Hi-Lite
High Integrity Lint
Integrated with Testing and Execution

Porteur : AdaCore
Yannick Moy - Romain Berrendonner
Tél. 01.49.70.87.75 – moy@adacore.com
Tél. 01.49.70.67.21 – berrendo@adacore.com

12 mars 2010

Table des matières

1 Objet du projet

1.1 Objectifs

L'objectif général du projet Hi-Lite est de promouvoir les méthodes formelles lors du développement d'applications industrielles critiques. Il s'agit de permettre aux développeurs de telles applications non seulement de vérifier complètement les propriétés de sûreté ou les propriétés logiques de leur réalisation, mais aussi de combiner ces méthodes avec les techniques de test et d'analyse statique, le tout de manière totalement intégrée dans leur environnement de développement habituel.

Actuellement, la vérification des propriétés attendues d'un logiciel est assurée le plus souvent de façon partielle, par l'application d'un processus de production du code (comportant par exemple des règles de codage et des phases de revue) et l'utilisation de techniques de **test** (simulation ou test à l'exécution ; entrées-sorties ou instrumentation du code). Avec l'accroissement de la taille des systèmes logiciels, le coût des tests rend cette approche de moins en moins pratique.

En parallèle, les techniques d'**analyse statique** (exécution symbolique, interprétation abstraite) ont atteint un degré d'industrialisation qui permet de les utiliser de manière automatique sur de très grandes applications. Ces techniques génèrent des avertissements (*warnings*) mais ne donnent pas de garanties (notamment au regard des *faux négatifs*), et le traitement des avertissements erronés (*faux positifs*) nécessite une expertise coûteuse. Par ailleurs, elles sont souvent reléguées, dans la pratique industrielle, à la fin du processus de développement logiciel, ce qui diminue leur intérêt pour la détection précoce des problèmes.

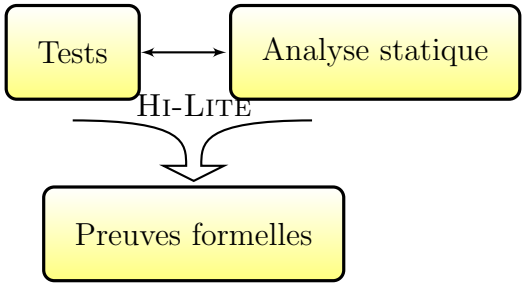
Seuls quelques acteurs se sont tournées vers les techniques de **preuve formelle**[1], en général pour garantir des propriétés de sûreté des applications critiques. Ces techniques requièrent une grande expertise des développeurs afin d'annoter le code avec des propriétés exprimées dans un langage logique, et d'analyser les cas où les propriétés ne sont pas prouvées automatiquement.

Ces trois familles de techniques (tests, analyse statique, preuves formelles) couvrent entièrement le spectre des techniques de vérification de programmes et offrent des garanties croissantes sur la conformité du logiciel aux exigences, ainsi que des limites intrinsèques. Le projet Hi-Lite se propose de dépasser ces limites en introduisant différentes **collaborations** entre techniques, ce qui facilitera le passage des tests à l'analyse statique et aux preuves formelles au sein du développement logiciel.

Ainsi, les tests permettront de mettre au point facilement, par exécution et débogage, les annotations qu'il sera possible d'utiliser pour l'analyse statique ou les preuves formelles. Les résultats de l'analyse statique pourront être traduits en annotations logiques qui permettront de vérifier la complétude des tests et qui faciliteront

[1] *Formal Methods : Practice and Experience*, Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui et John Fitzgerald, `http://www-users.cs.york.ac.uk/~jim/fmsurvey.pdf`

l'établissement des preuves formelles.



Nous viserons en priorité les programmes écrits dans le **langage Ada**, largement utilisé pour la programmation d'applications critiques, aussi bien pour la programmation directement en Ada que comme langage cible de traduction depuis des langages modèles de plus haut niveau (Simulink, SCADE, *etc.*). Nous viserons également les **programmes mixtes Ada/C**, très courants dans les applications critiques, où l'essentiel du développement est fait en Ada, avec certaines bibliothèques ou fonctionnalités développées en C. Pour la partie du projet relative au C, nous nous appuierons sur la plate-forme existante Frama-C développée par un partenaire du projet.

Nous proposons de définir pour chaque langage cible (Ada, C) un **langage d'annotation** commun entre les tests, l'analyse statique et les preuves formelles, qui facilitera le passage d'une technique à l'autre. Cet élément clé de collaboration entre techniques définira des annotations logiques **exécutables** permettant l'expression de **tests unitaires**. Il existe déjà par ailleurs des langages de spécification, permettant la vérification de propriétés de programmes, et des annotations exécutables (assertions), qui permettent de tester dynamiquement ces propriétés. Mais l'utilisateur qui désire à la fois tester et vérifier une propriété doit l'exprimer aujourd'hui de deux façons différentes. Le langage d'annotation que nous proposons permettra de regrouper dans un même fichier source les spécifications, le code et les tests, qui sont aujourd'hui séparés.

L'atout principal du projet Hi-Lite sera de combiner des **outils de vérification** parmi les meilleurs existant, en utilisant le langage d'annotation comme vecteur de communication. Il s'agit notamment d'additionner les capacités de vérification de chaque outil, et d'associer les outils producteurs et consommateurs d'annotations logiques pour obtenir de meilleurs résultats qu'avec chaque outil séparément. Du fait de la complexité combinée des outils considérés individuellement et de la complexité additionnelle des interactions envisagées entre outils, il sera essentiel de proposer à l'utilisateur une **intégration complète** des outils et de leurs interactions à travers une interface de développement adaptée. Le projet Hi-Lite bénéficiera pour cela de l'expertise des membres du projet, à l'origine de tous les outils de vérification et les interfaces de développement utilisés. Les développements liés à Hi-Lite seront disponibles sous **licence libre**, afin de favoriser la collaboration effective sur les outils et la diffusion des résultats du projet.

Notre but est de fournir des outils et techniques d'utilisation « légère » (*lite*) pour le développement d'applications critiques, qu'un développeur pourra appliquer à tout

instant du processus à une réalisation en cours, sur sa machine de développement, pour vérifier certaines propriétés de sûreté et certaines propriétés logiques.

