1 Introduction

De plus en plus souvent, l'homme est remplacé ou aidé par des logiciels informatiques dans des domaines critiques tels que les transports ferroviaires, l'avionique, le spatial, ou le médical. Une défaillance de ces logiciels peut coûter beaucoup, à la fois matériellement et en terme de vies humaines. Par conséquent, ils sont soigneusement vérifiés afin de s'assurer qu'ils fonctionnent correctement. Historiquement, cette vérification se fait principalement par test. Cependant, comme ces tests sont de plus en plus coûteux, ils sont parfois remplacés ou complétés par des méthodes dites de vérification formelle basées sur l'analyse par ordinateur du code source.

Elles fonctionnent de la manière suivante : au lieu d'exécuter le code source, on l'analyse de façon mathématique de manière à établir des garanties sur chaque exécution possible du logiciel. Ces techniques ont déjà été utilisées avec succès dans l'industrie, par exemple par Airbus [74] et Siemens [11]. Grâce au soutien des autorités de certification correspondantes, les méthodes formelles, et notamment la méthode B sont largement utilisées dans le domaine ferroviaire. Récemment, les autorités de certification pour l'avionique ont présenté les méthodes formelles comme une alternative au test (voir DO-333) ce qui devrait leur ouvrir un nouveau marché.

Les méthodes formelles incluent une grande variété de techniques. Nous nous intéressons en particulier à la vérification déductive. Les outils de vérification déductive de programmes fonctionnent en transformant les propriétés du programme en formules logiques appelées obligations de preuve. La validité de ces formules logiques, qui implique la correction du programme, peut alors être vérifiée par des prouveurs automatiques.

1.1 Une procédure de décision à partir d'une axiomatisation en logique du premier ordre

Les obligations de preuve générées par la vérification déductive d'un programme utilisent souvent de nombreux symboles appartenant à des théories prédéfinies, telles que l'arithmétique, les tableaux applicatifs, les champs de bits... Certains prouveurs automatiques supportent spécifiquement certaines théories. Les solveurs SMT (pour satisfiabilité modulo théories), décident habituellement de manière complète et terminante la satisfiabilité de formules sans quantificateurs modulo un certain nombre de théories. Ce sont des *procédures de décision* pour la satisfiabilité de ces formules. Bien sûr, toutes les théories utiles ne sont pas supportées par tous les solveurs SMT et il existe de nombreuses théories qui ne sont supportées par aucun solveur. L'ajout d'une nouvelle théorie à un solveur SMT est une tâche longue et complexe qui nécessite souvent une connaissance précise de l'implémentation du solveur ainsi que l'accès à son code source.

De nombreuses théories utiles peuvent facilement être décrites en utilisant un ensemble de formules du premier ordre. Une telle axiomatisation peut être utilisée directement par un solveur SMT, s'il supporte les quantificateurs. Pour donner quelques exemples, Simplify [34], CVC3 [39], CVC4 [8], Z3 [30], et Alt-Ergo [14] supportent la logique du premier ordre. Comme la logique du premier ordre est indécidable, tout prouveur automatique est au mieux semi-complet sur les problèmes de premier ordre et même la semi-complétude est inaccessible lorsque le prouveur supporte des théories non - triviales telles que l'arithmétique entière. Pour améliorer ses chances de trouver une preuve, la plupart des solveurs SMT donnent à l'utilisateur un certain

contrôle sur l'instanciation des formules quantifiées, en permettant d'annoter les quantificateurs avec des *motifs d'instanciation* également appelés *déclencheurs*.

Intuitivement, les déclencheurs sont utilisés comme motifs pour filtrer l'ensemble des termes connus, c'est-à-dire des termes apparaissant dans un fait supposé par le solveur, et déterminer ceux qui doivent être utilisés pour l'instanciation d'un quantificateur. Cette technique, appelée filtrage modulo égalité, a été introduite dans le Stanford Pascal Verifier [62] et est maintenant utilisée dans la plupart des solveurs SMT supportant les quantificateurs. Il a été démontré que, en restreignant l'instanciation à l'aide de déclencheurs, on pouvait obtenir à la fois la complétude et la terminaison du processus d'instanciation pour certaines axiomatisations, les transformant par là même en procédures de décision. Un des exemples les plus connus est la procédure de décision pour la théorie des tableaux fonctionnels décrite par Greg Nelson [62], que nous allons examiner plus en détail ci-dessous. Plus récemment, le même travail a été fait pour la spécification de structures de données plus complexes [59, 19].

Exemple 1.1. Voici une axiomatisation de la théorie des tableaux non-extensionnelle telle que définie par Greg Nelson. Cette axiomatisation utilise deux symboles de fonction, le premier, nommé *get*, pour l'accès dans un tableau et le second, nommé *set*, pour la modification d'un tableau. Il contient deux axiomes qui décrivent la façon dont le contenu d'un tableau est transformé lors d'une modification. Le premier indique que, un tableau a été modifié pour y stocker l'élément e à l'indice i, alors l'accès à l'indice i dans ce tableau renvoie e. Le second, dont on donne deux versions avec deux déclencheurs différents, indique que l'accès au tableau à tout autre indice renvoie l'élément précédemment stocké à cet indice.

$$W_{array} = \left\{ \begin{array}{l} \forall a, i, e.[set(a,i,e)]\ (get(set(a,i,e),i) \approx e) \\ \forall a, i, j, e.[get(set(a,i,e),j)]\ (i \not\approx j \rightarrow get(set(a,i,e),j) \approx get(a,j)) \\ \forall a, i, j, e.[set(a,i,e), get(a,j)]\ (i \not\approx j \rightarrow get(set(a,i,e),j) \approx get(a,j)) \end{array} \right\}$$

Le déclenchement du premier axiome exprime qu'il est suffisant de l'instancier avec les termes a, i et e si le terme $set(a,i,e)$ apparaît dans le problème. Pour le deuxième axiome, il y a deux cas différents où on doit l'instancier : si le terme $get(set(a,i,e),j)$ apparaît dans le problème ou si les deux termes $set(a,i,e)$ et $get(a,j)$ apparaissent dans le problème. Ces deux cas permettent de réécrire l'égalité $get(set(a,i,e),j) \approx get(a,j))$ dans les deux sens.

Malheureusement, l'utilisateur ne peut pas espérer prouver qu'un solveur SMT du premier ordre donné est complet et termine sur un ensemble particulier d'axiomes avec des déclencheurs, ce qui lui permettrait d'obtenir une procédure de décision. En effet, les déclencheurs ne sont pas destinés à changer la satisfiabilité d'une formule du premier ordre. Ils sont plutôt considérés comme des conseils, permettant de déterminer les instances les plus susceptibles d'être utiles pour la preuve. De ce fait, un solveur SMT peut aussi bien fonder ses décisions sur les déclencheurs donnés par l'utilisateur que sur des déclencheurs qu'il aurait lui même déduit ou sur une quelconque autre heuristique. Pour être aussi complet que possible, un solveur peut utiliser toute stratégie d'instanciation qu'il juge utile. Il peut même ignorer complètement les déclencheurs fournis par l'utilisateur. Et pourtant, si nous voulons utiliser notre axiomatisation comme une procédure de décision, nous devons être capables de contrôler l'instanciation des axiomes d'une manière précise et fiable.

1.2 Résumé des Contributions

Nous proposons dans la section 2 un cadre permettant d'ajouter une nouvelle théorie à un solveur SMT par l'intermédiaire d'une axiomatisation de premier ordre avec des déclencheurs. Afin de limiter l'instanciation de manière déterministe, nous donnons une sémantique formelle à la logique du premier ordre avec des déclencheurs, qui promeut les déclencheurs au statut des gardes interdisant toutes les instances qui ne correspondent pas au motif. En utilisant cette sémantique, nous définissons, indépendamment de l'implémentation d'un solveur spécifique mais modulo les théories qu'il supporte, trois propriétés d'un ensemble d'axiomes du premier ordre avec des déclencheurs, à savoir, la correction, la complétude et la terminaison, nécessaires pour qu'un solveur se comporte comme une procédure de décision pour cette axiomatisation.

Dans la section 3, nous expliquons comment un solveur SMT peut être étendu pour supporter notre logique. Nous considérons pour cela le cadre théorique habituellement utilisé pour modéliser les solveurs SMT, appelé DPLL abstrait modulo théories [65]. Nous décrivons une variante de ce cadre qui supporte les formules du premier ordre avec des déclencheurs. Nous montrons que, pour toute axiomatisation qui répond nos trois conditions de correction, de complétude et de terminaison, un solveur SMT conforme à notre description se comportera comme une procédure de décision pour la théorie axiomatisée. Plus précisément, considérons un solveur SMT qui décide efficacement les problèmes sans quantificateurs modulo une théorie T. Dans le cas le plus simple, T peut être la théorie de l'égalité avec des symboles de fonction non-interprétés (EUF). Il peut également s'agir de la théorie arithmétique linéaire, de celle des champs de bits, des tableaux associatifs, ou de toute combinaison de ce qui précède. Un utilisateur de ce démonstrateur veut étendre la théorie T avec une nouvelle théorie, par exemple, une théorie pour des structures de données, et obtenir une procédure de décision pour les problèmes sans quantificateurs modulo cette théorie élargie que l'on note T'. Pour ce faire, l'utilisateur écrit un ensemble d'axiomes du premier ordre avec des déclencheurs et prouve que cette axiomatisation est correcte et complète vis à vis de T' et termine modulo T. Comme les trois conditions sont formulées en termes purement logiques, aucune connaissance spécifique des mécanismes internes du prouveur n'est requise. S'il met en oeuvre notre extension de DPLL (T) ou toute autre méthode traitant les axiomes avec des déclencheurs conformément à notre sémantique, le solveur permettra de décider tout problème sans quantificateur modulo T' en un temps fini. La méthode n'est pas destinée à étendre les solveurs SMT ne supportant pas les quantificateurs à la logique du premier ordre. Nous ne prétendons pas non plus donner une sémantique ultimes pour les déclencheurs, sur laquelle tous les solveurs SMT premier ordre devraient converger. En effet, notre traitement restrictif et rigoureux des quantificateurs et des déclencheurs n'est destiné qu'à être appliqué aux axiomes de la théorie que nous voulons décider, et pas à des formules du premier ordre venant d'un problème utilisateur. En effet, alors que nous devons restreindre l'instanciation dans le premier cas pour garantir la terminaison, nous ne gagnerions rien en appliquant les mêmes restrictions à des formules sur lesquelles aucune preuve n'a été faite. Au contraire, nous sommes susceptibles d'empêcher le solveur de trouver des preuves qui autrement, auraient été découvertes. En outre, les contrôles supplémentaires nécessaires à la mise en œuvre de cette restriction risquent d'entraver la performance du solveur. Nous avons implémenté notre extension de DPLL (T) dans le solveur SMT du premier ordre Alt-Ergo.

