible candidates those checkers do not accept.

The implicates forwarded by GPiD can be reused for any other purpose when returned within the Abdulot library. They are simply printed on the terminal when a test executable is used.

After an implicate has been found (or after we are sure that no more implicate can be derived by adding more abducible literals to the current hypothesis), the hypothesis engine will backtrack to the last abducible literal selection it made and pursue the exploration by adding to the hypothesis an abducible literal not previously tried at this level.

Finally, the execution can be interrupted by the usual interruption signals. These signals are not forwarded to the SMT-solver however. This can explain the delay between the detection of the interruption and the actual termination of the execution.

In the implementation, the wrapping interface can be instantiated through a C++ template `GPiDAlgorithm<EngineT, LitGenT, IHandlerT>` within the Abdulot library. Its `Engine`, the class used to select hypotheses, also contains the reference to the SMT-solver interface that should be used and its corresponding problem loader. The `LitGenT` object allows one to load and generate core abducible literals that will be used by the engine. It is the object that is plugged as the *problem data interface*. Finally, the class `IHandlerT` will be used to handle the implicates generated during the execution of the algorithm. It is the object that is plugged as the *implicate interface*. Additional hypothesis checkers can also be forwarded to the template, to possibly reject some of the hypotheses the engine will be tempted to make.

7.3 A GPiD hypothesis engine for selecting the candidates and handling the hypotheses

As explicitly induced by the structure of GPiD, it can be plugged with various hypothesis engines, possibly implementing some or all of the algorithm refinements presented in Chapter 3.2. This section provides the description of an implementation providing all the refinements of Chapter 3.2 except the use of unit clauses. A sample description for the standard behaviour of this engine is provided in Figure 7.2 .

The engine stores the abducible literal candidates it recovers for the system it is embedded in as well as the currently selected hypothesis (set of such literals). Following the commands of the parent object, it will add or remove elements from this currently selected hypothesis before forwarding the latter when asked to. When it has to add literals to its hypothesis, it will take into consideration all the elements it has available and apply the corresponding selections as presented in the theoretical sections. The elements it can usually count on are the models forwarded through its interface, the known implicates it stores and any external checker it can access via its wrapping interface. If necessary, it also usually comes with an internal SMT-solver interface to perform some minimal queries in order to prune some candidates that may be inconsistent or leading to implicates that

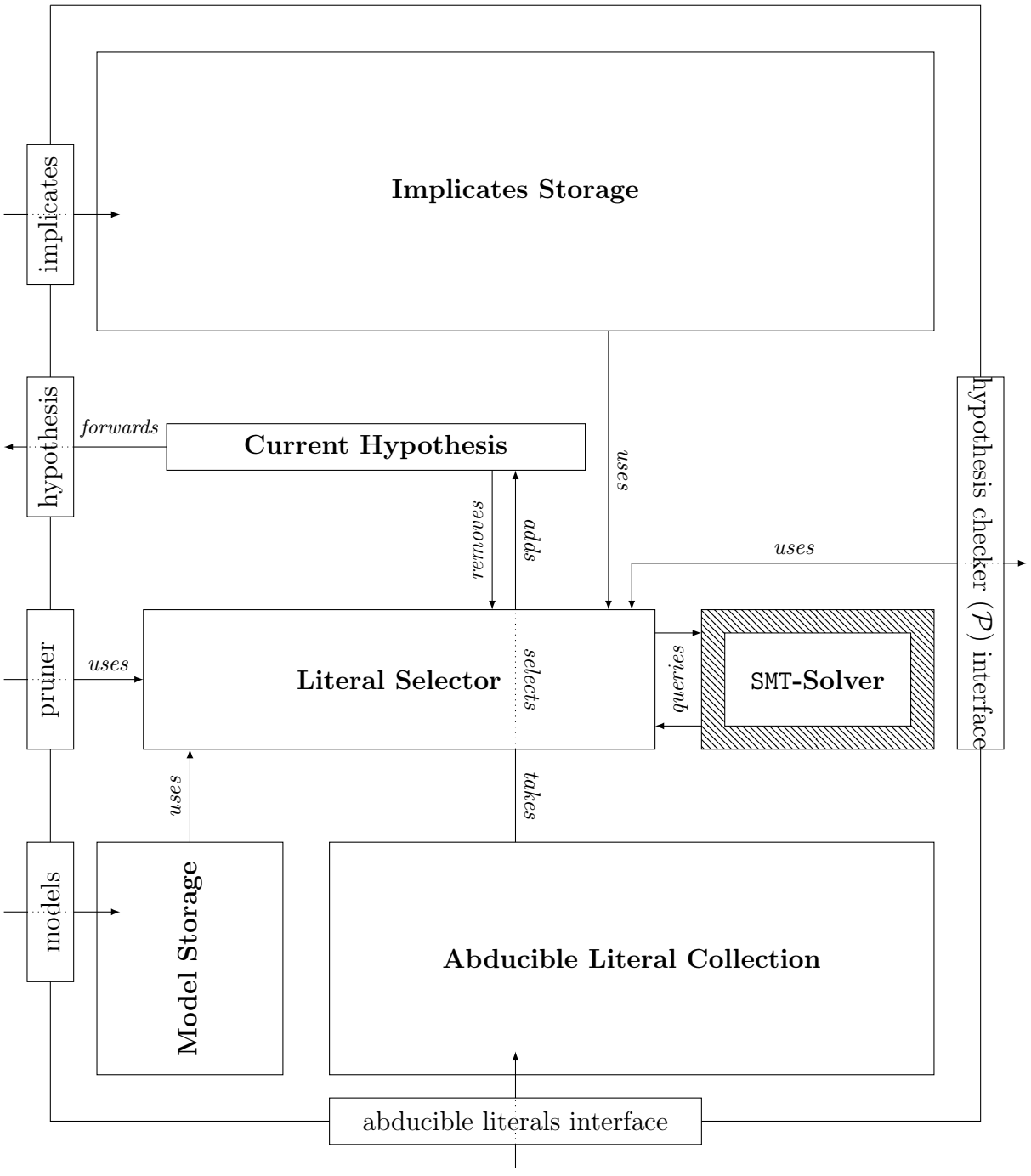


Figure 7.2: Standard structure of a GPID Hypothesis Engine

are already part of the initial problem for instance. The hypothesis engine implemented in the ABDULOT framework uses all these improvements, though they can be optionally deactivated. The framework also contains a so-called *naive* hypothesis engine which only uses minimal information to select abducible literals, and thus may not use any of those. The latter can be used for development or testing reasons.

7.4 Defining, generating and handling abducible literals

The hypothesis engine requires an initial set of abducible literals. As such literals are necessary on all the systems of this thesis, and as their formulation will most probably depend on the theory of the problem they relate to, we need some sort of generic method to describe, load and/or generate them. This is what we describe now.

We start by recalling that one can always write a custom abducible literal generator and plug it within the system. Two generators of this sort are provided within the ABDULOT framework: one for propositional logic problems, used by default jointly with the MINISAT interface on DIMACS [60] problems, and one prototype for SMT-LIB [9] problems. The first one generates the literals corresponding to all the propositional variables declared in the problem and their negation. The second generates all the equalities and disequalities between declared variables of the same sort. If none of those generators are useable and if the user does not want to provide their own generator, the framework also contains a generic method to build abducible literals from the problem loader interface of the SMT-solver. This is done using files we will call *abducible files*, that will be parsed and executed to generate source abducible text elements. All the text elements generated will then be parsed and converted into working abducible literals for the solver via its problem loader.

It is basically an SMT-LIB-oriented file format to define core formula templates and to unroll them using macros. It is best used for defining abducible literals as SMT-LIB elements but can also be used for any other file format. The commands of an abducible file are executed sequentially and before abducible literals can be recovered from it. We give the full grammar of the abducible files in Appendix A and detail the semantics of the commands below. A practical illustration will be given in Example 150.

- **size** ⇀ Set the number of literals to load. If **auto**, it will be computed as the total number of literals generated.
- **rsize** ⇀ Set the number of references to load. References are literals that are specially marked. They can typically be used when handling the implicates generated if a post-treatment requires such distinction (such as back translating a formula derived from a program containing both pointers and non-pointers variables).
- **abduce/ible** ⇀ Define one or more abducible literals.
- **ref(erence)** ⇀ Define one or more reference literals.
- **declare** ⇀ Define (non abducible) literals of one or more sorts. For instance, the command `(declare x y as int)` defines two literals `x` and `y` in the sort `int`. Such

```
(size 4)
(abduce (= a b))
(abduce (= a c))
(abduce (= b c))
(abduce (distinct a b))
(abduce (distinct a c))
(abduce (distinct b c))
```

$$\downarrow$$

$$\mathcal{A} = \{a \simeq b, a \simeq c, b \simeq c, a \not\simeq b\}$$

Figure 7.3: A simple abducible file.

literals are temporarily stored in a set indexed by the name of the sort. Multiple such sets can be created if multiple sorts are declared. The literals in these sets will not be forwarded as abducible literals.

- **extend** ⇀ Make all the literals in the given sets abducible literals. Only affects literals contained in the sets at the moment.
- **(declare-)lambda** ⇀ Define a macro. When applied, it will create a new literal by textually replacing its named parameters by their value in its data.
- **apply** ⇀ Create new literals by applying a function. The function must have been declared. The parameters follow the order of the definition and are either the name of a sort, in which case literals will be created by iterating over the elements of the corresponding set, or of the form `(strict <data>)`, in which case the exact content of `<data>` is used. All the literals generated are stored in the *target* set. Additional options can be specified such as `symmetric` for symmetric operations and `nonself` for non reflexive operations.
- **copy** ⇀ Copy the elements in the sources sets into the target sets. This can be used to factor the computation of some functions. If one wants to apply, *e.g.*, the usual sum function `+` to literals referenced as both `int` and `real`, it can first copy all these literals in a new set `intOrReal` beforehand with the command `(copy int real as intOrReal)`.
- **annotate** ⇀ Attach a forwardable annotation to some literals. Such annotations are forwarded to the problem loader or to the system extracting literals from the generator, while the names of the sets they were contained in is not. This can be used the same way reference literals are used to forward more complex information to the system (such as, *e.g.*, specific conversion rules that were used to generate them from a program).

Example 150. We give in Figures 7.3 and 7.4 some sample abducible files for abducibles written as SMT-LIB formulas, with the corresponding set of abducible literals that is built from them. In Figure 7.3, we define a simple abducible set directly by providing six literals. As we mentioned in the first line that this file contains four abducible literals,

```
(size auto)
(declare a b c as int)
(lambda eq (l r) (= l r))
(lambda neq (l r) (distinct l r))
(apply eq (over int int) res symmetric)
(apply neq (over int int) res symmetric)
(extend res)
```

$$\downarrow$$

$$\mathcal{A} = \{a \simeq b, a \simeq c, b \simeq c, a \not\simeq b, a \not\simeq c, b \not\simeq c\}$$

Figure 7.4: A more complex abducible file.

only the first four will be generated. In Figure 7.4, we define the same abducible literals but using the computational commands of the DSL. We start by declaring our three constants a, b and c as well the macros for equality and disequality. Then we apply these two predicates over our constants and store the resulting formulas in a set names `res`. Finally, we specify that the formulas of `res` should be considered as abducible literals. Because this time we did not explicitly set the number of abducible literals to extract, we will obtain the six literals. Note that if we did not add the `symmetric` keyword when applying the predicates, we would have also generated the literals with the constants in the other order, as well the tautologies $a \simeq a$, ..., and the contradictions $a \not\simeq a$, ...

The Abdulot framework provides an executable to parse and verify such files (called `abdulot-parser`). If abducible files are provided to any executable of the framework, they will have precedence over the generators we presented before. This means that the abducible literals that will be forwarded to the algorithm will be the one loaded from the file, and not the one generated by any other mean.

A version of the ImpGen-PiD algorithm re-expressed within this framework is provided in Algorithm 7.4.

It actually consists in a standardized version of the ImpGen-PiD algorithm where all the work related to the selection of the hypothesis has been delegated to the associated engine within the framework.

7.5 Parametrization of the GPiD tool

We finally take a look at the options accepted by the GPiD tool. It is possible to toggle the use of models, implicate storage and external checker within the hypothesis engine. It is also possible to toggle additional SMT-solver checks within the engine to prevent the generation of inconsistent hypotheses. This is deactivated by default for propositional logic as the only inconsistent hypotheses we could construct would contain both a literal and its complement and we handle such cases by automatically removing from the set of abducible literals the complement of any literal added to the hypothesis; links between literals and there complements are created during the automatic generation of abducible literals by the system for propositional logic problems. Additional SMT-solver checks should also be deactivated by the user if the original set of abducible literals does not

Algorithm 7.4: ImpGen-GPiD (Problem P, Hypothesis Engine E, SMT-solver S, σ)

```
1  S.addConstraints(P) ;
2  while ¬E.complete ∧ ¬(interrupted by the user or system) do
3      let h ← E.select() ;
4      S.push() ;
5      S.addConstraint(h) ;
6      if σ(S.check()) = Sat then
7          E.sendModel(S.getModel()) ;
8      else if σ(S.check()) = Unsat then
9          sendImplicate(h̄) ;
10     else
11         return Error ;
12     S.pop() ;
```

contain any subset that is unsatisfiable. If it is deactivated in other cases, some of implicates generated by the GPiD tool may be inconsistent. One can also configure the engine to skip hypotheses containing more than a fixed, given number of abducible literals, or to stop after the system has generated a given number of implicates. We also offer the possibility to add additional SMT-solver checks to prune abducible literals that are consequences of the current hypothesis. Finally, both global timeouts and local SMT-solver checks timeouts can be set.

The tool can be tried with the test executables provided through the implemented solver interfaces described in Chapter 8. Given one of those interfaces `solver`, a problem file accepted by the chosen SMT-solver `problem.sdt`, and a file containing abducible literals `abducibles.abd`, the generation of prime implicates can be launched with the command `gpid-solver -i problem.sdt -l abducibles.abd -p`. A complete description of the command line options to configure the execution as wanted is provided in Appendix C.

7.6 Implementation of Ilinva

In this section, we describe the implementation of the problem strengthener presented in Chapter 5, which we named Ilinva. A graphical description of its structure is provided in Figure 7.5. It is also presented as a generic system that will be instantiated with an interface for a program prover, that is, a system that generates the verification condition and manage the guide formula for a class of verification structures. The implementation of the interface for program verification and loop invariant generation will be described in Chapter 8.

We present the Ilinva tool within its interface, which takes as an input a problem we know how to convert into a verification structure and that can be solved by a given tool. The interface also accepts additional information for the generation of the strengthenings invariants (such as a core set of abducible elements linked to the problem at hand). We

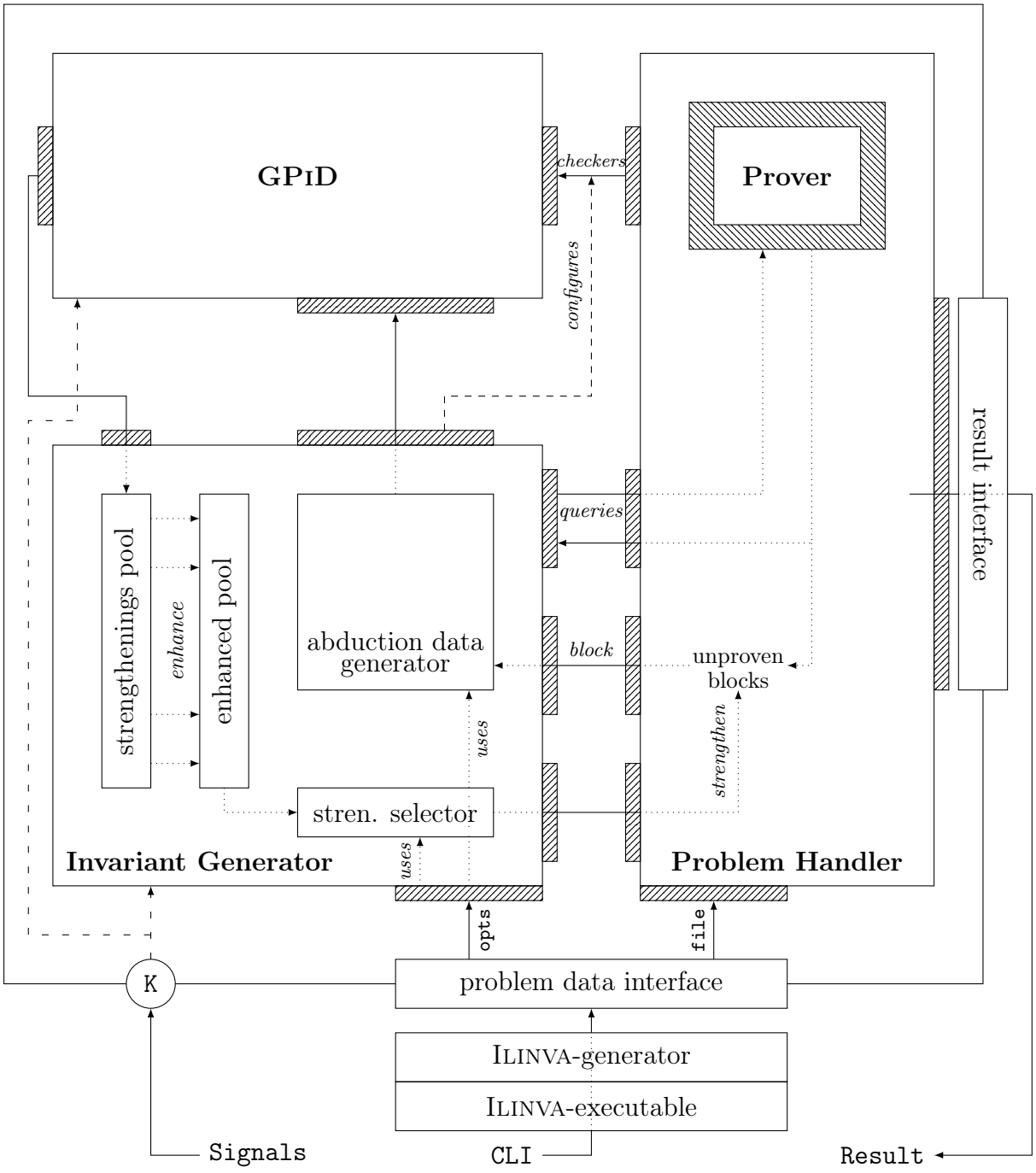


Figure 7.5: Structure of ILINVA within the ABDULOT framework

will expect as a result a modified problem for which the tool is able to produce a complete proof.

When sent to the ILINVA tool, the problem will be forwarded to the problem handler, which handles the task of constructing the associated verification structure and will communicate with the core tool to generate the verification conditions and check the status of the strengthened problems. Regarding the verification structure, exactly which part of the work is done by the tool itself and which part is done by the interface is left unspecified. We will assume that the problem handler as a whole acts as a checker for verification structures and implements an interface that handles the technical details required to do so. All the operations on verification conditions, complete proofs and strengthenings will be done under the command of the main component of the system which is the *invariant generator* — this last component basically runs the ILINVA algorithm.

We know recall how the algorithm is implemented within the framework. The invariant generator will, depending on the data sent to the ILINVA system, ask the problem handler to select a strengthenable part of the problem (couple verification condition – guide formula, see Lines 5 and 6 of Algorithm 5.2). It will then construct an abduction problem, which essentially means constructing a set of abducible literals and a configuration for the GPID tool, and request the abduction system to produce solutions for it (implicates we will convert to implicants). It would naturally be possible to plug another abduction procedure into the system instead of GPID, but this functionality is not explicitly provided, which means that a user would have to change the code initializing GPID instances within ILINVA to do so. While GPID computes the implicates of the problem it was sent, the invariant generator will recover them, sort them in the order they are returned, possibly prune some or construct others from the first ones (typically disjunction of the implicants it recovered) — this is what happens during the *enhancement* part on the figure — and then select which one it should use to strengthen the guide formula it was provided by the problem handler. From this point, it can ask the handler to recheck the problem, conclude if it succeeded or pursue the computation by trying to satisfy other verification conditions, backtrack, *etc.* Note that the construction of GPID instances and their consequent computation of abduction solutions are done in parallel of the main system. This means that while a GPID instance is generating abduction solutions for a given problem part, the invariant generator can try the solutions it receives at the same time, and even possibly explore subtrees of strengthenings if the assignment of one of those strengthenings was deemed useful by the problem handler.

Finally, we emphasize that additional checkers, representing additional information of the expected solution related to the prover used or the problem at hand can be used by GPID, provided they are made available by the problem handler and possibly configured by the invariant generator. This is typically what we will use to remove candidates that do not satisfy loop initialization verification conditions.

Inside the ABDULOT library, this strengthening system is available as a C++ template `IlinvaAlgorithm<EngineT<ProblemHandler, Interface> >`, which will be instantiated by the problem handler we use and by the SMT-solver interface that should be used by GPID. One again, the ILINVA system is made available through a test-executable which will depend on the problem handler it uses. Details on the problem handlers we built are exposed in Chapter 8.

In order to make the system independent on the problem handler we use, we specify in

Interface 7.5 the interface formalism we will consider for problem handler. Any problem prover that can be enclosed into such a structure is accepted by the system.

Interface 7.5: SMT-solver Problem Handler Template

```
1 interface ProblemProof contains
2 |  valid : void ⟶ b ∈ {⊤, ⊥};
3 interface ProblemHandler contains
4 |  proof : void ⟶ ProblemProof;
5 |  selectStrengthenablePart : void ⟶ guide formula/verification condition;
6 |  strengthen : GuideFormula : φ, Strengthening : ϕ ⟶ void;
7 |  release : void ⟶ void;
```

According to this, we rewrite in Algorithm 7.6 the ILINVA algorithm enclosed within the invariant generator component of the technical framework of the ILINVA tool. We store in a stack the sequence of verification conditions that we are currently computing strengthenings for. This stack also serves as a means to derecursify the initial algorithm. Non-valid verification conditions are selected for the current state of the problem at Line 6 and we detect Line 7 if it is possible to strengthen the selected verification condition (see Section 6.4). If this is the case, we create at Lines 18 and 19 a new parallel execution of GPID for this verification condition and remember this choice Line 20. The strenghtening of the candidate loop invariant with the first implicant is performed on the following line. If it was not the case, we have either explored all the possibilities and failed to solve our input problem (Line 8), or we need to backtrack to explore other possible implicants (Line 16) or verification conditions (Lines 13-14).

The ABDULOT framework provides a problem handler for generating loop invariants on WHY3 programs. The details of this implementation are described in Chapter 8. The corresponding executable is `ilinva-why3-wrapped`, which is already plugged to all the SMT-solver interfaces that were available for GPID on the system the framework was build on. A complete list of options that can be used to configure the ILINVA tool is available in Appendix C, as well as a more complete user manual. For a minimal usage, if one wants to generate loop invariants of a given WHY3 program `program.mlw`, using the solver `solver` for solving abduction problems with GPID, and with a core set of abducible elements `abducibles.abd`, one can use the command `ilinva-why3-wrapped -i program.mlw -g solver -a abducibles.abd`.

Algorithm 7.6: ILINVA (Problem P, Problem Handler H, GPID-Constructor C)

```
1  AbductionDataGenerator.configure(P) ;
2  let Stack ← ⟨⟩ ;
3  while ¬(interrupted by the user or system) do
4      if H.proof().valid() then
5          break ;
6      let φ, vc ← H.selectStrengthenablePart() ;
7      if φ, vc = error then
8          if Stack = ⟨⟩ then
9              break ;
10         else
11             H.release() ;
12             if ¬Stack.top().hasImplicant() then
13                 Stack.top().stop() ;
14                 Stack.pop() ;
15             else
16                 H.strengthen(Stack.top().φ, Stack.top().getImplicant()) ;
17             continue ;
18     let 𝔊 ← C.new(AbductionDataGenerator.generate(φ, vc)) ;
19     𝔊.start() ;
20     Stack.push(𝔊) ;
21     H.strengthen(Stack.top().φ, Stack.top().getImplicant()) ;
```

Chapter 8

Concrete interfaces for concrete problems

In Chapter 7, we presented the framework for a generic implementation of the prime implicate generation Algorithm ImpGen-PiD (see Chapter 3) into the GPiD tool and of the precondition strengthener Algorithm Ilinva (see Chapters 5) into the Ilinva tool. We showed that its generic design allowed us to plug it with interfaces of any SMT-solvers and program verifiers. In this chapter, we will describe how we designed concrete interfaces in order to generate actual tools that can be used to solve real problems. The resulting tools will also be used in Chapter 9 to evaluate and discuss the efficiency of these algorithms.

We will start by presenting how we plug widely used SAT-solvers and SMT-solvers into the SMT-solver interface framework we presented. We will build such interfaces for MiniSAT [70], CVC4 [7] and Z3 [50]. Moreover, we will design a generic SMT-solver interface that can work with any SMT-solver that exposes an SMT-lib interface [9]. Then, we will see how we can plug the Why3 [21, 74] verification platform into an interface to use it within Ilinva and generate loop invariants for programs it can handle. This chapter will be concluded by an explanation on how the corresponding tool are compiled in the Abdulot library and on how one can write additional interfaces for other solvers or provers.

8.1 SAT-solver and SMT-solver interfaces

The main guideline we considered when designing interfaces for solvers was to reuse as much as possible the elements they provide. Those include their parsers, formula internal representations and formula printers.

8.1.1 An interface for MiniSAT

MiniSAT is an open-source SAT-solver the design of which is intended to be minimal while still providing very good performance. It handles propositional clauses (disjunction of literals) expressed in the Dimacs format [60].

Example 151. Let us consider the set of propositional clauses we introduced in Example 52: $\Phi = \{p \vee \neg q \vee r, q \vee \neg r \vee \neg s, \neg p \vee \neg q \vee s\}$ over the four propositional variables p, q, r, s.

```
c Dimacs
p cnf 4 3
1 3 -2 0
2 -4 -3 0
4 -2 -1 0
```

Figure 8.1: DIMACS representation of the set of clauses $\{p \vee \neg q \vee r, q \vee \neg r \vee \neg s, \neg p \vee \neg q \vee s\}$

Figure 8.1 gives the DIMACS representation of this set of clauses, where p is represented by the integer 1, q by 2, r by 3 and s by 4.

As MINISAT is open-source and provides a C++ API, we have access to this content and can write an interface plugged directly within the solver, and bypass its external interface reading DIMACS data on the command line. We will still use the DIMACS parser of MINISAT but by sending data directly to it. If we use the interface framework that was presented as Interface 7.1, we plug the representation of the clausal constraints defined in the MINISAT API to the *Constraint* interface and use the clause printer of the solver to convert them into printable strings. The incremental interface for the solver (see Section 1.3) is not implemented directly within the MINISAT API, as it was not made explicitly available in the version we used. It is simulated using an external stack of literals that is forwarded as a set of assumptions during each satisfiability check call.

For these reasons, the problems that will be handled by GPID when instantiated with the MINISAT interface must be presented in the DIMACS format. The given file will be parsed exactly as it would have been by MINISAT. GPID will then generate all the prime implicates of the set of clauses it is given from an initial set of abducible. As computing prime implicates in propositional logic is fairly standard, we implemented a generic abducible generator that constructs from a given DIMACS file all the usual propositional literals we can extract from it. This generator extracts all the propositional variables declared in the file and generates all the corresponding positive and negative literals. If the user wants to reduce the number of abducible literals (which can typically happen when the set of clauses has to be obtained by the Tseitin transformation [139] of an initial formula), he or she will have to use an abduction file in the format presented in Section 7.4.