Par rapport aux projets Usine Logicielle et Lambda déjà labélisés par le pôle System@tic, qui mettent davantage l'accent sur un paradigme d'ingénierie par les modèles et dont les objectifs d'analyse statique sont différents, Hi-Lite apportera un certain nombre de nouveautés. Il complétera « par le bas » ces projets en offrant des méthodes innovantes de développement au niveau langage, dans les interstices de la programmation par modèles. Il les complétera également « en parallèle », en offrant des outils d'analyse statique fonctionnelle, alors que ces projets mettent eux l'accent sur l'analyse des propriétés temporelles, de mémoire, ou de stabilité numérique.

Par rapport aux outils existants d'analyse statique utilisés industriellement sur des programmes Ada ou C critiques, tels que PolySpace, Astrée (AbsInt), C Global Surveyor (NASA) ou Penjili (EADS), Hi-Lite apporte une solution aux problèmes de manque de modularité et d'absence d'interaction avec l'utilisateur bien connus des utilisateurs de ces outils. Par l'utilisation exclusive de **techniques modulaires**, y compris en analyse statique avec l'outil CodePeer, Hi-Lite facilitera la vérification de programme plus tôt dans le cycle de développement (pas besoin de tout le programme), par un développeur non-expert (pas d'outil « boîte noire » puisque les contrats de fonction sont disponibles), sur une machine de développement standard (vérifications peu coûteuses car à l'échelle des fonctions), qui pourra prendre en compte les multiples processeurs et cœurs disponibles sur une machine moderne (vérifications en parallèle pour chaque fonction).

Longtemps après la première version du populaire logiciel Lint (1977) pour trouver des erreurs de programmation dans les programmes C, après Splint (1994) qui y ajoutait un langage de spécification, nous proposons de construire un *High-integrity Lint*.

1.2 Approche et positionnement

1.2.1 Propriétés + langages

Propriétés d'intérêt Nous nous intéressons essentiellement aux propriétés de sûreté des applications critiques. Parmi les propriétés logiques, nous nous intéressons principalement à celles qui ont un effet sur la sûreté du programme, ce qui inclut parfois (mais pas toujours) la vérification formelle de certaines fonctionnalités du programme. Nous excluons explicitement les propriétés temporelles, aussi bien celles qui concernent l'enchaînement d'événements que celles qui bornent le temps d'exécution.

Parmi les propriétés visées, on distingue :
- l'absence d'**erreurs à l'exécution**, qui contient la sûreté de la mémoire et du typage : déréférencement de pointeur nul, accès en dehors des bornes d'un tableau (*buffer overflow*), dépassement de valeur entière (*integer overflow*), *etc.* ;
- **l'absence de code inutile**, symptôme d'une erreur logique : code mort, condition toujours vraie ou toujours fausse, affectation de variable non lue, *etc.* ;

- **l'absence de partage mémoire** (*aliasing*) implicite entre entités adressables lues ou écrites dans le même sous-programme ;
- le respect des **contrats de sous-programmes**, exprimés comme relations d'entrées-sorties (précondition et postcondition), et/ou comme ensembles de lectures et d'écritures en mémoire (effets, *footprint* ou *frame condition*) ;
- la propriété de **sous-typage entre classes dérivées**, ou LSP (*Liskov Substitution Principle*), telle que décrite dans le document de travail de la norme avionique DO-178C[2]. Cette propriété consiste à garantir que les méthodes d'une classe dérivée respectent les contrats des méthodes de la classe parent, ce qui permet de vérifier un programme Orienté Objet de façon modulaire.

Toutes ces propriétés sont de nature modulaire, c'est-à-dire qu'elles peuvent être exprimées et vérifiées au niveau des sous-programmes individuels, étant donné un langage d'annotation approprié. Les deux dernière propriétés (contrats et LSP) supposent explicitement l'existence d'un langage d'annotation dans lequel l'utilisateur exprime une propriété d'un programme. Les autres propriétés peuvent se vérifier sur un programme non annoté, mais cela nécessite alors de disposer du programme dans son intégralité. En plus de permettre une vérification modulaire, la présence d'annotations peut faciliter de façon importante la vérification de ces propriétés, en explicitant des informations utiles sur le comportement attendu du programme.

Langage de programmation L'étendue des vérifications envisageables pour un programme dépend autant du langage de programmation utilisé que de la complexité et de la taille du programme considéré. Il est donc crucial de choisir un langage de programmation donnant des garanties fortes et permettant d'exprimer des contraintes riches afin de pouvoir vérifier complètement des propriétés complexes des programmes. Le langage Ada se distingue des autres langages utilisés pour la programmation d'applications critiques (C, C++, C#, Java) par sa richesse d'expression :

- types pointeurs non-nuls (Ada 2005) ;
- types entiers arbitrairement bornés ;
- effets de lecture et/ou écriture sur les paramètres de sous-programmes (paramètres `in, out, in out`).

De façon générale, le langage Ada est largement orienté vers la vérification (statique ou dynamique) de propriétés de programmes, ce qui en fait un langage idéal pour notre projet. Plus particulièrement, nous nous intéressons aux applications critiques programmées en Ada, qui utilisent en général un ensemble restreint du langage : peu de pointeurs, pas ou peu d'allocations dynamiques, interactions réduites entre tâches, *etc.* Nous nous concentrerons sur l'analyse de ces programmes en exploitant certaines restrictions quand c'est utile.

Le langage SPARK définit un tel ensemble de restrictions du langage Ada, qui permet la preuve de propriétés de flot de données, de flot d'information et de propriétés fonctionnelles. Les restrictions fortes de SPARK (fonctions sans effets de bord, initialisation globale des tableaux, pas de pointeurs, pas de partage mémoire, *etc.*) restreignent aussi

[2]Document qui fixe les conditions de sécurité applicables aux logiciels critiques de l'avionique

son utilisation aux parties les plus critiques des applications. Il est courant de programmer le cœur d'une application en SPARK et les parties périphériques en Ada, sans les restrictions de SPARK.