2 Logique du premier ordre avec déclencheurs

Dans les solveurs SMT du premier ordre, les déclencheurs sont utilisés pour favoriser l'instanciation de formules universellement quantifiées avec les termes "connus" ayant une forme donnée. Intuitivement, un terme est dit connu s'il apparaît dans un fait supposé par le solveur. Voici, par exemple, une formule avec un déclencheur dans la syntaxe SMT-LIB version 2 [10] :

```
(forall ((x Int)) (! (= (f x) c):pattern ((g x))))
```

Le symbole "`!`" sous le quantificateur universel dénote une sous-formule annotée et le motif `(g x)` apparaît après le mot-clé `:pattern`. Le sens communément admis de la formule ci-dessus est :

`(= (f t) c)` *est vraie pour tout terme* `t` *de type* `Int` *tel que* `(g t)` *est connu.*

Le concept de déclencheurs peut être étendu aux littéraux. Si un axiome ne peut déduire des faits nouveaux que si on l'instancie avec des termes ayant une propriété donnée P, il peut être inutile de l'instancier avec un terme t si on ne sait pas a priori que $P(t)$ est vrai. En d'autres termes, cela permet de restreindre l'instanciation non seulement par la forme de termes connus, mais aussi par ce qui est connu à leur sujet. Par exemple, dans la théorie des tableaux extensionnels, il suffit d'appliquer l'axiome d'extensionnalité sur les paires de termes de type tableaux que l'on sait être différents [41] :

$$\forall a_1, a_2 : array.[a_1 \not\approx a_2]\ (a_1 \not\approx a_2 \rightarrow (\exists i : index.\ get(a_1, i) \not\approx get(a_2, i))$$

Dans cette section, nous étendons la logique du premier ordre avec une construction pour les déclencheurs. Par souci de simplicité, notre formalisation est non typée, même si tous nos exemples utilisent des sortes. Ensuite, nous définissons ce que signifie le fait qu'une formule avec des déclencheurs est vraie dans le contexte d'un ensemble de faits connus. Enfin, nous présentons des propriétés de correction, de complétude et de terminaison d'un ensemble de formules du premier ordre avec des déclencheurs.

2.1 Notions Préliminaires

Nous travaillons en logique du premier ordre non typée et utilisons les notations habituelles pour les formules du premier ordre et pour les termes. Les formules sont appelées φ ou ψ, les littéraux l, les termes s ou t et les substitutions σ ou μ. D'autres conventions de notation seront introduites au cours du texte. Pour simplifier nos définitions, nous travaillons sur des formules sous forme normale négative. La syntaxe des formules et des littéraux est décrite ci-dessous, A étant un atome :

$$\begin{aligned} \varphi &::= l \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \exists x.\varphi \\ l &::= A \mid \neg A \end{aligned}$$

La transformation en forme normale négative utilise les définitions habituelles :

$$\begin{aligned}\neg(\varphi_1 \vee \varphi_2) &\Longrightarrow \neg\varphi_1 \wedge \neg\varphi_2\\ \neg(\varphi_1 \wedge \varphi_2) &\Longrightarrow \neg\varphi_1 \vee \neg\varphi_2\\ \neg(\forall x.\varphi) &\Longrightarrow \exists x.\neg\varphi\\ \neg(\exists x.\varphi) &\Longrightarrow \forall x.\neg\varphi\end{aligned}$$

Nous disons qu'un terme, un littéral, ou une formule est *close* si elle n'a pas de variable libre. Nous utilisons $\mathscr{T}(t)$, $\mathscr{T}(l)$, $\mathscr{T}(S)$ pour désigner l'ensemble de tous les sous-termes contenus dans, respectivement, un terme t, un littéral l ou un ensemble de termes ou littéraux S. Nous raisonnons modulo une théorie T, que nous supposons être fixée pour le reste de cette section. Dans le cas le plus simple, T peut être la théorie de l'égalité avec des symboles de fonction non interprétés (EUF). Nous supposons que la signature de T contient au moins un symbole de constante pour permettre la construction d'univers de Herbrand et qu'elle peut être étendue à volonté avec des symboles de fonctions non interprétées pour permettre la skolémisation. Nous utilisons la définition suivante pour les atomes, $\approx$ étant le symbole de l'égalité :

$$A ::= \top \mid t_1 \approx t_2 \mid \ldots$$

Les points représentent d'autres formes de prédicats spécifiques à la théorie utilisée, par exemple la comparaison pour l'arithmétique linéaire. Nous utilisons des modèles de Herbrand dans notre formalisation c'est à dire que nous appelons modèle un ensemble de littéraux L contenant tout les littéraux valides. Notez qu'une formule du premier ordre est satisfiable si et seulement si elle a un modèle de Herbrand.

Nous utilisons la notation $L \vDash \varphi$ pour indiquer que la formule du premier ordre φ est valide dans un modèle de Herbrand L.

Définition 2.1 (Satisfiabilité modulo T)**.** On dit qu'une formule φ du premier ordre est valide dans un modèle L modulo T, écrit $L \vDash_T \varphi$, si φ est valide dans L et L est aussi un modèle de T. Si une formule φ du premier ordre a un modèle modulo T, nous disons qu'elle est T-satisfiable ou, de façon équivalente, qu'elle est satisfiable modulo T.

Nous utilisons parfois des clauses, qui sont des ensembles disjonctifs de littéraux. On dit qu'une clause est *unitaire* si elle ne contient qu'un seul littéral. La clause vide est équivalente à faux, c'est-à-dire $\neg\top$.

2.2 Syntaxe et Sémantique

Nous introduisons deux nouveaux types de formules :

- Une formule φ protégée par un déclencheur l que nous notons $[l]\varphi$. Elle peut se lire *si le littéral l est valide et si tous ses sous-termes sont connus alors φ est valide*.
- Une notation duale pour $[l]\varphi$, appelée *témoin*, que nous notons $\langle l\rangle\varphi$. Elle peut se lire *le littéral l est valide, tous ses sous-termes sont connus et φ est valide*.

Notez que ni les déclencheurs ni les témoins ne sont nécessairement attachés à un quantificateur. La syntaxe étendue de formules peut être résumée comme suit :

$$\varphi ::= l \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \forall x.\varphi \mid \exists x.\varphi \mid [l]\varphi \mid \langle l \rangle \varphi$$

Pour les nouvelles constructions, la mise sous forme normale négative est faite en utilisant les équivalences $\neg\langle l\rangle\varphi \Longrightarrow [l]\neg\varphi$ et $\neg[l]\varphi \Longrightarrow \langle l\rangle\neg\varphi$. Si une formule contient plusieurs déclencheurs ou témoins consécutifs, nous écrivons $[l_1,\ldots,l_n]\varphi$ pour $[l_1]\ldots[l_n]\varphi$ et $\langle l_1,\ldots,l_n\rangle\varphi$ pour $\langle l_1\rangle\ldots\langle l_n\rangle\varphi$. Une formule du premier ordre avec des déclencheurs est évaluée dans le contexte d'un ensemble donné de faits supposés et de termes connus :

Définition 2.2 (Monde modulo T)**.** Nous appelons *monde* un ensemble T-satisfiable de littéraux clos. Un monde L est habité si il y a au moins un terme apparaissant dans L, c'est à dire si $\mathscr{T}(L)$ est non vide. Un monde L est complet si, pour tout littéral clos l dans la signature de T, $l \in L$ ou $\neg l \in L$.

Définition 2.3 (Terme Connu modulo T)**.** Un terme t est *connu* dans le monde L si et seulement si il existe un terme $t' \in \mathscr{T}(L)$ tel que $L \models_T t \approx t'$.

L'intuition clé pour comprendre les mondes est qu'un littéral clos l ne peut être évalué dans un monde L que si chaque terme t de $\mathscr{T}(l)$ est connu dans L. Si, au contraire, il existe un terme t de $\mathscr{T}(l)$ qui n'est pas connu dans L, nous "refusons" d'évaluer le littéral, c'est-à-dire que ni l ni $\neg l$ est vrai dans L. Pour exprimer cette contrainte facilement, nous utilisons un symbole de prédicat unaire *known* que nous supposons être frais dans T et dans le problème. En utilisant ce symbole, le fait que t est un terme connu dans L, peut être écrit de façon équivalente $L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(t)$.

Définition 2.4 (Valeur de Vérité modulo T)**.** Étant donné un monde L et une formule close φ, nous définissons quand φ est *vraie* dans L, noté $L \models_T \varphi$, par induction sur φ :

$L \rhd_T l$	$L \models_T l$ et $L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(\mathscr{T}(l))$
$L \rhd_T \varphi_1 \vee \varphi_2$	$L \rhd_T \varphi_1$ ou $L \rhd_T \varphi_2$
$L \rhd_T \varphi_1 \wedge \varphi_2$	$L \rhd_T \varphi_1$ et $L \rhd_T \varphi_2$
$L \rhd_T \forall x.\varphi$	pour tout terme t dans $\mathscr{T}(L)$, $L \rhd_T \varphi[x \leftarrow t]$
$L \rhd_T \exists x.\varphi$	il existe un terme t dans $\mathscr{T}(L)$ tel que $L \rhd_T \varphi[x \leftarrow t]$
$L \rhd_T [l]\varphi$	si $L \rhd_T l$ alors $L \rhd_T \varphi$
$L \rhd_T \langle l\rangle\varphi$	$L \rhd_T l$ et $L \rhd_T \varphi$

On dit qu'une formule close φ est *fausse* dans L lorsque $L \models_T \neg\varphi$. Nous disons que φ est *faisable* s'il existe un monde dans lequel φ est vraie.

Comme nous l'avons noté, une formule qui contient un terme inconnu dans un monde peut n'être ni vraie ni fausse dans ce monde. D'autre part, il est impossible pour une formule d'être à la fois vraie et fausse dans le même monde. En d'autres termes, il n'y a pas de formule close φ et de monde L tels que $L \models_T \varphi$ et $L \models_T \neg\varphi$. La preuve se fait par induction sur la structure de

φ. D'après notre définition, une formule avec un témoin $\langle l \rangle \varphi$ est traitée comme la conjonction $l \wedge \varphi$. Par contre, une formule avec un déclencheur $[l]\varphi$ n'est pas équivalente à la disjonction $\neg l \vee \varphi$. En effet, considérons un littéral l qui contient un terme inconnu dans L, de sorte que ni l ni $\neg l$ n'est vrai dans L. Nous avons alors $L \models_T [l]\bot$ mais pas $L \models_T \neg l \vee \bot$. Toutefois, si L est un monde complet, alors tout littéral clos est soit vrai soit faux dans L, et nous pouvons remplacer tous les déclencheurs par des implications.