When MINISAT answers that a given problem is `Sat`, the assignment of the variables it used to satisfy all the clauses can be recovered and used as a model. This model is always total, which means that all the propositional variables of the problem are assigned a boolean value. The structure of the API sadly ensures that whenever a new satisfiability check call is made by the solver, this model will stop being available (as it will be modified by the solver to compute the result of the new formula it was given). This means that if we want to keep these models, we have to store them ourselves. This storage is thus mandatory if we want to check which abducible literals are entailed by the models directly within the exploration loop, as described at Line 9 of Algorithm 3.2. To prevent this additional storage requirement, we can configure the GPID implementation to instead remove all the candidates implied by the model at the moment we recover it from the satisfiability test. With this method, the model is only used once, when it is still valid,

and can be broken by the MiniSAT solver afterwards while being sure that no additional invalid access to it will be made.

8.1.2 Interfaces for the C++ API of CVC4 and Z3

In order to solve problems formalized in more expressive theories, it is clear that we need interfaces to SMT-solvers. CVC4 and Z3 are both widely used, handle a wide range of theories and offer a C++ API from which we can construct an efficient interface for our framework. Similarly to what we did for the MiniSAT interface, we reused as much as possible from the solvers' available code. This ensures that, as both solvers can handle multiple file formats, the problems we can handle will also be expressible in these formats. The API of CVC4 provides parsers for SMT-lib, TPTP [168], and its own CVC4 script file format. Likewise, Z3 provides parsers for files written in SMT-lib or TPTP. This means that when using the interfaces we built on these solvers, the user can inform the Abdulot tool it is using of the format their problem is in and forward the problem as it is. It is important to note however that the parser we select to load the problem file is also the one we will use to parse the abducible literals. In the current version of the Abdulot framework, the problem and the adbucibles must be expressed in the same format.

The fact that we use the formula framework of the SMT-solvers' API means that we recover the context manager they provide, that is, the class they use to allocate, construct and manage formulas. This also means that all the internal work they can do to process formulas forwarded to them via those methods will also be active. It is also from this context manager that we obtain the formula printers of the solvers to return the solutions we generate, usually expressed in the SMT-lib format.

Incremental functions are already implemented in both SMT-solvers and used by our tools. Contrarily to the MiniSAT interface, we do not need to simulate it with an additional stack packaged in the interface. Also, both SMT-solvers provide classes to represent models that we can use to prune our candidate hypotheses. It is interesting to note however that they also both provide a specific method to check whether a given formula is satisfied by the model they currently contain (*i.e.*, the model of the last formula they processed, if the formula was found satisfiable). Similarly to what we did in the MiniSAT interface, we can use these methods to prune the candidate hypotheses once only which avoids the need to store the models in memory.

We provide for both solvers simple generation rules that generate either all the equalities and disequalities between the constants of the problem, or all the inequalities between those constants. For any other use of abducible literals, the user has to provide an abducible file.

It is important to take into account the fact that SMT-solver APIs tend to evolve rather frequently alongside the tool they belong to. This means, in particular, that the interfaces built on them can be broken in future versions. The exact versions of the SMT-solvers handled by the interfaces are provided in the documentation of the tool. Overcoming this development instability is the purpose of the standard, more stable interfaces presented in the next section.

8.1.3 A generic interface for SMT-LIB-compliant SMT-solvers

In order to obtain a stable interface for SMT-solvers, and in order to be able to easily plug our framework into many such solvers, we have implemented a generic interface for SMT-solvers providing an SMT-LIB command-line interface. The principle of this interface is to load input files and abducible files in the SMT-LIB format and to represent the corresponding formulas as strings of characters. It can then implement simple formula constructors (conjunction, disjunction and negation) to permit the creation of satisfiability queries for abduction hypotheses and finally invoke a compatible SMT-solver program by sending to it the adapted problem (rewritten in the SMT-LIB format) within a file. It is clear that some definitions of the problem files the interface handles are dependent on the solver that should be used to solve them. This is typically the case when notifying the theory in which the problem is expressed or when using quantified formulas. The purpose of this generic interface is not to provide a type of universal representation of any problem that could permit it to be sent to any SMT-solver. Rather, it uses a standard representation for satisfiability modulo theories problems that can allow the user to plug the correct solver to the ABDULOT framework for a given problem easily.

In order to handle such problems, we implemented a simple LISP parser in the ABDULOT system. Given an SMT-LIB problem file, it will return the associated abstract syntax tree from which we can extract the SMT-LIB commands of the file as well as their arguments. The configuration commands as well the constants and function declarations are recovered and stored in memory. They will be added to the header of any file later constructed by the interface. The assert commands containing the facts that the problem contains are then stored as strings and can be negated when necessary. Similarly, the abducible literals obtained from an abducible file will be stored as strings and added to the assertions of the problems we construct when necessary.

The interface then performs as expected: the incremental structure is implemented internally using a stack and, when a satisfiability test is queried, the interface constructs an SMT-LIB problem file (juxtaposing the configurations and declarations of the initial problem file, the assertions of the initial problem file and any additional facts added through the incremental interface) and sends it to the executable of the SMT-solver it was configured with. It then recovers the result from the SMT-solver to answer the query. It is standard for SMT-solvers to additionally output a model of the formula that they computed, when this formula was found satisfiable. Technically, it could be possible to parse such models and use them within our framework. However, it appears that the format of those models is less standard than the SMT-LIB file format. Moreover, as no model verification algorithm on SMT-LIB formulas represented by character strings was implemented in the interface, we would require an additional call to the SMT-solver executable to check whether or not a given formula is satisfied by those recovered models. For those reasons, the recovery and usage of these models is not available in the current version and has been left for future work. When asked for a model of a given formula previously found satisfiable by the SMT-solver, the interface will act as if no such model was available.

Instantiations of this standard interface are available in the ABDULOT framework for ALT-ERGO [39], CVC4, VERIT [24] and Z3. If a user wants to plug another SMT-solver into the ABDULOT framework, they can copy the configuration file of one of those

instances and replace the executable of the chosen solver with one of its own.

8.2 A Why3 interface for Ilinva

In this section, we will present the implementation of a problem handler interface for the Ilinva algorithm and a tool using the Why3 platform to generate loop invariants of programs.

8.2.1 Presenting the Why3 platform

Why3 [21, 74, 75] is a widely-used platform for deductive program verification. It is written in OCaml and uses an internal logic formula representation (in the form of the Why language) as well an internal ML-oriented language (WhyML) to define programs and express logical properties on them. From these two representations, Why3 is able to generate verification conditions that can be sent to SMT-solvers to ensure the correctness of the property they express. The main advantages of the Why3 platform is that we can express a very large set of programs and properties in it, and that it can be plugged into most of the SMT-solvers and theorem provers available. These features are extremely useful in our context as we also aim to build a generic system that can plugged to many SMT-solvers to handle various theories.

The use of the Why3 platform also permits to avoid implementing a front-end for a program language, which would have otherwise been necessary to evaluate programs and their loop invariants. Moreover, we can delegate to Why3 the back-end verification condition generators that construct from a program they are given the verification conditions necessary to prove its property. Why3 provides back-ends for all the SMT-solvers it can work with. Moreover, there exists a variety of tools [46, 74] that allow one to automatically translate classes of programs written in well-known programming languages into a format that Why3 can handle.

Why3 is thus a deductive program verification platform able to perform most of the operations required by the Ilinva algorithm on verification structures (see Definition 112). It is a very good system to build an Ilinva interface upon as it will perform most of the work by itself. The following paragraph of this section details how we build the elements we have to implement to do that: wrappers to obtain an efficient communication between the Why3 tool and the Abdulot framework.

We emphasize that the purpose of the implementation of this interface is to study the feasability of the generation of loop invariants through abduction in many theories more than to obtain a tool as efficient as possible to do so. It is also intended to be inserted in the generic framework of the Ilinva algorithm. If one wants to implement an efficient system to compute the loop invariants of programs using the Ilinva algorithm, a better choice would be to extract directly the instantiated version of the Ilinva algorithm for the generation of program loop invariants and to implement this algorithm directly inside a deductive program verification system such as Why3 (see *e.g.* [20]).

8.2.2 Recovering and using the verification conditions generated by Why3

We will now describe the structure of the Ilinva interface we built to plug Why3 into our tool. The primary goal of this interface is to be able to represent the WhyML program, to modify its invariants and to communicate with Why3 in order to check whether a proof can be obtained and/or to obtain verification conditions for a particular property expected from the program. For these reasons, we designed a component that (loosely) parses the WhyML program it is given, extracts the loops and the associated invariants, and then constructs a structure linking these candidate invariants to the location they appear at within the program. From this structure, the component is able to export a WhyML program that would correspond to the one it was previously given where all the invariants are replaced by the candidates the structure contains. This exported WhyML program can then be returned outside of the interface (a functionality that will be used when we have obtained a proof to return a result) or forwarded to the Why3 platform to check the status of the program decorated with the current candidate invariants.

In the latter case, the interface will recover the verification conditions that Why3 generates for the program it was given, as well as the result it obtained out of the computation (that is, whether the generated formulas were found to be valid or not). It will then associate each verification condition to the loop invariant location it depends on (see Section 6.4) and generate relevant couples of loop invariant locations and verification conditions that it forwards to the main algorithm. In practice, we generate all possible couples of this type and try them one after the other: if the selection of one couple does not permit to generate a solution to the problem, we will backtrack until this selection was made and try the next couple. When the main system obtains a strengthening from abduction for a given couple, it will send this strengthening to our Why3 interface that will use it to strengthen the designated candidate loop invariants. Similarly, when backtracking, the main algorithm will delegate the task of relaxing previously strengthened invariants to the interface.

8.2.3 Configuring GPID and its SMT-solver interfaces

As we use GPID to compute the abduction solutions of the system, we detail in this section the configuration we forward to it from the Ilinva tool. We mentioned in Chapter 7 that the GPID implementation is generic (with respect to the SMT-solver). However, when using it in practice, we need to decide which SMT-solver to use. As mentioned in Section 8.2.1, the Why3 platform already uses SMT-solvers to decide whether the verification conditions it generates are valid or not. As we recover those verification conditions, it is natural to use the SMT-solver that Why3 targeted when it generated them. For this reason, we will ask the user to choose which SMT-solver Why3 should use to prove the program and use the same SMT-solver within GPID. It is clear that one of the major advantages of the Why3 platform lies in the fact that one can use a distinct prover for any given verification condition, as it is very possible that, within a problem, some verification conditions will be found more easily by one particular prover. This suggests that some work should be carried out on trying to check the verification conditions with the various SMT-solvers available in Why3 or even to do the same for

the computation of the abduction solutions. However, as the use of such a functionality would greatly increase the number of satisfiability tests without further refinements, it has not been implemented.

Also, as we explained in Section 6.4, we decided not to try to strengthen candidate loop invariants when verification conditions related to the initial conditions of the loop are not valid. In particular, when we do not allow the invariant generator to produce disjunctions of implicants, this means that we can, instead of removing such possible strengthenings after they have been generated, use a filtering predicate within GPID to prevent their generation directly. When the algorithm will construct its hypotheses, it will check, before adding any literal, that these literals necessarily hold when the loop is entered (see also Section 3.2). This verification will be done by the WHY3 interface, to which the GPID wrapped execution will forward its candidate hypotheses for verification before pursuing its computation. This technique cannot be used if disjunctive loop invariants are allowed, since, in this case, such implicants (those invalidating the loop initialisation verification condition) generated by GPID may still be used in a disjunction that could, as it will be less restrictive implicants, satisfy the corresponding verification condition.

8.2.4 Abducible literals for the strengthening of WHYML programs loops

We now present some practical details on how we can generate and handle abducible literals for WHYML programs. As we explained in Chapter 6.4, a natural way to obtain relevant abducible literals on programs is to consider the symbols they contain (program variables, program functions) and to generate the literals we can obtain from those. In order to do that, the ILINVA implementation contains a set of tools that are able to (loosely) parse WHYML programs, extract such symbols and generate the corresponding abducible literals expressed in the abducible format handled by our system. This initial set of abducible literals will be used by the algorithm as its core set of abducible literals. When tackling a particular abduction problem however, we do not need all those abducible literals to be considered. The WHY3 interface will be tasked with the removal from this initial set of the literals that are not related to the current verification condition. Those include the ones that contain symbols that either are undefined at the location of the condition or that do not appear in it. This updated set will be the one forwarded to GPID to perform abduction on the verification condition.

The translation that is required to adapt abducible literals to a specific verification condition (and backward, to translate the solutions it returns into a correctly expressed loop invariant strengthening), as it can differ depending on the type of verification condition at hand (see again Section 6.3), should also be done by the interface. As it happens deeply within the GPID system however, we cannot expect it to be executed this transparently in the implementation. In order to obtain the result, the interface will wrap alongside the loop invariant condition/verification condition couple a conversion toolbox that will perform this translation. The GPID tool will handle the abducible literal it was provided blindly. However, whenever it exports a problem for its SMT-solver, the practical construction of problem will not use the abducible literal itself but the result of its conversion by this toolbox. Similarly, the implicates it generates will transit by this conversion toolbox to be translated before being returned. This allows us to perform the

```
let main () : unit
    diverges
  = let t = ref 0 in
    while !t < 10 do
        invariant { true }
      t := !t + 1
    done;
    assert { !t = 10 }
```

Figure 8.2: Simple WhyML loop

necessary adaptation of the abducible literals for the problem without explicitly defining a distinct behaviour within the GPID tool.

We also remark that, in addition to the the minimal translation required by the ILINVA Algorithm presented in Chapters 5 and 6, additional technical conversions are necessary and will be performed by the same toolbox. This is typically the case for WHYML program variables that are WHYML references. Indeed, such a distinction is usually omitted within the verification conditions but is mandatory for the program to be contextually valid. By using the reference keyword on variables within the abducible file we generate from WHYML programs, the conversion toolbox will be able to detect whether a given variable should be written as a reference or not and perform the translation of the implicates accordingly.

Example 152. Let us consider the sample WHYML program presented in Figure 8.2. A satisfying loop invariant to prove the assertion of this program would be, *e.g.*, $t \leq 10$. However, as t is represented in the program by a reference variable `t`, this invariant should be written in WHYML as `{ !t <= 10 }`. As the verification condition generated by WHY3 makes no distinction between reference and non-reference variables, we will need to define t as a reference variable within our abducible file.

Now let us unroll a complete execution for this sample program. First, we need to obtain the set of abducible literals we will use for the computation of abduction solutions. This program contains only one variable (`t`) and two integer constants: 0 and 10 (we do not consider the incrementation). The predicates contained in the program are the usual equality for integers $=$ and the inequality contained in the loop condition $<$. From this information, we can generate an abducible file applying the generation rules we described in this chapter. This result is exposed in Figure 8.3 The files will generate 18 abducible literals that will be used by GPID to generate implicates.

We can now send the files of Figures 8.2 and 8.3 to the ILINVA tool, select an SMT-solver for the computation, *e.g.* CVC4, and try to generate conjunctive invariants for our WHYML program (command line: `ilinva-why3-wrapped -i tenten.mlw -a tenten.abd -g ccvc4 -H "solver:cvc4" -c`). The tool will then forward the program to WHY3, which in turn will answer that the assertion it contains cannot be proved and return the associated verification condition (see Appendix B). This condition can be used directly for the abduction in GPID. Moreover, as this example program is very simple, all the abducible literals will be kept for the execution. ILINVA will thus call GPID with this verification condition and recover implicants of the formula expressed in SMT-LIB

```
(size auto)

(lambda eq (l r) (= l r))
(lambda neq (l r) (distinct l r))
(lambda le (l r) (<= l r))
(lambda lt (l r) (< l r))

(declare t as int)
(declare 0 10 as int)
(reference t)

(apply eq (over int int) bf symmetric)
(apply neq (over int int) bf symmetric)
(apply le (over int int) bf nonself)
(apply lt (over int int) bf nonself)

(extend bf)
```

Figure 8.3: Abducible file for the WhyML program of Figure 8.2

```
let main () : unit
    diverges
  = let t = ref 0 in
    while !t < 10 do
        invariant { !t <= 10 }
      t := !t + 1
    done;
    assert { !t = 10 }
```

Figure 8.4: Simple WhyML loop with a valid loop invariant

such as `(< t 10)` and `(<= t 10)`. The conversion toolbox will be able to translate these implicants back to syntactically valid WhyML conditions: `!t < 10` and `!t <= 10` (note the reintroduction of the derefencing operator `!`, induced by the content of the abducible file and required because `t` is a reference variable).

As the first implicant does not propagate (and cannot be strengthened to propagate), the tool will backtrack and try with the second implicant generated by GPID, which happens to be a valid loop invariant for problem. ILINVA will thus return the updated WhyML program (Figure 8.4).

8.3 Documentation, compilation structure and distribution of the implementation

8.3.1 Dummy interfaces for documentation

Two additional interfaces (called documentation interfaces) are available in the Abdulot framework. They are named Tisi interfaces and do not implement any real satisfiability solving algorithm or any precondition verification algorithm. Instead, they serve as generation proxies for a documentation presenting the exact technical requirement of an interface. If a user wants to implement an interface, either for an SMT-solver or for problem prover, they can refer to the documentation generated for these two interfaces to obtain the exact definitions, classes and functions prototypes, preconditions and expected behaviours in order to design it. Additional method descriptions are provided in the documentation to help this design process.

8.3.2 A note on the compilation framework

The compilation of interfaces is included within the compilation structure of the Abdulot framework. The compilation scripts start by generating the additional libraries required by the framework to work, then compile the algorithms and standard tools in the Abdulot library. Once this first step is done, it will detect the SMT-solvers available in the system and generate additional libraries containing the compiled interfaces for those SMT-solvers. The same process is applied to the problem handler interfaces used by Ilinva. We point out that the case of the MiniSAT interface is particular as the compilation scripts will automatically download, patch and compile a correct version of the solver to generate an interface that can be plugged in the Abdulot framework. This additional step is required to let us create an access to some of the functionalities of MiniSAT that are not available in its usual public API (most notably to compute statistics). All the other solvers must have been previously installed on the system by the user (as well as their API if one wants to use the interfaces built upon them). Finally, after all the interfaces have been compiled into libraries, the compilation scripts will compile tool executables for both GPiD and Ilinva using all the interfaces available. These tool instances are the one that be used by the user on actual problem files. They are also the ones we will use in Chapter 9 to evaluate the algorithms developed in this thesis.

8.3.3 Distribution

This concludes the presentation of the implementation of a generic infrastructure implementing the ImpGen-PiD algorithm presented in Chapter 3.2 and the Ilinva algorithm presented in Chapter 5, as well as their instantiations to compute prime implicates modulo theories and to generate loop invariants for programs. The complete Abdulot framework (which includes all the required libraries, algorithm and the interfaces presented in this chapter) is available under a three-clause BSD license on GitHub [160]. The version in use at the time of this thesis was composed of 16000 lines of code, among which 1500 concern to the compilation structure, 1500 to the implementation of the ImpGen-PiD

algorithm, 600 to the implementation of the Ilinva algorithm, 3500 to the SMT-solvers interfaces and 3500 to the Why3 interface.

Chapter 9

Experimental evaluation of GPID and ILINVA

In Chapters 7 and 8, we presented an implementation of the prime implicate generation algorithm we devised in Part I and of the loop invariant generator we devised in Part II. In the current chapter, we are going to use this implementation in order to perform an experimental evaluation of those methods. We will take a particular interest in showing that the methods devised are indeed generic and that they can solve problems expressed in many different theories, even though, when competing with other tools specifically designed for some particular theory, they do not always perform as efficiently. We will also take some time to discuss which lines of research we think should be explored to improve the prime implicate generator in regard of the application we used it for, namely, loop invariant generation.

9.1 Experimental infrastructure

All experiments presented in this chapter have been performed on a computer powered by a dual-core Intel i5-4250U processor running at 1.3 GHz and running macOS version 10.14.3. The computer possessed 4 GB of DDR3 RAM running 1.6 GHz and a SSD Hard Drive. The version of ABDULOT framework used for the experiments in this Chapter is 0.7.11, compiled with CLANG (Apple version 10.0.1-0.46.4). The SMT-solvers used are CVC4 (a prerelease of version 1.7), Z3 (version 4.7.1) and ALT-ERGO (version 2.3.0). The version of the MINISAT SAT-solver we used is a patched compilation of version 2.2.0. Finally, the version of the WHY3 platform we used is 1.2.0. The exact version of the systems and tools we compare ourselves to will be mentioned when such tools are introduced within the chapter. Most of the raw results were processed using either Python or TOEV scripts available in the public repository of the ABDULOT tool before being presented here. Such scripts were used mostly to compute statistical values on the results, recheck the obtained results to ensure that they are correct and draw the graph included in this work. The benchmarks and set of examples we used are described in detail when introduced within the chapter. They mostly consist of known examples obtained from standard repositories and of crafted examples used for testing and analysing specific properties of the algorithms we study.

9.2 Evaluating the ImpGen-PiD algorithm in propositional logic

We start by evaluating our prime implicate generator in propositional logic, where it is reasonable to try and to generate all the prime implicates of a given set of clauses, and where we can compare the efficiency of our approach with other state-of-the-art alternatives. In order to evaluate the ImpGen-PiD algorithm in this context, we implemented in the Abdulot framework the model minimization (see also Chapter 11) and unit propagation unfolding we presented in 3.4. This was done by partly merging the main algorithmic component with its MiniSAT interface. We compared the resulting implementation with two state-of-the-art prime implicate generator for propositional logic: ZRes [163] and primer [141]. We will look at whether or not these algorithms are able to generate all the implicates of a given benchmark or some implicates of a given benchmark. Moreover, we will discuss the results we can obtain by reducing the size of the set of abducible literals and what is says on the behaviour of the algorithm. We will also discuss the use of the filtering predicate within GPiD to generate implicate of small size more efficiently than with the other tools.

9.2.1 Benchmarks

We first present the benchmarks we used for prime implicate generation in propositional logic.

- `uf20`: Set of 1000 uniform random satisfiable almost unsatisfiable problems (few models). 20 variables and 91 clauses (4.3 clause-to-variable ratio, *i.e.* 4.3 times more clauses than variables) with clauses of length 3. Benchmark available on the SAT-lib [93, 94]
- `mid20-8`: Set of 1000 uniform random satisfiable problems. 20 variables, 91 clauses with clauses of length 8.
- `high20-18`: Set of 1000 uniform random satisfiable problems. 20 variables, 91 clauses with clauses of length 18.
- `small`: Set of 298 uniform random problems with 7 variables and 21 clauses of length 3.
- `hcp`: 5 problems with 100 variables and two problems with 250 variables rendered almost unsatisfiable from a first-hand uniform random generation with a 4.3 clause-to-variable ratio. Clauses of length 3.
- `rider`: Uniform random problems with a 4.3 clause-to-variable ratio and clauses of length 3. 20 problems for each number of variables between 20 and 300.

These benchmarks have been chosen to evaluate the various properties GPiD is supposed to possess. Typically, the fact that as expected, it performs best on almost unsatisfiable problems (with a small number of models). Apart from the `uf20` benchmark, all the benchmarks were generated manually. For all the experiments of this section, the

implicit set of abducible literals we consider contains all the propositional literals that are contained in the problem handled.

9.2.2 Evaluating GPID alone

To evaluate the general performance of GPID in propositional logic, we started by studying how well it performs on the benchmarks we chose. On the `uf20`, `high20-18` and `small` benchmarks, the tool was able to generate all the prime implicates of any example within ten seconds. For the problems contained in `mid20-8` however, there are no examples for which it was able to generate all the prime implicates under a minute. We then used the `hcp` benchmark to evaluate how the tool performs on almost unsatisfiable problems. The five examples with a hundred variables were solved (in the sense that we generate all of their prime implicates) under ten seconds by GPID. One of the problems containing 250 variables was also solved under ten seconds, however, the time to solve the other one exceeded fifteen minutes.

We continued with the evaluation of GPID by studying the performance of the algorithm when we reduce the number of abducible literals of the problem we send it. We did this on the `rider` benchmark by randomly selecting a set of abducible literals among the literals of every problem. The average evaluation time of the problems for every given number of variable has been represented on Figure 9.1, where GPID$[x]p$ is the evaluation of the GPID algorithm with x percent of abducible literals. It shows that as expected, the lower the number of abducible literals, the faster the algorithm performs. A profiling of some of the examples with a large number of variable showed that most computation time was spent within the satisfiability tests performed by MINISAT and during unit propagation. As those are parts of a state of the art SAT-solver implementation, this is a limitation of the GPID algorithm that is hard to overcome.

9.2.3 Comparison with ZRES and PRIMER

The last part of the evaluation consisted in comparing the results of GPID to those obtained by other state-of-the-art prime implicate generators in propositional logic. We compared with implementations of algorithms introduced in Chapter 2: ZRES [163] and PRIMER [141].

On Figure 9.2, we represent the comparison of the execution time (in seconds) with ZRES (left) and PRIMER (right) on the `small` (top) and `uf20` (bottom) benchmarks. We observe that we are generally faster than ZRES and a little slower than PRIMER on these benchmarks. Moreover, while ZRES and PRIMER appear to be uniform on these benchmarks (all the problems are solved in almost the same time), we can see that GPID is not on the `uf20`. Indeed, some of the examples take much more time to solve than the others (the ones with more implicates). This result scales on almost unsatisfiable problems, which means that when we solve such problems fast enough, we solve them faster than ZRES and a little slower than PRIMER. However, the larger the number of variables, the larger the number of problems with a small number of implicates for which we fail to answer in a reasonable time. This is also shown on the `hcp` benchmark: for the six problems for which we answer under ten seconds, we compute the prime implicates between one to two seconds slower than PRIMER; for the other one, while PRIMER gives

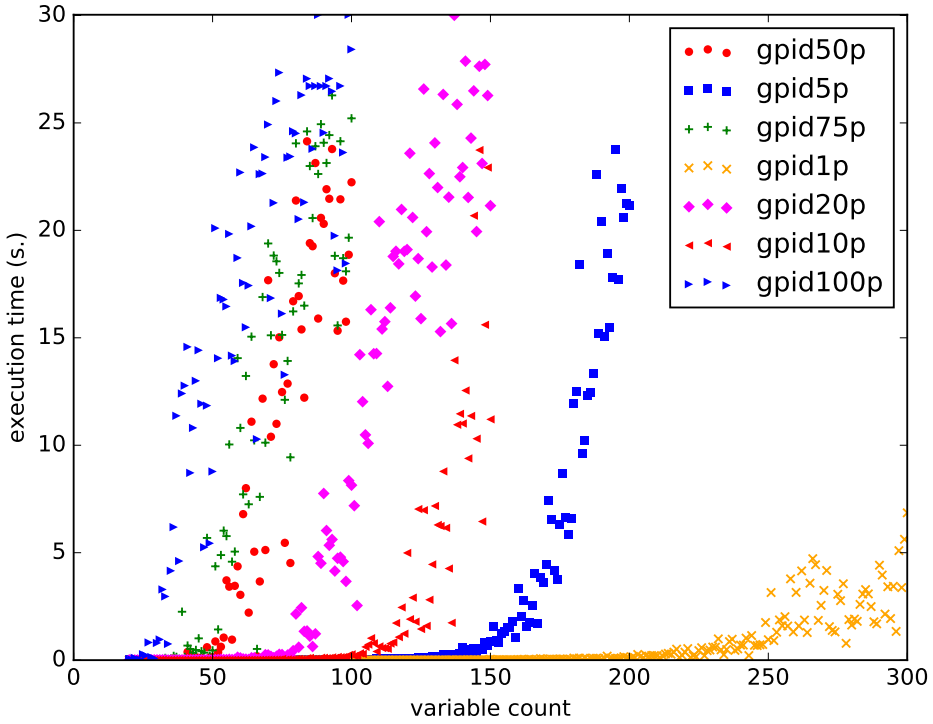


Figure 9.1: Execution of the GPiD algorithm as a function of the number of variables in almost unsatisfiable problems with various abducible ratios.

an answer in almost the same time, we do not. For this particular example we fail to solve, primer finds 374 implicates. On the other example with 250 variables we solved, there are 255 implicates. We emphasize that ZRes did not give an answer in a reasonable time for these examples.