Les meilleurs experts des langages Ada et SPARK sont réunis dans les sociétés AdaCore et SC^2 by Altran, membres du projet Hi-Lite. À ce titre, nous définirons un nouvel ensemble de restrictions du langage Ada qui relâche certaines restrictions de SPARK non essentielles pour la vérification des propriétés de sûreté. Nous prévoyons de traduire le code Ada dans une représentation intermédiaire SPARK, afin de bénéficier des outils de vérification existant pour SPARK. Il existe des précédents de ce type de traduction : le greffon Jessie de Frama-C traduit les programmes C en code intermédiaire Why, ESC/-Java traduit les programmes Java dans un langage intermédiaire de commandes gardées, Spec# traduit les programmes C# en code intermédiaire Boogie. Ces plates-formes font le choix d'une traduction de tous les programmes, ce qui a pour conséquence de rendre la preuve de certaines propriétés particulièrement difficile pour les programmes qui utilisent certaines caractéristiques complexes (comme les pointeurs). Un objectif du projet Hi-Lite sera de déterminer quelles restrictions du langage Ada permettent une traduction idéale pour la preuve automatique, tout en offrant une plus grande facilité d'expression que le langage SPARK.

Langage d'annotation En plus de définir un sous-langage d'Ada, le langage SPARK contient un langage d'annotation, qui permet d'exprimer et de vérifier la plupart des propriétés visées sur les programmes SPARK. D'autres langages d'annotation similaires existent pour C (ACSL), Java (JML) et C# (Spec#). Ces langages d'annotation contiennent tous les expressions booléennes sans effets de bord du langage de programmation correspondant, plus un certain nombre d'annotations non exécutables :

- annotations d'effet des sous-programmes (écritures, lectures) ;
- types logiques et fonctions définies par axiomatisation, c'est-à-dire par un certain nombre de propriétés, et non par une liste de champs (pour un type) ou d'instructions (pour une fonction) ;
- certaines applications des quantificateurs pour lesquelles il n'est pas possible d'énumérer toutes les valeurs de la variable quantifiée ;
- code *ghost* ou modèle, qui n'est utilisé que pour la vérification ;
- axiomatiques (ensemble d'axiomes) utilisées à différentes fins : définition du modèle d'exécution ou du modèle mémoire, définition de propriétés logiques, aide à la preuve automatique, *etc.*

Les annotations non exécutables permettent d'exprimer des propriétés riches, mais elles posent des difficultés supplémentaires par rapport aux annotations exécutables. D'une part, les annotations non exécutables reposent sur une sémantique complexe avec lesquelles les programmeurs ne sont pas familiers. D'autre part, les annotations non exécutables rendent le débogage des spécifications beaucoup plus difficile, car il n'est pas possible de tester ces annotations à l'exécution ou d'exécuter ces annotations pas-à-pas dans un débogueur. Enfin, les annotations non exécutables reposent le plus souvent sur la définition d'axiomes, ce qui peut facilement aboutir à des contradictions qui invalident tous les résultats de preuve obtenus, comme l'a par exemple déjà expérimenté

Astrium sur un de ses projets.

Un exemple classique d'annotation exécutable est l'assertion, qui permet de vérifier en un point de programme qu'une expression booléenne du programme est vraie. En règle générale, on peut même dire que l'assertion n'est vue que comme une annotation exécutable, et pas comme une annotation vérifiable, ce qui autorise par exemple les effets de bord dans les assertions, même si ce n'est pas recommandé. La méthodologie de *Design by Contract* raffine les assertions en préconditions, devant être garanties par l'appelant d'un sous-programme, en postconditions, devant être garanties par le sous-programme appelé, et en invariants maintenus par un ensemble de sous-programmes. À l'origine une spécificité du langage Eiffel, le *Design by Contract* est aujourd'hui repris dans les *Code Contracts* de la plate-forme .NET, les pragmas préconditions et postconditions du compilateur GNAT Pro pour Ada d'AdaCore[3], et les futures préconditions, postconditions et invariants de type pour la prochaine révision de la norme Ada, appelée Ada 201X, proposés par l'ARG (*Ada Rapporteur Group*)[4].

Les tests unitaires peuvent également être considérés comme des annotations exécutables (e.g. clause "for_example" en JML). En effet, un test spécifiant des valeurs d'entrées pour les paramètres entrant d'un sous-programme et des valeurs de sorties pour les paramètres sortant définit un contrat partiel pour ce sous-programme. Cette expression des tests comme contrats partiels est particulièrement intéressante pour effectuer la *Local Type Consistency Verification* prévue dans la future norme DO-178C. En effet, la norme prévoit que cette activité de vérification pourra être assurée par vérification formelle ou par test. L'expression des tests comme contrats partiels permettra facilement de passer de l'un à l'autre et d'accroître si besoin le niveau de garanties, tout en résolvant par construction le problème de traçabilité entre les programmes, les spécifications et les tests.

Nous proposons de définir un langage d'annotation pour Ada, appelé ALFA dans la suite de ce projet, pour *Annotation Language For Ada*. Ce langage devra permettre l'expression des propriétés visées sur les programmes Ada, sous forme de contrats pour les sous-programmes, d'assertions, d'invariants, *etc.* Nous nous fixons comme objectifs principaux de ce langage d'annotation la possibilité d'exprimer les tests unitaires comme contrats partiels, et la possibilité d'exécuter les annotations lors de tests. Un des buts de cette intégration est de permettre aux équipes de développement de vérifier une même propriété à plusieurs niveaux d'assurance, en garantissant la cohérence des vérifications effectuées.

Programmes multi-langages De nombreuses applications écrites majoritairement en Ada contiennent également des morceaux de code écrits en C (utilisation de bibliothèques C préexistantes, optimisation de parties du programme dont l'efficacité est critique, code automatiquement généré par des outils externes, *etc.*). Établir des propriétés pour ces applications requiert donc de spécifier aussi bien le code écrit en Ada que celui écrit en

[3] *Gnat Pro Reference Manual*, Chapter 1, Implementation Defined Pragmas
[4] *Ada 2005 Issues* AI05-0145 et AI05-0146 (`http://www.ada-auth.org/AI05-SUMMARY.HTML`)

C. Dans un tel cadre, les langages de spécification utilisés pour Ada et C doivent être suffisamment proches pour être en mesure d'exprimer le même type de propriétés dans les deux langages.