Définition 2.5 (Modèle modulo T)**.** Soit φ une formule close. Un monde L est un *modèle* de φ si $L \models_T \varphi$ et L est complet. Nous disons que φ est satisfiable si elle a un modèle.

2.3 Correction et Complétude

Lorsqu'un utilisateur souhaite étendre la théorie T supportée par un solveur à l'aide d'un ensemble d'axiomes avec des déclencheurs, il doit prouver que cette axiomatisation est une représentation adéquate de la théorie étendue T' modulo T.

Définition 2.6 (Correction modulo T)**.** Une axiomatisation W est *correcte* par rapport à T' si, pour tout ensemble T'-satisfiable de littéraux clos L, $W \cup L$ est T-satisfiable.

Définition 2.7 (Complétude modulo T)**.** Une axiomatisation W est *complète* par rapport à T' si, pour chaque ensemble L de littéraux clos tel que $W \cup L$ est faisable modulo T, L est T'-satisfiable.

Remarque 2.1. Notez que les définitions de correction et de complétude ne sont pas symétriques. En effet, nous voulons que le solveur soit autorisé à s'arrêter dès qu'il a trouvé un monde dans lequel l'axiomatisation est vraie en déduisant que le problème est satisfiable. En particulier, nous voulons que le solveur instancie les quantificateurs universels qu'avec les termes de L et qu'il ignore les formules protégées par un déclencheur l si l n'est pas vrai dans L.

Très souvent, T est la théorie définie par le même ensemble d'axiomes W où tous les déclencheurs et témoins sont effacés. Plus précisément, nous commençons avec une axiomatisation du premier ordre de notre théorie, que nous annotons avec des déclencheurs et des témoins afin de limiter l'instanciation et de garantir la terminaison de la recherche de preuves. Dans ce cas, pour prouver la correction, nous devons montrer que les témoins ajoutés ne nous permettent pas de déduire des faits qui ne sont pas impliqués par l'axiomatisation initiale. Pour la complétude, nous devons montrer que les déclencheurs ajoutés et la sémantique restreinte des quantificateurs ne nous empêchent pas de prouver les faits clos déductibles dans la première axiomatisation.

Exemple 2.1. La preuve que l'ensemble d'axiomes W_{array} présenté dans l'exemple 1.1 est complet modulo EUF ressemble beaucoup à la preuve par Greg Nelson dans [62]. Nous ne refaisons pas cette preuve ici, mais nous expliquons pourquoi des variantes plus simples ou plus intuitives de cette axiomatisation sont incomplètes.

- Soit W^1_{array} l'ensemble W_{array} où le déclencheur du premier axiome est remplacé par le terme $get(set(a,i,e),i)$. Considérons l'ensemble de littéraux clos $L_1 = \{set(a,i,e_1) \approx set(a,i,e_2), e_1 \not\approx e_2\}$. L_1 est insatisfiable dans la théorie des tableaux puisque nous avons à la fois $get(set(a,i,e_1),i) \approx e_1$ et $get(set(a,i,e_2),i) \approx e_2$. Pourtant, W^1_{array} est vrai dans le monde L_1, puisque L_1 ne contient pas de terme correspondant au déclencheur.

- Soit W^2_{array} l'ensemble W_{array} Sans le second axiome. Considérons l'ensemble de littéraux $L_2 = \{get(set(a,i_1,e),j) \not\approx get(set(a,i_2,e),j), i_1 \not\approx j, i_2 \not\approx j\}$. L_2 est insatisfiable dans la théorie des tableaux puisque $get(set(a,i_1,e),j) \approx get(a,j)$ et $get(set(a,i_2,e),j) \approx get(a,j)$. Pourtant, W^2_{array} est vrai dans le monde $L_2 \cup \{get(set(a,i_1,e),i_1) \approx e, get(set(a,i_2,e),i_2) \approx e\}$.
- Soit W^3_{array} l'ensemble W_{array} sans le troisième axiome. Considérons l'ensemble de littéraux clos $L_3 = \{set(a_1,i,e) \approx set(a_2,i,e), i \not\approx j, get(a_1,j) \not\approx get(a_2,j)\}$. L_3 est insatisfiable dans la théorie des tableaux puisque $get(set(a_1,i,e),j) \approx get(a_1,j)$ et $get(set(a_2,i,e),j) \approx get(a_2,j)$. Pourtant, l'ensemble W^3_{array} est vrai dans le monde $L_3 \cup \{get(set(a_1,i,e),i) \approx e, get(set(a_2,i,e),i) \approx e\}$.

2.4 Terminaison

Une fois qu'il a été établi qu'un ensemble d'axiomes avec des déclencheurs est correct et complet pour notre théorie, nous devons montrer qu'un solveur utilisant cette axiomatisation termine sur tout ensemble de formules sans quantificateurs ni déclencheur. Nous appelons une telle axiomatisation *terminante* et le reste de cette section est consacrée à la définition de cette propriété.

Il n'y a pas définition unique de la terminaison d'une axiomatisation. Différentes variantes de l'algorithme du solveur peuvent terminer sur différentes catégories de problèmes, être plus ou moins aisées à décrire et rendre le raisonnement plus ou moins difficile. Nous nous efforçons de trouver une "bonne" définition, qui, d'une part, permet une implémentation efficace, et d'autre part, est assez simple pour que les preuves de terminaisons soient faisables. Ci-dessous, nous présentons ce que nous considérons être une assez bonne définition. Elle sert de base à la procédure de décision basée sur DPLL décrite dans la section 3.

Pour nous rapprocher de l'implémentation d'un solveur, nous commençons par éliminer les quantificateurs existentiels et par mettre nos axiomes en forme clausale.

Skolémisation : La transformation de skolémisation, notée SKO, traverse une formule de haut en bas en remplaçant les quantificateurs existentiels par des témoins utilisant des fonctions de Skolem :

$$\text{SKO}(\exists x.\varphi) \triangleq \langle c(\bar{y})\rangle \, \text{SKO}(\varphi[x \leftarrow c(\bar{y})]),$$

où $\bar{y}$ est l'ensemble des variables libres de $\exists x.\varphi$ et c est un symbole de fonction frais.

Lemme 2.1. *La skolémisation préserve la faisabilité et la satisfiabilité.*

Démonstration. La preuve peut être faite par induction sur φ. Nous construisons un monde pour $\text{SKO}(\varphi)$ en donnant aux termes de Skolem l'interprétation des termes correspondant dans le monde que nous avions pour φ. Inversement, si $\text{SKO}(\varphi)$ est vraie dans un monde, alors φ est vraie dans le même monde.

L'utilisation des témoins est cruciale ici. En effet, $\text{SKO}(\exists x.[x]\bot)$ donne $\langle c\rangle[c]\bot$ qui est également infaisable, tandis que la formule $[c]\bot$ est vraie dans n'importe quel monde où c est inconnue. □

La skolémisation ne préserve la correction et la complétude d'un ensemble d'axiomes que dans la mesure où les symboles de Skolem ne sont pas utilisés dans les mondes pour la version skolemisée. Par exemple, si T' est la théorie $\exists x.P(x)$, alors la formule skomémisée $\langle c\rangle P(c)$ n'est une représentation adéquate de T que si c est frais dans les mondes considérés. En effet, le littéral $\neg P(c)$ est T'-satisfiable, alors que $\langle c\rangle P(c) \cup \neg P(c)$ n'a pas de modèle. Cela ne pose pas de problème pour nous : les théorèmes de correction et de complétude dans la section 3 ne nécessitent pas que les axiomatisations soient skolemisées.

Clausification :

Définition 2.8 (Pseudo-clause)**.** Nous appelons *pseudo-littéral* un littéral l, un déclencheur $[l]C$, un témoin $\langle l\rangle C$, ou une formule universellement quantifiée $\forall x.C$, où C est une disjonction de pseudo-littéraux, appelée *pseudo-clause*.

Dans ce qui suit, nous traitons les pseudo-clauses (et les autres types de clauses) comme des ensembles disjonctifs, c'est-à-dire que nous ignorons l'ordre de leurs éléments et les doublons. Comme en logique traditionnelle, toute formule skolemisée peut être transformée en une forme clausale, le cas des déclencheurs et des témoins étant réglé en utilisant l'équivalence entre les formules $[l](\varphi_1 \wedge \varphi_2)$ et $[l]\varphi_1 \wedge [l]\varphi_2$, et les formules $\langle l\rangle(\varphi_1 \wedge \varphi_2)$ et $\langle l\rangle\varphi_1 \wedge \langle l\rangle\varphi_2$.

Avant de donner sa définition, nous donnons quelques explications informelles de la propriété de terminaison. Pour raisonner sur la terminaison, nous avons besoin d' une représentation abstraite de l'évolution de l'état du solveur. Il est commode de voir cette évolution comme un jeu où nous choisissons une formule universellement quantifiée à instancier et notre adversaire décide de la façon d'interpréter le résultat de l'instanciation, c'est-à-dire, de choisir l'ensemble des nouveaux faits que l'on peut supposer. Chaque fois que nous arrivons à un ensemble de faits qui est insatisfiable ou saturé de sorte qu'aucune nouvelle instance ne peut être faite, le jeu se termine et nous gagnons. Si, quelles que soient les instances que nous choisissons, l'adversaire peut trouver de nouvelles formules universelles à nous proposer, le jeu se poursuit indéfiniment. Une axiomatisation est terminante s'il existe une stratégie gagnante. En d'autres termes, peu importe le modèle partiel que nous explorons, il y a une séquence d'instanciations - que notre solveur finira par faire par équité - permettant d'arriver soit à une contradiction, soit à un modèle partiel saturé.

Les mouvements de l'adversaire sont représentés par une *affectation de vérité*. Intuitivement, étant donné un ensemble de faits supposés par le solveur, une affectation de valeur de vérité est une série d'autres faits que le solveur peut supposer en utilisant seulement un raisonnement propositionnel, sans instanciation. Une fois que cette affectation est faite, nous pouvons choisir une formule universelle et un terme connu pour effectuer une nouvelle instance et ainsi de suite. Un arbre qui inspecte toutes les affectations de vérité possibles pour certains choix d'instances (ie toutes les réponses possible de l'adversaire à une stratégie particulière du joueur) est appelé arbre d'instanciation. Une axiomatisation est terminante si, pour tout ensemble de littéraux clos, nous pouvons construire un arbre d'instanciation fini.