The experiments on less constrained problems (with more implicates) show that we are not as fast as the two other implementations. On the `high-20-18` benchmark, while both ZRes and PRIMER compute the set of prime implicates under ten seconds for all the problems, we are above this time on forty percent of them. Moreover, there are no examples where we appear to be faster than any of the two. The `mid-20-8` benchmark provides another interesting information: on this benchmark, ZRes is the only algorithm that answers under ten seconds (both PRIMER and GPiD taking more time, PRIMER still being the fastest of these two). It is thus possible that such problems are harder to solve with decomposition-based algorithms. In a more general way, even if slower, the results we have in computation time over the various benchmarks are closer to the ones PRIMER gives.

9.2.4 Summary

While not performing as fast as PRIMER, it appears that GPiD falls in the range of the state-of-the-art for solving highly constrained prime implicates generation problems

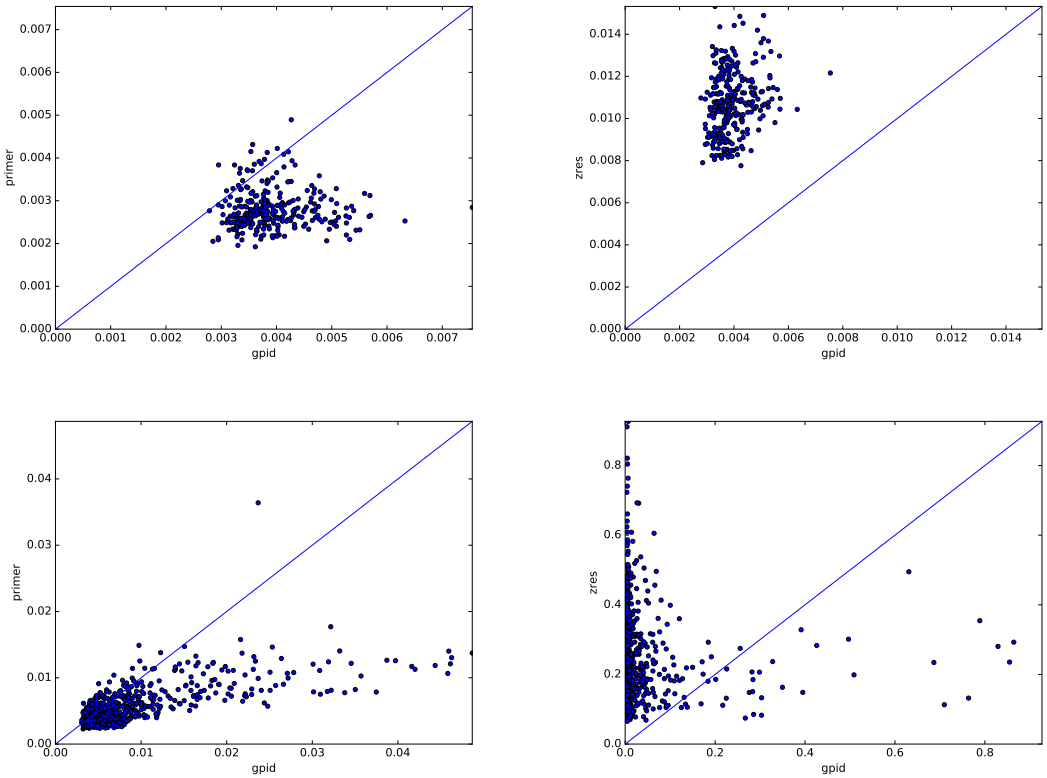


Figure 9.2: Comparison of GPiD with ZRes and PRIMER on the `uf20` (bottom) and `small` benchmarks

(with a small number of models). Indeed, experiments have shown that on such problems, we solve problems between two to ten times faster than ZRes and slightly slower than PRIMER. When the number of variables increases however, it appears that the GPiD algorithm takes more time than the other algorithms for solving a problem. Analyses on these examples show that they have more prime implicates than the ones we solved, which tends to demonstrate that the GPiD efficiency heavily depends on the number of prime implicates. This makes it hard for GPiD to compete with the other algorithms on examples with a large number of implicates, which is not necessarily the case for, *e.g.*, ZRes. However, those are not the examples we are interested in solving.

Even though GPiD does not prove fast enough on general problems, it is still almost as fast as PRIMER on highly constrained problems, even with a large number of variables, as long as the number of prime implicates remains low. GPiD also offers the possibility to control the literals we need in the implicates. This is a distinctive advantage compared to other algorithms, especially when working on problems converted from actual programs. It also reflects that the flexibility we gain on the form of both problems and solutions can have a negative impact on the efficiency of the implicate generation.

9.3 Evaluating the ImpGen-PiD algorithm in the general context

We now tackle the direct evaluation of the ImpGen-PiD Algorithm via the GPiD tool in the general context (modulo any given theory for which an implementation of a decision procedure for testing satisfiability is available). In this case, it is not reasonable to try to generate all the implicates of a given formula (as they can be particularly numerous, even for simple problems), the goal of this section is restricted to the evaluation of whether the system is able to generate some implicates for a wide range of problems and, if so, how many (and/or how fast) it can generate. As a secondary experiment, we will also evaluate how this implementation compares to another prime implicate generator in the theory it handles.

In order to obtain a large range of logical problems in various theories, we built our benchmarks from problems extracted from the SMT-lib [9, 8] library. We considered the theories of quantifier-free uninterpreted functions (which are contained within the QF_UF folder of the SMT-lib library) and of quantifier-free linear integer arithmetic with uninterpreted functions (which belong to the QF_UFLIA folder). From those sets, only the satisfiable examples have been kept for analysis. As it is induced by the implementation, the sets of abducible literals are part of the problem input. We generated them by considering all ground equalities and disequalities with a maximal depth provided by the user; all the experiments were conducted using a maximal depth of 1 and the average number of abducible literals is around 13397 (min. 1741, max. $17 \cdot 10^6$). For the experiments, we chose not to apply unit propagation simplifications to the considered sets of clauses. The reason for this decision is that efficiently performing such simplifications can be difficult and strongly depends on the theory. We also define $\texttt{fix}(,,)$ (see Definition 63) as the complementation on literals and $\mathcal{P}$ (see Definition 79) as either $\top$ or a predicate ensuring $\textsc{Card}(M) \leq n$ to generate $\mathcal{T}$-implicates of size at most n. In all the experiments, the prime implicates filter (SubMin(), see Definition 43) was not active, so that $\mathcal{T}$-implicates can be generated on the fly. This implicitly means that, when an $\mathcal{T}$-implicate is generated and returned at the same time, there is no guarantee that no more general $\mathcal{T}$-implicate will be generated at a later time. It is guaranteed, however, that no more general $\mathcal{T}$-implicate have already been generated. Finally, we recovered the models from the SMT-solvers in order to further prune the set of abducible literals (see Definition 72 and Proposition 73).

Tables 9.1 and 9.2 show the number of examples for which the GPiD tool generates at least one $\mathcal{T}$-implicate for a given timespan, for the QF_UF and QF_UFLIA benchmarks respectively. The results show that our tool is quite efficient, since it fails to generate any $\mathcal{T}$-implicate within 35 seconds for only 2% (resp. 1%) of the QF_UF (resp. QF_UFLIA) benchmarks. Figure 9.3 additionally shows the proportion of the QF_UFLIA set for which GPiD generates an implicate in less than 15 seconds, depending on the maximal size constraint. For the QF_UF benchmark, the proportion decreases from 97% for a maximal size constraint of 1 to 95% when there are no size restrictions. We also point out that for 57% of the QF_UF benchmark, we are actually able to generate all the $\mathcal{T}$-implicates of size 1 in less than 15 seconds.

We ran additional experiments to compare this approach with a previous one based on a superposition-based approach [68, 69] and implemented in the cSP tool (see also

Size/Time	$[0, 0.5[$	$[0.5, 1[$	$[1, 1.5[$	$[1.5, 2[$	$[2, 5[$	$[5, 10[$	$[10, 35[$	None
1	2235	75	28	16	33	32	61	69
2	2236	81	27	16	30	23	67	69
3	2236	79	27	16	34	23	65	69
4	2230	84	23	18	33	24	68	69
5	2231	79	27	12	36	22	73	69
6	2234	73	29	15	30	24	75	69
7	2231	81	23	15	33	22	75	69
8	2233	78	23	16	33	21	76	69

Table 9.1: Number of problems for which at least one $\mathcal{T}$-implicate of a given maximal size can be generated in a given amount of time (in seconds), for the QF_UF SMTLib benchmark (2549 examples).

Size/Time	$[0, 0.5[$	$[0.5, 1[$	$[1, 1.5[$	$[1.5, 2[$	$[2, 5[$	$[5, 10[$	$[10, 35[$	None
1	120	23	46	76	100	6	25	4
2	120	23	6	0	0	0	247	4
3	120	23	6	0	96	4	147	4
4	120	23	6	0	0	0	247	4
5	120	23	6	0	0	0	247	4
6	120	22	7	0	0	0	247	4
7	121	22	6	0	0	0	247	4
8	116	24	6	3	0	0	247	4

Table 9.2: Number of problems for which at least one $\mathcal{T}$-implicate of a given maximal size can be generated in a given amount of time (in seconds), for the QF_UFLIA SMTLib benchmark (400 examples).

Chapter 2). As far as we are aware, CSP is the only other available tool for implicate generation in the theory of equality with uninterpreted function symbols. Previous experiments (see, e.g., [68, 69]) showed that CSP is already more efficient than approaches based on a reduction to propositional logic for generating implicates of ground equational formulas. This is why we did not run comparisons against tools for propositional implicate generation. We chose to compare the tools by focusing on their ability to generate at least one $\mathcal{T}$-implicate of a given size. Indeed, generating all (prime) $\mathcal{T}$-implicates is unfeasible within a reasonable amount of time except for very simple formulas, and comparing the raw number of $\mathcal{T}$-implicates generated is not relevant because some of these may actually be redundant with respect to non-generated ones. We believe that, in practice, being able to efficiently compute a small number of $\mathcal{T}$-implicates for a complex problem is more useful than computing huge sets of $\mathcal{T}$-implicates but only for simple formulas. This is typically the case when we use this algorithm for the generation of loop invariants within the ILINVA Algorithm, where we want to obtain a relevant strengthening as fast as possible, not a huge set of uninteresting strengthenings leading to dead-ends. The following experiments are only based on the previously used benchmarks that can be

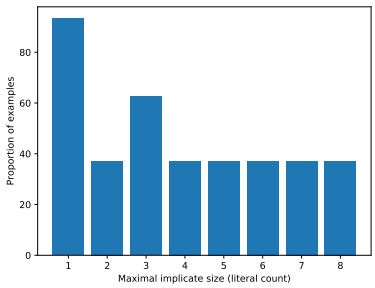
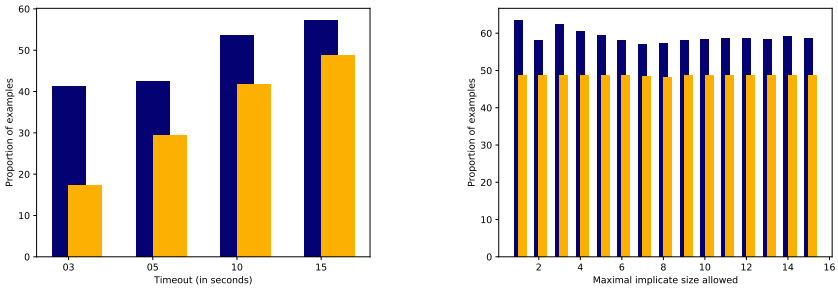


Figure 9.3: Proportion (out of 100) of examples of the QF_UFLIA benchmark where GPID generates at least one implicate under 15 seconds.



(a) (b)

Figure 9.4: Number of examples from the QF_UF benchmark set for which GPID (on the left, darker color) and CSP (on the right, lighter color) generate at least one $\mathcal{T}$-implicate within a given time (a) and generate at least one implicate of a given maximal size under 15 seconds (b)

solved by both prototypes. As CSP is not capable of handling integer arithmetics, this only includes the examples of the QF_UF benchmark.

We represented on Figure 9.4 the number of examples for which both tools can generate at least one $\mathcal{T}$-implicate with a given maximal size constraint for various timeouts (a) and generate at least one $\mathcal{T}$-implicate within a given time limit for various maximal size constraints (b). Figure 9.4 shows that GPID is able to generate at least one $\mathcal{T}$-implicate for more examples than CSP. It is able to generate at least one $\mathcal{T}$-implicate for more than twice the number of examples CSP handles when the time limit is low, though this gap tends to be reduced the more time we allow. Moreover, if we limit the size of the $\mathcal{T}$-implicates we generate (which can be done internally by both prototypes), the experiments show that GPID is still able to generate at least one $\mathcal{T}$-implicate for more examples than CSP. Here also, the difference between the two prototypes tends to be reduced when we allow for larger $\mathcal{T}$-implicates. This result was to be expected since $DPLL(\mathcal{T})$-based approaches are more efficient than engines using the superposition calculus for testing the satisfiability of quantifier-free formulas with a large combinatorial structure. Furthermore, the superposition engine used in the previous approach had to

Examples	Sources	Theories
Ox	[57, 85]	LIA
534	[114, 22, 172]	Array, NLIA
509, H04, H05	[114, 22, 172]	NLIA
BM	[157]	LIA
Scmp, Dmd	[157]	(N)LIA, Float
listx	Crafted	List
arrayx	Crafted	Array
expox	Crafted	NLIA
square	Crafted	NLIA
realx	Crafted	LRA
B00	Crafted	NLIA
DIVx	Crafted	NLIA

Table 9.3: Benchmark sources and content for the experimental evaluation of the Ilinva prototype.

be specifically tuned for implicate generation, and thus is already less efficient than state-of-the-art systems such as Vampire, E or Spass. This is part of the advantage of having a generic algorithm using decision procedures as black boxes.

9.4 Evaluating the loop invariant generator Ilinva

We now consider the evaluation of the Ilinva tool for generating loop invariants for WhyML programs. In order to evaluate both the efficiency and the genericity of the approach, we devised a benchmark set containing both examples expressed in theories that are usually considered by such tools (such as linear arithmetic) and examples that are extracted or crafted in more expressive theories, such as those for which SMT-solvers show difficulties to decide the satisfiability. We collected our benchmarks from several sources [57, 85, 157, 114, 22, 172] corresponding to those considerations. We also added examples corresponding to standard algorithms for division and exponentiation (involving lists, arrays, and nonlinear arithmetic). A detailed view of each benchmark source and of the theories it uses is given in Table 9.3.

Some of these benchmarks have been translated by hand from C or Java into WhyML when they were not already expressed in the language. In all the experiments, the initial invariant associated with each loop was $\top$. We configured the Ilinva and GPiD tools with Z3 for the benchmarks containing real arithmetic, with Alt-Ergo for lists and arrays and with CVC4 in all the other cases. All those examples, as not easily recoverable from their sources (due to the WhyML translation that was performed), are distributed with the source of the Ilinva tool.

9.4.1 Results of the generation

We ran ILINVA on each example, first without disjunctive invariants then with disjunctions of size 2. The results are reported in Tables 9.4 and 9.5. For each example, we report whether our tool was able to generate invariants allowing WHY3 to check the soundness of all program assertions before the timeout. If this is the case, we also report the time ILINVA took to do so (columns `T(C)` when generating conjunctions only and `T(D)` when generating implicants containing disjunctions). We also report the number of candidate invariants that have been tried (columns `C(D)` and `C(D)`) and the number of abducible literals that were sent to the GPID algorithm (column `Abd`). Note that the number of candidate invariants does not correspond to the number of satisfiability queries that are made by the system: those made by GPID to generate these candidates are not taken into account. The timeout is set to 20 min. For some of the examples that represent key crafted examples expressed in more difficult theories, we allowed the algorithm to run longer. We report those cases by putting the results between parentheses. Light gray cells indicate that the program terminates before the timeout without returning any solution, and dark gray cells indicate that the timeout was reached. Empty cells mean that the tool could not generate any candidate invariant. The last column of both tables report the time WHY3 takes to prove all the assertions of an example when correct invariants are provided.

An essential point concerns the handling of local solver timeouts. Indeed, most calls to the SMT-solver in the abductive reasoning procedure will involve satisfiable formulas, and the solvers usually take a lot of time to terminate on such formulas (or in the worst case will not terminate at all if the theory is not decidable, *e.g.*, for problems involving first-order axioms). We thus need to set a timeout after which a call will be considered as satisfiable (see Section 7.1). Obviously, we neither want this timeout to be too high as it can significantly increase computation time, nor too low, since it could make us miss solutions. We decided to set this timeout to 1 second, independently of the solver used, after measuring the computation time of the WHY3 verification conditions already satisfied (for which the solver returns `Unsat`) across all benchmarks. We worked under the assumption that the computation time required to prove the other verification conditions when possible would be roughly similar.

9.5 Discussing the results of the evaluation of ILINVA

As can be observed, ILINVA is able to generate solutions for a wide range of theories, although the execution time is usually high. The number of invariant candidates is relatively high, which has a major impact on the efficiency and scalability of the approach.

When applied to examples involving arithmetic invariants, the program is rather slow, compared to the approach based on minimization and quantifier elimination [57]. This is not surprising, since it is very unlikely that a purely generic approach based on a model-based tree exploration algorithm involving many calls to an SMT-solver can possibly compete with a more specific procedure exploiting particular properties of the considered theory. We also wish to emphasize that the fact that our framework is based on an external program verification system (which itself calls external solvers) involves a significant overcost compared to a more integrated approach: for instance, for the `Oxy` examples

	Abd	T(C)	C(C)	T(D)	C(D)	Why3
001	36	9.68	7	11.89	10	0.26
002	536	3′18.9	66		1126	0.45
004	108	50.47	32	2′31.4	156	0.26
005	266	1′9.07	5	1′3.2	5	0.31
006	390	6′13.6	56	18′5.1	552	0.72
007	594	1′50.1	13	15′40.6	355	0.38
008	210	2′35.5	61	9′35.8	528	0.42
009	390		0		0	0.56
010	90	1′39.54	65	12′56.9	0	0.35
011	180	2′17.7	63		942	0.26
012	782		0		0	0.53
013	296	2′4.5	0		1621	0.30
014	270		0		0	0.34
015	36	32.53	21		888	0.27
016	60	12.54	8	29.72	32	0.26
017	36	40.88	26	2′42.5	134	0.33
018	38	58.49	38	6′53.3	0	0.30
019	60	1′59.5	111		1620	0.31
020	546		380		870	0.49
021	90	0.76	0	0.76	0	0.38
022	270	2′10.1	20	2′11.9	20	0.48
023	36	4.6	5	4.7	5	0.28
025	60	1′23.4	20	2′38.4	44	0.39
026	396	6′23.2	21	7′13.9	66	0.83
028		2′3.9	137	16′22.8	1331	0.31
029	61776		0		0	0.65
030	36	31.43	26	41.66	45	0.26
031	67050		0		0	0.49
032	40	0.865	0	0.833	0	0.50
033	90	1′11.3	12	1′19.9	21	0.45
034	6768	0.798	0	0.79	0	0.44
035	18	18.42	25	2′7.9	200	0.26
036	61778		0		0	1.09
037	36	0.752	0	0.769	0	0.34
038	630		444	3′54.4	0	0.48
039	546		1581		1840	0.40
040	272		0		0	0.84
041			0		0	0.37
042	271	1′50.4	25		605	1.12
043	60	4.27	2	3.67	2	0.29
044		22.481	13	5′7.8	290	0.35
045			0		0	1.50
046			513		813	0.61

Table 9.4: Experimental Results of the Ilinva tool (Part 1).

(taken from [57, 85]), the time used by Why3 to check the verification conditions once the correct invariants have been generated is often greater than the total time reported in [57] for computing the invariants and checking all properties. Of course, our choice also has clear advantages in terms of genericity, generality and evolvability.

When applied to theories that are harder for SMT-solvers, the algorithm can still generate satisfying invariants. However, due to the high number of candidates it tries, combined with the heavy time cost of a verification (which can be several seconds), it may take some time to do so.

The number of abducible literals also has a strong impact on the efficiency of the process, leading to timeouts when the considered program contains many variables or

	Abd	T(C)	C(C)	T(D)	C(D)	Why3
509	130	$(1h50')$	(95)		0	0.66
534	$172k$		8		0	0.62
H04	120	$2'54.8$	223		1383	0.31
H05	1260		0		0	0.37
list0	60	$6'30.4$	77		1722	0.40
list1	20	40.82	3	$3'26.2$	385	0.47
list2	720		40		0	0.40
list3	126	$3'35.1$	11		930	0.44
list4	816		18		0	
list5	468		22		0	0.44
array0			0		0	0.72
array1			0		0	0.50
array2			0		0	0.50
array3			0		0	0.82
expo0	171	$(6h36')$	(9)		0	0.40
expo1	2130		0		0	
square	705		62		148	
real0	965		81		213	0.55
real1	965		73		115	0.55
real2	240		9		2	0.40
real00	36	$4'9.6$	25	$5'32.18$	40	0.47
realS	66	$1'5.3$	5	$1'0.1$	5	0.33
real3	17460		0		0	
BM	1260	$3'15.2$	74		33	3.35
Scmp			0		0	0.83
Dmd	42		6		0	$1'44.9$
B00	$639k$		0		0	0.76
DIV0	560	$3'58$	35		534	0.83
DIV1	310	14.6	19	14.6	19	0.44
DIVE	42250		0		0	

Table 9.5: Experimental Results of the Ilinva tool (Part 2).

function/predicate symbols. It can be observed that the abduction depth is rather low in all examples (1 or 2).

Our prototype has some technical limitations that have a significant impact on the time cost of the execution. For instance, we use SMT-lib files for communication between GPiD and CVC4 or Z3, instead of using the available APIs. We went back to this solution, which is clearly not optimal for performance, because we experienced many problems coping with the numerous changes in the specifications when updating the solvers to always use the latest versions. The fact that Why3 is taken as a black box also yields some time consumption, first in the (backward and forward) translations (*e.g.*, to associate program variables to their logical counterparts), but also in the verification tasks, which have to be rechecked from the beginning each time an invariant is updated. Our aim was not to devise an efficient system, but rather to assess the feasibility and usefulness of this approach.

This finally gives us some additional information on the behaviour of the GPiD tool. When we used it within the Ilinva tool, it was shown to be the main bottleneck of the system. This is due to both the cost of the numerous calls to the SMT-solvers and the size of the search tree of the abduction procedure, especially for hard theories (*e.g.*, non-linear arithmetic) for which most calls with satisfiable formulas were stopped by the local timeout of Ilinva. This suggests that the focus we took on reducing the search

space in Chapter 3.2 was important, and that more work should be invested in reducing this set further. It may even interesting to relax the completeness of the algorithm if we can produce "good" implicates by trying less hypotheses.

Still these experiments show that both systems are already of practical use to solve generic problems where no specific or more adapted solutions is known, or even more classical problems in the case of GPiD, even though it does not compete extensively against state-of-the-art tools aimed at solving a specific class of problems.

Part IV

A model minimization algorithm for performing abduction in separation logic

Chapter 10

Multi-level implicant minimization by formula component-based generalization

In this part, we present the bases of another approach to generate implicants of a formula. The idea we investigate is based on model construction. Once a model of the considered formula has been found, we try to deduce a first implicant from it. In some cases, this implicant can be obtained by considering the conjunction of all the abducible literals that are true in the model. It can be viewed as an abstract description of a class of interpretations satisfying the considered formula. If it is possible to obtain this implicant, the proposed approach will then refine it to produce a new implicant as general as possible. This refinement can be done by either removing hypotheses from the initial implicant or replacing them by weaker ones. When no such operation allows to obtain a formula that remains an implicant of the initial problem, we conclude that we have obtained the best implicant we could get from this method.

In order to compute this refinement of the initial implicant, we present in this part an algorithm to minimize the model-obtained implicants when they can be expressed as a conjunction of smaller, base logical formulas for which we assume basic weakening rules are given. We then perform a weakening on each component by trying to replace it by a more general formula and backtracking if this replacement prevents the new conjunction from remaining an implicant of the considered formula. The method we present is an extension of [26], where the authors present an algorithm to efficiently minimize models of propositional logic formulas. In this context, minimizing a formula is equivalent to minimizing the set of literals that are true in the model. The approach of [26] uses the divide-and-conquer principle to perform this minimization. It is also proven that their approach performs a number satisfiability checks in both $O((\mathfrak{l}' - 1) + \mathfrak{l}' \log(\frac{\mathfrak{l}}{\mathfrak{l}'}))$ and $\Omega(\mathfrak{l}' + \mathfrak{l}' \log(\frac{\mathfrak{l}-\mathfrak{l}'}{\mathfrak{l}'}))$ where $\mathfrak{l}$ is the size of the initial model and $\mathfrak{l}'$ is the size of the minimal model returned by their algorithm. In our context, we can review this propositional logic approach as follows. The method of [26] first constructs a formula from a propositional model by considering the conjunction of all the literals that are satisfied by the model. To obtain a more general implicant, the weakening that is performed consists in removing some of the literals of the formula. Indeed, this gives a formula that is more general than the previous one and thus, if it keeps the initial problem as a logical consequence, gives a

more general implicant (from which we can immediately deduce a smaller, partial model) of the problem. In other theories however, the weakening of the literals of a formula is not that simple. For instance, a literal can be a logical consequence of another (such as $x \simeq 1 \models x \leq 1$ in linear arithmetic). The framework we devise can be applied to other logics and provides a finer control on the components of the model we try to minimize (finer than just deciding between keeping or discarding a literal, as it is done in [26]).

We will start by presenting the framework for this algorithm, based on the same divide-and-conquer paradigm, and then detail instantiations to minimize implicants in propositional logic and separation logic (a well-known logic for reasoning on programs [151]). We will also introduce a simple algorithm to perform abduction on separation logic formulas based on this minimization algorithm. In Chapter 11, we will present and evaluate an implementation of this method and discuss the observed behaviour.

10.1 A framework for representing implicants

We assume given a class of logical formulas $\mathfrak{S}$.

In all the following, we are going to consider what we call the *generalization* of logical formulas according to a *generalization relation*. For this reason, we will assume the existence of a predefined generalization relation $\rightarrow_g$ on the formulas of $\mathfrak{S}$ (saying that ϕ_2 is more general than ϕ_1 when $\phi_1 \rightarrow_g \phi_2$).