Il existe déjà un langage de spécification pour le langage C, ACSL (*ANSI C Specification Language*) défini par le CEA LIST et l'INRIA ProVal qui sont membres du projet Hi-Lite. Ainsi, outre la définition du langage ALFA pour exprimer les propriétés visées sur les programmes Ada, nous proposons d'extraire d'ACSL un sous-ensemble cohérent avec ALFA. Pour atteindre ce but, nous adapterons le langage ACSL si besoin est. De plus, nous veillerons tout particulièrement à ce que le sous-ensemble extrait soit totalement exécutable, contrairement à ACSL pris dans son intégralité. Ce sous-ensemble exécutable d'ACSL sera appelé E-ACSL (*Executable ANSI C Specification Language*) dans la suite de ce projet.

1.2.2 Automatisation

La vérification automatique des propriétés des programmes annotés est la condition indispensable de la démocratisation des méthodes formelles. Cela nécessite la collaboration de différents outils autour d'une représentation commune, telle qu'envisagée dans l'*Evidential Tool Bus* de John Rushby[5]. Ce soucis de collaboration entre outils est présent dans la plate-forme Frama-C pour la vérification des programmes C et dans la plate-forme Spec# pour la vérification des programmes C#.

Nous proposons de construire le projet Hi-Lite autour de la spécialisation et de la coopération de différents logiciels libres existants (figure 1) représentant l'état de l'art en vérification et développement de logiciels critiques, développés par différents membres du projet : AdaCore, SC^2 by Altran, le CEA LIST et l'équipe ProVal de l'INRIA. L'utilisation optimale de chacun de ces outils suppose une expertise forte. L'utilisation combinée de différents outils nécessite donc une expertise dans chacun des outils ainsi combinés. Nous utiliserons l'expertise de chacun des membres du projet dans les outils qu'il fournit pour proposer une intégration complète de ces outils et de leur collaboration à travers des environnements de développement, condition essentielle pour permettre une utilisation par des programmeurs non-experts.

GNAT Pro GNAT Pro est le compilateur pour Ada développé par AdaCore depuis 1994, à partir du compilateur du même nom développé à la New-York University depuis 1980. GNAT Pro repose sur la chaîne de compilation GCC. La version commerciale du compilateur est utilisée pour le développement d'applications critiques. Elle est disponible sur une vingtaine de plates-formes natives, et une trentaine de plates-formes croisées. La suite logicielle GNAT Pro comprend le compilateur et un ensemble d'utilitaires pour l'analyse, le test et la navigation dans le code.

[5] *Harnessing Disruptive Innovation in Formal Verification*, John Rushby, International Conference on Software Engineering and Formal Methods, 2006

logiciel	catégorie	experts	licence
GNAT Pro	compilateur	AdaCore	GNU GPL
CodePeer	analyseur	AdaCore	GNU GPL
Examiner	vérificateur et générateur d'OP[6]	SC^2 by Altran	GNU GPL
Simplifier	prouveur	SC^2 by Altran	GNU GPL
Why	générateur d'OP	ProVal	GNU LGPL
Alt-Ergo	prouveur	ProVal	CeCILL-C
Frama-C	analyseur et vérificateur	CEA LIST et ProVal	GNU LGPL

Fig. 1 – Logiciels servant de base à Hi-Lite

CodePeer CodePeer est un générateur de contrats de sous-programmes et un détecteur de bogues pour Ada développé depuis 2008 par SofCheck et AdaCore, sur la base du logiciel Inspector développé depuis 2002 par SofCheck. CodePeer applique des techniques d'exécution symbolique de façon modulaire, depuis les sous-programmes feuilles du graphe d'appel. Les contrats générés sont utilisés à la fois de façon interne lors de la propagation modulaire, et de façon externe lors de phases de relecture de code par un utilisateur.

Examiner L'Examiner est un vérificateur et générateur d'obligations de preuve pour SPARK, développé depuis 1983 par SC^2 by Altran. D'une part, l'Examiner vérifie que le programme respecte les règles SPARK, ce qui comprend notamment la vérification des flux de données et des flux d'information, ce qui garantit entre autres l'absence de lecture de variables non initialisées. D'autre part, l'Examiner génère des obligations de preuve pour garantir l'absence d'erreurs à l'exécution et le respect des contrats de sous-programmes.

Simplifier Le Simplifier est un prouveur automatique pour la logique du premier ordre FDL (format interne défini par SC^2 by Altran), développé depuis 1983 par SC^2 by Altran, sur la base d'un moteur de règles de réécriture en Prolog. D'après les données collectées sur les projets publics en SPARK (Tokeneer, iFACTS), le Simplifier permet de prouver automatiquement 95% des obligations de preuve générées par l'Examiner.

Why La plate-forme Why regroupe un ensemble de logiciels pour la vérification de programmes, qui permettent la preuve automatique ou manuelle de propriétés de programmes grâce à l'intégration d'un grand nombre d'assistants de preuve et de prouveurs automatiques. Depuis la première version en 1998, elle a été utilisée notamment pour prouver des propriétés de programmes C et Java. La plate-forme Why fait parti intégrante de la plate-forme Frama-C.

Alt-Ergo Alt-Ergo est un prouveur automatique de technologie SMT (*Satisfiability Modulo Theories*, une architecture de prouveur aujourd'hui plébiscitée pour la preuve de programmes) développé depuis 2005 au sein de l'équipe ProVal de l'INRIA. Alt-Ergo a été utilisé notamment pour prouver automatiquement des propriétés de programmes C et Java. Alt-Ergo fait parti des plates-formes Caveat et Frama-C pour la vérification des programmes C, utilisées par Airbus pour la certification d'applications embarquées critiques. Dans ce cadre, l'utilisation d'Alt-Ergo fait actuellement l'objet d'une qualification auprès des autorités aéronautiques.