Pour éviter d'appliquer les substitutions lors des instances, nous utilisons des *clotures*. Une cloture est une paire $\varphi \cdot \sigma$ constituée d'un pseudo-littéral φ et d'une substitution σ qui associe un terme clos à chaque variable libre de φ. Nous écrivons $\varphi\sigma$ l'application de σ à φ, et $\varnothing$ la

substitution vide. Si deux substitutions σ et σ' ont le même domaine D, nous écrivons $\sigma \approx \sigma'$ pour $\bigwedge_{x \in D} x\sigma \approx x\sigma'$. Si C est une pseudo-clause, nous écrivons $C \cdot \sigma$ pour l'ensemble disjonctif de clotures $\{\varphi \cdot \sigma' \mid \varphi \in C \text{ et } \sigma' \text{ est } \sigma \text{ restreinte aux variables libres de } \varphi\}$. Comme ils viennent de l'axiomatisation de notre théorie, nous appelons *clauses de théorie* les ensembles disjonctifs de clotures.

Nous définissons l'ensemble des faits qui sont directement disponibles dans un ensemble de clauses de théorie V, sans avoir besoin d'éliminer de déclencheurs ni de témoins, d'instancier une formule universellement quantifiée ou de décider quelle partie d'une disjonction supposer :

Définition 2.9. Étant donné un ensemble de clauses de théorie V, nous définissons l'ensemble des littéraux $\lfloor V \rfloor \triangleq \{l\sigma \mid l \cdot \sigma \text{ est une clause unitaire de } V\}$.

Définition 2.10 (Affectation de Vérité modulo T)**.** Une affectation de vérité pour un ensemble de clauses de théorie V est un ensemble de clauses de théorie qui peut être construit à partir de V par l'application exhaustive des règles suivantes :

- si $(\varphi_1 \vee \ldots \vee \varphi_n) \cdot \sigma \in A$, ajouter un sous-ensemble quelconque de l'ensemble de clotures $\varphi_1 \cdot \sigma, \ldots, \varphi_n \cdot \sigma$ à A,
- si $[l]C \cdot \sigma \in A$ et $\lfloor A \rfloor \rhd_T l\sigma$, ajouter $C \cdot \sigma$ à A,
- si $\langle l \rangle C \cdot \sigma \in A$, ajouter $l \cdot \sigma$ et $C \cdot \sigma$ à A.

Nous disons que l'affectation de vérité A est T*-satisfiable* si l'ensemble de littéraux $\lfloor A \rfloor$ est T-satisfiable. Une affectation de vérité T-satisfiable A est dite *finale* si toutes les instances possibles sont redondantes dans A, c'est-à-dire si, pour chaque cloture $\forall x.C \cdot \sigma \in A$ et chaque terme $t \in \mathscr{T}(\lfloor A \rfloor)$, il existe une substitution σ' telle que que $C \cdot \sigma' \in A$ et $\lfloor A \rfloor \vDash_T (\sigma \cup [x \mapsto t]) \approx \sigma'$. Dans ce qui suit, nous écrivons $\mathscr{T}(A)$ pour $\mathscr{T}(\lfloor A \rfloor)$ et $A \vDash_T l$ pour $\lfloor A \rfloor \vDash_T l$.

Comme les affectations de vérité ne font que décomposer les formules existantes et sans jamais introduire de nouvelles instances, tout ensemble fini de clauses de théorie a un nombre fini d'affectations de vérité possibles.

Remarquons que, pour l'implémentation du solveur, cette définition signifie que, si nous demandons que le solveur élimine les déclencheurs et les témoins dès qu'ils apparaissent, il est permis de reporter la décision sur les disjonctions. Ce report correspond à l'ajout d'un ensemble vide de clotures dans le premier cas de la définition ci-dessus. Ainsi, le solveur n'est pas obligé de faire tout de suite des choix qu'il devra annuler plus tard, et peut au contraire attendre que les instances à venir réduisent son espace de choix.

Définition 2.11 (Arbre d'Instanciation modulo T)**.** Un arbre d'instanciation pour un ensemble de pseudo-clauses W est un arbre où chaque nœud est étiqueté par un ensemble de clauses de théorie, la racine contenant $W \cdot \varnothing$, et chaque arête sont marquées par une affectation de vérité non-finale telle que :

- Un nœud étiqueté par V a une arête marquée par chaque affectation de vérité T-satisfiable et non-finale de V,
- Une arête marquée par une affectation de vérité A conduit à un nœud étiqueté par $A \cup C \cdot (\sigma \cup [x \mapsto t])$, avec $\forall x.C \cdot \sigma \in A$ et $t \in \mathscr{T}(A)$.

Définition 2.12 (Terminaison modulo T)**.** Un ensemble de pseudo-clauses W est *terminant* si, pour tout ensemble fini de littéraux clos L, $W \cup L$ admet au moins un arbre d'instanciation fini.

Remarque 2.2. Pour être terminant, un ensemble de doit avoir un arbre d'instanciation fini et pas n'admettre que des arbres d'instanciation finis. En effet, nous supposons que l'implémentation sera équitable de sorte que l'instance spécifique attendue par l'arbre d'instanciation sera toujours faite en un nombre de pas fini. Le fait que le solveur puisse également faire des instances supplémentaires n'est pas problématique puisque, en un nombre fini d'étapes, il atteindra soit un état insatisfiable, soit un état où toutes les instances supplémentaires seront redondantes

3 Procédure de Décision

Dans cette section, nous présentons une extension de DPLL abstrait modulo théories [65] qui supporte les formules avec des déclencheurs et des témoins. Si un ensemble d'axiomes est correct et complet par rapport à une théorie T' qui étend la théorie T supportée par le solveur, notre procédure est correcte et complète par rapport à T'. En outre, sous certaines restrictions d'équité sur les dérivations, notre procédure se termine sur tout problème utilisateur si l'axiomatisation se termine. Dans toute cette section, la théorie T supportée par le solveur est fixée.

3.1 Préliminaires

Nous décrivons un solveur qui prend en entrée un ensemble d'axiomes du premier ordre avec des déclencheurs et des témoins, noté Ax, et un ensemble de clauses sans quantificateurs, noté G. Comme pour DPLL classique, nous commençons par skolemiser et clausifier les axiomes de Ax de manière à obtenir un ensemble de pseudo-clauses W , comme décrit dans la section 2. Ensuite, nous convertissons W en un ensemble de clauses de théorie (c'est-à-dire de disjonctions de clotures) en leur associant la substitution vide : $W \cdot \varnothing$. Nous lançons la procédure sur $W \cdot \varnothing$ et G , avec trois résultats possibles :

- Le solveur répond *Unsat*, ce qui signifie que l'union $Ax \cup G$ est insatisfiable, et donc, si Ax est correcte par rapport à T', que G est T'-insatisfiable ;
- Le solveur répond *Sat*, ce qui signifie qu'il existe une formule G' telle que $G' \models_T G$ et l'union $Ax \cup G'$ est faisable - par conséquent, si Ax est complète par rapport à T', G' est T'-satisfiable et G est T'-satisfiable ;
- Le solveur s'exécute indéfiniment, si W est terminante, cela ne peut pas arriver.

Si Ax n'est pas à la fois correcte et complète, l'union $Ax \cup G$ peut être à la fois faisable (vraie dans un monde) et insatisfiable (fausse dans tout monde complet). Dans ce cas, le solveur est non déterministe. Par exemple, soit Ax le singleton $[a]\bot$ et G le singleton $a \approx a \vee \top$. Le solveur peut retirer $\top$ de G, apprendre la constante a, retirez le déclencher et obtenir la contradiction, renvoyant donc *Unsat*. Alternativement, il peut jeter toute la clause de G, puisqu'elle est redondante, et répondre *Sat* : l'union $Ax \cup G$ est vraie dans le monde vide.

Remarquons que l'explication du cas *Sat* est un peu compliquée : au lieu de trouver directement un monde dans lequel $Ax \cup G$ est vrai, le solveur se contente de la faisabilité de l'union de Ax avec n'importe quel antécédent de G modulo T, qui n'a pas à contenir les mêmes termes ni à se comporter de la même manière que G vis à vis de la relation de $\rhd_T$. C'est un élément important de notre approche : le problème utilisateur G est traité classiquement modulo T et le solveur est libre de faire n'importe quelle simplification autorisée par T, sans s'inquiéter des

termes connus. De cette manière, nous restons compatibles avec la sémantique traditionnelles de DPLL. D'autre part, l'axiomatisation Ax est traitée conformément à la sémantique dans la section 2.

Pour maintenir cette distinction, le solveur fonctionne avec deux types de clauses. Les clauses provenant de l'axiomatisation sont des clauses de théorie, c'est-à-dire des disjonctions de clotures qui accumulent les substitutions provenant des instances successives. Les clauses venant de G sont les disjonctions de littéraux clos ; nous les appelons *clauses utilisateur* pour les distinguer des clauses de théorie. La clause vide $\bot$ est considéré comme une clause utilisateur. Un *super- clause* est soit une clause de théorie soit une clause utilisateur.

Outre l'ensemble des clauses à traiter (qui peut être modifié au cours de la recherche), la procédure DPLL maintien l'ensemble de faits clos supposés. Dans notre procédure, ces faits, que nous appelons *super-littéraux*, peuvent être de trois types :

- Un littéral l ;
- Une cloture $\varphi \cdot \sigma$;
- Une anti-cloture $\neg(\varphi \cdot \sigma)$.

Nous étendons la fonction $\mathscr{T}$ représentant l'ensemble des sous-termes d'un objet aux clotures et aux anti-clotures :

$$\begin{aligned} \mathscr{T}(l \cdot \sigma) &\triangleq \mathscr{T}(l\sigma) \\ \mathscr{T}(\varphi \cdot \sigma) &\triangleq \mathscr{T}(\sigma) \qquad \text{si } \varphi \text{ n'est pas un littéral} \\ \mathscr{T}(\neg(\varphi \cdot \sigma)) &\triangleq \varnothing \end{aligned}$$

Les clotures $\varphi \cdot \sigma$, où φ est une formule avec un déclencheur, un témoin ou une formule universellement quantifiée, sont traités comme des boîtes opaques de sorte que les seuls termes auxquels nous avons accès sont ceux apportés par la substitution φ. Une anti-cloture $\neg(\varphi \cdot \sigma)$ ne fournit pas de nouveau terme (et ne doit donc pas être confondue avec $(\neg\varphi) \cdot \sigma)$. En effet, si le solveur décide de supposer une cloture et revient plus tard sur cette décision, il ne devrait pas retenir les termes qu'il avait tirés de cette supposition.