Assumption 153. *We assume that $\rightarrow_g$ is well-founded. This property permits to prevent the possibility to generate some infinite generalization sequence leading to the non-termination of our algorithm.*

Assumption 154. *We assume that $\rightarrow_g$ is functional (*i.e. $\forall\phi_1.\forall\phi_2.\forall\phi_3.\phi_1 \rightarrow_g \phi_2 \wedge \phi_1 \rightarrow_g \phi_3 \Rightarrow \phi_2 = \phi_3$*). This property permits to obtain a deterministic generalization relation, which in turns reduces the search space of exploration-based algorithms.*

We will generalize the components of an initial formula that can be expressed as a conjunction of formulas of $\mathfrak{S}$ using this generalization relation.

As our goal is to represent a formula by a set of generalizable components, we first define a way to represent these components and the state they are in (whether they have been generalized or not). To apply the algorithm, we will need a way to keep a memory of the states the components are in, of how many times they were generalized and of their previous states. This information will permit to backtrack the generalization that were made if the resulting formula is not an implicant for the problem anymore. In order to represent these generalizable components, we will thus use sequences of formulas. These sequences will track the consecutive generalizations of the source component while keeping in memory all the previous generalization steps.

Definition 155. We consider non-empty finite sequences of formulas of $\mathfrak{S}$ that we call $\mathfrak{S}$*-sequences*. Such sequences are of the form $\langle \phi_1, \phi_2, \ldots, \phi_n \rangle$ where $\forall i \in [\![1, n]\!], \phi_i$ is a formula of $\mathfrak{S}$.

For an $\mathfrak{S}$-sequence $s = \langle \phi_1, \phi_2, \ldots, \phi_n \rangle$, we will call $\mathbf{base}(s) \stackrel{\text{def}}{=} \phi_1$ the *base formula* of s and $\mathbf{active}(s) \stackrel{\text{def}}{=} \phi_n$ the *active formula* of s.

Definition 156. For $s = \langle \phi_1, \ldots, \phi_n \rangle$ and $t = \langle \psi_1, \ldots, \psi_m \rangle$ two (possibly empty) $\mathfrak{S}$-sequences, we define the *concatenation of s and t* as $s + t = \langle \phi_1, \ldots, \phi_n, \psi_1, \ldots, \psi_m \rangle$.

Example 157. Let us consider the following $\mathfrak{S}$-sequences: $s = \langle x \simeq y, x \leq y, \top \rangle$ and $t = \langle t \simeq r \rangle$. We have $\mathbf{active}(s) = \top$, $\mathbf{base}(t) = \mathbf{active}(t) = t \simeq r$ and $\mathbf{base}(s) = x \simeq y$. Moreover, $t + s = \langle t \simeq r, x \simeq y, x \leq y, \top \rangle$.

Definition 158. We say that an $\mathfrak{S}$-sequence $\langle \phi_1, \phi_2, \ldots, \phi_n \rangle$ is a *generalization sequence* if $\forall i \in [\![1, n[\![, \phi_i \rightarrow_g \phi_{i+1}$.

Intuitively, when representing a generalizable component, the base formula of a generalization sequence is the initial description of the component and the active formula is its current description. The main algorithm starts with a representation of a formula containing only generalization sequences of one element. It updates these sequences so that, at the end of its execution, we are able to construct a minimal model by taking the active formulas of all the generalization sequences that it built.

Definition 159. Given two generalization sequences s_1 and s_2, we say that s_1 is a *prefix* of s_2 if there exists a generalization sequence s_3 such that $s_1 = s_2 + s_3$. Similarly, we will say that s_2 is a *suffix* of s_1 if there exists a generalization sequence s_3 such that $s_3 = s_1 + s_2$.

We will additionally say that a generalization sequence s_1 is a *strict prefix* (resp. *strict suffix*) of a generalization sequence s_2 if it a prefix (resp. suffix) of it and if $s_1 \neq s_2$.

Definition 160. We define $<_g$ the *well-founded order induced by $\rightarrow_g$ on generalization sequences* as follows: for two generalization sequences s_1, s_2, we write that $s_1 <_g s_2$ if s_2 is a strict prefix of s_1.

This implies that s_1 and s_2 start with the same elements but ϕ_1 contains more generalizations and therefore admits an active formula more general than the active formula of ϕ_2.

Remark 161. Notice that because $\rightarrow_g$ is functional, for any two generalization sequences s_1, s_2 such that $\mathbf{base}(s_1) = \mathbf{base}(s_2)$, we have necessarily that either s_1 is a prefix of s_2 or s_2 a prefix of s_1.

In order to simplify the recovery of initial components and the association of various possible generalizations of the same initial formula, we define the following notations:

Definition 162. Given a set of $\mathfrak{S}$-sequences W, we will define the *set of base formulas* $\mathbf{base}(W) \stackrel{\text{def}}{=} \{\mathbf{base}(s) \mid s \in W\}$ and, given a formula $\phi \in \mathbf{base}(W)$, we will denote by $\mathbf{base}^-[\phi](W)$ an arbitrary element of W with a base equal to ϕ and verifying $\forall s \in W, \mathbf{base}(s) = \phi \Rightarrow |s| \geq |\mathbf{base}^-[\phi](W)|$.

Remark 163. Note that $\mathbf{base}^-[]()$ is a partial function. Note also that, if the set of $\mathfrak{S}$-sequences W was obtained from an initial set of $\mathfrak{S}$-sequences of size one by appending to them any given number of their successive generalizations by $\rightarrow_g$, then for any formula ϕ that is the base of one of the sequences of W, $\mathbf{base}^-[\phi](W)$ exists and is unique (because $\rightarrow_g$ is functional).

Definition 164. In order to define the way we construct the main formula from its components, we assume, in the following, the existence of a *representation function* $\mathcal{S}$ from a set of $\mathfrak{S}$-sequences to formulas of $\mathfrak{S}$.

An example of such a function can be the conjunction of the active formulas of the set.

Definition 165. We say that a representation function $\mathcal{S}$ is *monotonous for a generalization relation* $\rightarrow_g$ if for any two sets of $\mathfrak{S}$-sequences W and W' such that W' is obtained by replacing an element $\langle \phi_1, \ldots, \phi_n \rangle$ of W by an $\mathfrak{S}$-sequence $s = \langle \phi_1, \ldots, \phi_n, \psi \rangle$ such that $\phi_n \rightarrow_g \psi$, we have that $\mathcal{S}(W') \models \mathcal{S}(W)$.

Example 166. Let us consider the generalization relation $\rightarrow_g$ such that for any $\phi \in \mathfrak{S}$, $\phi \rightarrow_g \top$. Let us also consider the representation function $\mathcal{S}_\wedge : \{s_1, \ldots, s_n\} \mapsto \bigwedge_{i \in [\![1,n]\!]} \mathbf{active}(s_i)$. Now let us consider a set of $\mathfrak{S}$-sequences $W = \{s_1, \ldots, s_n\}$. For any $i \in [\![1, n]\!]$, $\bigwedge_{j \in [\![1,n]\!]} \mathbf{active}(s_j) \models \bigwedge_{j \in [\![1,n]\!] \setminus \{i\}} \mathbf{active}(s_j) \wedge \top$. This ensures that $\mathcal{S}_\wedge$ is monotonous for $\rightarrow_g$.

Note also that $\mathcal{S}_\wedge$ is monotonous when $\rightarrow_g$ is such that $\phi \models \psi$ every time $\phi \rightarrow_g \psi$.

In all the following, we will always assume that our representation functions are monotonous for the generalization relations we consider.

We now discuss the way we will regroup the base formulas to obtain a coherent representation of an implicant.

Definition 167. Let W be a finite set of generalization sequences. We say that W is a *representation set* if it verifies all the following :

1. $\textsc{Card}(W) = \textsc{Card}(\mathbf{base}(W))$ (no two elements in W have the same base.)
2. $\mathcal{S}(W)$ is satisfiable.

This is meant to ensure both that the generalized formula will be satisfiable and that every generalization sequence in W will be unambiguously related to a unique component of the original formula.

Example 168. For instance $\{\langle x \simeq y, \top \rangle, \langle r \simeq t, \top \rangle\}$ is a valid representation set while $\{\langle x \simeq y, \top \rangle, \langle x \simeq y, x \leq y \rangle\}$ is not, because its two generalization sequences have the same base.

We also need to define additional properties to ensure that the representation set we modify is and remains the representation of an implicant of the original formula.

Definition 169. Let W be a representation set. We say that $\mathcal{S}(W)$ is the *formula represented* by W. We also use the notations $\mathcal{M} \models W$ or $W_1 \models W_2$ when we mean $\mathcal{M} \models \mathcal{S}(W)$ or $\mathcal{S}(W_1) \models \mathcal{S}(W_2)$.

Definition 170. We say that a representation set W is an *implicant representation set* for a formula ϕ if $W \models \phi$.

Finally, we present a notation for representing the fact that we recover the first implicant representation set from our source formula.

Definition 171. We assume the existence of a possibly partial function IEXTRACT that, given a satisfiable formula ϕ, will return an implicant representation set for ϕ. We call such a function an *implicant extractor*.

Example 172. In propositional logic, an intuitive example of implicant extractor would be the following. Given a finite set of propositional literals $\mathcal{L}$ from which we construct propositional formulas, for any satisfiable formula ϕ constructed on $\mathcal{L}$, IEXTRACT would first compute a model $\mathcal{M} \models \phi$ and return $\{\langle l \rangle \mid l \in \mathcal{L} \wedge \mathcal{M} \models l\}$.

10.2 A framework for generalizing implicants

In this part, we present the generalization formalism used to replace components of implicants by more general ones.

Notation 173. In what follows, we will denote by ${\rightarrow_g}^+$ the transitive closure of $\rightarrow_g$ over the formulas of $\mathfrak{S}$. We also consider the usual associated notation ${\rightarrow_g}^*$.

The main idea with this order is to define possible generalization sequences for components. We then try to apply these predefined generalization steps to the components we have at hand.

Definition 174. We consider a partial function *Gen* mapping any pair (W, s) where W is a representation set and $s \in W$ to a set of sequences defined as follows:

1. For any pair (W, s) in the domain of *Gen*, $W \models Gen(W, s)$.
2. For any pair (W, s) in the domain of *Gen*, $Gen(W, s) = (W \setminus \{s\}) \cup \{s'\}$ where $s' <_g s$.

We call this function an *atomic generalization function.*

This function is defined as partial in order to make a distinction between active components we are allowed to generalize and the ones we are not allowed to generalize (typically because it would break the implicant property of the result). This function can be viewed as an extension of the generalization order to generalize representation sets.

Proposition 175. *For* $(W, s) \in$ DOM(Gen)*,* $Gen(W, s)$ *is a representation set.*

Proof. Let us consider $(W, s) \in$ DOM(Gen).

1. Because W is a representation set, and because the construction does not, by definition, add any new base in the returned set, we cannot introduce a duplicate base in the set. Thus we have CARD$(Gen(W, s))$ = CARD$(\mathbf{base}(Gen(W, s))))$. Moreover, if $s' <_g s$, then $\mathbf{base}(s') = \mathbf{base}(s)$. Therefore, $Gen(W, s)$ contains the same set of bases as W.
2. $\mathcal{S}(W)$ is satisfiable because W is a representation set. Moreover, we have that $\mathcal{S}(W) \models \mathcal{S}(Gen(W, s))$ by definition of *Gen*. This means that $\mathcal{S}(Gen(W, s))$ admits at least one model (the same model as W). $\mathcal{S}(Gen(W, s))$ is therefore satisfiable.

Thus $Gen(W, s)$ is indeed another $\mathcal{S}$-representation set. □

Definition 176. Let W_1 and W_2 be two representation sets. We say that W_2 is a *generalization* of W_1 according to an atomic generalization function Gen (and note $W_2 \preceq_{Gen} W_1$) if either $W_2 = W_1$ or $\exists W_I$ a representation set and $s \in W_I$ such that $W_2 = Gen(W_I, s)$ and $W_I \preceq_{Gen} W_1$.

We say that W_2 is a *strict generalization* of W_1 according to an atomic generalization function Gen (and note $W_2 \prec_{Gen} W_1$) if it is a generalization of W_1 and distinct from W_1.

Proposition 177. *Let us consider an atomic generalization function Gen and the relation $\preceq_{Gen}$ on representation sets as described in Definition 176.*

1. *$\preceq_{Gen}$ is an order.*
2. *$\prec_{Gen}$ is well-founded.*

Proof. By definition, if $W_1 \prec_{Gen} W_2$, then the set $\{|s| \mid s \in W_1\}$ is strictly lower than $\{|s| \mid s \in W_2\}$ according to the multiset extension of the reverse ordering on natural number. This entails that $\prec$ is well-founded. Note also that as $\prec$ is well-founded, it is necessarily antisymmetric. □

Definition 178. Let Gen be an atomic generalization function and let W be a representation set. We say that W is *generalizable* according to Gen if there exists $s \in W$ such that $(W, s) \in \textsc{Dom}(Gen)$. We say that $s \in W$ is *generalizable* in W according to Gen if $(W, s) \in \textsc{Dom}(Gen)$.

The algorithm we will use to minimize implicants will perform a decomposition of the initial representation set into smaller, more general ones. In order to reconstruct the correct representation of the complete implicant from these sets, we need to recover the generalization sequence they contain and keep only those for which the active formula is the least general for a given base. Indeed, when we generalize the active formulas of a given representation set, it is possible that the resulting formulas do not entail the one we are constructing an implicant for. This means that at least one component has been generalized more than possible and that this generalization must be undone. We thus need to backtrack this generalization to obtain once more an implicant representation set using the active formulas that were kept in the other sets. This is the purpose of the following definition.

Definition 179. Let W_1 and W_2 two representation sets. We define the *merging operation* $W_1 \otimes W_2 \stackrel{\text{def}}{=} \left\{\mathbf{base}^-[\phi](W_1 \cup W_2) \;\middle|\; \phi \in \mathbf{base}(W_1 \cup W_2)\right\}$.

Using this operator, we can recover previous states of the generalization from one representation set used to test a new generalization and one keeping components in a representation known to provide an implicant of the initial formula.

Example 180. Taking for instance $W_1 = \{\langle p_1 \wedge p_2\rangle, \langle p_3 \wedge p_4, p_3\rangle\}$ and $W_2 = \{\langle p_1 \wedge p_2, p_2\rangle, \langle p_3 \wedge p_4\rangle\}$, we will obtain $W_1 \otimes W_2 = \{\langle p_1 \wedge p_2\rangle, \langle p_3 \wedge p_4\rangle\}$. The context would typically be that we tried to generalize $p_1 \wedge p_2$ by p_2 in one part and $p_3 \wedge p_4$ by p_3 in the other but then discovered that it would have broken the properties we needed thus wanted to backtrack to the previous result. The advantage of the merging operator being that it allows us to backtrack all breaking generalizations even if made separately.

Proposition 181. *If W_1 and W_2 are two representation sets, then $W_1 \otimes W_2$ is also a representation set.*

Proof. Reusing the two points of Definition 167. The fact that $\textsc{Card}(W_1 \otimes W_2) = \textsc{Card}(\mathbf{base}(W_1 \otimes W_2))$ is ensured by construction. The said construction also ensures that $W_1 \otimes W_2 \subset W_1 \cup W_2$, which proves that $\mathcal{S}(W_1 \otimes W_2)$ is satisfiable as the parameters are both representation sets. □

Proposition 182. *As we have a deterministic way to compute the $\boldsymbol{base}^-[]()$ function for representation sets, the merging operator is associative and commutative.*

Proof.

- Commutativity is trivial when applying the properties of the set union operation.
- Associativity uses the same properties after noting that for a representation set W and a formula ϕ, we have $\mathbf{base}(\mathbf{base}^-[\phi](W)) = \phi$.

  Thus, $\mathbf{base}(\{\mathbf{base}^-[\phi](W) \mid \phi \in \mathbf{base}(W)\}) = \mathbf{base}(W)$ and for two representation sets W_1 and W_2, $\mathbf{base}(W_1 \cup W_2) = \mathbf{base}(W_1) \cup \mathbf{base}(W_2)$. This permits to ensure that $\mathbf{base}^-[\phi](W_1 \cup W_2) = \mathbf{base}^-[\phi](\mathbf{base}^-[\phi](W_1) \cup W_2)$ as we look for the minimal sequences on the same set of bases.

□

It is now possible to tackle the construction of our implicant minimization algorithm. In propositional logic, the usual algorithm that is used to minimize an initial implicant built from the conjunction of all the literals satisfied by an initial model consists in splitting these literals into two parts. For each part, the algorithm tries to minimize the set of propositional literals by removing some of them while assuming that all the literals of the other part are present in the model. Of course, only the literals that do not break the implicant property of the whole formula are removed. To perform this simplification, it recursively calls itself. This first step gives the algorithm a minimal implicant for the first part, assuming the second. It can then minimize the second part while assuming that the literals of the first are present in the final conjunction. Once the two parts have been minimized under the support the other, the algorithm is able to return a minimal implicant of the initial formula.

What we present below is an adaptation of this algorithm that permits to handle more complex cases where the weakening of a literal does not consist in its removal. Moreover, we also extend the algorithm to be able to decompose the initial set in more than two parts, hence the following definition.

Definition 183. Let W be a representation set. We say that $\{W_1, W_2, \ldots, W_n\}$ is a *generalization partition* of W if all the following are verified:

1. $\bigotimes_{i \in [\![1,n]\!]} W_i = W$.
2. For any $W_i, i \in [\![1,n]\!]$, $W_i \prec_{Gen} W$.
3. For any $W_i, W_j, i, j \in [\![1,n]\!], i \neq j, s_i \in W_1 \wedge s_j \in W_j \Rightarrow \mathbf{base}(s_i) \neq \mathbf{base}(s_j)$.

Example 184. Let us consider the generalization function that transforms equality between terms into lower inequality and lower inequality to $\top$. This means that, for any two terms t_1, t_2, we have $t_1 \simeq t_2 \rightarrow_g t_1 \leq t_2 \rightarrow_g \top$.

Now let us consider the constants a, b, c, d, e, f, g, h and the representation set

$$W = \{\langle a \simeq b\rangle, \langle c \simeq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\}.$$

The set $\{\{\langle a \simeq b, a \leq b\rangle, \langle c \simeq d, c \leq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\}, \{\langle a \simeq b\rangle, \langle c \simeq d\rangle, \langle e \simeq f, e \leq f\rangle, \langle g \simeq h, g \leq h\rangle\}\}$ is a generalization partition of W as well as the set $\{\{\langle a \simeq b, a \leq b, \top\rangle, \langle c \simeq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\}, \{\langle a \simeq b\rangle, \langle c \simeq d, c \leq d\rangle, \langle e \simeq f, e \leq f\rangle, \langle g \simeq h, g \leq h\rangle\}\}$.

The set $\{\{\langle a \simeq b, a \leq b, \top\rangle, \langle c \simeq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\}, \{\langle a \simeq b, a \leq b\rangle, \langle c \simeq d, c \leq d\rangle, \langle e \simeq f, e \leq f\rangle, \langle g \simeq h, g \leq h\rangle\}\}$ however is not a generalization partition of W. Indeed $\{\langle a \simeq b, a \leq b, \top\rangle, \langle c \simeq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\} \otimes \{\langle a \simeq b, a \leq b\rangle, \langle c \simeq d, c \leq d\rangle, \langle e \simeq f, e \leq f\rangle, \langle g \simeq h, g \leq h\rangle\} = \{\langle a \simeq b, a \leq b\rangle, \langle c \simeq d\rangle, \langle e \simeq f\rangle, \langle g \simeq h\rangle\}$ which is distinct from W.

After decomposing representation sets into smaller parts, we also need to remove from the considered parts of the partition any element that would be replaced when merging it back with the remaining parts of the partition. This cleaning process is the purpose of the following procedure.

Definition 185. Considering two representation sets W_1 and W_2, we define the *clean-representation set of W_1 according to W_2* as the set

$$\mathbf{Clean}[W_2](W_1) \stackrel{\text{def}}{=} \{s \in W_1 \mid \forall t \in W_2, \mathbf{base}(t) = \mathbf{base}(s) \Rightarrow t <_g s\}$$

.

Proposition 186. *For any two representation sets W_1, W_2, we have*

$$W_1 \otimes W_2 = \boldsymbol{Clean}[W_2](W_1) \otimes W_2$$

.

Proof. As W_1 and W_2 are representation sets and $\rightarrow_g$ is functional, we have that $\forall s, t \in W_1 \cup W_2, \mathbf{base}(s) = \mathbf{base}(t) \wedge |s| = |t| \Rightarrow s = t$.

Let $s \in W_1 \otimes W_2$. By construction, we know that s is at least in one of the two sets W_1 and W_2. We also have that $\forall t \in W_1 \cup W_2, \mathbf{base}(t) = \mathbf{base}(s) \Rightarrow t \leq_g s$ because $t = \mathbf{base}^{-}[\mathbf{base}(s)](W_1 \otimes W_2)$. Adding the premise of the property, this means that $\forall t \in W_1 \cup W_2, \mathbf{base}(t) = \mathbf{base}(s) \Rightarrow t \leq_g s$.

- If $s \in W_2$ and $s \notin \mathbf{Clean}[W_2](W_1) \otimes W_2$, then by definition this entails that either $\mathbf{Clean}[W_2](W_1)$ or W_2 contains a sequence s' such that $s <_g s'$. Necessarily s' is contained in $W_1 \cup W_2$, which contradicts the previous property. This ensures that $s \in \mathbf{Clean}[W_2](W_1) \otimes W_2$.

- If $s \notin W_2$, then necessarily $s \in W_1$. As no sequence distinct from s but with the same base in $W_1 \cup W_2$ is smaller than s, this gives us that $s \in \mathbf{Clean}[W_2](W_1)$ and then $s \in \mathbf{Clean}[W_2](W_1) \otimes W_2$.

Now let $s \in \mathbf{Clean}[W_2](W_1) \otimes W_2$. By construction, $s \in W_1 \cup W_2$. The same reasoning on the sequences lengths allows us to deduce that s is the smallest element with its base in $W_1 \cup W_2$, thus that it will be kept when constructing $W_1 \cup W_2$.

This completes the set equality proof. □

We now describe our algorithm to minimize an initial representation set. The algorithm is presented in Algorithm 10.1 and depends on a given atomic generalization function.

The main idea of Algorithm 10.1 for generalizing formulas goes as follows: first we ensure that we are not already considering a minimal set of formulas. If it is not the case, we try various possible generalizations we suspect could preserve the property of the original set. If no such generalization actually works, we decompose our set in a generalization partition and try to generalize each independently, assuming elements of other parts are fixed. An additional representation set Sup is kept to store the maximal generalization level allowed for each component.

Algorithm 10.1: MinimalPart(ψ, Gen, W, Sup)

```
let W = Clean[Sup](W);
if there exist no W' ≺_Gen W then
    return W ;
else if we can detect that there are no more general hypotheses W' ≺_Gen W such
  that W' ⊗ Sup ⊨ ψ then
    return W ;
else if W = {s} then
    if Gen(W, s) ⊗ Sup ⊨ ψ then
        return MinimalPart(ψ, Gen, Gen(W, s), Sup) ;
    else
        return W ;
    end
foreach W_H in an arbitrary set of hypotheses such that W_H ≺_Gen W do
    if W_H ⊗ Sup ⊨ ψ then
        return MinimalPart(ψ, Gen, W_H, Sup) ;
    end
end
let W_1, ..., W_n be a generalization-partition of W ;
foreach W_i, i ∈ [[1, n]] do
    let W_i = MinimalPart(ψ, Gen, W_i, Sup ⊗ ⊗_{j∈[[1,n]],j≠i} W_j) ;
end
return ⊗_{i∈[[1,n]]} W_i ;
```

Lemma 187. *Let $\psi \in \mathfrak{S}$ and Gen be an atomic generalization function. Let W and Sup be two representation sets such that $W \otimes Sup \models \psi$. The two following properties hold.*

1. *$W' \otimes Sup' \models \psi$ for any recursive call* MinimalPart(ψ, Gen, W', Sup') *made by the call* MinimalPart(ψ, Gen, W, Sup).

2. If $\textsc{MinimalPart}(\psi, Gen, W, Sup)$ *terminates, then*

$$\textsc{MinimalPart}(\psi, Gen, W, Sup) \otimes Sup \models \psi$$

.

Proof. First, we notice that the first and second parameters of MinimalPart do not change in any recursive call, which justifies the fact that we keep the notations ψ and Gen within this proof.

We are going to prove the two points simultaneously by induction on the number of recursive calls, as a pre (1) and post (2) condition.

(2) holds if we are in one of the following base cases of the algorithm. If we return at line 3, 5 or 10, we are actually returning $\mathbf{Clean}[Sup](W)$ where W is the entry parameter of the algorithm. Proposition 186 gives us that $\mathbf{Clean}[Sup](W) \otimes Sup = W \otimes Sup$. This allows us to conclude that (2) holds using our initial hypothesis.

For all the other cases, let us assume as an induction hypothesis (*) that (2) holds for any recursive call $\textsc{MinimalPart}(\psi, Gen, W', Sup')$ verifying (1) : $W' \otimes Sup' \models \psi$.

If we happen to return from line 8 or line 14, (1) is ensured by the tests preceding these invocations (line 7 and line 13 respectively). Applying the induction hypothesis (*), we can conclude that (1) holds for these two cases.

This leaves us with the partition case. Let $W_1, \ldots, W_n$ be our initial generalization partition of W. As $\otimes$ is associative and commutative (Proposition 182), we have that $W_1 \otimes (Sup \otimes \bigotimes_{j \in [\![2,n]\!], j \neq 1} W_j) = Sup \otimes \bigotimes_{i \in [\![1,n]\!]} W_i = W \otimes Sup$. This implies (1) as a consequence of our initial hypothesis and Proposition 186.

Applying our induction hypothesis (*), we deduce that $W_1 \otimes (Sup \otimes \bigotimes_{j \in [\![2,n]\!], j \neq 1} W_j) \models \psi$. We reapply the same process for every part of the generalization partition, replacing each time W in the reasoning by $\bigotimes_{i \in [\![1,n]\!]} W_i$ for which the implicant property has been obtained in the previous step. This allows to conclude that, at the end of the partitioning loop, we have $Sup \otimes \bigotimes_{i \in [\![1,n]\!]} W_i \models \psi$, which proves point (2) for the return at line 21.

This completes the proof of the initial lemma by induction. □

Lemma 188. *For any formula* ψ*, atomic generalization function* Gen*, representation sets* W, Sup*, such that* $W \otimes Sup \models \psi$*,* $\textsc{MinimalPart}(\psi, Gen, W, Sup)$ *terminates.*

Proof. Termination comes from the well-founded order used by $\rightarrow_g$. We prove that any recursive call made to MinimalPart uses as its representation set a strict generalization of the previous parameters. Because $\prec_{Gen}$ is well-founded, this ensures that we can only make a finite number of recursive calls before falling in a base case.

This is trivial for the recursive call at line 8 as $Gen(W, s) \prec_{Gen} W$ for any representation set W and any $s \in W$ by construction.

For the recursive call of line 14, this property is ensured by construction as a restriction on the block hypothesis allowed to be tried (see the condition in the loop line 12).