Frama-C Frama-C (*Framework for Modular Analyses of C*) est une plate-forme dédiée à l'analyse et à la vérification de programmes C (`http://frama-c.cea.fr`). Ce logiciel est développé depuis 2004 par le CEA LIST, en collaboration avec l'équipe ProVal de l'INRIA (en particulier dans le cadre des projets ANR-RNTL CAT puis, aujourd'hui, U3CAT), dans le but de remplacer les outils Caveat (développé par le CEA LIST) et Caduceus (développé par ProVal). Grâce à son architecture logicielle extensible, elle facilite la programmation de tout type d'analyseurs de programmes C, des plus simples à déployer (analyseurs syntaxiques et outils de *reverse-engineering* par exemple) à ceux offrant le plus de garanties formelles (comme les interpréteurs abstraits et les outils fondés sur les méthodes déductives). En outre, la plate-forme est également conçue pour faciliter les collaborations inter-analyses grâce au langage ACSL. Ainsi, chaque analyseur peut exprimer les propriétés lui servant d'hypothèses ainsi que celles qu'il est en mesure de vérifier.

Schéma d'interaction La figure 2 présente les objectifs du projet Hi-Lite en terme d'interaction entre les logiciels existants utilisés au niveau du langage Ada. Les bulles du schéma désignent les langages d'expression du code (langage de programmation, langage d'annotation) ou des données intermédiaires. Les boîtes désignent les outils présentés précédemment. Le flux de données définissant l'interaction entre ces outils est représenté par des flèches pleines pour les transformations existantes, et par des flèches en pointillés pour les transformations que nous proposons de définir et de réaliser dans le projet Hi-Lite. Il est utile de remarquer que les transformations que nous proposons auront des répercussions sur les transformations existantes, qu'il faudra donc faire évoluer.

En plus des langages Ada, SPARK et ALFA, ce schéma mentionne cinq autres langages :

- SCIL : *SofCheck Intermediate Language*, le langage intermédiaire utilisé par l'outil CodePeer ;
- Why : le langage intermédiaire utilisé par la plate-forme Why ;
- FDL : *First-order Development Logic*, le langage de logique du premier ordre utilisé par les outils SPARK ;
- SMTLIB : *Satisfiability Modulo Theories Library*, le langage commun de logique du premier ordre utilisé par les prouveurs SMT.
- C/E-ACSL : le langage C et son langage d'annotation exécutable E-ACSL, utilisés en complément d'ADA et d'ALFA dans les applications clientes.

Nous proposons de définir six nouvelles transformations :

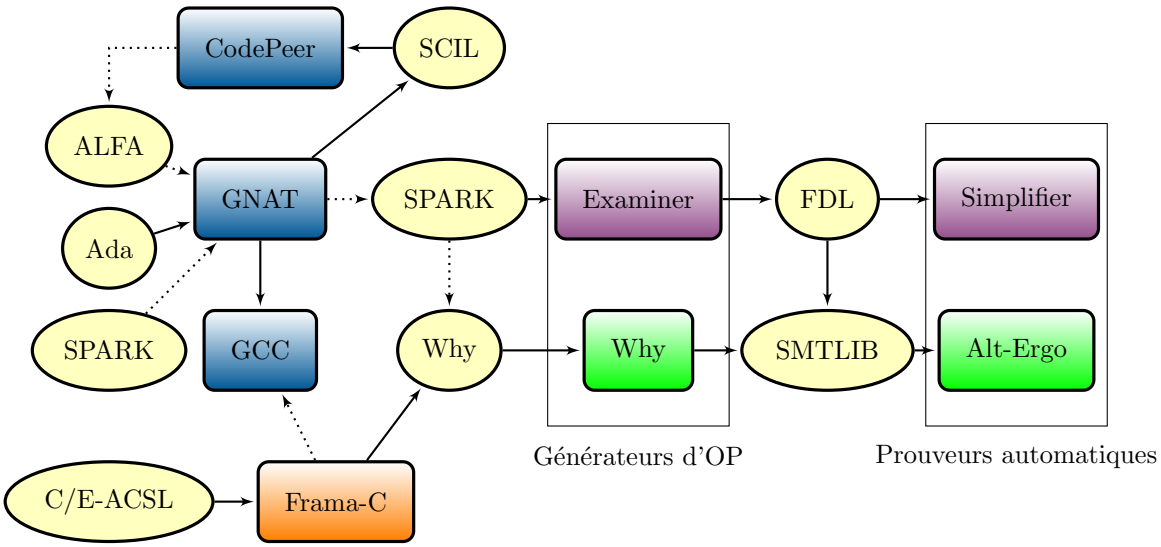


Fig. 2 – Schéma descriptif de l'interaction proposée

- ALFA et SPARK → GNAT : il s'agit de transformer le compilateur Ada afin d'analyser lexicalement, syntaxiquement et sémantiquement les annotations définies par le langage ALFA, une fois celui-ci défini, ainsi que le sous-ensemble des annotations SPARK qui correspondent à des annotations ALFA.
- CodePeer → ALFA : CodePeer génère aujourd'hui des annotations qui lui sont propres. Afin de permettre une utilisation de ces annotations par d'autres outils, il faudra traduire ces annotations dans le langage ALFA.
- GNAT → SPARK : quand le code Ada respecte les restrictions SPARK (ou qu'il est possible de le traduire vers SPARK), et quand les annotations ALFA permettent une traduction vers SPARK, il sera nécessaire de traduire le code annoté vers SPARK.
- SPARK → Why : afin d'utiliser l'outil Why pour générer des obligations de preuve qui soient ensuite prouvées avec l'outil Alt-Ergo, il faudra traduire le code SPARK en code Why.
- Frama-C → GCC : il s'agit de faire générer par Frama-C une instrumentation du code C étudié pour vérifier à l'exécution les annotations E-ACSL fournies en entrée.

Il est important de noter que les transformations existantes forment des blocs de base maîtrisés chacun par un des membres du projet.

La génération d'annotations ALFA par CodePeer contient notamment la génération de cas de tests unitaires, ou vecteurs de test, destinés à exercer une combinatoire importante des chemins d'exécution dans le code. Nous évaluerons la possibilité de compléter ainsi la couverture du code par les tests, en particulier pour assurer l'objectif MC/DC de la norme DO-178, complétant ainsi le travail déjà effectué par AdaCore au sein du projet Couverture du FUI 5. Un outil dédié sera développé pour extraire du code les annotations ALFA représentant des cas de tests et pour générer les *fixtures* permettant de créer l'environnement initial pour ces tests (paramètres des procédures, variables globales). Il sera ensuite possible de lancer automatiquement les suites de tests ainsi générés à travers la plate-forme AUnit développée par AdaCore.