Étant donné un ensemble de super-littéraux M, nous définissons $\textsc{Lit}(M)$ comme l'ensemble des littéraux de M et $\textsc{Clo}(M)$ comme l'ensemble des clotures de M. Étant donné un ensemble de super-clauses F, nous définissons $\textsc{Lit}(F)$ comme l'ensemble des clauses utilisateur unitaires de F, et $\textsc{Clo}(F)$ comme l'ensemble des clauses de théorie unitaires de F.

Pour modéliser les déclencheurs, nous devons trouver un moyen de protéger une super-clause de sorte que ses éléments ne soient disponibles qu'une fois qu'une certaine condition est remplie. Nous définissons une *clause gardée* comme une paire $H \to C$, où la garde H est un ensemble conjonctif des clotures et C est une super-clause. Si M est un ensemble de super-littéraux et F un ensemble de clauses gardées, on définit l'ensemble des super-clauses *disponibles* comme l'ensemble de super-clauses de F dont la garde est contenue dans M :

$$\textsc{Avb}(F,M) \triangleq \textsc{Lit}(M) \cup \textsc{Clo}(M) \cup \{C \mid H \to C \in F \text{ and } H \subseteq M\}$$

Tout raisonnement plus complexe sur les gardes est laissé à DPLL Nous utilisons aussi l'ensemble des gardes de F, défini comme $\textsc{Grd}(F) \triangleq \{H \mid H \to C \in F\}$.

Nous étendons maintenant les définitions 2.4 et 2.5 aux super-littéraux et aux clauses gardées.

Définition 3.1 (Valeur de Vérité modulo T)**.** Étant donné un monde L, nous définissons ce que signifie qu'un super-littéral, une super-clause, une garde ou une clause gardée est vrai est L, noté $L \blacktriangleright_T F$:

$L \blacktriangleright_T l$	$L \models_T l$
$L \blacktriangleright_T \varphi \cdot \sigma$	$L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(\mathscr{T}(\sigma))$ et $L \rhd_T \varphi\sigma$
$L \blacktriangleright_T \neg(\varphi \cdot \sigma)$	si $L \cup \mathit{known}(\mathscr{T}(L)) \models_T \mathit{known}(\mathscr{T}(\sigma))$ alors $L \not\rhd_T \varphi\sigma$
$L \blacktriangleright_T C$	C est une clause utilisateur et $L \models_T C$
$L \blacktriangleright_T C$	C est une clause de théorie et il existe $\varphi \cdot \sigma \in C$ tel que$L \blacktriangleright_T \varphi \cdot \sigma$
$L \blacktriangleright_T H$	H est une garde et, pour tout $\varphi \cdot \sigma \in H$, $L \blacktriangleright_T \varphi \cdot \sigma$
$L \blacktriangleright_T H \to C$	si $L \blacktriangleright_T H$ alors $L \blacktriangleright_T C$

On dit qu'un super-littéral est *faux* dans L lorsque sa négation est vraie dans L. Nous appelons un super-littéral, une super-clause, une garde ou une clause gardée *faisable* s'il existe un monde dans lequel il est vrai. Nous appelons une super-clause, un super-littéral, une garde ou un clause gardée *satisfiable* s'il existe un monde complet dans lequel il est vrai. Nous appelons alors ce monde un *modèle*.

Sur les littéraux usuels et les clauses utilisateur, $\blacktriangleright_T$ coïncide avec $\models_T$: une clause utilisateur C est vraie dans un monde L si et seulement si elle est vraie dans tous les modèles de L. Sur les clotures et les clauses de théorie , $\blacktriangleright_T$ se comporte comme $\rhd_T$: une clause de théorie est vraie dans L si et seulement si l'un de ses éléments est vrai en L. Par un léger abus de notations, nous réutilisons les termes introduits dans les définitions 2.4 et 2.5 ; dans cette section, nous utilisons la définition 3.1.

Nous définissons une première version de l'implication qui traite les clotures comme des atomes opaques dont les arguments sont données par la substitution accumulé. C'est l'implication utilisée dans le solveur DPLL, la sémantique de cloture étant prise en charge par des règles spécifiques.

Définition 3.2. Nous définissons un encodage des super-littéraux et des clauses gardées vers des littéraux et des clauses. Dans les règles ci-dessous, P_φ est un symbole de prédicat frais que nous associons à chaque pseudo-littéral . L'arité de P_φ est le nombre de variables libres dans φ.

$$
\begin{aligned}
[\![l]\!] &\triangleq l \\
[\![l \cdot \sigma]\!] &\triangleq l\sigma \\
[\![\varphi \cdot \sigma]\!] &\triangleq P_\varphi(\mathit{vars}(\varphi))\sigma \qquad \text{si } \varphi \text{ n'est pas un littéral} \\
[\![\neg(\varphi \cdot \sigma)]\!] &\triangleq \neg [\![\varphi \cdot \sigma]\!] \\
[\![e_1 \vee \cdots \vee e_m]\!] &\triangleq [\![e_1]\!] \vee \cdots \vee [\![e_m]\!] \\
[\![(g_1 \wedge \cdots \wedge g_n) \to (e_1 \vee \cdots \vee e_m)]\!] &\triangleq \neg [\![g_1]\!] \vee \cdots \vee \neg [\![g_n]\!] \vee [\![e_1]\!] \vee \cdots \vee [\![e_m]\!]
\end{aligned}
$$

Soit S un ensemble conjonctif de super-littéraux et/ou des clauses gardées. Soit E un super-littéral, une super-clause, ou une clause gardée. Nous définissons $S \models^\star_T E$ comme $[\![S]\!] \models_T [\![E]\!]$.

$\models^\star_T$ est une extension conservative de l'implication habituelle du premier ordre aux clauses gardées. Plus généralement, si $S \models^\star_T E$ alors tout modèle de S et aussi un modèle de E.

Nous avons également besoin d'une version plus faible de l'implication qui préserve à la fois la faisabilité et la satisfiabilité. Nous voulons que cette implication soit aussi proche que possible de l'implication habituelle $\models_T$ sur clauses utilisateur tout en restant facilement calculable par un solveur sur les clauses de théorie. Nous utilisons l'ensemble $\lfloor V \rfloor$ de littéraux clos accessibles à partir d'un ensemble de clauses de théorie V (voir définition 2.9) :

Définition 3.3. Soit F un ensemble de super-clauses et C une super-clause. Nous écrivons $F \vdash^\star_T C$ si et seulement si une des conditions suivantes est vérifiée :

- C est une clause utilisateur unitaire et $\text{LIT}(F) \cup \lfloor \text{CLO}(F) \rfloor \models_T C$;
- C est une clause utilisateur non-unitaire $\{C' \mid C' \text{ est une clause utilisateur de } F\} \cup \lfloor \text{CLO}(F) \rfloor \models_T C$;
- C est une clause de théorie $D \cdot \sigma$ et il existe un littéral $l \in D$ tel que $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(l\sigma))$ and $F \vdash^\star_T l\sigma$;
- C est une clause de théorie $D \cdot \sigma$et il existe une clause de théorie $C' \cdot \sigma' \in F$ telle que $C' \subseteq D$, $F \vdash^\star_T \sigma|_{Dom(\sigma')} \doteq \sigma'$, and $F \cup \mathit{known}(\mathscr{T}(\text{CLO}(F))) \vdash^\star_T \mathit{known}(\mathscr{T}(\sigma|_{Dom(\sigma')}))$.

On peut remarquer que $\vdash^\star_T$ ne coïncide pas avec implication usuelle modulo T sur les clauses utilisateur unitaires. En effet, $\vdash^\star_T$ est utilisé en particulier pour décider qu'une clause n'est pas nécessaire pour la preuve et peut donc être oubliée ou non générée. Dans la définition d'une affectation de vérité, nous imposons au solveur de supposer les clauses unitaires dès qu'elle apparaissent alors qu'il est autorisé à reporter la décision sur les littéraux des clauses non-unitaires. Par conséquent, même si un ensemble de clauses non-unitaire implique une clause unitaire C, le solveur ne peut se permettre d'oublier C sans compromettre la terminaison. Par exemple, considérer l'ensemble d'axiomes :

$$W = \{c \approx c, f(c) \approx f(c), f(c) \approx c, \forall x[f(c) \approx c].f(x) \approx x, \forall x.f(x) \approx f(x)\}$$

L'ensemble W est terminant (chaque terme introduit par le dernier axiome peut-être rendu égal à un terme connu par l'avant dernier). Pourtant, considérons l'ensemble $G = \{f(c) \approx c, f(c) \approx c \vee c \not\approx c\}$. Nous avons $F \setminus \{f(c) \approx c\} \cdot \varnothing \cup G \vdash^\star_T f(c) \approx c \cdot \varnothing$, donc $f(c) \approx c$ peut-être retiré de F. Nous avons également $f(c) \approx c \vee c \not\approx c \models_T f(c) \approx c$. Supposons que nous pouvons retirer $f(c) \approx c$ de G. Alors, le solveur peut produire un nombre infini de termes à partir de $F \setminus \{f(c) \approx c\}$. Il ne peut jamais choisir de déduire $f(c) \approx c$ à partir de $f(c) \approx c \vee c \not\approx c$, ce qui permettrait à tous ces termes de devenir égaux.

Dans les deux derniers cas de la définition3.3, les termes connus sont fournis uniquement par les clotures (c'est-à-dire, les clauses de théorie unitaires) de F et non par les clauses utilisateur. En effet, comme nous l'avons dit plus tôt, nous traitons les clauses utilisateur en utilisant la sémantique du premier ordre usuelle, où un littéral peut être remplacé par un littéral équivalent indépendamment de ses sous-termes .

Étant donnés un ensemble de super-littéraux M, nous écrivons $M \vdash^\star_T C$ pour $\text{LIT}(M) \cup \text{CLO}(M) \vdash^\star_T C$. En d'autres termes, nous traitons les littéraux et les clotures de M comme des clauses utilisateur des des clause de théorie unitaires, et nous ignorons les anti-clotures. En utilisant cette définition, nous avons $M \vdash^\star_T \bot$ dès que l'ensemble $\text{LIT}(M) \cup \text{CLO}(M)$ est insatisfiable.