Finally, for the recursive call at line 19, this property is ensured by the Definition (183) of the generalization partition. □

Lemma 189. *Let* $\psi \in \mathfrak{S}$*,* Gen *be an atomic generalization function and let* W, Sup *be two representation sets such that* $W \otimes Sup \models \psi$*.*

We note $W_R = \textsc{MinimalPart}(\psi, Gen, W, Sup)$ *(which exists by Lemma 188). There exists no* $W_H \prec_{Gen} W_R$ *such that both* $\textbf{\textit{Clean}}[Sup](W_H) \neq \textbf{\textit{Clean}}[Sup](W_R)$ *and* $W_H \otimes Sup \models \psi$.

Proof. We proceed by induction on the structure of Algorithm 10.1. The result is immediate for the base cases at Lines 3, 5 and 10. It is also immediate by the induction hypothesis for the tail-recursive calls at Lines 8 and 14.

For the last case, the induction hypothesis ensures that, for any $i \in [\![1, n]\!]$, we have at the end of Line 19 that there exist no $W_{H_i} \prec_{Gen} W_i$ such that $W_{H_i} \otimes Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j \models \psi$. Now let us assume that there exists $W_{H_{1\to i}} \prec_{Gen} \bigotimes_{j\in[\![1,i]\!]} W_j$ such that $W_{H_{1\to i}} \otimes Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j \models \psi$. Then there exists a generalization sequence in $s \in W_{H_{1\to i}}$ that could replace $\textbf{base}^-[\textbf{base}(s)](\bigotimes_{j\in[\![1,i]\!]} W_j)$ in $\bigotimes_{j\in[\![1,i]\!]} W_j$ and still produce an implicant representation set when merged with Sup.

If $\textbf{base}^-[\textbf{base}(s)](\bigotimes_{j\in[\![1,i]\!]} W_j) <_g \textbf{base}^-[\textbf{base}(s)](Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j)$ then we have $\textbf{Clean}[W_{H_{1\to i}}](Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j) = \textbf{Clean}[W_i](Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j)$. Otherwise either $\textbf{base}^-[\textbf{base}(s)](\bigotimes_{j\in[\![1,i]\!]} W_j) \in W_i$, in which case we have by induction that $\textbf{Clean}[W_{H_{1\to i}}](Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j) = \textbf{Clean}[W_i](Sup \otimes \bigotimes_{j\in[\![1,n]\!],j\neq i} W_j)$, or that $\textbf{base}^-[\textbf{base}(s)](\bigotimes_{j\in[\![1,i]\!]} W_j) \in W_{k\in[\![1,i[\![}$, in which case we have the same result from the induction in the previous step. □

We finally describe with the following result how the MinimalPart algorithm can be used to find a general implicant of a given formula.

Proposition 190. *Let* $\psi \in \mathfrak{S}$ *and* $\mathcal{M} \models \psi$. *Let Gen be an atomic generalization function and* IExtract *a implicant extractor. Finally, let*

$$W = \textsc{MinimalPart}(\psi, Gen, \textsc{IExtract}(\psi, \mathcal{M}), \emptyset).$$

We have all the following:

1. $W \models \psi$
2. *For any strict generalization* $W_H \prec_{Gen} W$, $W_H \not\models \psi$.

Proof. These are immediate from Lemmas 187 and 189, whose application conditions are met by the result of the extractor (see Definition 171). □

10.3 Instantiating the algorithm

10.3.1 Minimizing propositional logic models

We start by giving an instantiation of the framework of Section 10.1 that makes it equivalent to the algorithm of [26].

Definition 191. In this section, we consider the class of *propositional logic formulas*. We give an inductive definition of a propositional logic formula ϕ:

$$\phi : \top \mid \bot \mid v \mid \neg\psi_1 \mid \psi_1 \vee \psi_2 \mid \psi_1 \wedge \psi_2$$

where v is a boolean constant and both ψ_1 and ψ_1 are propositional logic formulas.

Definition 192. Let ϕ be satisfiable propositional logic formula and $\mathcal{M}$ a model of ϕ. We consider the following implicant extractor: $\mathfrak{prex}(\phi) \stackrel{\text{def}}{=} \bigcup_{[\![l]\!]_{\mathcal{M}}=\top} \{\langle l \rangle\}$.

Definition 193. We consider the generalization relation $\rightarrow_g$ on propositional logic formula that is such that for any propositional literal l, we have $l \rightarrow_g \top$, and such that no other formulas are in relation.

The generalization using this order on the formulas returned by $\mathfrak{prex}$ (that only contains literals) will thus consist in replacing a given literal from the model by $\top$, which is equivalent to removing it in the representation of the formula. If we only consider generalization partitions of size two, then Algorithm 10.1 will basically consist in randomly separating the literals of the model in two parts and trying to minimize these parts independently. This instance produces the algorithm that is described in [26] and of which this work is a nontrivial extension. This is moreover, the algorithm we use to minimize propositional logic models in Section 3.4.

10.3.2 Minimizing separation logic implicants

In this section, we describe an instantiation of the minimization algorithm to separation logic. Most real-life programs handle more or less complex data structures that are represented in a memory such as memory tables or linked lists. When one wants to express some information on a program in order to prove properties on them, it is natural to look for a way to work with this memory. This requires practical representations and methods to express properties over such memories. Many solutions have been developed in order to tackle this issue. Separation logic [151] is one of them.

Definition 194. We give an inductive definition of a *separation logic formula* $P \in \mathfrak{S}$:

$$P : \phi \mid E \mapsto (v_1, \ldots, v_n) \mid \bot \mid P_1 \wedge P_2 \mid P_1 \Rightarrow P_2 \mid emp \mid P_1 * P_2 \mid P_1 \mathrel{-\!\!*} P_2$$

where ϕ is an equality or disequality between variables, $E, v_1, \ldots, v_n$ are variables, $n \geq 1$ and P_1, P_2 are separation logic formulas.

In separation logic, we can naturally express properties over separated memory blocks. Conceptually, the logic describes a mapping of variables to memory locations. On top of that, it also defines two new logical operators. One is the separating conjunction, denoted by $P * Q$ for two properties P and Q, and represents the fact that the properties are verified on two separate parts of the memory. The second is the separating implication, denoted by $P \mathrel{-\!\!*} Q$, which represents the fact that the memory obtained by any extension using a block verifying P will always verify Q. Intuitively, this second operator is meant to express how one adds elements in memory. When we want to verify properties in a program, as we did in Chapter 6, and if we want to express properties on the shape of the memory handled by this program, it can be useful to do so in separation logic. It can then be interesting the discard the previously introduced abduction problem and to consider instead the problem of bi-abduction, which will be presented in Section 10.4.

We will thus consider in this part that $\mathfrak{S}$ is the set of separation logic formulas. We also define:

$$\begin{aligned}\mathcal{S} : \{s_1, \ldots, s_n\} \mapsto & \bigwedge_{\{s_i \mid \exists v_1, v_2. \mathbf{base}(s_i) \in \{v_1 = v_2, v_1 \neq v_2\}\}} \mathbf{active}(s_i) \\ & \wedge *_{\exists v_1, v_2. \mathbf{base}(s_i) \in \{v_1 \mapsto v_2\}} \mathbf{active}(s_i) \\ & \wedge_{\exists i \in [\![1,n]\!]. \mathbf{active}(s_i) = emp} emp\end{aligned}$$

Definition 195. We call a tuple (D, s, h) an *interpretation for separation logic* when D is a countable set, s is a total function mapping variables to elements of D and h is a partial function from D to D^n.

Definition 196. Let (D, s, h) be an interpretation for separation logic.

Given a variable v in the domain of s, we call *evaluation of v in (D, s, h)* the value $[\![v]\!]_{(D,s,h)} = s(v)$.

Given ϕ an equality (resp. disequality) between two variables v_1 and v_2 of s, we call *evaluation of ϕ in (D, s, h)* the boolean value $[\![\phi]\!]_{(D,s,h)}$ such that $[\![\phi]\!]_{(D,s,h)} = \top$ if and only if $s(v_1) = s(v_2)$ (resp. $s(v_1) \neq s(v_2)$).

Definition 197. Let D be a countable set and h_1, h_2 be two partial functions from D to D^n. We say that h_1 and h_2 are *disjoint* (and write $h_1 \# h_2$) if $\textsc{Dom}(h_1) \cap \textsc{Dom}(h_2) = \emptyset$.

Definition 198. Let D be a countable set and h_1, h_2 be two disjoint functions of elements of D to tuple of elements of D. We define the *disjoint union* of h_1 and h_2 and denote by $h_1 * h_2$ the function from D to D^n such that we have all the following:

- $\textsc{Dom}(h_1 * h_2) = \textsc{Dom}(h_1) \cup \textsc{Dom}(h_2)$.
- $\forall v \in \textsc{Dom}(h_1).\ h_1 * h_2(v) = h_1(v)$.
- $\forall v \in \textsc{Dom}(h_2).\ h_1 * h_2(v) = h_2(v)$.

Definition 199. Let P be a separation logic formula. Let (D, s, h) be an interpretation for separation logic. We define the satisfaction relation between interpretations and formulas as follows: we say that (D, s, h) is a model of P and write $(D, s, h) \models P$ if one of the following holds:

$(D, s, h) \models \phi$	iff $[\![\phi]\!]_s = \top$
$(D, s, h) \models E \mapsto v_1, \ldots, v_n$	iff $\{[\![E]\!]_s\} = \textsc{Dom}(h)$ and $h([\![E]\!]_s) = [\![\psi_1]\!]_s, \ldots, [\![\psi_n]\!]_s$
$(D, s, h) \models \bot$	never
$(D, s, h) \models P_1 \wedge P_2$	iff $(D, s, h) \models P_1$ and $(D, s, h) \models P_2$
$(D, s, h) \models P_1 \Rightarrow P_2$	iff $(D, s, h) \models P_2$ when $(D, s, h) \models P_1$
$(D, s, h) \models emp$	iff $\textsc{Dom}(h) = \emptyset$
$(D, s, h) \models P_1 * P_2$	iff $\exists h_0, h_1.\ h_0 \# h_1, h_0 * h_1 = h$, $(D, s, h_0) \models P_1$ and $(D, s, h_1) \models P_2$
$(D, s, h) \models P_1 \mathrel{-\!\!*} P_2$	iff $\forall h'.$ if $h' \# h$ and $(D, s, h') \models P_1$ then $(D, s, h * h') \models P_2$

Definition 200. We say that a separation logic formula P is *satisfiable* if there exists (D, s, h) such that (D, s, h) is a model of P.

Definition 201. Let (D, s, h) an interpretation for separation logic. For any $x \in \textsc{Dom}(h)$, we call *equivalence class of x in (D, s, h)* the set

$$\mathcal{E}_{(D,s,h)}(x) = \{v \text{ variable} \mid s(v) = x\}.$$

For any $x \in \textsc{Dom}(h)$ such that $\mathcal{E}_{(D,s,h)}(x) \neq \emptyset$, we call *representation of x in (D, s, h)* and denote by $\widehat{\mathcal{E}_{(D,s,h)}(x)}$ an arbitrary element of $\mathcal{E}_{(D,s,h)}(x)$.

Definition 202. Let P be a (satisfiable) separation logic formula and $(D, s, h) \models P$. We consider the implicant extractor $\mathfrak{slex}$ defined as follows:

$$\begin{aligned}
\mathfrak{slex}(P) \stackrel{\text{def}}{=} & \bigcup_{\substack{v_1,v_2 \in \textsc{Dom}(s) \\ v_1 \neq v_2 \\ s(v_1)=s(v_2)}} \{\langle v_1 = v_2 \rangle\} \\
& \cup \bigcup_{\substack{v_1,v_2 \in \textsc{Dom}(s) \\ s(v_1)\neq s(v_2)}} \{\langle v_1 \neq v_2 \rangle\} \\
& \cup \bigcup_{\substack{x_0 \in \textsc{Dom}(h), h(x_0)=x_1,\ldots,x_n \\ \mathcal{E}_{(D,s,h)}(x_0)\neq\emptyset \\ \forall i \in [\![1,n]\!].\mathcal{E}_{(D,s,h)}(x_i)\neq\emptyset}} \{\langle \widehat{\mathcal{E}_{(D,s,h)}(x_0)} \mapsto \widehat{\mathcal{E}_{(D,s,h)}(x_1)}, \ldots, \widehat{\mathcal{E}_{(D,s,h)}(x_n)} \rangle\} \\
& \cup_{\text{if } \textsc{Dom}(h)=\emptyset} \{\langle emp \rangle\}
\end{aligned}$$

We justify this definition by the following proposition.

Proposition 203. *If P is a satisfiable separation logic formula, the result of $\mathfrak{slex}(P)$ is a representation set.*

Proof. The single base is ensured by the fact that we return a set of sequences of length 1, and the satisfiability of active elements is trivial by construction.

$\mathfrak{slex}(P)$ is consistent: (D, s, h) the model of P used to construct the representation set is also a model of $\mathcal{S}(\mathfrak{slex}(P))$.

It is not certain, however, that $\mathfrak{slex}(P)$ entails P. Indeed, the formalism we use to represent the models by formulas does not include much information that appears in the model such as the exact domain of h. It could be possible to extend $\mathfrak{slex}$ by inserting components extracting this information from the model but in order to keep the presentation simple, this has not been done. Instead, we will check if $\mathfrak{slex}(P)$ entails P before proceeding with its minimization and fail if this is not the case. □

Definition 204. We present here the generalization relation $\rightarrow_g$ we define on separation logic formulas:

1. $\neg emp \rightarrow_g \top$.
2. $emp \rightarrow_g \top$.

3. If $v_0, v_2, \ldots, v_n$ are variables, $v_0 \mapsto (v_1, \ldots, v_n) \rightarrow_g \neg emp$.

4. If v_1 and v_2 are variables, $v_1 = v_2 \rightarrow_g \top$.

5. If v_1 and v_2 are variables, $v_1 \neq v_2 \rightarrow_g \top$.

6. Other elements are not in relation.

Justification. We first notice that any formula added in the first representation set by our extractor $\mathfrak{slex}$ is indeed generalizable with this generalization relation. Second, the associated order is well founded (any generalizable formula can be reduced to one of the base cases). □

To obtain the associated atomic generalization function Gen, we still need to define its domain.

Definition 205. Let W be a representation set and $s \in W$. $(W, s) \in \textsc{Dom}(Gen)$ if and only if at least one of the following is verified:

- $\mathbf{active}(s) \in \{emp, \neg emp\}$.
- $\exists v_1, v_2$ variables such that $\mathbf{active}(s) \in \{v_1 = v_2, v_1 \neq v_2\}$.
- $\exists v_1, v_2, \ldots, v_n$ variables such that $\mathbf{active}(s) \in \{v_1 \mapsto v_2, \ldots, v_n\}$.

We can then apply the framework of Section 10.1 to minimize separation logic implicants.

10.4 Applying this algorithm as a subservient tool for bi-abduction in separation logic

Definition 206. Given a separation logic formula P that we call *problem constraint* and a separation logic formula Q that we call *conclusion* such that $P \not\models Q$, the *bi-abduction problem* consists in constructing missing hypotheses F (also called the *antiframe*) and/or missing conclusions G (also called the *frame*) such that $P * F \models Q * G$.

Those will represent the missing preconditions in memory (antiframe) to ensure the conclusion holds as well as information on parts of the memory unaffected by it (frame). In the context of program verification, antiframes are usually used to express the necessary conditions on the initial state of the memory and the frames are used to infer which memory cells are left unchanged by the program. In this work, we will only develop the computation of antiframes, that is, the computation of missing hypotheses F such that $P * F \models Q$. Intuitively, the idea we explore is to generalize an intuitive solution of this problem.

Proposition 207. *Given two separation logic formulas P, Q, the formula $F = P \mathrel{-\!\!*} Q \wedge \neg(P \mathrel{-\!\!*} \bot)$ satisfies $P * F \models Q$.*

Remark 208. Note that the formula $P \mathrel{-\!\!*} Q$ is actually the most general solution to the bi-abduction problem. However, it is possible that a model of this formula would prove unusable by stating that any extension verifying P would contradict itself. To prevent this cases to occur, we add the restriction $\neg(P \mathrel{-\!\!*} \bot)$ to the solution we consider in order to filter such models.

Unfortunately, such a solution is not in a form that makes it easily usable. Indeed, practical tools based on separation logic rarely handle the separating implication, and tend to work with a subclass of separation logic formulas called *symbolic heaps* [32, 13, 61], which intuitively represent the exact mappings of memory cells. It is also interesting to note that the formula $P \mathrel{-\!\!*} Q$ is not explicit regarding the actual shape the memory should have. Indeed, the formula describes the memory indirectly by stating a property some of its extensions must verify, and not the memory itself. For these reasons, we need to refine this solution to obtain a formula that can be handled by such tools. To do that, we choose to obtain the formula from model of this solution, then refine this formula further in order to construct another solution with a more practical form. The first step of this method, obtaining an initial model of the most general solution, can be done using a solver for separation logic. CVC4 provides one such solver. From this model, we can use the Algorithm presented in Sections 10.1 and 10.3.2 to obtain a more general solution. This process is summarized in Algorithm 10.2.

Algorithm 10.2: SL-ABDUCE(P, Q)

```
1 if P ⊨ Q then
2     return ;
3 let (D, s, h) be a model of F = P −∗ Q ∧ ¬(P −∗ ⊥) ;
4 return MINIMALPART(F, Gen, 𝔰𝔩𝔢𝔵(F, ((D, s, h))), ∅) ;
```

Theorem 209. *Given two separation logic formulas P, Q such that $P \not\models Q$, the formula $F = \text{SL-ABDUCE}(P, Q)$ satisfies $P * F \models Q$.*

Proof. This is a direct consequence of Propositions 207 and 190. □

We will present in Chapter 11 an evaluation of the algorithm to see when this effect occurs and whether or not it remains possible to solve even simple abduction problems with this method.

Chapter 11

A simple implementation and evaluation of the minimization algorithm and of its use for abduction in separation logic

This chapter presents an implementation and an experimental evaluation of the implicant minimization algorithm and of the separation logic abduction algorithm we developed in Chapter 10. As we will see, the efficiency of the separation logic checkers and the complexity of the formulas we generate make it difficult for the general bi-abduction algorithm to work efficiently. This lack of efficiency makes it difficult to evaluate the minimization algorithm itself directly from its instantiation in separation logic. In order to evaluate the minimization algorithm itself, before considering its practical use in separation logic, we will define an alternative instantiation, the only purpose of which is the evaluation of the algorithm.

11.1 A instantiation for evaluating the minimization algorithm

As the MinimalPart algorithm is defined for minimizing implicants independently of the concrete definition of the interpretations of the formulas, it is clear that the computation cost of a minimization execution heavily depends on the definition used. In particular, the complexity of the operation verifying if the current generalization still entails the original formula impacts on the efficiency the instantiated algorithm. In order to evaluate the efficiency of MinimalPart independently of this external cost, we devise an instantiation for which this verification can be done easily.

Example 210. Consider a sequence of stacks of tokens, such as the one represented on Figure 11.1. Now assume that, for some reason, we want to reproduce this sequence but that the only information we can access is whether all the stacks of our representation contain less tokens than their counterpart in the source sequence.

To solve this problem, we may use the MinimalPart algorithm. Intuitively, we can first represent each stack by a formula of which the successive generalization represent

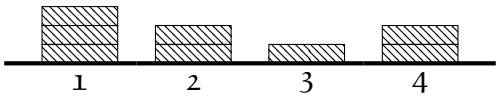


Figure 11.1: A sequence of stacks of tokens

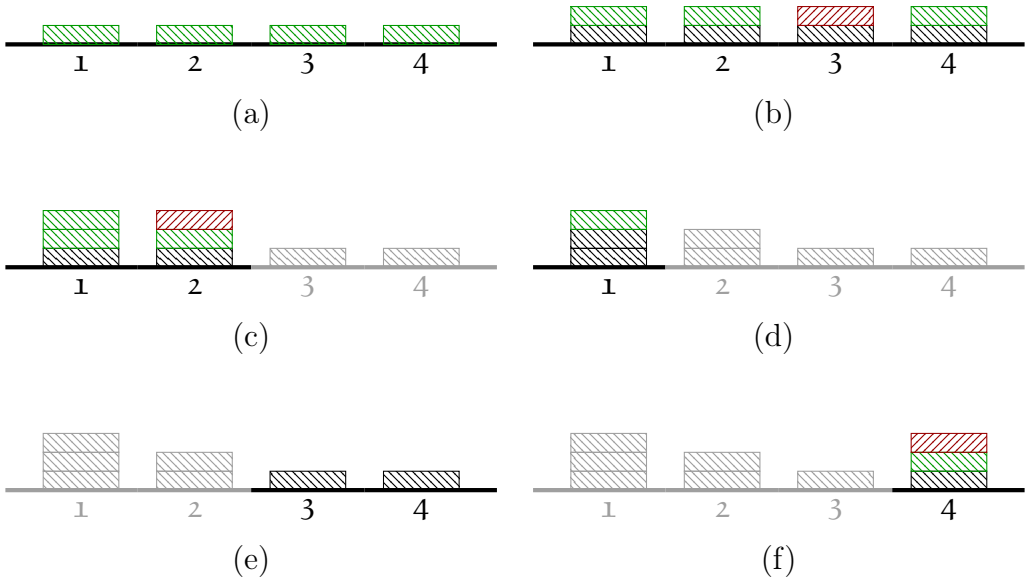


Figure 11.2: Generalization for sequences of stacks of tokens

the various possible number of tokens a stack can contain. Second, we can view our information access by the verification that our sequence is an "implicant" of the source sequence, in the sense of Definition 170.

In this context, we can apply our minimization algorithm for, *e.g.*, a partition of size two. Let us take the example of Figure 11.1 as a source sequence and execute the algorithm (see also Figure 11.2).

Assuming the partitions we consider always split the set of stacks we consider in two, as all the stacks of the source contain at least one token, the algorithm will proceed and generalize our representation by adding these tokens (a). The second time, the generalization will not be accepted because our third stack would then contain more tokens that the one in the source (b). The algorithm will thus split the stacks, fix our third and fourth stacks and minimize the other two in this context. It will then manage to generalize both stacks once more by adding one token, then it will fail again as the second stack in the source contains only two tokens (c). By splitting the set of stacks it works on again, it will obtain sets of one stack and thus generalize them as much as possible. First by adding a new token to the first stack (d), then by noticing than no more tokens can be added, neither in the first nor in the second one. By backtracing until its first split, the algorithm will now have to obtain the stacks it is looking for in the right of the sequence. As the two first sequences have already been obtained, it will fix them and work on the other part (e). Because it is not possible to generalize the whole set, the algorithm will also split the stacks at this moment. It will then try to minimize

the third one, which is not possible, and finish by minimizing the last stack as much as possible (f). Finally, the algorithm will backtrack its splits and reconstruct the sequence of stacks of tokens that was proposed in Figure 11.1.

We will use this simple application to evaluate our implicant minimization algorithm. Therefore, we provide below the formal definition of the related instantiation.

Definition 211. Given a finite sequence of symbols $\mathfrak{Y}$ and a *depth limit* $n \in \mathbb{N}$, we call *token-stack formula* all the elements of the following set:

$$\left\{ \bigwedge_{\mathfrak{y} \in \mathfrak{Y}} \mathfrak{y}^{\mathfrak{k}_\mathfrak{y}} \;\middle|\; \forall \mathfrak{y} \in \mathfrak{Y}. \mathfrak{k}_\mathfrak{y} \in [\![0, n]\!] \right\}$$

Similarly to what we did in propositional logic, we will use $\mathcal{S} : \{s_i \mid i \in [\![1, m]\!]\} \mapsto \bigwedge_{i \in [\![1,m]\!]} \mathbf{active}(s_i)$.

Example 212. Given a sequence of symbols $\mathfrak{Y} = \langle \mathfrak{1}, \mathfrak{2}, \mathfrak{3}, \mathfrak{4} \rangle$, the example of Figure 11.1 could be represented by the token-stack formula $\mathfrak{1}^3 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^2$.

Definition 213. Given a sequence of symbols $\mathfrak{Y}$, a depth limit $n \in \mathbb{N}$ and a token-stack formula, we define an *interpretation of the symbols* as a sequence $\mathfrak{I}$ of natural integers indexed by the symbols of $\mathfrak{Y}$.

Definition 214. Given a sequence of symbols $\mathfrak{Y} = \langle \mathfrak{y}_1, \ldots, \mathfrak{y}_k \rangle$, a token-stack formula $\mathfrak{f}$ and an interpretation of the symbols $\mathfrak{I} = \langle n_1, \ldots, n_k \rangle$, we define $[\![\mathfrak{f}]\!]_\mathfrak{I}$ the *evaluation of* $\mathfrak{f}$ *in* $\mathfrak{I}$ inductively as follows:

- $[\![\top]\!]_\mathfrak{I} = \top$
- $[\![\mathfrak{y}_i^\mathfrak{k} \wedge \mathfrak{f}']\!]_\mathfrak{I} = \mathfrak{k} \leq n_i \wedge [\![\mathfrak{f}']\!]_\mathfrak{I}$.

Definition 215. Given an interpretation of the symbols $\mathfrak{I}$ and a token-stack formula $\mathfrak{f}$, we say that $\mathfrak{I}$ *is a model of* $\mathfrak{f}$ if $[\![\mathfrak{f}]\!]_\mathfrak{I} = \top$.

We say that a token-stack formula $\mathfrak{f}$ is satisfiable if it admits a model.

Given two set-symbols-formulas $\mathfrak{f}_1$ and $\mathfrak{f}_2$, we say that $\mathfrak{f}_2$ is a logical consequence of $\mathfrak{f}_1$ if all the models of $\mathfrak{f}_1$ are also models of $\mathfrak{f}_2$.

Example 216. Consider again the token-stack formula $\mathfrak{1}^3 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^2$ representing the example of Figure 11.1. The sequence $\langle 2, 2, 1, 1 \rangle$ is a model of this formula while the sequence $\langle 3, 2, 1, 3 \rangle$ is not. Thus, we can write $\mathfrak{1}^2 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^1 \models \mathfrak{1}^3 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^2$ and $\mathfrak{1}^3 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^3 \not\models \mathfrak{1}^3 \wedge \mathfrak{2}^2 \wedge \mathfrak{3}^1 \wedge \mathfrak{4}^2$.

Proposition 217. *Let* $\mathfrak{f}$ *be a token-stack formula. The sequence* $\langle 0 \mid \mathfrak{y} \in \mathfrak{Y} \rangle$ *is a model of* $\mathfrak{f}$.

Definition 218. Let $\mathfrak{f}$ be a token-stack formula and $\mathfrak{I}$ a model of $\mathfrak{f}$. We consider the implicant extractor $\mathfrak{syex}$ defined as follows:

$$\mathfrak{syex}(\mathfrak{f}) \stackrel{\text{def}}{=} \bigcup_{n_{\mathfrak{y}_i} \in \mathfrak{I}} \{\langle \mathfrak{y}_i^{n_{\mathfrak{y}_i}} \rangle\}$$

Proposition 219. *If* $\mathfrak{f}$ *is a satisfiable token-stack formula, the result of* $\mathfrak{sync}(\mathfrak{f})$ *is a representation set.*

Proof. The unicity of the base is ensured by the definition of the interpretation and the satisfiability of the active elements is trivial. □

Definition 220. We now define the generalization relation $\rightarrow_g$ we use token-stack formulas: $\mathfrak{y}^{\mathfrak{k}_1} \rightarrow_g \mathfrak{y}^{\mathfrak{k}_2}$ if and only if $\mathfrak{k}_1 < \mathfrak{k}_2$.