Complémentarité de preuve Certaines propriétés ne s'expriment pas sous forme d'assertions dans le programme : effets de lecture/écriture, absence de code mort ou de code inutile, propriété de sous-typage, *etc.* Ces propriétés ne seront prouvables que par certains outils de notre plate-forme. En dehors de ces propriétés spécifiques, la plupart des propriétés s'expriment sous forme d'assertions en un ensemble de points du programme, qui peuvent donc être prouvées par différents outils, ou par une combinaison d'outil prouvant chacun certaines assertions. Nous proposons principalement trois alternatives de preuve : CodePeer, les outils SPARK (Examiner + Simplifier) et l'utilisation combinée de Why et Alt-Ergo. D'autres variantes sont envisageables, par exemple en traduisant les obligations de preuve générées par l'Examiner dans le format SMTLIB pour utiliser ensuite Alt-Ergo.

Nous ne considérons pas la possibilité d'effectuer des preuves formelles de façon manuelle à l'aide d'un assistant de preuve comme Coq, ce qui requiert une grande expertise de l'utilisateur. En revanche, nous proposons d'utiliser différents prouveurs automatiques, de technologies différentes, afin d'augmenter le ratio d'obligations de preuve prouvées automatiquement. Une **obligation de preuve** se décompose en un but, correspondant par exemple à une assertion du programme, et un ensemble d'hypothèses, qui détaillent le chemin parcouru dans le sous-programme pour atteindre cette assertion. Nous proposons d'intégrer dans Hi-Lite un prouveur de chacune des deux grandes familles de prouveurs automatiques : le Simplifier est un prouveur dit « en arrière », qui part du but pour retrouver les hypothèses ; Alt-Ergo est un prouveur dit « en avant », qui commence par supposer que le but n'est pas atteint et explore les conséquences de cette hypothèse.

Chaque alternative de preuve reposant sur des technologies différentes, on cherchera à tirer le meilleur parti de chaque technologie, en spécialisant les outils correspondants pour certaines catégories d'obligations de preuve. Idéalement, on pourra bénéficier de l'union des résultats individuels obtenus.

1.2.3 Modularité

La modularité est la capacité à vérifier un sous-programme ou un module (un paquetage en Ada) indépendemment des autres sous-programmes ou modules. Cette capacité à décomposer un problème global complexe en un ensemble de sous-problèmes locaux plus simples est la seule solution permettant une vérification « légère » d'une implémentation en cours par un développeur, sur sa machine de développement. C'est le cas de l'approche de compilation séparée suivie par la quasi totalité des compilateurs (comme GNAT Pro). Au contraire, l'absence de modularité est la raison principale du coût prohibitif de certaines analyses statiques. De plus, l'accroissement prévisible du nombre de processeurs et de cœurs dans les ordinateurs individuels bénéficiera principalement aux outils reposant sur des techniques modulaires. Nous avons donc choisi pour Hi-Lite des techniques modulaires : compilation avec GNAT Pro, exécution symbolique avant/arrière avec l'outil CodePeer, preuves formelles avec SPARK et la plate-forme Why.

L'ajout d'annotations ALFA aux programmes est essentielle afin de pouvoir rem-

placer le corps d'un sous-programme par son contrat lors de la vérification modulaire. Un problème essentiel de la vérification modulaire de programmes est donc le coût d'annotation des programmes. Il est courant d'avoir autant d'annotations que de code quand on désire prouver une propriété non triviale d'un programme. On distingue les annotations sur les constructions externes du programme (contrats de sous-programmes, invariants de types) des annotations sur les constructions internes du programme (invariants de boucle, assertions en un point de programme). Les annotations externes correspondent à l'interface naturelle des spécifications de comportement du programme, qui permettent par exemple de blâmer le sous-programme appelant ou le sous-programme appelé quand un appel échoue. Les annotations internes correspondent à des étapes de preuve nécessaires pour garantir la validité des annotations externes. Aussi, les programmeurs ont en général beaucoup plus de facilité à écrire les annotations externes qu'ils n'en ont à écrire les annotations internes, y compris pour les annotations les plus courantes.

Nous proposons d'utiliser l'outil CodePeer afin de générer la plupart des annotations courantes, en particulier pour les annotations internes. Plusieurs modes de génération sont possibles, selon les annotations que l'utilisateur voudra générer. Quel que soit le mode sélectionné, CodePeer génère des annotations nécessaires, toujours correctes vis-à-vis du code analysé, et des annotations suffisantes, qui peuvent être incorrectes car trop imprécises. Il reviendra donc à l'utilisateur de valider les annotations générées, pour inclusion définitive dans le code.

Les annotations d'effet des sous-programmes sont un cas particulier important d'annotations que CodePeer pourrait générer exhaustivement. En particulier, on pourra générer les annotations d'effet attendues par SPARK à partir d'un code partiellement annoté ou non-annoté.

1.2.4 Interactions avec l'utilisateur

Il n'est pas possible d'envisager de vérifier une propriété complexe d'un programme sans interactions avec l'utilisateur. L'absence d'interactions dans la plupart des outils d'analyse statique est la raison principale pour laquelle ces outils sont limités à la découverte de bogues, même quand tous les problèmes potentiels sont identifiés (par exemple, avec PolySpace Verifier). À l'inverse, nous ne recherchons pas des interactions trop détaillées avec les outils de preuve, qui nécessitent une expertise coûteuse. En particulier, nous ne considérons pas la possibilité d'interagir avec un assistant de preuve dans le but de guider la preuve. Nous nous intéressons plutôt à des interactions au niveau du code source, comprenant à la fois le code (Ada, SPARK) et les annotations (ALFA), qui est le niveau idéal d'interactions avec l'utilisateur.