Dans notre algorithme , nous utilisons des termes clos provenant des clauses utilisateur pour instancier les formules universellement quantifiées et ouvrir des déclencheurs. Pour ces termes soient utilisés par la relation de $\vdash_T^\star$, nous convertissons les littéraux de l'ensemble des faits supposés en clotures. Étant donné un ensemble de super-littéraux M, on définit $\lceil M \rceil = M \cup \{l \cdot \varnothing \mid l \in \text{LIT}(M)\}$. Par conséquent, pour tout termes $t \in \mathscr{T}(M)$, $t \in \mathscr{T}(\text{CLO}(\lceil M \rceil))$.

3.2 Description de DPLL(T) avec des déclencheurs

La méthode présentée ci-dessous adapte les principes DPLL modulo théories (suivant [65]) à des ensembles de super-littéraux et et de clauses gardées . Nous appelons cette adaptation DPLL$^\star(T)$ ou DPLL* puisque T est fixé .

3.2.1 Règles de déduction de DPLL$^\star(T)$

Les règles sont donnés dans les figures 1 et 2. La procédure tente de construire un modèle pour un ensemble de clauses gardées F. Le modèle partiel en cours de construction est représenté comme un ensemble de super-littéraux M, supposés vrais à un instant donné. Nous disons que la paire $M \parallel F$ est l'*état courant* du solveur. Un super-littéral e est *défini* dans M si e ou $\neg e$ appartient à M.

On peut donner une valeur de vérité arbitraire à un élément d'une clause disponibles en utilisant la règle `Decide`. Les super-littéraux de M dont la valeur de vérité a été choisi arbitrairement sont marqués par la lettre `d` et appelés *super-littéraux de décision*. Si tous les éléments d'une clause est faux sauf un, cet élément doit être vrai pour la clause doit être valide. Il peut être propagé en utilisant `UnitPropagate`. Si tous les éléments d'une clause disponible sont faux, alors la clause gardée correspondante est appelée *clause de conflit*. Si il y a une clause de conflit dans F et si M ne contient pas super-littéraux de décision, on passe dans un état spécial, nommé *fail*, pour terminer la recherche. Cela signifie qu'aucun modèle n'a pu être trouvé pour F. La règle `Restart` peut être utilisée pour relancer la recherche à partir de zéro. Si un super-littéral apparaît dans des clauses disponibles ou dans des gardes de F dont la négation conduit à une contradiction modulo T dans les tous modèles de M, on peut le propager en utilisant T`-Propagate`.

L'ensemble des clauses gardé F peut être modifiée au cours de la recherche en utilisant les règles T`-learn` et T`-Forget`. Contrairement à la version classique de DPLL, nous imposons des conditions différentes sur les clauses qui peuvent être apprises et les clauses qui peuvent être oubliées. On peut apprendre toute clause $H \to C$ telle que $F, H \models_T^\star C$, ce qui implique que tout modèle de F est aussi un modèle de $H \to C$. Par contre, nous sommes plus restrictifs sur la définition des clauses pouvant être oubliées.On peut oublier une clause gardée $H \to C$ si $\text{AVB}(F, H) \vdash_T^\star C$. Cette restriction est nécessaire pour la terminaison.

Enfin, si tous les éléments d'une clause disponible de F sont faux et s'il y a au moins un littéral de décision dans M, la règle T`-backjump` peut être appliquée. Elle permet de supprimer une ou plusieurs décisions de M et d'ajouter un nouvel élément impliqué par M et F.

Des règles spécifiques sont nécessaires pour récupérer les informations de contenues dans les clotures. Les formules ajoutées par ces règles à l'ensemble des clauses gardées F sont des tautologies dans la sémantique de formules du premier ordre avec des déclencheurs . La règle

`UnitPropagate` :

$$M \parallel F, H \to C \vee e \Longrightarrow Me \parallel F, H \to C \vee e \quad \textbf{si} \begin{cases} H \wedge \neg C \subseteq M \\ e \text{ n'est pas défini dans } M \end{cases}$$

`Decide` :

$$M \parallel F \Longrightarrow Me^{\mathsf{d}} \parallel F \quad \textbf{si} \begin{cases} e \text{ ou } \neg e \text{ apparaît dans } \textsc{Avb}(F, M) \\ e \text{ n'est pas défini dans } M \end{cases}$$

`Fail` :

$$M \parallel F, H \to C \Longrightarrow \mathit{fail} \quad \textbf{si} \begin{cases} H \wedge \neg C \subseteq M \\ M \text{ ne contient pas de littéral de décision} \end{cases}$$

`Restart` :

$$M \parallel F \Longrightarrow \varnothing \parallel F$$

T`-Propagate` :

$$M \parallel F \Longrightarrow Me \parallel F \quad \textbf{si} \begin{cases} e \notin M \text{ et :} \\ \quad M \models^{\star}_{T} e \text{ et } e \text{ ou } \neg e \text{ apparaît dans } \textsc{Avb}(F, M), \text{ ou} \\ \quad \lceil M \rceil \vdash^{\star}_{T} e \text{ et } e \text{ apparaît dans } \textsc{Grd}(F) \end{cases}$$

T`-Learn` :

$$M \parallel F \Longrightarrow M \parallel F, H \to C \quad \textbf{si} \begin{cases} \text{tous les atomes de } H \text{ apparaissent dans } \textsc{Grd}(F) \cup \lceil M \rceil \\ \text{tous les atomes de } C \text{ apparaissent dans } \textsc{Avb}(F, H) \cup \textsc{Lit}(M) \\ F, H \models^{\star}_{T} C \end{cases}$$

T`-Forget` :

$$M \parallel F, H \to C \Longrightarrow M \parallel F \quad \textbf{si} \begin{cases} \text{toutes les clotures de } C \text{ définies dans } M \text{ apparaissent dans } \textsc{Avb}(F, H) \\ \textsc{Avb}(F, H) \vdash^{\star}_{T} C \end{cases}$$

T`-Backjump` :

$$Me^{\mathsf{d}}N \parallel F \Longrightarrow Me' \parallel F \quad \textbf{si} \begin{cases} \text{il existe } H \to C \in F \text{ telle sue } H \wedge \neg C \subseteq Me^{\mathsf{d}}N \\ \text{il existe } D \subseteq M \text{ tel que :} \\ \quad F, D \models^{\star}_{T} e', \\ \quad e' \text{ n'est pas défini dans } M, \text{ et} \\ \quad e' \text{ ou } \neg e' \text{ apparaît dans } \textsc{Avb}(F, M) \cup \textsc{Lit}(Me^{\mathsf{d}}N) \end{cases}$$

FIGURE 1 – Régles de transition de DPLL$^{\star}(T)$ sur des clauses guardées

`Instantiate` :

$$M \parallel F \implies M \parallel F, (\forall x.C \cdot \sigma) \wedge x \approx x \cdot [x \mapsto t] \to C \cdot (\sigma \cup [x \mapsto t]) \textbf{ si} \begin{cases} \forall x.C \cdot \sigma \in M \\ t \in \mathscr{T}(M) \\ \textsc{Avb}(F, M) \not\vdash^{\star}_{T} C \cdot (\sigma \cup [x \mapsto t]) \end{cases}$$

`Witness-Unfold` :

$$M \parallel F \implies M \parallel F, \langle l \rangle C \cdot \sigma \to l \cdot \sigma, \langle l \rangle C \cdot \sigma \to C \cdot \sigma \quad \textbf{si} \begin{cases} \langle l \rangle C \cdot \sigma \in M \end{cases}$$

`Trigger-Unfold` :

$$M \parallel F \implies M \parallel F, [l] C \cdot \sigma \wedge l \cdot \sigma \to C \cdot \sigma \quad \textbf{si} \begin{cases} [l] C \cdot \sigma \in M \\ \lceil M \rceil \vdash^{\star}_{T} l \cdot \sigma \end{cases}$$

FIGURE 2 – Nouvelles règles de transition pour DPLL$^{\star}(T)$ sur des clauses gardées

`Instantiate` crée une nouvelle instance d'une formule universellement quantifiée de M avec un sous-terme de M. La règle `Witness-Unfold` traduit un témoin $\langle l \rangle C$ comme la conjonction $l \wedge C$. La règle `Trigger-Unfold` utilise le mécanisme des gardes pour protéger les éléments contenus dans la pseudo-clause sous le déclencheur de sorte qu'ils ne puissent être décidés ou propagés qu'une fois que la garde est contenue dans M. Une application de l'une de ces trois règles est dite *redondante* dans F, si les clauses gardées ajoutées sont redondantes dans F, et un clause gardée $H \to C$ est dite redondante dans F si $\text{AVB}(F,H) \vdash^{\star}_{T} C$.

3.2.2 Critères de terminaison de DPLL$^{\star}(T)$

Un solveur implantant DPLL$^{\star}(T)$ tente de construire un modèle pour ensemble de clauses gardées en appliquant les règles décrites dans les figures 1 et 2 d'une manière arbitraire. Nous définissons maintenant quand un tel solveur sera autorisé à s'arrêter, c'est-à-dire à déduire la satisfiabilité ou insatisfiabilité du problème utilisateur G modulo l'extension de la théorie T décrite par l'axiomatisation W :

Propriété 3.1. Le solveur peut répondre *Unsat* sur G si $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^{\star} \_ \parallel \mathit{fail}$.

Propriété 3.2. Le solveur peut répondre *Sat* sur G si $\varnothing \parallel W \cdot \varnothing \cup G \Longrightarrow^{\star} M \parallel F$ où :

(i) $M \vdash^{\star}_{T} \text{AVB}(F,M)$,

(ii) $M \nvdash^{\star}_{T} \bot$, et

(iii) pour toute clause gardée $H \to C$ qui peut être ajoutée par `Instantiate`, `Witness-Unfold`, ou `Trigger-Unfold`, nous avons $\text{AVB}(F,M) \vdash^{\star}_{T} C$.

Remarque 3.1. Quand il n'y a pas de clotures ni de formule quantifiée, le calcul ci-dessus coïncide avec la version classique de DPLL modulo théories à l'exception du fait que l'on ne peut oublier les clauses unitaires que si elles sont impliquées par d'autres clauses unitaires. En conséquence, les modifications de DPLL présentées ici peuvent être implantés comme l'extension d'une implémentation existante.