Definition 221. We define the domain of the associated atomic generalization function *Gen*: $(W, s) \in \textsc{Dom}(Gen)$ if and only if $\mathbf{active}(s)$ is a token-stack formula of the form $\mathfrak{y}^{\mathfrak{k}}$ and such that $\mathfrak{k} < n$ where n is the depth limit.

Proposition 222. *Let* $\mathfrak{f}$ *be a token-stack formula.* $\mathcal{S}(\textsc{MinimalPart}(\mathfrak{f}, Gen, \mathfrak{sync}(\mathfrak{f}, \langle 0 \mid \mathfrak{y} \in \mathfrak{Y} \rangle), \emptyset)) = \mathfrak{f}$.

This instantiation offers two major advantages for an experimental evaluation: first, it is easy to verify if a formula is a logical consequence of another (this can be checked with a number of integer comparisons linear in the size of the formula), second, it is easy to verify if the implementation works correctly by returning a minimal implicant of the initial formula, thanks to Proposition 222.

11.2 Implementation of the minimization algorithm

We now present an implementation of our implicant minimization algorithm. The MinimalPart algorithm has been implemented within the Abdulot framework [160], in an additional library called `minpart`. The implementation is generic in the two following senses. First, it can be plugged with any heuristic algorithm to generate the partitions on which the MinimalPart algorithm performs the recursive minimization. Two simple such heuristics implementations are also included in the library. Second, as presented in the theoretical approach, it is independent of the problem context, that is, the class of formulas we minimize and the associated generalization function and representation set extractors.

11.2.1 Implementing representation sets

We start by presenting the implementation of representation sets. We decided to implement the representation sets as vectors of integers, representing the number of times the generalization rules has been applied to the subformulas they contain. Given an initial satisfiable formula and a model for it provided at the beginning of the execution, we can deduce the number of elements its representation set will contain. Hence, this initial formula will be represented by a vector containing as many zeros as it has sequences of formulas. During an execution, all the representation sets for this problem will be vectors of the same size, allowing the integer at a given index in the vector to always represent the same generalization sequence, independently of the concrete representation set that is considered. In practice, all the representation sets generated for this execution will represent the same generalization sequence in different states. Every time a generalization

sequence is generalized, the value of the corresponding index within the corresponding vector will be increased by one. The actual formula this represents will then be computed from the initial formula and the number of times the generalization relation has to be applied by a `Context` component handling the concrete formulas considered and implementing the generalization rules (see Section 11.2.2).

In order to maintain some consistency between all the representation sets that will be used and required by the MinimalPart algorithm, we have implemented a representation set engine component. This component is designed to allocate the integer vectors when required, to perform the operations that can be computed directly on this structure (such as merging two representation sets) and forward any context-related operation to the `Context` component it will include. The current implementation allocates a new vector every time a new representation is needed. Note that, as various sequence indices may represent distinct generalization structures, the maximal depth of the vector representation of representation sets may differ from index to index. The fact that all the representation set-related operations are performed within the engine allows to prevent the framework to transfer integer vectors between its components. This is done by generating an identifier for each vector representation. This identifier will be the only representation transferred outside of this engine component and manipulated by the main algorithm.

Within the implementation, this engine is thus implemented as a C++ template `GSetEngine<Context>` which can be instantiated for any given `Context` component. It exposes methods to perform all the generalization-set related operations and only externalize the vectors it contains by their indices.

11.2.2 Context Interfaces

The `minpart` framework uses a `Context` component that will be tasked to perform all the operations that depend on the instantiation of the MinimalPart algorithm. These operations are the application of the generalization relation, the verification that a representation set is an implicant of the initial formula and that it is consistent, and an information access to the initial model that was initially minimized (returning data such as the size of the corresponding representation set). This context component has to be provided by the user of the `minpart` framework to apply the minimization algorithm in the instantiation they wish to solve. It will then be given information on the implicant to minimize (such as the formula it should be an implicant of) at the beginning of the execution of the MinimalPart algorithm. The interface that such a component must have is presented in Interface 11.1.

The Abdulot framework contains two instantiations of this component that are presented below.

Token-stack formulas

The implementation for the token-stack formulas is done internally by comparing the indices of generalizations of the symbols contained within a given formula. At the beginning of an execution, it is presented the generalization depth and the number of symbols that are considered and implements the verification rules as they are presented in Section

Interface 11.1: Generalization Context Interface Template

```
1 interface Context contains
2     Context : φ ⟶
3     getHypothesisSize : void ⟶ size ;
4     isLastGeneralization : vector<int>, index ⟶ {⊤, ⊥} ;
5     isGeneralizable : vector<int>, index ⟶ {⊤, ⊥} ;
6     isValidHypothesis : vector<int> ⟶ {⊤, ⊥} ;
7     isConsistentHypothesis : vector<int> ⟶ {⊤, ⊥} ;
8     print : vector<int> ⟶ void ;
```

11.1. Similarly, the generalization of a subformula will be performed by increasing its index until it reaches the maximal depth allowed.

Propositional logic

The implementation for propositional logic has been intertwined with the C++ library of MiniSAT [70]. The components takes a propositional formula expressed in the Dimacs format [60]. It starts by calling MiniSAT on this formula to obtain the first model and stores the resulting MiniSAT C++ instance in memory. Then, the component has to check whether a given subset of literals of the model it returns remains a model of the initial formula. Thanks to an instrumentation of the source code of MiniSAT, this can be performed without negating the input formula and querying a new satisfiability test. Indeed, the MiniSAT C++ instance stores enough information to deduce which clauses (if any) may stop being validated when a given literal is removed from the model. We therefore use this information to test if our representation sets are still models of the initial formula.

Separation logic

The implementation of the separation logic `Context` component has been implemented using the satisfiability procedure of CVC4 [150]. Similarly to what we did in Chapter 8, we intertwined the component with the C++ library of the SMT-solver. The component thus builds the logical formulas corresponding to the entailment checks and forward them to CVC4 to obtain the result. In order to build the formula corresponding to the representation set it is given, the component will have to know which generalization rule to apply for a given subformula. This is done using the initial formula that it is provided at the beginning of the execution. Indeed, this initial implicant and its corresponding representation set contain enough information for the `Context` to know which rule should be applied to each formula and which new formula should be obtained from them. Given the number of generalizations for a given index within the representation set, the component will take the formula corresponding to the given index within the initial implicant and apply the correct generalization rule accordingly, as many times as the number of generalizations instructs. This permits to obtain the corresponding active formula for the current representation set, and thus the formula to check the modelling and consistency properties for.

11.2.3 Generating partitions

Another major component of the `minpart` framework targets the generation of partitions and generalization blocks. As the actual method to generate such partitions was left open in Chapter 10, we decided to implement a generic component that can be instantiated with various generation methods. The external behaviour of the component is fairly simple: given the identifier of a representation set, it will generate and return a partition of this set, querying the representation set engine to know which subformulas can be generalized and which ones cannot. Similarly, the partition generator will be tasked to produced the generalization blocks we try before looping on the representation sets of the partition, which may or may not be equivalent.

Within the `minpart` framework, we implemented an instantiation of the partition generator. Our generator simply splits the vectors in a given n parts of the same size and generates a partition of size n corresponding to the generalization of all the formulas of each part. This decomposition in n parts can be done in the order of the vector indices, randomized at the beginning of the execution or randomized every time a new partition is requested. For each generalized component within the set of a partition, the atomic generalization function is applied a given k number of times.

11.2.4 Structure of the implementation

We finally describe the complete implementation model of the `minpart` framework, which is represented in Figure 11.3. It is available as a library and can be tried from two example executables for token-stack formulas and separation logic formulas.

Given an input consisting in an initial logical formula accompanied by its representation set representation, the framework forwards the formula to the context component to allow it to handle the modelling verification tests for it and instantiate an integer vector for its representation. The index of this vector is then sent to the minimization algorithm and will be used as the initial set within the minimization algorithm. The minimization component then executes the MinimalPart algorithm, requesting the partition generator for partitions and the representation set engine for cleaning, merging, and verifying representation sets. Depending on the results of the modelling tests, the minimization component will update its representation sets and support as detailed in the MinimalPart algorithm. Once a minimal set has been obtained, it will return it. The user (or the executable) should then apply the `Context` component one last time to get the corresponding logical formula.

11.2.5 Distribution

The whole `minpart` library, as well as two executables: one for set-symbols literals and one for separation logic, are publicly available within the Abdulot framework [160]. In order to use the separation logic context furnished, one should first install the C++ library of the CVC4 SMT-solver.

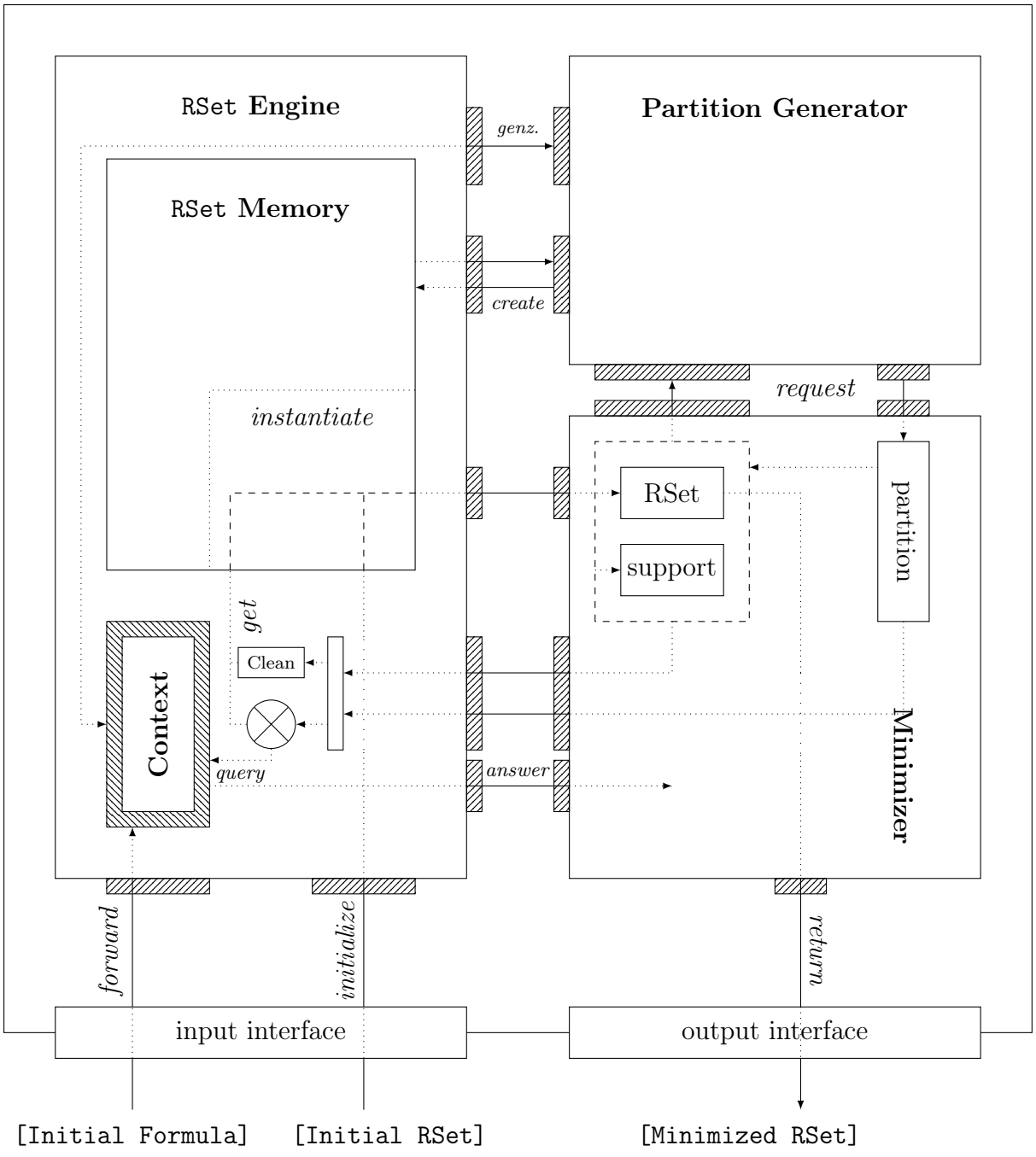


Figure 11.3: Structure of `minpart` within the ABDULOT framework

11.3 Testing the model minimization algorithm

We now present an experimental evaluation of the MinimalPart algorithm through this implementation framework. The experimental infrastructure is the same as the one presented in Chapter 9. In this first section, we evaluate the minimization algorithm using its token-stack formulas instantiation. As measuring an execution time is of no interest in this context, we will rather count the number of modelling tests that are made, as this is expected to be the most costly operation within the framework.

Throughout the section, the examples for which we look for a minimal model are randomly generated using the following process: we start by defining a set of symbols (of which we will abstract the content and give the size) and a depth limit, then we generate a formula containing all the provided symbols with a random generalization depth taken between zero and the depth limit. In all the examples, it has always been verified that the minimal model returned by the experiment was indeed equal to the randomly generated problem.

11.3.1 Shape of the problem

In Figure 11.4, we present the evolution of the number of satisfiability tests required to solve the minimization problem as a function of the size of the initial formula (as in the size of the initial representation set) when the depth limit is 10, the size of the partition 2 and the depth of atomic generalization function application 1. In Figure 11.5, we present the same value as a function of the depth limit when the size of the input formula is 100, the size of the partition 2 and the depth of atomic generalization function application 1. For all inputs, the experiments has been run on a 100 random problems of which the distribution is drawn. Note also that the partitions were generated using the most simple heuristic: separating the set into the given number of equally sized generalized sets. The theoretical mean number of satisfiability tests required by the naive method to solve the same problem with the given inputs is also represented in the figures, in the form of blue crosses. This naive method is recalled in Algorithm 11.2.

Algorithm 11.2: *Minimize*$(\mathfrak{f}, \mathfrak{Y})$

```
1 let ℑ ← ⟨0_i | η_i ∈ 𝔜⟩;
2 foreach η_i ∈ 𝔜 do
3     while ℑ ⊨ 𝔣 do
4         Increase the interger at index i in ℑ;
5 return ℑ;
```

Proposition 223. *Given a set of symbols of size s and a depth limit d, the mean number of satisfiability tests made by Algorithm 11.2 on the related token-stack formulas is $s(1 + \frac{d}{2})$.*

The results show that the MinimalPart algorithm is more efficient whenever the number of possible generalizations for a component is high. It also shows that the number of satisfiability tests it requires increases less than the ones of the naive algorithm

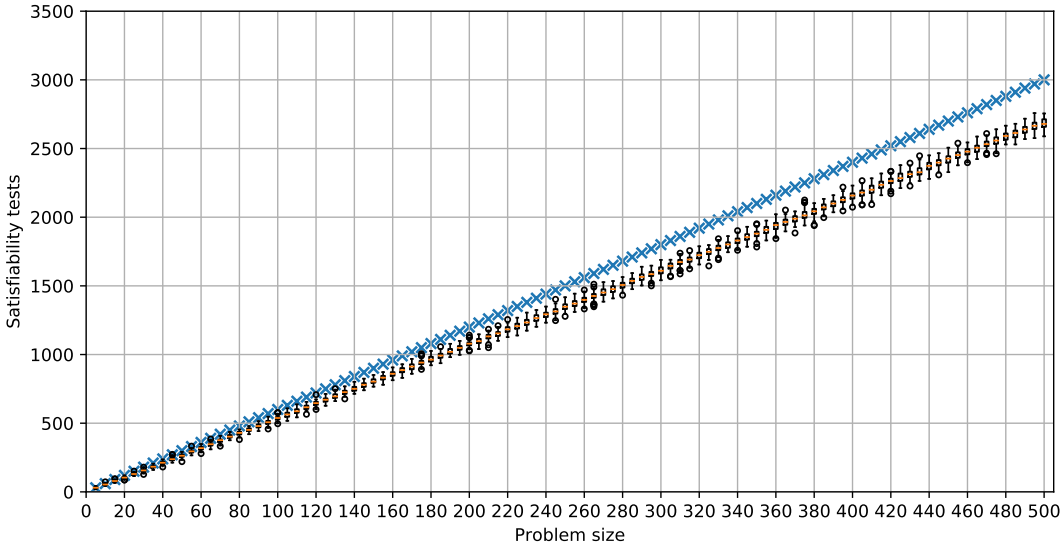


Figure 11.4: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas of a given size, for a depth limit of 10 and partitions of size 2 and of depth 1.

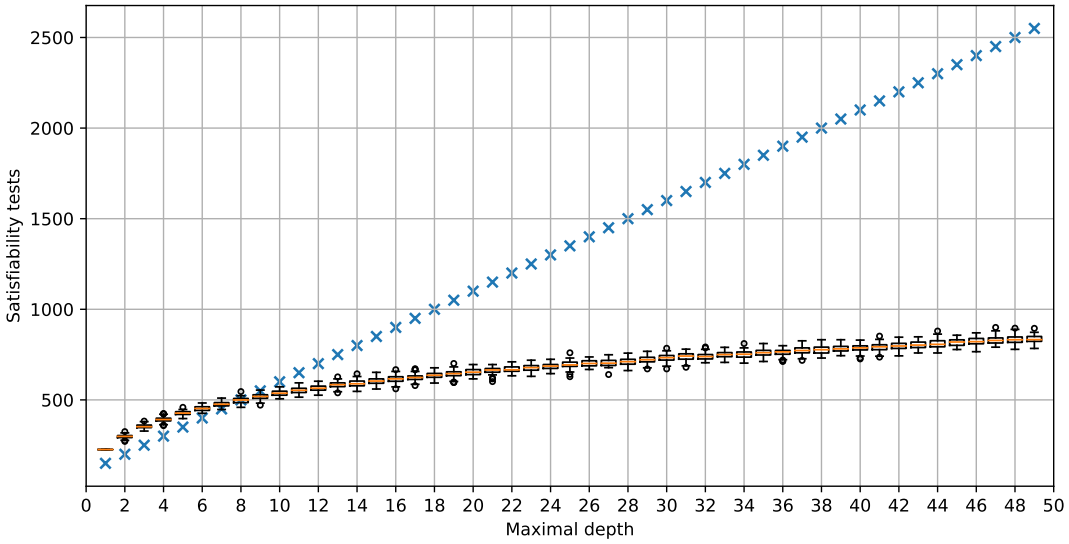
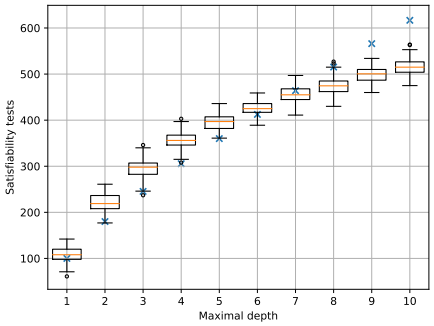
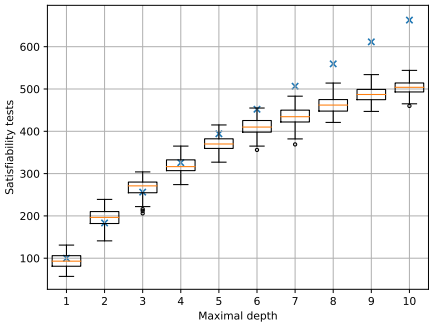


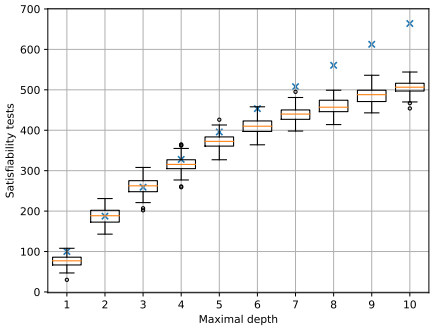
Figure 11.5: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas of a given depth limit, for a number of symbols of 100 and partitions of size 2 and of depth 1.



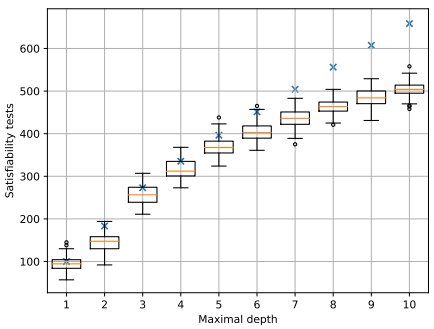
$P_v = [0.015, 0.061, 0.092, 0.104, 0.104,$
$0.104, 0.104, 0.104, 0.104, 0.104, 0.104]$



$P_v = [0.0055, 0.0270, 0.1075, 0.1075, 0.1075,$
$0.1075, 0.1075, 0.1075, 0.1075, 0.1075, 0.1075]$



$P_v = [0.004, 0.027, 0.057, 0.114, 0.114,$
$0.114, 0.114, 0.114, 0.114, 0.114, 0.114]$



$P_v = [0.004, 0.019, 0.076, 0.096, 0.115,$
$0.115, 0.115, 0.115, 0.115, 0.115, 0.115]$

Figure 11.6: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas of a given depth limit, for a number of symbols of 100 and partitions of size 2 and of depth 1 for various probability distributions.

when the size of the problem increases. Note that the fact that the MinimalPart algorithm invokes more satisfiability tests than the naive algorithm on uniformly randomized problems when the number of possible generalizations is low is expected. Indeed, when only few components can be generalized, the satisfiability queries trying to generalize many components at once will mostly fail, forcing the algorithm to fallback to base cases which corresponds to the satisfiability tests made by the naive algorithm. As it has already queried the satisfiability tests for these global generalization strategies, this will result in more tests in general. Figure 11.6 shows the distributions for a set of symbols of size 100 as a function of the maximal depth but for different generalization depth probabilities. It shows that our algorithm is also interesting when the components of a representation set have a good chance to be generalizable. Indeed, the higher this probability is, the more likely it is for the naive algorithm to perform much more redundant tests.

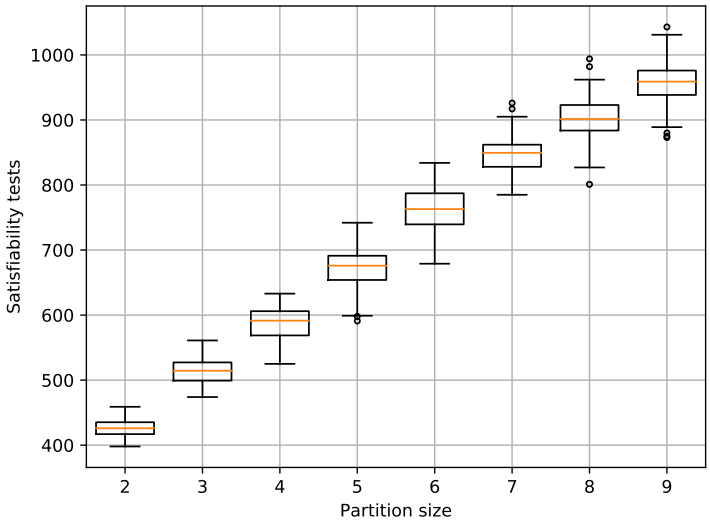


Figure 11.7: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas with a given partition size, for a number of symbols of 100, a depth limit of 5 and partitions of depth 1.

11.3.2 Partitions

We now study various heuristics for the generalization partitions we can use to minimize token-stack formulas.

In Figure 11.7 (resp. Figure 11.8), we present the distribution of the number of satisfiability tests required to minimize 100 token-stack formulas as a function of the size of the partition, for a set of symbols of size 100, a depth limit of 5 (resp. 10) and a generalization depth of 1 in the partitions.

We observe that in both cases, the number of satisfiability tests that is required to minimize random problems increases with the number of partitions. Once again, this can be explained by the fact that increasing the number of partitions increases the number of satisfiability tests that have to be made in the recursive calls of the MinimalPart algorithm. This also degrades the general performance of the algorithm as this can lead to redundant satisfiability queries when generalizing the many parts a partitions under the support of the others.

Figure 11.9 presents the influence of the generalization depth applied on the representation sets of the partition. Again, we draw the distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas built on 100 symbols with a depth limit of 10. The size of the partition is reset to 2 and the result is expressed as a function of the number of times the atomic generalization function is applied to each element the generalized representation set.

This shows that it can be interesting to generalize the components of the sets of a partition more than once to help the algorithm converge faster to a given solution. It also shows that, as expected, if the depth of this systematic generalization is too high,

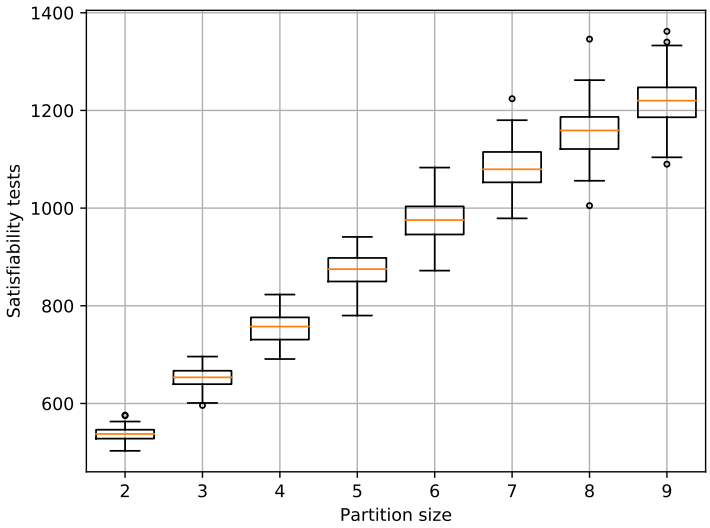


Figure 11.8: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas with a given partition size, for a number of symbols of 100, a depth limit of 10 and partitions of depth 1.

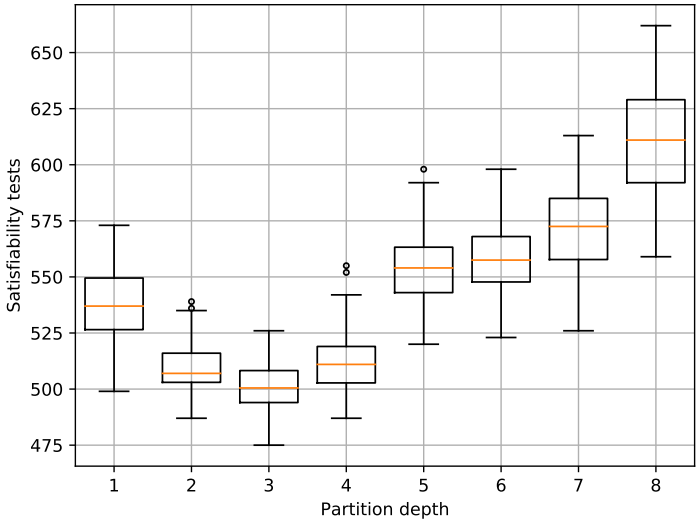


Figure 11.9: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas with a given partition depth, for a number of symbols of 100, a depth limit of 10 and partitions of size 2.