Les conclusions de la plus grande expérience industrielle d'annotation de programmes[7] (400.000 annotations sur plusieurs millions de lignes de code) nous apprennent qu'il est essentiel de prévoir une courbe d'apprentissage progressive pour l'utilisateur, qui apporte

[7] *Modular Checking for Buffer Overflows in the Large*, Brian Hackett, Manuvir Das, Daniel Wang et Zhe Yang, International Conference on Software Engineering, 2006

à la fois des bénéfices immédiats à la première utilisation et des bénéfices incrémentaux au-fur-et-à-mesure de l'investissement de l'utilisateur. C'est cette approche que nous désirons suivre, ce qui aura des conséquences sur la définition des langages d'annotation ALFA et E-ACSL, et sur les interactions prévues avec l'utilisateur.

Dans la suite, nous indiquons les points d'interaction avec l'utilisateur. D'une part, l'utilisateur a la possibilité d'ajouter des annotations dans le programme, qui seront utilisées par les outils. D'autre part, les différents outils produisent des avertissements et des résultats de preuve (positif ou négatif) associés à une localisation dans le programme, pour que l'utilisateur puisse les visualiser sur le code. Cette interaction passe par trois nouveaux outils : l'*Evidential Tool Bus* (mentionné explicitement) et les environnements de développement GPS et GNATbench (implicitement, pour toutes les interactions avec l'utilisateur).

Evidential Tool Bus L'*Evidential Tool Bus* est un outil proposé par John Rushby[8] pour communiquer les résultats de vérification entre outils. Les plates-formes Frama-C pour la vérification des programmes C, Spec# pour la vérification des programmes C#, Mobius pour le code Java certifié et l'outil POGS d'SC^2 by Altran pour la vérification des programmes SPARK sont des exemples d'*Evidential Tool Bus*. Chaque outils intégré dans Hi-Lite a la possibilité de prouver certaines propriétés. L'*Evidential Tool Bus* sera le moyen par lequel l'utilisateur et les différents outils pourront communiquer (renseigner/interroger) sur les propriétés prouvées. La communication avec l'utilisateur se fera au moyen des environnements de développement GPS et GNATbench.

GPS GPS est un environnement de développement initialement dédié aux applications écrites en Ada, développé par AdaCore depuis 2000. GPS propose une interface commune pour tous les outils de développement fournis par AdaCore : compilateur GNAT Pro, débogueur visuel basé sur GDB, vérification de règles de codage GNATcheck, *etc.* Les versions les plus récentes de GPS proposent également une intégration individuelle des outils CodePeer et SPARK (Examiner, Simplifier). GPS est disponible sous licence libre GNU GPL.

GNATbench GNATbench est un greffon (*plug-in*) de l'environnement de développement libre Eclipse, proposé par AdaCore depuis 2005 pour le développement d'applications en Ada. Comme GPS, GNATbench propose une interface intégrée pour tous les outils de développement fournis par AdaCore, y compris les outils CodePeer et SPARK pour les versions récentes. GNATbench est disponible sous licence libre EPL.

Lancement des analyses A travers l'IDE (GPS ou GNATbench), l'utilisateur pourra lancer la vérification à différents niveaux de granularité : le programme complet, le module

[8] *Harnessing Disruptive Innovation in Formal Verification*, John Rushby, International Conference on Software Engineering and Formal Methods, 2006

(ou paquetage en Ada), le sous-programme. Cette interaction fine est rendue possible par la modularité des techniques employées dans Hi-Lite. Idéalement, l'utilisateur devra pouvoir modifier un sous-programme et ne relancer que les générations d'annotations et les vérifications correspondantes, sans invalider d'autres résultats précédemment obtenus.

Interrogation des résultats Nous développerons l'*Evidential Tool Bus* en prenant modèle sur l'outil POGS développé par SC^2 by Altran, qui permet de détailler les résultats de vérification pour chaque obligation de preuve sur un projet. Il devra être possible d'accéder aussi bien aux annotations (utilisateur/générées) qu'aux résultats de preuve (valide/invalide/ne sait pas), de façon détaillée ou résumée.

Validation sélective des résultats Il devra être possible de valider partiellement ou totalement les contrats générés par CodePeer, pour adoption définitive dans le code sous forme d'annotations ALFA.

1.2.5 Relation avec les autres projets

Il convient d'examiner la manière dont Hi-Lite se place, d'une part, par rapport aux axes de développement du pôle System@tic et, d'autre part, par rapport au développement de systèmes complexes dans lesquels le logiciel occupe une part prépondérante.

En ce qui concerne les premiers, Hi-Lite est en ligne à la fois avec les priorités du groupe technique OCDS (Outils de conception et de développement de systèmes) et du groupe technique logiciel libre. Dans le premier cas, c'est à l'égard de la thématique « ingénierie logicielle », et plus particulièrement des approches de vérification de bout en bout qu'Hi-Lite est pertinent. Il permet en effet de s'assurer de la conformité fonctionnelle entre spécification (exprimée sous forme d'annotations) et implémentation. À ce titre, il correspond également aux approches relatives à la certification d'OCDS, puisqu'il permet la cohérence entre les exigences exprimées comme spécifications Ada et cette implémentation. A l'égard du groupe technique logiciel libre, Hi-Lite est en ligne avec la thématique « technologies et outils pour le développement de logiciels libres », notamment dans ses aspects relatifs aux compilateurs de dernière génération et à l'amélioration des techniques de test.

Concernant les projets ensuite, nous nous intéresserons ici uniquement aux projets Usine Logicielle, Lambda et Assert, à l'égard desquels Hi-Lite présente, comme on va le voir, une approche très complémentaire.

Ainsi, le premier sous-projet d'Usine Logicielle, OpenDevFactory, a développé des outils pour modéliser le comportement des applications, étudié les correspondances sémantiques entre modèles, et développé les outils de transformation correspondant, tandis que Hi-Lite ne s'intéresse qu'à la programmation au niveau langage.