3.3 Propriétés

Théorème 3.1 (Correction modulo T)**.** *Si le solveur répond* Unsat *sur un ensemble de clauses utilisateurs G avec un axiomatisation Ax correcte d'une extension T' de T alors G n'a pas de modèle dans la théorie T'.*

Théorème 3.2 (Complétude modulo T)**.** *Si le solveur répond* Sat *sur un ensemble de clauses utilisateurs G avec un axiomatisation Ax complète d'une extension T' de T alors G est T'-satisfiable .*

Pour la terminaison, nous avons besoin que l'instanciation soit *équitable*, c'est-à-dire que toutes les instances possibles soient produites en un nombre fini de pas. Pour définir l'équité, nous utilisons la notion de *niveau d'instanciation*. Si un terme t a un niveau d'instanciation n, alors t est le résultat de n pas d'instanciation. Plus formellement, si M est un ensemble de super-littéraux, le niveau d'instanciation $\text{level}_M(t)$ (resp. $\text{level}_M(e)$) d'un terme t (resp. d'un

super-littéral e) est soit un entier positif soit un élément spécial noté ∞. Il est défini comme la limite de la suite level^i_M calculée de la manière suivante :

sur un terme t	$\text{level}^i_M(t) \triangleq \min\{\text{level}^i_M(e) \mid e \in M \text{ et } t \in \mathscr{T}(e)\}$
sur un littéral l	$\text{level}^i_M(l) \triangleq 0$
sur une cloture ou une anti-cloture	$\text{level}^0_M(e) \triangleq 0$ si σ est vide et ∞ sinon
$\varphi \cdot \sigma$ ou $\neg(\varphi \cdot \sigma)$	$\text{level}^{i+1}_M(e) \triangleq 1 + \max\{\text{level}^i_M(x\sigma) \mid x \in Dom(\sigma)\}$

Les opérations min, max et $+$ sont définies de telle sorte que, si S est un ensemble non-vide, $\min(S \cup \infty) = \min(S)$, $min(\varnothing) = \infty$, $\max(S \cup \infty) = \infty$, $\max(\varnothing) = -1$, et $1 + \infty = \infty$. Cette suite est convergeante. En effet, le niveau d'instanciation de chaque terme ou super-littéral peut soit rester toujours infinie soit devenir finie à un certain rang i, auquel cas elle n'est plus modifiée par la suite.

En utilisant cette définition, nous définissons le niveau d'instantiation courant d'un ensemble de super-littéraux M comme $\text{level}(M) = \max\{\text{level}_M(e) \mid e \in M\}$. Nous assurons l'équité en exigeant que instances faisant croître le niveau d'instanciation courant ne soient possibles que lorsque :

- Une affectation de vérité, telle que définie à la section 2, a été atteinte, et
- Toutes les instances possibles depuis plus longtemps et d'un niveau d'instanciation inférieur ont déjà été exécutée.

Ces deux exigences sont obtenues grâce à une restriction sur les dérivations :

Définition 3.4 (Équité)**.** Nous disons que la dérivation est équitable si, pour toute étape $\_ \parallel F \Longrightarrow Me \parallel F$ telle que $\text{level}_M(e) > \text{level}(M)$, e est de la forme $x \approx x \cdot [x \mapsto t]$ et `Instantiate` peut être appliquée avec une formule universellement quantifiée $\forall x.\varphi \cdot \sigma$ et un terme t dans $M \parallel F$. Pour chacune de ces étape, si M' est le plus petit préfixe de M tel que $t \subseteq \mathscr{T}(M')$, alors il existe un préfixe N de M contenant M' et $\forall x.\varphi \cdot \sigma$ tel que :

(a) $N \not\vdash^\star_T \bot$,

(b) pour toute super-clause unitaire $e \in \text{AVB}(F, \varnothing)$, on a $\lceil N \rceil \vdash^\star_T e$,

(c) pour toute cloture $\langle l \rangle C \cdot \sigma \in N$, $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ et, si C est unitaire, $\lceil N \rceil \vdash^\star_T C \cdot \sigma$,

(d) pour toute cloture $[l]\varphi \cdot \sigma \in N$ telle que φest unitaire, si $\lceil N \rceil \vdash^\star_T l \cdot \sigma$ alors on a $\lceil N \rceil \vdash^\star_T \varphi \cdot \sigma$,

(e) pour toute cloture $\forall x.\varphi \cdot \sigma \in M'$ telle que φ est unitaire et pour tout terme $t \in \mathscr{T}(M')$ tel que $\text{level}_M(\varphi \cdot (\sigma \cup [x \mapsto t])) \leq \text{level}(M)$, on a $\lceil N \rceil \vdash^\star_T \varphi \cdot (\sigma \cup [x \mapsto t])$, et

(f) pour toute clause gardée $H \to C$qui peut être ajoutée à F en appliquant `Instantiate`, `Witness-Unfold` ou `Trigger-Unfold` sur une cloture de M', si $\text{level}_M(H) \leq \text{level}(M)$ alors $\text{AVB}(F, M) \vdash^\star_T C$.

Remarque 3.2. On n'est pas obligé d'utiliser les niveaux d'instanciation pour assurer l'équité. En effet, d'après notre définition, il suffit de s'occuper des clauses unitaires, des déclencheurs et des témoins en priorité et de sélectionner les instances dans l'ordre dans lequel elles deviennent possibles.

En utilisant cette définition de l'équité , nous présentons un ensemble de restrictions sur la dérivation qui permettent d'assurer la terminaison :

Théorème 3.3 (Termination)**.** *Il n'y a pas de dérivation Der infinie partant d'un état* $\varnothing \parallel G \cup W \cdot \varnothing$ *avec W terminante telle que :*

- *Der ne contient pas de sous-dérivation infinie composée uniquement d'applications de T* `-Learn`*, T* `-Forget`*, et d'applications redondantes de* `Witness-Unfold`*,* `Trigger-Unfold` *et* `Instantiate`*,*
- *la dérivation est équitable,*
- *pour toute sous-dérivation de Der de la forme :* $S_{i-1} \Longrightarrow S_i \Longrightarrow \ldots \Longrightarrow S_j \Longrightarrow \ldots \Longrightarrow S_k$ *les seules applications de* `Restart` *produisent* S_i*,* S_j *et* S_k *:*
  - *il y a plus de pas de DPLL** *qui ne sont ni des applications de T* `-Learn` *ni des applications de T* `-Forget` *ni des applications redondantes de* `Witness-Unfold`*,* `Trigger-Unfold` *ou* `Instantiate` *dans* $S_j \Longrightarrow \ldots \Longrightarrow S_k$ *que dans* $S_i \Longrightarrow \ldots \Longrightarrow S_j$*, ou*
  - *Une clause gardée ne contenant que des littéraux et des clotures avec une substitution vide est apprise dans* $S_j \Longrightarrow \ldots \Longrightarrow S_k$ *et n'est plus jamais oubliée dans Der.*

Finalement, nous avons besoin propriété de progrès. Elle assure que chaque dérivation qui ne permet pas au solveur de terminer peut être étendu en respectant les restrictions nécessaires à la terminaison.

Théorème 3.4 (Progress)**.** *Si le solveur ne peut pas répondre après une dérivation équitable* $\varnothing \parallel G \cup W \cdot \varnothing \Longrightarrow M \parallel F$*, alors il est une dérivation équitable* $M \parallel F \Longrightarrow^+ S$ *ne contenant pas d'application de* `Restart` *et au moins une étape qui ne soit ni une application de T* `-Learn` *ou de T* `-Forget`*ni une application redondante de* `Witness-Unfold`*,* `Trigger-Unfold` *ou* `Instantiate`*.*

4 Conclusion

Nous avons présenté une sémantique pour la logique du premier ordre avec des déclencheurs et des témoins. Grâce à cette sémantique, nous avons défini trois propriétés de correction, de complétude et de terminaison pour des axiomatisations du premier ordre avec des déclencheurs. Nous avons ensuite décrit l'implémentation d'un solveur SMT qui prend en entrée une axiomatisation de premier ordre avec des déclencheurs d'une théorie T' et se comporte comme une procédure de décision pour T' si l'axiomatisation est correcte, complète, et terminante selon nos définitions. Cette implémentation est formalisée comme une extension du cadre abstrait DPLL(T) et est intégrée dans le solveur SMT Alt-Ergo. Puisque nous avons démontrer dans notre formalisation que l'instanciation des quantificateurs termine pour une axiomatisation terminante, notre implémentation instancie les quantificateurs venant de l'axiomatisation de la théorie de manière avide, c'est-à-dire dès qu'ils apparaissent, sans attendre que tous les littéraux aient été affectés dans le modèle courant de DPLL(T).

Nous avons testé notre application sur plusieurs études de cas : des programmes Why3 écrits spécifiquement et utilisant une théorie pour les listes doublement chaînées impératives avec itérateurs, un ensemble important d'obligations de preuve venant de la vérification de programmes en B et utilisant de manière intensive une théorie des ensembles et des programmes

de SPARK inspirés de la compétition VSTTE et utilisant une bibliothèque de vecteurs. Nous avons comparé notre implémentation avec l'implémentation pour la gestion des quantificateurs de Alt-Ergo, qui instancie de manière paresseuse. En conclusion de ces tests, il apparaît qu'une instanciation avide augmente vraiment l'efficacité quand le problème nécessite beaucoup de raisonnement lié à la théorie. D'un autre coté, lorsque l'axiomatisation de la théorie est de taille importante, cette instanciation avide peut s'avérer coûteuse. C'est d'autant plus évident lorsque les tests nécessitent l'instanciation de beaucoup d'axiomes ne venant pas de la théorie, et donc traités de manière paresseuse et relativement peu de raisonnement spécifique à la théorie, ce qui est le cas pour nos programmes SPARK utilisant des vecteurs.

Dans les solveurs SMT existants, les déclencheurs sont une simple heuristique, ils n'ont pas de sémantique de sorte que choisir les bons déclencheurs pour une axiomatisation peut s'avérer très compliqué pour un utilisateur. Nous avons comparé le comportement des solveurs SMT existants supportant les quantificateurs sur les axiomatisations utilisés dans nos études de cas, avec et sans les déclencheurs fournis manuellement. Les résultats montrent que notre sémantique modélise de manière acceptable les heuristiques de ces solveurs et peut donc aider dans le choix des déclencheurs. En particulier, elle permet de comprendre pourquoi un axiome ralentit un test en induisant trop d'instances et quels sont les déclencheurs qui peuvent être utilisés pour empêcher cette explosion. D'un autre coté, en général, il n'est pas utile de chercher à atteindre la complétude ou la terminaison d'une axiomatisation dans ce contexte, puisque la plupart des solveurs SMT utilisent également d'autres heuristiques d'instanciation que les déclencheurs.