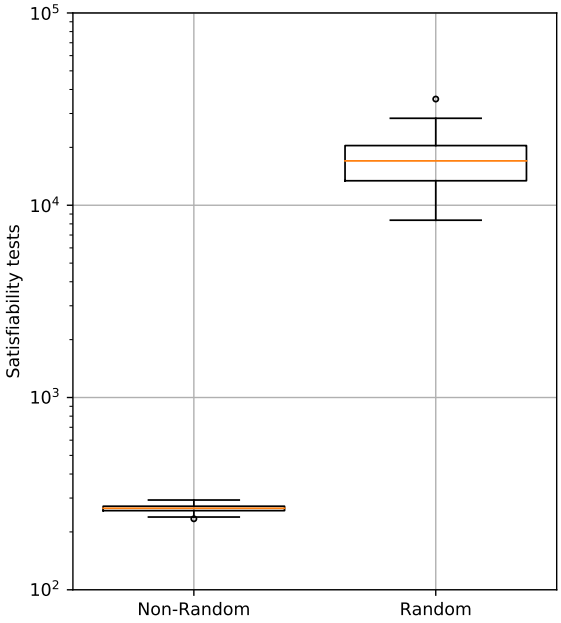


Figure 11.10: Distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas with or without randomizing the partitions, for a number of symbols of 100, a depth limit of 10 and partitions of size 2 and depth 1.

then the general performance will decrease as most of the satisfiability tests for these parts will fail (because we are past the maximal generalization possible for at least one component), and the algorithm will end up querying many unuseful tests before falling back into its base case and generalizing all the components independently.

Finally, Figure 11.10 shows the effect of choosing random sets of generalization sequences each time a new partition is needed. It shows the distribution of the number of satisfiability tests required to minimize 100 random token-stack formulas of size 50, with a depth limit of 10 and partitions of size 2 generalizing each of their components once. The distribution id drawn for both the case of random partitions and partition structure set at the beginning of the execution. Note that the number of satisfiability tests is presented in a logarithmic scale for this figure.

This clearly shows that randomizing the structure of the partitions each time is inefficient.

11.4 Testing the minimization of separation-logic formulas

We now apply our minimization algorithm in the context of separation logic. We start by trying the minimization algorithm on randomly generated simple separation logic

formulas. Such formulas are generated using Algorithm 11.3 where $\mathcal{V}$ denotes a set of variables and $\mathfrak{p}$ a depth probability, and by only keeping the generated formulas that are satisfiable and for which IEXTRACT returns an implicant. We use CVC4 to obtain an initial model for them and minimize the corresponding representation set.

Algorithm 11.3: $RandomSLFormula(\mathcal{V}, \mathfrak{p})$

```
if random() < p then
    let ∘ ⟵ randomChoice({∧, ⇒, ∗});
else
    let ∘ ⟵ randomChoice({ϕ, ↦, emp, ⊥});
if ∘ ∈ {∧, ⇒, ∗} then
    return RandomSLFormula(V, p) ∘ RandomSLFormula(V, p);
else if ∘ = ϕ then
    let ∘_e ⟵ randomChoice({=, ≠});
    let v_l ⟵ randomChoice(V);
    let v_r ⟵ randomChoice(V \ {v_l});
    return v_l ∘_e v_r;
else if ∘ = ↦ then
    let v_0 ⟵ randomChoice(V);
    let v_1, ..., v_n ⟵ randomChoices(V);
    return v_0 ↦ v_1, ..., v_n;
else if ∘ ∈ {emp, ⊥} then
    return ∘;
```

We now present the distribution of the number of satisfiability tests that are required to solve 78400 such randomly generated separation logic formulas with a number of variables ranging from 2 to 50 a depth probability ranging from 0.5 to 0.65. Note that following the results of the previous section, we decided to generate the structure of the partitions at the beginning of the execution only and that we used partitions of size 2 generalizing their components once. The mean minimization time required on these problems is at 2 seconds. The number of satisfiability tests is detailed as a function of the number of variables in Figure 11.11 and as a function of the depth probability in Figure 11.12 where the three hardest problems have been discarded (which required more than 1000 satisfiability tests). All the distribution subsets of these graphs are evenly sized.

The influence of the number of variables in the problem is expected as it results in larger generalization sets generated by IEXTRACT. It is interesting to remark that, considering the number of possible generalizations this instantiation offers (either one or two), the number of satisfiability tests required is significantly below the results we obtained for uniformly random problems. This is a consequence of the fact that most of the components can be generalized in the implicant we obtained, which in turn reduces the number of satisfiability required tests to ensure that representation sets with few generalized components still represent an implicant. This in turns permits to obtain the final result from less base-case generalizations, resulting in less satisfiability tests. It is important, however, to understand that part of this effect results from the simplification

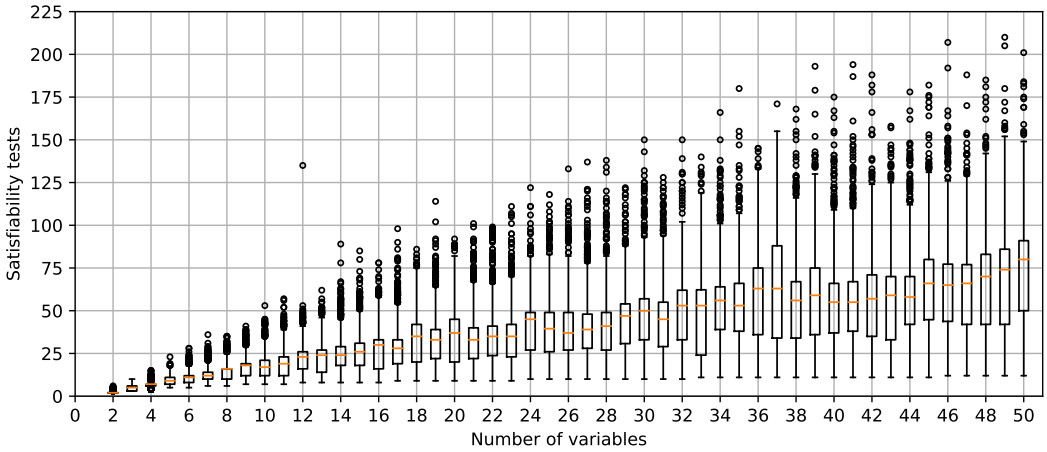


Figure 11.11: Distribution of the number of satisfiability tests required to minimize implicants of 78400 randomly generated separation logic formulas as a function of the number of variables in the random generation.

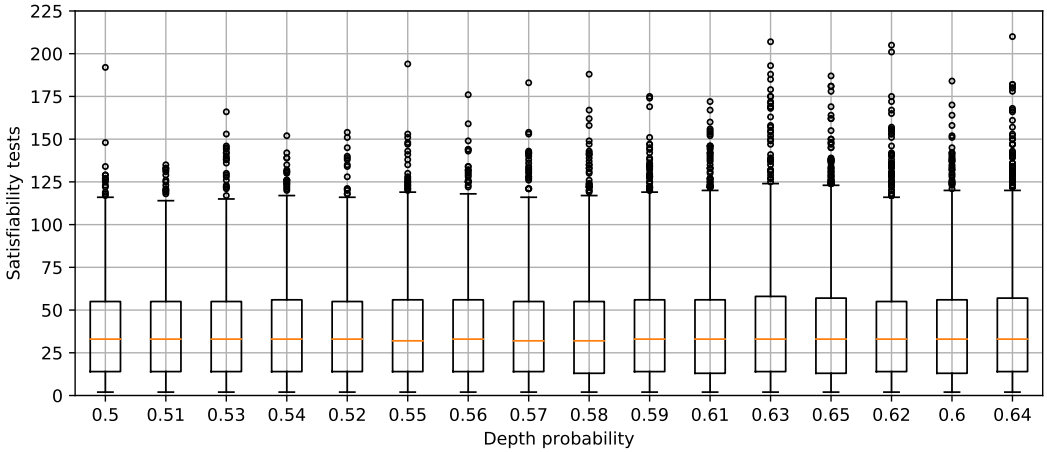


Figure 11.12: Distribution of the number of satisfiability tests required to minimize implicants of 78400 randomly generated separation logic formulas as a function of the probability depth in the random generation.

of the redundant equalities and disequalities between variables that are present in the initial implicant.

It is also interesting to notice that the depth probability of the random generalization seemingly has no influence on the number of satisfiability tests that are performed. This is due to the fact that it neither impacts the size of the representation set nor the maximal generalization depth. It has, however, an influence on the complexity of the formulas and thus on the time necessary to compute the satisfiability tests.

11.5 Wrapping the `minpart` for performing abduction

Finally, we evaluated an implementation of Algorithm 10.2 for computing abduction results in separation logic. We implemented this algorithm as an executable wrapper of the separation logic formula minimizer used in the previous section. In order to do perform its evaluation, we generated some random separation logic abduction problems using Algorithm 11.3 and we constructed formulas of the form $\mathit{RandomSLFormula}(\mathcal{V}, \mathfrak{p}) \mathrel{-\!\!*} \mathit{RandomSLFormula}(\mathcal{V}, \mathfrak{p})$. We checked that the obtained formula were indeed satisfiable and that the formula returned by IEXTRACT was an actual implicant before trying to minimize them. We generated 23900 such formulas, again with a depth probability ranging from 0.5 to 0.65 and a number of variables between 2 and 25. For 9600 of these problems, the algorithm was able to produce a minimal implicant in less than two minutes. The distribution of the corresponding examples is presented on Figure 11.13. Most of these problems correspond to formulas with only a small number of constraints, which results in easier satisfiability tests for the SMT-solver. Only few harder problems (those requiring more satisfiability tests) were solved by the system, contrarily to what we could observe on formulas not containing any separating implication. This shows that the impact of the computation cost of the satisfiability tests in this context prevents the algorithm to generate solution on problems that have more than a very limited number of constraints. This effect reduces the interest of the implicant minimization strategy to solve the bi-abduction problem by minimizing an initial implicant of the most general solution of the problem, as the practical computation cost of the required satisfiability tests appears to be too heavy.

11.6 Conclusion

Experiments show the practical use of the MINIMALPART on large representation sets, when the generalization function can produce many successive generalizations and on problems where the number of generalized components in the result is important. The experiments have additionally shown that out of these classes of problems, the satisfiability tests required by the decomposition become an overcost compared to the naive algorithm. The evaluation of its instantiation in separation logic has shown that it is able to generate minimal implicants of separation logic formulas from the models that are returned by CVC4 in a reasonable time, as long as these formulas do not include any quantification or separating implication. This last result compromises the direct application of the method for solving abduction problems as proposed by the SL-ABDUCE

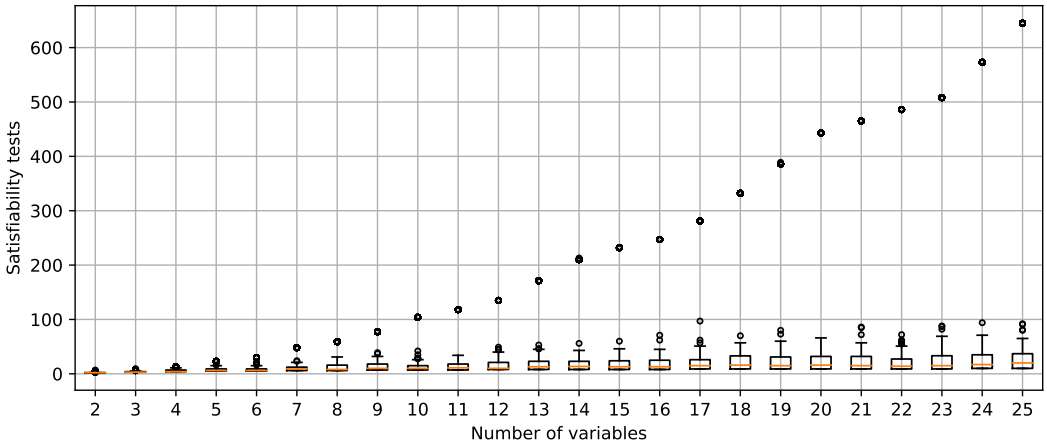


Figure 11.13: Distribution of the number of satisfiability tests required to minimize implicants of 9600 randomly generated separation logic formulas with separating implications as a function of the number of variables in the random generation.

algorithm. It could still be interesting to try or develop other satisfiability checkers for separation logic, aimed specifically at solving these types of formulas, instead of CVC4.

Conclusion

In the present work, we devised a theory-independent algorithm for abductive reasoning via implicate generation, over a set of user-defined abducible literals. This algorithm works by successively trying possible conjunctions of abducible literals until it can derive a contradiction with its input formula. It additionally uses an order on the literals and models of the formulas it works with to reduce as much as possible the number of conjunctions it has to try. These latter methods are mandatory to obtain an efficient algorithm. After presenting general definitions in Chapter 1, we proposed in Chapter 2 an overview of related prime implicate generation methods and of loop invariant generation. We then formally defined the proposed abduction method in Chapter 3 and showed that it is sound and complete. This means that it generates and returns the set of prime implicates of any given formula (up to redundancy). We then proposed in Chapter 4 an efficient structure for representing, storing and reusing the implicates computed by our algorithm.

In Part II, we applied this algorithm to program verification. We started by defining in Chapter 5 a generic framework for applying abductive reasoning in the context of program verification, inspired by the work of Dillig *et al.* [57]. Starting from an abstract representation of the structure of a program containing formulas to prove (assertions) and formulas that can be modified to obtain these proofs (such as candidate loop invariants), the algorithm uses external verification conditions generators to produce logical problems ensuring the validity of the assertions. Then it uses abduction, typically the prime implicated generator we devised in Part I, to infer interesting updates of the modifiable formulas of the program that should lead to a complete proof. This framework was instantiated in Chapter 6 to be applied to the generation of property-directed loop invariants of programs.

In Chapter 7, we investigated the issues related to the implementation of the methods introduced in the previous parts. We presented a generic structure that permits to obtain in practice both the abstraction of the theory in the prime implicate generator of Part I and the abstraction of the programs and verification conditions of the loop invariant generator of Part II. For the first abstraction, we designed generic SMT-solvers interface wrappers to plug the main system to any available satisfiability decision procedure. For the second one, we designed a program representation wrapper allowing the system to be plugged to any concrete program representation and associated verification tools. The issues related to the implementation of such interfaces were discussed in Chapter 8: we presented the design of SMT-solver interfaces based on both library API and SMT-LIB command-line interfaces for the widely used SMT-solvers CVC4, Z3, Alt-Ergo, VeriT, as well as a detailed program representation interface based on Why3 for an easy use in our program verification framework. We concluded Part III by performing an

experimental evaluation of both systems, on which additional discussion will be presented below.

Finally, Part IV was focused on our work on an algorithm for minimizing sets of variables with many possible values, which was inspired by a dichotomic propositional formula model minimizer [26]. This algorithm was formally defined in Chapter 10. We also provided an algorithm using this method to perform some sort abductive reasoning on separation logic formulas (relying on its separating conjunction instead of the usual boolean conjunction). These two algorithms were implemented and briefly evaluated in Chapter 11 where we found that the model minimizer was of practical interest but that its use in abduction for separation logic was relying on operators for which currently available separation logic satisfiability checkers are unfortunately unable to perform efficiently.

In summary, the main achievement of this work is the design of a prime implicate generation algorithm which is generic both in theory (through an abstraction of theory-related operations by satisfiability tests) and in practice (through the use of SMT-solvers interfaces) and its application to generate loop invariants of programs, even in non usually targeted theories. Experiments showed that this approach was able to compete with state-of-the-art propositional logic prime implicate generators for classes of highly constrained formulas and managed to generate more implicates than state-of-the-art approaches modulo theories under a given time. By combining this work with the Why3 platform to generate loop invariants, we additionally obtained a system able to check properties of WhyML programs and to automatically infer loop invariants for them. Practical results showed that, in its current state, the system is able to generate loop invariants of programs both on usual numerical domain and on more complex theories. It was typically able to generate loop invariants proving the result of exponentiation algorithms, list algorithms and floating point data handling algorithms. As practical work on abducible literals, solution guiding and information reuse are mostly inexistent in the current prototype, we consider these results very encouraging.

Future work

These results allowed us to extract three main point that have to be investigated to develop more efficient abduction procedures and facilitate their usage for solving practical problems. First, it is clear that the costly operations in our prime implicate generator are the satisfiability tests. Improving the efficiency of the satisfiability tests themselves stays clearly out of the scope of the project, though it is clear that by entirely delegating them to external SMT-solvers, we can always benefit from the improvements these tools receive. Reducing this cost within the ImpGen-PiD algorithm could be done by reducing the number of satisfiability tests we require. Methods should be investigated to cut more exploration branches from which we could be hinted that no relevant implicate will be derived. Obtaining more intelligence from the models we recover from satisfiability tests, such as the literals that appear on many models for instance, or deriving consequences beforehand from the information we already have, are possible solutions for this problem. This could be used to define a more relevant order, maybe built dynamically, that could help the algorithm build a search tree finding minimal implicates sooner. Depending on the application we target the implicate generation for, it could also be interesting to

investigate changes to the algorithm that would permit to cut more exploration branches at the expense of completeness, if such cuts can lead to the generation an implicate relevant to the problem faster. As a very simple illustration, counting the number of occurrences of a symbol in the source problem, or more generally, building some sort of correlation table exhibiting which symbols are tightly intertwined to the problem and which are less, could help us build heuristics in favour of their corresponding literals.

Second, as we previously mentioned, the use we made of abducible literals for the application to loop invariant generation was mostly blind and contained many systematic literals that would immediately appear as irrelevant to any human user. It is thus clear that work should be done to devise more interesting and knowledgable methods to generate and use abducible literals. This includes pruning literals, for instance from contextual information extracted from the program, or obtained for larger knowledge bases on program invariants. It could also be interesting, in a more distant perspective, to develop methods generating higher-level literals that could permit to generate solutions for problems requiring higher level properties than the one directly present within the program. Merging such a literal generator with a method to extract interesting predicates that can represent relevant properties of a program could thus also permit to generate invariants for more complex programs. Other than that, it will also be important to look for techniques, heuristics or learning procedures, to order the abducible literals more efficiently such that those we expect to lead to the generation of a relevant solution are tried first. Within the loop invariant generation algorithm, this ordering should also be different for each call to GPID, as we try to strengthen possibly very different abduction problems with them.

Third, it is clear that within its application to loop invariant generation, many redundant computations are performed (on similar abduction problems, between the verification condition checks performed within the program prover and those within the prime implicate generator, *etc.*) and could thus be reduced. Possible ways to reduce redundant computations include methods to reuse information obtained from the verification conditions checks performed by the program prover to guide the exploration within the ImpGen-PID algorithm. It should also be possible to extract from the program some additional properties that could be forwarded to the prime implicate generation and used there to cut exploration branches or, as previously suggested, reorder abducible literals. This could then be used to hint GPID not to generate implicates that appear relevant in the current problem but that we can detect cannot produce a solution in the broader context. The need for such information retrieval from the programs additionally suggests that a better integration between the static analyser and the abductive reasoning tool would be beneficial.

Additionally, it would be interesting to refine the loop invariant generation algorithm in order to intertwine it better with its abductive component. This could typically ease the implementation of our second and third points, as the two algorithms follow a similar structure and could probably share parts of their execution (*e.g.* for a program with only one loop invariant, GPID could be adapted to the loop invariant construction directly, confounding its satisfiability tests with the ones made by the program checker). This could also help switch between SMT-solvers within the GPID algorithm, and taking advantage of this functionality present in Why3 to be able to derive implicates more automatically, without the need to previously select the most adapted tool for the problem at hand.

It could also be interesting to tweak our method to integrate it in a semi-automatic infrastructure where it could communicate with a user, sending them information on the failure of the tasks to help them tweak the inputs to solve the problem. Finally, it is clear that the higher the expressivity of the properties we want to derive, the more we need methods to deduce contextual information to reduce the search space. In the long run, such methods will be necessary to increase further the scope of the program we can verify.

Appendix A

Grammar of the abducible file format used by the ABDULOT framework

⟨*file*⟩ ::= ⟨*command*⟩*

⟨*command*⟩ ::= ⟨*sizecommand*⟩
| ⟨*abducecommand*⟩
| ⟨*refcommand*⟩
| ⟨*extendcommand*⟩
| ⟨*declarecommand*⟩
| ⟨*lambdacommand*⟩
| ⟨*applycommand*⟩
| ⟨*copycommand*⟩
| ⟨*annotatecommand*⟩

⟨*sizecommand*⟩ ::= '`(`' ⟨*sizekw*⟩ ('`auto`' | INT) '`)`'

⟨*sizekw*⟩ ::= '`size`'
| '`rsize`'

⟨*abducecommand*⟩ ::= '`(`' ⟨*abducekw*⟩ ⟨*data*⟩+ '`)`'

⟨*abducekw*⟩ ::= '`abduce`'
| '`abducible`'

⟨*refcommand*⟩ ::= '`(`' ⟨*refkw*⟩ ⟨*data*⟩+ '`)`'

⟨*refkw*⟩ ::= '`reference`'
| '`ref`'

⟨*extendcommand*⟩ ::= '`(`' ⟨*extendkw*⟩ ⟨*annot*⟩+ '`)`'

⟨*extendkw*⟩ ::= '`extend`'

⟨*declarecommand*⟩ ::= '`(`' ⟨*declarekw*⟩ ⟨*data*⟩+ ⟨*askw*⟩ ⟨*annot*⟩+ '`)`'

⟨*declarekw*⟩ ::= ‘`declare`’

⟨*askw*⟩ ::= ‘`as`’

⟨*lambdacommand*⟩ ::= ‘`(`’ ⟨*lambdakw*⟩ ⟨*annot*⟩ ‘`(`’ (⟨*annot*⟩)* ‘`)`’ ⟨*data*⟩ ‘`)`’

⟨*lambdakw*⟩ ::= ‘`lambda`’
| ‘`declare-lambda`’

⟨*applycommand*⟩ ::= ‘`(`’ ⟨*applykw*⟩ ⟨*annot*⟩ ‘`(`’ ⟨*overkw*⟩ (⟨*data*⟩ | ⟨*strictkw*⟩ ⟨*data*⟩)* ‘`)`’
⟨*annot*⟩ ⟨*annot*⟩*‘`)`’

⟨*applykw*⟩ ::= ‘`apply`’

⟨*overkw*⟩ ::= ‘`over`’

⟨*strictkw*⟩ ::= ‘`strict`’

⟨*copycommand*⟩ ::= ‘`(`’ ⟨*copykw*⟩ ⟨*annot*⟩+ ⟨*askw*⟩ ⟨*annot*⟩+ ‘`)`’

⟨*copykw*⟩ ::= ‘`copy`’

⟨*annotatecommand*⟩ ::= ‘`(`’ ⟨*annotatekw*⟩ ⟨*annot*⟩ ⟨*data*⟩+ ‘`)`’

⟨*annotatekw*⟩ ::= ‘`annotate`’

⟨*annot*⟩ ::= any alphanumeric identifier.

⟨*data*⟩ ::= any unspaced text (or any text embedded in parentheses).

Appendix B

WHY3 verification condition used as an abduction problem in Example 152

```
;;; generated by SMT-LIB2 driver
;;; SMT-LIB2 driver: bit-vectors, common part
;;; SMT-LIB2: integer arithmetic
(declare-sort uni 0)

(declare-sort ty 0)

(declare-fun sort (ty uni) Bool)

(declare-fun witness (ty) uni)

;; witness_sort
(assert (forall ((a ty)) (sort a (witness a))))

(declare-fun int () ty)

(declare-fun real () ty)

(declare-fun bool () ty)

(declare-fun match_bool (ty Bool uni uni) uni)

;; match_bool_sort
(assert
(forall ((a ty))
(forall ((x Bool) (x1 uni) (x2 uni)) (sort a (match_bool a x x1 x2)))))

;; match_bool_True
(assert
(forall ((a ty))
(forall ((z uni) (z1 uni)) (=> (sort a z) (= (match_bool a true z z1) z)))))

;; match_bool_False
(assert
(forall ((a ty))
(forall ((z uni) (z1 uni))
(=> (sort a z1) (= (match_bool a false z z1) z1)))))
```

```
(declare-fun index_bool (Bool) Int)

;; index_bool_True
(assert (= (index_bool true) 0))

;; index_bool_False
(assert (= (index_bool false) 1))

;; bool_inversion
(assert (forall ((u Bool)) (or (= u true) (= u false))))

(declare-sort tuple0 0)

(declare-fun tuple01 () ty)

(declare-fun Tuple0 () tuple0)

;; tuple0_inversion
(assert (forall ((u tuple0)) (= u Tuple0)))

;; CompatOrderMult
(assert
(forall ((x Int) (y Int) (z Int))
(=> (<= x y) (=> (<= 0 z) (<= (* x z) (* y z))))))

(declare-fun ref (ty) ty)

(declare-fun mk_ref (ty uni) uni)

;; mk ref_sort
(assert (forall ((a ty)) (forall ((x uni)) (sort (ref a) (mk_ref a x)))))

(declare-fun contents (ty uni) uni)

;; contents_sort
(assert (forall ((a ty)) (forall ((x uni)) (sort a (contents a x)))))

;; contents_def
(assert
(forall ((a ty))
(forall ((u uni)) (=> (sort a u) (= (contents a (mk_ref a u)) u)))))

;; ref_inversion
(assert
(forall ((a ty))
(forall ((u uni)) (=> (sort (ref a) u) (= u (mk_ref a (contents a u)))))))

(declare-fun t () Int)

;; H
(assert (not (< t 10)))

(assert
;; VC_main
```

```
;; File "tenten/../tenten.mlw", line 5, characters 5-9
(not (= t 10)))
(check-sat)
```

Appendix C

User manual for the implementation

C.1 Distribution

The Abdulot framework contains the implementation of all the algorithms that are presented in this thesis, as well as the additional libraries, tools and interfaces necessary to compile it and plug it to SMT-solvers and to the Why3 platform. It is publicly available on GitHub (https://github.com/sellamiy/GPiD-Framework) under the BSD 3cl licence. This appendix describes all the information necessary to compile it and make it run on some minimal examples.

C.2 Structure of the repository

We detail in this section the content of the directories the sources of the Abdulot framework contain.

C.2.1 Directories

- `cmake/include` : Local CMake sources to help in the global compilation structure.
- `cmake/modules` : Additional CMake sources to detect and configure external tools such as SMT-solvers.
- `cmake/patches` : MiniSAT patches necessary for building the MiniSAT solver interface.
- `framework` : Sources of the structures and algorithms of the geneiric prime implicate generator and loop invariant generator.
- `iphandle` : Sources of the program abstraction interfaces.
- `iphandle/why3-wrapped` : Sources of the Why3 interface.
- `lib/antlr4` : CMake sources for recovering and compiling ANTLR [136].
- `lib/cxxopts` : CMake sources for recovering and compiling CxxOpts (https://github.com/jarro2783/cxxopts).

- `lib/lcdot` : Graph library, used for drawing execution search trees.
- `lib/lisptp` : LISP parser, used in particular to parse SMT-LIB and abducible files.
- `lib/minpart` : Implementation of the MINIMALPART algorithm and of its associated structures.
- `lib/smtlib2tools` : Set of tools for handling SMT-LIB problems.
- `lib/snlog` : Logging library.
- `lib/starray` : Static emptyable arrays library (see also [161]).
- `lib/stdutils` : Miscellaneous C++ 11 utilities.
- `lib/ugly` : Miscellaneous algorithms.
- `lib/why3cpp` : Set of tools for handling WHY3 calls and WHYML programs.
- `mpcontext` : Sources of context components for the minpart algorithm.
- `mpcontext/prop-minisat` : Sources of propositional minpart context.
- `mpcontext/sl-cvc4` : Sources of separation logic minpart context.
- `solvers` : Sources of the solver interfaces.
- `test` : Some tests.
- `tools` : Sources of the example executables used for the evaluation.
- `utils` : Miscellaneous scripts, used in particular to configure the solver interfaces during the compilation.

C.2.2 Compiling

The compilation is performed via CMake. The configuration accepts the following options:

- `BUILD_TESTS` (default `ON`) : Build the executables performing tests on the algorithms of the framework.
- `INSTRUMENTATION` (default `OFF`) : Build the framework with instrumentation instructions that allow to draw the exploration graphs of the algorithms and permit to verify the intermediate results of the computation. Can also significantly reduce the execution speed.
- `BUILD_GPID` (default `ON`) : Build the prime implicate generator executables.
- `BUILD_ILINVA` (default `ON`) : Build the loop invariant generator executables. the use of the option `BUILD_GPID=ON` is mandatory when if `BUILD_ILINVA` is set.

- `BUILD_MINPART` (default `ON`) : Build the contextualized implicant minimizer based on `minpart`.
- `SKIP_SOLVER_INTERFACE` (list of interfaces) : Allows one to skip the compilation and generation of a given solver interface (as well as of the corresponding executables). For instance, to skip the generation of the Z3 interface, one can add the configuration option `-SKIP_SOLVER_INTERFACE=z3`. Note that when an SMT-solver is not present in the system, the generation of the corresponding interfaces is automatically skipped.
- `SKIP_ILINVA_HANDLER` (list of interfaces) : Identical to the previous option but for the program representation interfaces of the loop invariant generator.
- `SKIP_MINPART_CONTEXT` (list of interfaces) : Identical to the previous option but for the `minpart` context components.

All the libraries of the framework are built in the `lib` directory of the build directory. Similarly, all the executables are built in the `bin` directory of the build directory.

C.2.3 Configuring Why3

The Ilinva test executable of the Abdulot uses the Why3 platform to generate and check the verification conditions of programs. This requires that the Why3 platform is installed and correctly configured before starting the compilation of the tool. Why3 can be installed via the OCaml package manager OPAM. Before compiling its Ilinva interface, it is also mandatory to make sure that Why3 can use at least one SMT-solver on the system and that it has been configured for it (by running `why3 config` once).

C.2.4 Executables

The sample executables that are generated by the compilation of the while framework are listed below. Their correspondence to the algorithms presented in this thesis are also mentioned.

- `abdulot-parser` : Executable to parse and check abducible files.
- `gpid-lcvc4` : GPiD algorithm plugged to the C++ library version of CVC4.
- `gpid-ccvc4` : GPiD algorithm plugged to the SMT-lib interface of CVC4.
- `gpid-verit` : GPiD algorithm plugged to the SMT-lib interface of VeriT.
- `gpid-lz3` : GPiD algorithm plugged to the C++ library version of Z3.
- `gpid-cz3` : GPiD algorithm plugged to the SMT-lib interface of Z3.
- `gpid-saltergo` : GPiD algorithm plugged to the SMT-lib interface of Alt-Ergo.
- `gpid-minisatp` : GPiD algorithm plugged to the C++ library of MiniSAT.
- `gpid-tisi` : Compilation artifact; do not use.

- `gpid` : GPID algorithm plugged to all the available interfaces. The interface to use has to be selected through a command-line option.
- `smt2abduce` : Script wrapping to help the generation of abducible literals.
- `ilinva-why3-wrapped` : ILINVA algorithm plugged to WHY3.
- `smtlib2-tools` : Miscellaneous fun command line utilities to use on SMT-LIB files.
- `why3-tools` : Miscellaneous fun but hard to use command line helpers to use on WHYML files.
- `minpart-lits` : `minpart` algorithm instantiation used to minimize a random token-set-formula.
- `minpart-prop-minisat` : `minpart` algorithm instantiation used to minimize propositional models.
- `minpart-sl-cvc4` : `minpart` algorithm instantiation used to minimize separation logic formulas.
- `abdulot-select` : High-level framework wrapper to use to encapsulate calls to any of the executables of the ABDULOT framework. May also be configured by CMake to encapsulate a single ABDULOT executable. May be a Bash script or a Python3 script depending on the system and on the configuration.

C.3 Running simple examples

We present in this section simple command-line examples to use the executables of the framework to perform some very simple, non-configured, tasks.

C.3.1 Generate propositional logic implicates

To generate the implicates of a propositional logic formula (assuming it is in the DIMACS format and can be parsed by MINISAT), one can use the `gpid-minisatp` tool. Assuming the DIMACS problem is called `problem.dimacs`, one can run `gpid-minisatp -i problem.dimacs -autogen-abducibles all -P` to generate all the prime implicates of the problem and print them on the terminal.

C.3.2 Generate the implicates of an SMT-LIB file

Given an SMT-LIB problem `problem.smt2` and a corresponding set of abducible literals `problem.abd`, one can run, *e.g.*, `gpid-lcvc4 -i problem.smt2 -l problem.abd -P` to generate all the prime `problem.abd`-implicates of the problem.

C.3.3 Generate the loop invariants of a WhyML program

Given a WhyML program `prog.mlw` and a set of core abducible literals extracted from it `prog.abd`, one can run, *e.g.*, `ilinva-why3-wrapped -i prog.mlw -a prog.abd -g cz3 -H "solver:z3" -c -smt-time-limit=1` to try to generate satisfying loop invariants for the program of `prog.mlw`.

Index

Bibliography

[1] S. B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, (6):509–516, 1978.

[2] N. Azmy and C. Weidenbach. Computing tiny clause normal forms. In *Automated Deduction - CADE-24 - 24th International Conference on Automated Deduction, Lake Placid, NY, USA, June 9-14, 2013. Proceedings*, pages 109–125, 2013.

[3] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *J. Log. Comput.*, 4:217–247, 1994.

[4] L. Bachmair and H. Ganzinger. Resolution theorem proving. *Handbook of Automated Reasoning*, 1, 02 2001.

[5] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 203–213, New York, NY, USA, 2001. ACM.

[6] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In W. A. Hunt and F. Somenzi, editors, *Computer Aided Verification*, pages 118–121, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovi'c, T. King, A. Reynolds, and C. Tinelli. Cvc4. In G. Gopalakrishnan and S. Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, jul 2011. Snowbird, Utah.

[8] C. Barrett, P. Fontaine, and C. Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB), 2016. Available at `http://smtlib.cs.uiowa.edu/benchmarks.shtml`.

[9] C. Barrett, P. Fontaine, and C. Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at www.SMT-LIB.org.

[10] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009.

[11] C. Benzmüller and T. Raths. Hol based first-order modal logic provers. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 127–136, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[12] C. Benzmüller, J. Otten, and T. Raths. Implementing and evaluating provers for first-order modal logics. volume 242, 01 2012.

[13] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 52–68, Berlin, Heidelberg, 2005. Springer-Verlag.

[14] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant synthesis for combined theories. 01 2007.

[15] M. Bienvenu. Prime implicates and prime implicants in modal logic. In *Proceedings of the 22Nd National Conference on Artificial Intelligence - Volume 1*, AAAI'07, pages 379–384. AAAI Press, 2007.

[16] M. Bienvenu. Prime implicates and prime implicants: From propositional to modal logic. *J. Artif. Int. Res.*, 36(1):71–128, Sept. 2009.

[17] G. Bittencourt. Combining syntax and semantics through prime form representation. *J. Log. Comput.*, 18:13–33, 02 2008.

[18] P. Blackburn, J. Van Benthem, and F. Wolter. *Handbook of Modal Logic*. Springer, 2006.

[19] J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending sledgehammer with SMT solvers. In *Proceedings of the 23rd International Conference on Automated Deduction*, CADE'11, pages 116–130, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. The Why3 platform 1.2.0, Jan. 2019. Tutorial and Reference Manual; available at http://why3.lri.fr/manual.pdf.

[21] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011. https://hal.inria.fr/hal-00790310.

[22] S. Boldo. Gallery of proved programs. Available at https://www.lri.fr/~sboldo/research.html.

[23] G. Boole. *The Calculus of Logic*. 1848.

[24] T. Bouton, D. Oliveira, D. Déharbe, and P. Fontaine. verit: An open, trustable and efficient smt-solver. pages 151–156, 08 2009.

[25] M. Bozga, P. Habermehl, R. Iosif, F. Konecný, and T. Vojnar. Automatic verification of integer array programs. In *CAV*, 2009.

[26] A. Bradley and Z. Manna. Checking safety by inductive generalization of counterexamples to induction. pages 173–180, 12 2007.

[27] A. Bradley and Z. Manna. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20:379–405, 07 2008.

[28] A. R. Bradley. Sat-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 70–87, Berlin, Heidelberg, 2011. Springer-Verlag.

[29] A. R. Bradley. Ic3 and beyond: Incremental, inductive verification. In *CAV*, 2012.

[30] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[31] D. Brand. Proving theorems with the modification method. *SIAM Journal on Computing*, 4(4):412–430, 1975.

[32] J. Brotherston, N. Gorogiannis, M. Kanovich, and R. Rowe. Model checking for symbolic-heap separation logic with inductive predicates. volume 51, pages 84–96, 04 2016.

[33] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. pages 174–177, 03 2009.

[34] R. Caferra. *Logic for computer science and artificial intelligence*. 2013.

[35] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. volume 44, pages 289–300, 01 2009.

[36] M. Cialdea Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Logic Journal of IGPL*, 1, 01 1997.

[37] A. Cimatti, A. Griggio, S. Mover, and S. Tonetta. Ic3 modulo theories via implicit predicate abstraction. 10 2013.

[38] M. Colón, S. Sankaranarayanan, and H. B. Sipma. Linear invariant generation using non-linear constraint solving. In *CAV*, 2003.

[39] S. Conchon, A. Coquereau, M. Iguernlala, and A. Mebsout. Alt-Ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, Oxford, United Kingdom, July 2018.

[40] T. Coquand and G. Huet. The calculus of constructions. Technical Report RR-0530, INRIA, May 1986.

[41] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. volume 3576, pages 462–475, 07 2005.

[42] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. pages 238–252, 01 1977.

[43] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *European Symposium on Programming*, pages 21–30. Springer, 2005.

[44] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 84–96, New York, NY, USA, 1978. ACM.

[45] C. Csallner, N. Tillmann, and Y. Smaragdakis. Dysy: Dynamic symbolic execution for invariant inference. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 281–290, New York, NY, USA, 2008. ACM.

[46] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c: A software analysis perspective. In *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag.

[47] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, July 1962.

[48] M. Davis and H. Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, July 1960.

[49] J. de Kleer. An improved incremental algorithm for generating prime implicates. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92, pages 780–785. AAAI Press, 1992.

[50] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[51] L. De Moura and N. Bjørner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, Sept. 2011.

[52] A. Degtyarev and A. Voronkov. Equality elimination for the tableau method. In *DISCO*, 1996.

[53] W. Del-Pinto and R. A. Schmidt. Forgetting-based abduction in alc. In *SOQE*, 2017.

[54] W. Del-Pinto and R. A. Schmidt. Extending forgetting-based abduction using nominals. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems*, pages 185–202, Cham, 2019. Springer International Publishing.

[55] I. Dillig and T. Dillig. Explain: A tool for performing abductive inference. In *CAV*, 2013.

[56] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 181–192, New York, NY, USA, 2012. ACM.

[57] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive invariant generation via abductive inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 443–456, New York, NY, USA, 2013. ACM.

[58] I. Dillig, T. Dillig, K. McMillan, and A. Aiken. Minimum satisfying assignments for smt. pages 394–409, 07 2012.

[59] T. Dillig, I. Dillig, K. McMillan, and A. Aiken. Mistral smt solver. Available at http://www.cs.utexas.edu/~tdillig/mistral/index.html.

[60] Dimacs challenge - satisfiability: Suggested format, 2008. Available at https://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/satformat.ps.

[61] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'06, pages 287–302, Berlin, Heidelberg, 2006. Springer-Verlag.

[62] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27:1165 – 1178, 08 2008.

[63] J. Du, H. Wan, and H. Ma. Practical tbox abduction based on justification patterns. In *AAAI*, 2017.

[64] B. Dutertre and L. de Moura. A fast linear-arithmetic solver for dpll(t). In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 81–94, Berlin, Heidelberg, 2006. Springer-Verlag.

[65] M. Echenim, N. Peltier, and Y. Sellami. A Generic Framework for Implicate Generation Modulo Theories. In *IJCAR*, Oxford, United Kingdom, July 2018.

[66] M. Echenim, N. Peltier, and Y. Sellami. Ilinva: Using abduction to generate loop invariants. In A. Herzig and A. Popescu, editors, *Frontiers of Combining Systems*, pages 77–93, Cham, 2019. Springer International Publishing.

[67] M. Echenim, N. Peltier, and S. Tourret. An approach to abductive reasoning in equational logic. pages 531–537, 08 2013.

[68] M. Echenim, N. Peltier, and S. Tourret. Quantifier-free equational logic and prime implicate generation. volume 9195, pages 311–325, 08 2015.

[69] M. Echenim, N. Peltier, and S. Tourret. Prime implicate generation in equational logic. *J. Artif. Int. Res.*, 60(1):827–880, Sept. 2017.

[70] N. Eén and N. Sörensson. An extensible sat-solver. In *SAT*, 2003.

[71] T. Eiter and G. Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, Jan. 1995.

[72] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 213–224, New York, NY, USA, 1999. ACM.

[73] B. Errico, F. Pirri, and C. Pizzuti. Finding prime implicants by minimizing integer programming problems. *Proceedings of the 8th Australian Joint Conference on Artificial Intelligence*, pages 355–362, 01 1995.

[74] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. pages 173–177.

[75] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.

[76] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for esc/java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517, Berlin, Heidelberg, 2001. Springer-Verlag.

[77] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[78] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, C. Tinelli, R. Alur, and D. Peled. Dpll(t): Fast decision procedures. 06 2004.

[79] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[80] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by smt solving: Termination and invariant synthesis. *Logical Methods in Computer Science*, 6, 10 2010.

[81] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *Proceedings of the 13th International Conference on Static Analysis*, SAS'06, pages 144–160, Berlin, Heidelberg, 2006. Springer-Verlag.

[82] S. Graf and H. Saïdi. Construction of abstract state graphs with pvs. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

[83] A. Griggio. A practical approach to satisfiability modulo linear integer arithmetic. *JSAT*, 8:1–27, 01 2012.

[84] A. Gupta. Formal hardware verification methods: A survey. In *Computer-Aided Verification*, pages 5–92. Springer, 1992.

[85] A. Gupta and A. Rybalchenko. Invgen: An efficient invariant generator. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 634–640, Berlin, Heidelberg, 2009. Springer-Verlag.

[86] C. Haase. A survival guide to presburger arithmetic. *ACM SIGLOG News*, 5(3):67–82, July 2018.

[87] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 339–348, New York, NY, USA, 2008. ACM.

[88] L. Henocque. The prime normal form of boolean formulas, Mar 2002. Available at http://laurent.henocque.com/oldsite/element/2002Henocque-PNF-ResearchReport/RR-LSIS-02-002.pdf.

[89] J. Henry. *Static analysis of program by Abstract Interpretation and Decision Procedures*. PhD thesis, 2014. Thèse de doctorat dirigée par Monniaux, David et Moy, Matthieu Informatique Grenoble 2014.

[90] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Proceedings of the 10th International Conference on Model Checking Software*, SPIN'03, pages 235–239, Berlin, Heidelberg, 2003. Springer-Verlag.

[91] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[92] K. Hoder and N. Bjørner. Generalized property directed reachability. In A. Cimatti and R. Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 157–171, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[93] H. H. Hoos and T. Stützle. Satlib: An online resource for research on sat. pages 283–292. IOS Press, 2000.

[94] H. H. Hoos and T. Stützle. Satlib: Benchmarks, 2000. Available at https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[95] R. Hähnle. Tableaux and related methods. *Handbook of Automated Reasoning*, 1, 12 2001.

[96] K. Inoue. Consequence-finding based on ordered linear resolution. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'91, pages 158–164, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[97] K. Iwanuma, H. Nabeshima, and K. Inoue. Toward an efficient equality computation in connection tableaux: A modification method without symmetry transformation. In *FTP*, 2009.

[98] S. Jabbour, J. Marques-Silva, L. Sais, and Y. Salhi. Enumerating prime implicants of propositional formulae in conjunctive normal form. pages 152–165, 09 2014.

[99] P. Jackson. Computing prime implicates incrementally. In *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, pages 253–267, London, UK, UK, 1992. Springer-Verlag.

[100] P. Jackson and J. Pais. Computing prime implicants. In *Proceedings of the Tenth International Conference on Automated Deduction*, CADE-10, pages 543–557, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[101] J. R. Josephson and S. G. Josephson. *Abductive inference: Computation, philosophy, technology*. Cambridge University Press, 1996.

[102] D. Kapur. A quantifier-elimination based heuristic for automatically generating inductive assertions for programs. *Journal of Systems Science and Complexity*, 19:307–330, 01 2006.

[103] A. Karbyshev, N. Bjørner, S. Itzhaky, N. Rinetzky, and S. Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM*, 64:1–33, 03 2017.

[104] E. G. Karpenkov, D. Monniaux, and P. Wendler. Program analysis with local policy iteration. In *Proceedings of the 17th International Conference on Verification, Model Checking, and Abstract Interpretation - Volume 9583*, VMCAI 2016, pages 127–146, New York, NY, USA, 2016. Springer-Verlag New York, Inc.

[105] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6(2):133–151, June 1976.

[106] A. Kean and G. Tsiknis. An incremental method for generating prime implicants/implicates. *J. Symb. Comput.*, 9(2):185–206, Feb. 1990.

[107] S. Klarman, U. Endriss, and S. Schlobach. Abox abduction in the description logic alc. *J. Autom. Reasoning*, 46:43–80, 2011.

[108] E. Knill, P. T. Cox, and T. Pietrzykowski. Equality and abductive residua for horn clauses. *Theor. Comput. Sci.*, 120:1–44, 1993.

[109] D. E. Knuth. Postscript about np-hard problems. *SIGACT News*, 6(2):15–16, Apr. 1974.

[110] L. Kovacs and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. 5503, 09 2009.

[111] L. Kovács and A. Voronkov. First-order theorem proving and vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification*, pages 1–35, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[112] L. Kovács and A. Voronkov. Interpolation and symbol elimination. pages 199–213, 07 2009.

[113] J. V. Leeuwen. *Handbook of Theoretical Computer Science: Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.

[114] J.-J. Lévy and C. Ran. Sorting algorithms. Available at http://pauillac.inria.fr/~levy//why3/sorting/.

[115] J. Madre and O. Coudert. A new method to compute prime and essential prime implicants of boolean functions. *Advanced Research in VLSI and Parallel Systems*, 01 1992.

[116] V. M. Manquinho, A. L. Oliveira, and J. P. M. Silva. Models and algorithms for computing minimum-size prime implicants. In *In Proc. International Workshop on Boolean Problems (IWBP'98*, 1998.

[117] P. Marquis. Extending abduction from propositional to first-order logic. In *Proceedings of the International Workshop on Fundamentals of Artificial Intelligence Research*, FAIR '91, pages 141–155, London, UK, UK, 1991. Springer-Verlag.

[118] P. Marquis. Consequence finding algorithms. 2000.

[119] A. Matusiewicz, N. Murray, and E. Rosenthal. Prime implicate tries. pages 250–264, 07 2009.

[120] A. Matusiewicz, N. Murray, and E. Rosenthal. Tri-based set operations and selective computation of prime implicates. pages 203–213, 06 2011.

[121] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In M. Sagiv, editor, *European Symposium on Programming (ESOP'05)*, volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer-Verlag, 2005.

[122] S.-i. Minato. Zero-suppressed bdds for set manipulation in combinatorial problems. In *Proceedings of the 30th International Design Automation Conference*, DAC '93, pages 272–277, New York, NY, USA, 1993. ACM.

[123] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, Mar. 2006.

[124] A. Miné. Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation. *Foundations and Trends in Programming Languages*, 4(3-4):120–372, 2017.

[125] D. Monniaux. A survey of satisfiability modulo theory. *ArXiv*, abs/1606.04786, 2016.

[126] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[127] H. Nabeshima, K. Iwanuma, K. Inoue, and O. Ray. Solar: An automated deduction system for consequence finding. *AI Commun.*, 23:183–203, 2010.

[128] A. Newell, J. C. Shaw, and H. A. Simon. Empirical explorations of the logic theory machine: a case study in heuristic. In *IRE-AIEE-ACM '57 (Western)*, 1957.

[129] T.-H. Ngair. A new algorithm for incremental prime implicate generation. In *Proceedings of the 13th International Joint Conference on Artifical Intelligence - Volume 1*, IJCAI'93, pages 46–51, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[130] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest. Using dynamic analysis to discover polynomial and array invariants. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 683–693, Piscataway, NJ, USA, 2012. IEEE Press.

[131] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, Nov. 2006.

[132] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[133] J. Otten. Mleancop: A connection prover for first-order modal logic. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning*, pages 269–276. Springer International Publishing, 2014.

[134] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. Decidability of inferring inductive invariants. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 217–231, New York, NY, USA, 2016. ACM.

[135] L. Palopoli, F. Pirri, and C. Pizzuti. Algorithms for selective enumeration of prime implicants. *Artif. Intell.*, 111:41–72, 1999.

[136] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[137] J. H. Perkins and M. D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, SIGSOFT '04/FSE-12, pages 23–32, New York, NY, USA, 2004. ACM.

[138] D. Peuter and V. Sofronie-Stokkermans. On invariant synthesis for parametric systems. In P. Fontaine, editor, *Automated Deduction – CADE 27*, pages 385–405, Cham, 2019. Springer International Publishing.

[139] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3):293–304, 1986.

[140] K. Popper. *Conjectures and refutations: The growth of scientific knowledge*. Routledge, 1963.

[141] A. Previti, A. Ignatiev, A. Morgado, and J. Marques-Silva. Prime compilation of non-clausal formulae. 01 2015.

[142] W. V. Quine. A way to simplify truth functions. *The American Mathematical Monthly*, 62(9):627–631, 1955.

[143] A. Ramesh. Some applications of non clausal deduction.

[144] A. Ramesh, G. Becker, and N. V. Murray. Cnf and dnf considered harmful for computing prime implicants/implicates. *Journal of Automated Reasoning*, 18:337–356, 1997.

[145] M. Raut. An incremental knowledge compilation in first order logic. 10 2011.

[146] M. Raut. An incremental algorithm for computing prime implicates in modal logic. 04 2014.

[147] M. Raut and A. Singh. Prime implicants of first order formulas via transversal clauses. *Int. J. Comput. Math.*, 81:157–167, 02 2004.

[148] M. Raut and A. Singh. A survey on computing prime implicants and implicates in classical and non-classical logics. *Computer Systems Science and Engineering*, 29:327–340, 09 2014.

[149] M. K. Raut and A. Singh. Prime implicates of first order formulas. *IJCSA*, 1:1–11, 2004.

[150] A. Reynolds, R. Iosif, C. Serban, and T. King. A Decision Procedure for Separation Logic in SMT. In *Automated Technology for Verification and Analysis 14th International Symposium (ATVA 2016)*, volume 9938 of *Automated Technology for Verification and Analysis 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pages 244–261, Chiba, Japan, Oct. 2016.

[151] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS '02, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[152] T. Ribeiro and K. Inoue. Learning Prime Implicant Conditions From Interpretation Transition. In *The 24th International Conference on Inductive Logic Programming (ILP 2014)*, volume 9046 of *Inductive Logic Programming 24th International Conference, ILP 2014, Nancy, France, September 14-16, 2014, Revised Selected Papers*, Nancy, France, Sept. 2015. Springer.

[153] A. J. Robinson and A. Voronkov. *Handbook of automated reasoning*, volume 1. Gulf Professional Publishing, 2001.

[154] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42:443–476, 2007.

[155] R. Rymon. An se-tree-based prime implicant generation algorithm. *Annals of Mathematics and Artificial Intelligence*, 11:351–365, 1994.

[156] S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. volume 39, pages 318–329, 01 2004.

[157] Satconv benchmarks.

[158] A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1998.

[159] S. Schulz. System Description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

[160] Y. Sellami. Abdulot framework and tools. Available at https://github.com/sellamiy/GPiD-Framework.

[161] Y. Sellami. A dpll-based approach to prime implicate generation, 2016.

[162] J. a. P. M. Silva and K. A. Sakallah. Grasp—a new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '96, pages 220–227, Washington, DC, USA, 1996. IEEE Computer Society.

[163] L. Simon and A. del Val. Efficient consequence finding. In *IJCAI*, 2001.

[164] J. R. Slagle, C.-L. Chang, and R. C. T. Lee. A new algorithm for generating prime implicants. *IEEE Trans. Comput.*, 19(4):304–310, Apr. 1970.

[165] V. Sofronie-Stokkermans. Hierarchical reasoning for the verification of parametric systems. In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, pages 171–187, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[166] V. Sofronie-Stokkermans. Hierarchical reasoning and model generation for the verification of parametric hybrid systems. In M. P. Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 360–376. Springer, 2013.

[167] T. Strzemecki. Polynomial-time algorithms for generation of prime implicants. *J. Complex.*, 8(1):37–63, Mar. 1992.

[168] G. Sutcliffe. The tptp problem library, 2019. Available at http://www.tptp.org/TPTP/TR/TPTPTR.shtml.

[169] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 132–143, New York, NY, USA, 1977. ACM.

[170] A. Tarski. Equational logic and equational theories of algebras. In H. A. Schmidt, K. Schütte, and H.-J. Thiele, editors, *Contributions to Mathematical Logic*, volume 50 of *Studies in Logic and the Foundations of Mathematics*, pages 275 – 288. Elsevier, 1968.

[171] P. Tison. Generalization of consensus theory and application to the minimization of boolean functions. *IEEE Transactions on Electronic Computers*, EC-16(4):446–456, Aug 1967.

[172] Gallery of verified programs. Available at http://toccata.lri.fr/gallery/.

[173] S. Tourret. *Prime implicate generation in equational logic*. PhD thesis, 2016. Thèse de doctorat dirigée par Peltier, Nicolas et Echenim, Bertrand Mnacho Informatique Grenoble Alpes 2016.

[174] D.-K. Tran, C. Ringeissen, S. Ranise, and H. Kirchner. Combination of Convex Theories: Modularity, Deduction Completeness, and Explanation. *Journal of Symbolic Computation*, 45(2):261–286, Feb. 2010. Special issue on Automated Deduction: Decidability, Complexity, Tractability. Final version of inria-00331479 (INRIA RR-6688). doi:10.1016/j.jsc.2008.10.006.

[175] M.-T. Trinh, Q. L. Le, C. David, and W.-N. Chin. Bi-abduction with pure properties for specification inference. In *Proceedings of the 11th Asian Symposium on Programming Languages and Systems - Volume 8301*, pages 107–123, Berlin, Heidelberg, 2013. Springer-Verlag.

[176] y. Wei, H. Roth, C. Furia, Y. Pei, A. Horton, M. Steindorfer, M. Nordio, and B. Meyer. Stateful testing: Finding more errors in code and contracts. *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings*, 08 2011.

[177] L. Wos, R. Overbeck, E. Lusk, and J. Boyle. Automated reasoning: introduction and applications. 1984.

[178] L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, pages 279–285, Piscataway, NJ, USA, 2001. IEEE Press.