Références

[1] W. Ahrendt, T. Baar, B. Beckert, M. Giese, E. Habermalz, R. Hähnle, W. Menzel, W. Mostowski, and P. H. Schmitt. The KeY system : Integrating object-oriented design and formal methods. In *Fundamental Approaches to Software Engineering*, pages 327–330. Springer, 2002.

[2] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2) :140 – 164, 2003. 12th International Conference on Rewriting Techniques and Applications (RTA 2001).

[3] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, and A. Thums. Formal system development with KIV. In *Fundamental approaches to software engineering*, pages 363–366. Springer, 2000.

[4] R. Bardou. *Verification of Pointer Programs Using Regions and Permissions*. Phd thesis, Université Paris-Sud, 2011.

[5] J. Barnes. Rationale for Ada 2012 : 1 contracts and aspects. *ADA USER*, 32(4) :247, 2011.

[6] J. Barnes. *SPARK : The Proven Approach to High Integrity Software*. Altran Praxis, 2012.

[7] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system : An overview. In *Construction and analysis of safe, secure, and interoperable smart devices*, pages 49–69. Springer, 2005.

[8] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg, 2011.

[9] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. *Handbook of satisfiability*, 185 :825–885, 2009.

[10] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB standard version 2.0. *Technical report, University of Iowa*, december 2010.

[11] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. METEOR : A successful application of B in a large project. In *FM'99—Formal Methods*, pages 369–387. Springer, 1999.

[12] N. Bjørner. Engineering theories with Z3. *Programming Languages and Systems*, pages 4–16, 2011.

[13] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1) :49–87, 1997.

[14] F. Bobot, S. Conchon, E. Contejean, and S. Lescuyer. Implementing polymorphism in SMT solvers. In *SMT'08*, volume 367 of *ACM ICPS*, pages 1–5. ACM, 2008.

[15] F. Bobot, J.-C. Filliâtre, C. Marché, A. Paskevich, et al. Why3 : Shepherd your herd of provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, 2011.

[16] T. Bouton, D. C. B. De Oliveira, D. Déharbe, and P. Fontaine. veriT : an open, trustable and efficient SMT-solver. In *Automated Deduction–CADE-22*, pages 151–156. Springer, 2009.

[17] A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays ? In *Verification, Model Checking, and Abstract Interpretation*, pages 427–442. Springer, 2006.

[18] D. Cansell and D. Méry. Foundations of the B method. *Computing and informatics*, 22(3-4) :221–256, 2012.

[19] S. Chatterjee, S. K. Lahiri, S. Qadeer, and Z. Rakamarić. A low-level memory model and an accompanying reachability predicate. *International journal on software tools for technology transfer*, 11(2) :105–116, 2009.

[20] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model checking*. MIT press, 1999.

[21] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC : A practical system for verifying concurrent C. In *Theorem Proving in Higher Order Logics*, pages 23–42. Springer, 2009.

[22] D. R. Cok and J. R. Kiniry. Esc/Java2 : Uniting Esc/Java and JML. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128. Springer, 2005.

[23] S. Conchon, E. Contejean, and M. Iguernelala. Canonized rewriting and ground AC completion modulo Shostak theories. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 45–59. Springer, 2011.

[24] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[25] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

[26] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's decision procedure for combinations of theories. In *Automated Deduction—CADE-13*, pages 463–477. Springer, 1996.

[27] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7) :394–397, 1962.

[28] L. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *CADE-21*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.

[29] L. de Moura and N. Bjørner. Engineering DPLL(T) + saturation. In *IJCAR 2008*, volume 5195 of *LNCS*, pages 475–490. Springer, 2008.

[30] L. de Moura and N. Bjørner. Z3 : An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[31] L. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *Formal Methods in Computer-Aided Design, 2009. FMCAD 2009*, pages 45–52. IEEE, 2009.

[32] L. de Moura and N. Bjørner. Satisfiability modulo theories : An appetizer. In *Formal Methods : Foundations and Applications*, pages 23–36. Springer, 2009.

[33] A. Degtyarev and A. Voronkov. Equality reasoning in sequent-based calculi. In *Handbook of Automated Reasoning, volume I, chapter 10*. Citeseer, 1996.

[34] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify : a theorem prover for program checking. *J. ACM*, 52(3) :365–473, 2005.

[35] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[36] C. Dross, J.-C. Filliâtre, and Y. Moy. Correct code containing containers. In *Proceedings of the 5th international conference on Tests and proofs*, TAP'11, pages 102–118. Springer-Verlag, 2011.

[37] C. G. Fermüller, A. Leitsch, U. Hustadt, and T. Tammet. Resolution decision procedures. In *Handbook of Automated Reasoning*, pages 1791–1849. Elsevier Science Publishers BV, 2001.

[38] H. Ganzinger and K. Korovin. Theory instantiation. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 497–511. Springer, 2006.

[39] Y. Ge, C. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *CADE-21*, volume 4603 of *LNCS*, pages 167–182. Springer, 2007.

[40] Y. Ge and L. De Moura. Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In *Computer Aided Verification*, volume 5643 of *LNCS*, pages 306–320. Springer, 2009.

[41] A. Goel, S. Krstić, and A. Fuchs. Deciding array formulas with frugal axiom instantiation. In *Proceedings of the Joint Workshops of the 6th International Workshop on Satisfiability Modulo Theories and 1st International Workshop on Bit-Precise Reasoning*, ACM ICPS, pages 12–17. ACM, 2008.

[42] R. Hahnle. Tableaux and related methods. *Handbook of automated reasoning*, 1 :101–178, 2001.

[43] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. In *ACM SIGPLAN Notices*, volume 43, pages 339–348. ACM, 2008.

[44] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[45] B. Jacobs and F. Piessens. The VeriFast program verifier. *CW Reports*, 2008.

[46] S. Jacobs and V. Kuncak. Towards complete reasoning about axiomatic specifications. In *Proceedings of VMCAI-12*, volume 6538 of *LNCS*, pages 278–293. Springer, 2011.

[47] D. Jovanović and C. Barrett. Polite theories revisited. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 402–416. Springer, 2010.

[48] M. Kaufmann and J. Strother Moore. ACL2 : An industrial strength version of Nqthm. In *Computer Assurance, 1996. COMPASS'96,'Systems Integrity. Software Safety. Process Security'. Proceedings of the Eleventh Annual Conference on*, pages 23–34. IEEE, 1996.

[49] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In *Automation of Reasoning*, pages 342–376. Springer, 1983.

[50] K. Korovin. iProver–an instantiation-based theorem prover for first-order logic (system description). In *Automated Reasoning*, pages 292–298. Springer, 2008.

[51] K. Korovin. An invitation to instantiation-based reasoning : From theory to practice. *Volume in memoriam of Harald Ganzinger, Lecture Notes in Computer Science. Springer*, 2009.

[52] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *Fundamental Approaches to Software Engineering*, pages 470–485. Springer, 2009.

[53] D. Larraz, E. Rodríguez-Carbonell, and A. Rubio. SMT-based array invariant generation. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *VMCAI*, volume 7737 of *Lecture Notes in Computer Science*, pages 169–188. Springer, 2013.

[54] K. R. M. Leino. Efficient weakest preconditions. *Inf. Process. Lett.*, 93(6) :281–288, 2005.

[55] K. R. M. Leino. This is Boogie 2. *Manuscript KRML*, 178, 2008.

[56] K. R. M. Leino. Dafny : An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 348–370. Springer, 2010.

[57] C. Lynch and B. Morawska. Automatic decidability. In *Logic in Computer Science, 2002. Proceedings. 17th Annual IEEE Symposium on*, pages 7–16. IEEE, 2002.

[58] C. Lynch, S. Ranise, C. Ringeissen, and D.-K. Tran. Automatic decidability and combinability. volume 209, pages 1026–1047. Elsevier, 2011.

[59] S. McPeak and G. C. Necula. Data structure specifications via local equality axioms. In *Computer Aided Verification*, pages 476–490. Springer, 2005.

[60] M. Moskal. Programming with triggers. In *Proceedings of the 7th International Workshop on Satisfiability Modulo Theories*, ACM ICPS, pages 20–29. ACM, 2009.

[61] Y. Moy and C. Marché. Modular inference of subprogram contracts for safety checking. *Journal of Symbolic Computation*, 45 :1184–1211, 2010.

[62] G. Nelson. Techniques for program verification. *Technical Report CSL81-10, Xerox Palo Alto Research Center*, 1981.

[63] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 11(4) :517–561, 1989.

[64] G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2) :245–257, 1979.

[65] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories : From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6) :937–977, 2006.

[66] I. O'Neill. SPARK – a language and tool-set for high-integrity software development. In J.-L. Boulanger, editor, *Industrial Use of Formal Methods : Formal Verification*. Wiley, 2012.

[67] S. Qadeer. Algorithmic verification of systems software using SMT solvers. In J. Palsberg and Z. Su, editors, *SAS*, volume 5673 of *Lecture Notes in Computer Science*, page 2. Springer, 2009.

[68] J. C. Reynolds. Separation logic : a logic for shared mutable data structures. 2002.

[69] A. Riazanov and A. Voronkov. Vampire 1.1. In *Automated Reasoning*, pages 376–380. Springer, 2001.

[70] P. Rümmer. E-matching with free variables. In *Logic for Programming, Artificial Intelligence, and Reasoning : 18th International Conference, LPAR-18*, volume 7180 of *LNCS*, pages 359–374. Springer, 2012.

[71] S. Schneider. *The B-method : An introduction*, volume 200. Palgrave Oxford, 2001.

[72] S. Schulz. E – A Brainiac Theorem Prover. *Journal of AI Communications*, 15(2/3) :111–126, 2002.

[73] R. E. Shostak. Deciding combinations of theories. In *6th Conference on Automated Deduction*, pages 209–222. Springer, 1982.

[74] J. Souyris, V. Wiels, D. Delmas, and H. Delseny. Formal verification of avionics software products. In *FM 2009 : Formal Methods*, pages 532–546. Springer, 2009.

[75] N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In H.-J. Boehm and C. Flanagan, editors, *PLDI*, pages 387–398. ACM, 2013.

[76] A. Tafat. *Preuves par raffinement de programmes avec pointeurs*. Phd thesis, Université Paris-Sud, 2013.

[77] C. Tinelli and C. Ringeissen. Unions of non-disjoint theories and combinations of satisfiability procedures. *Theoretical Computer Science*, 290(1) :291–353, 2003.

[78] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass version 2.0. In *Automated Deduction—CADE-18*, pages 275–279. Springer, 2002.