Hi-Lite a donc vocation à occuper d'une part les « interstices » de la programmation par modèle, c'est-à-dire à être utilisé pour réaliser cette partie du code que les modèles ne prennent pas en charge : intérieur des boîtes noires ou interfaces de bas niveau par exemple. Mais Hi-Lite utilise également des langages, comme SPARK, ALFA ou ACSL, qui ont une vocation naturelle à être le langage cible utilisé lors de la génération de code exécutable à partir des modèles. Ainsi sera-t-il possible de vérifier que le code généré présente, à un niveau plus microscopique, des propriétés de fonctionnement compatibles avec celles qui auront été démontrées, au niveau macroscopique, sur les modèles. La réalisation complète de ce type d'approche dépasse toutefois le cadre de Hi-Lite.

Hi-Lite, Usine Logicielle, Lambda et Assert sont aussi complémentaires dans leur approche de l'analyse statique. Là où le sous-projet MoDriVal d'Usine Logicielle a développé des outils d'analyse statique visant à démontrer l'absence de débordement mémoire et la stabilité des calculs numériques, deux problématiques critiques dans le monde des systèmes embarqués, là où Assert s'est penché sur les propriétés temporelles d'une application temps réel modélisée en AADL ou en HRT-UML/RCM, Hi-Lite s'intéresse lui à l'analyse statique du comportement fonctionnel du programme, réalisée à partir des annotations et du code.

De même, Usine Logicielle et Hi-Lite sont complémentaires dans leur approche du test : si MoDriVal a adopté une approche permettant la génération de tests de haut niveau en utilisant les exigences comme entrée, Hi-Lite opte quant à lui pour la génération de tests unitaires, à partir des contrats partiels exprimés sous forme d'annotations dans le code.

Le positionnement de Hi-Lite au niveau langage évite par ailleurs tout duplicat avec certains aspects d'Usine Logicielle et, surtout, de Lambda, qui sont spécifiques à l'ingénierie par les modèles. C'est notamment le cas pour la modélisation de la plate-forme d'exécution du logiciel, qui est indispensable au regard des propriétés qu'Usine Logicielle et Lambda cherchent à démontrer. C'est également le cas pour les techniques de transformation de modèles, dans lesquelles la préservation de la sémantique est un problème délicat. Du point de vue de la première, Hi-Lite se préoccupe d'une seule plate-forme d'exécution : le code binaire. Concernant la seconde, peu importent les transformations subies par le modèle dès lors qu'on ne cherche qu'à démontrer des propriétés au niveau du langage.

Ainsi, non seulement Hi-Lite apporte sans redondance des techniques complémentaires à celles mises en œuvre dans les projets existants, mais il prépare aussi le prochain saut conceptuel, celui du *Model Compiler* [9], qui permettra d'intégrer de manière étroite l'approche par modèles et l'approche langages.

Id	D ?	L ?	Livrable
L2.1.1	D		Recommandations pour la définition des langages ALFA et E-ACSL
L3.1.1	D		Manuel de référence du langage ALFA
L3.2.1		L	Version modifiée des outils SPARK
L3.3.1	D		Définition du profil Spark restreint d'Ada
L3.4.1	D		Manuel de référence du langage E-ACSL
L4.1.1		L	Version de GNAT Pro supportant les annotations ALFA
L4.2.1		L	Version de GNAT Pro supportant (partiellement) les annotations SPARK
L4.3.1		L	Outil de traduction de code Ada (respectant le profil Spark) vers SPARK
L4.4.1		L	Outil de traduction de code SPARK restreint vers Why (SPY)
L4.4.2		L	Outil de traduction de code SPARK complet vers Why (SPY)
L4.5.1		L	Greffon Frama-C traduisant E-ACSL vers C
L5.1.1		L	Version modifiée de CodePeer adaptée à ALFA
L5.2.1		L	Version améliorée de CodePeer
L5.3.1		L	Version modifiée de Why
L5.3.2		L	Version modifiée d'Alt-Ergo
L5.5.1		L	Version modifiée de Frama-C
L6.1.1	D		Spécification de la bibliothèque standard Ada de conteneurs
L6.1.2	D		Implémentation en SPARK de la bibliothèque standard Ada de conteneurs
L6.2.1		L	Outil de suivi des objectifs de preuve et des avertissements émis
L6.3.1		L	Version modifiée de GPS
L6.3.2		L	Version modifiée de GNATbench
L7.1.1	D		Rapport d'expérience préliminaire d'usage de la plate-forme
L7.1.2	D		Rapport d'expérience final d'usage de la plate-forme
L7.2.1	D		Rapport d'expérience préliminaire d'application industrielle
L7.2.2	D		Rapport d'expérience final d'application industrielle

Fig. 3 – Liste des livrables

2 Résultats attendus

Chacun des livrables sera doté d'un identifiant unique permettant de le référencer de manière univoque. La liste des livrables du projet est décrite dans la figure 3. Nous indiquons par une lettre la nature du livrable : D pour « Document » indique qu'il s'agit d'un rapport, tandis que L pour « Logiciel » indique qu'il s'agit d'un programme informatique.

[9] *Verifying Model Compilers*, Matteo Bordin et Franco Gasperoni, soumis à Embedded Real Time Software and Systems, 2010

3 Verrous technologiques

3.1 Génération d'annotations précises

La vérification de propriétés complexes d'une implémentation, telles que les propriétés de sûreté et les propriétés logiques, dépend de la présence d'annotations suffisamment précises. Différentes solutions innovantes permettraient chacune un gain de précision important.

Génération de contrats conditionnels

Les annotations doivent généralement prendre en compte différents scénarios d'exécution du code, ce qui s'exprime par la présence de disjonctions dans les formules logiques.

Les techniques d'analyse statique (exécution symbolique, interprétation abstraite, approche CEGAR, *model-checking*) représentent généralement les états abstraits sous forme de conjonctions, et la prise en compte de disjonctions conduit à une explosion combinatoire de ce nombre d'états. CodePeer définit déjà des techniques pour prendre en compte certaines disjonctions dans les postconditions sans explosion du nombre d'états.

Un des défis majeurs pour obtenir des annotations suffisamment précises sera de générer également des préconditions sous forme d'implications (qui sont une forme de disjonction), et de propager ces disjonctions des sous-programmes appelés aux sous-programmes appelants sans faire exploser le nombre d'états. Il sera utile de s'inspirer de ce qui est fait dans des outils comme PREfix[10] qui trouvent des erreurs en ne simulant qu'une partie des chemins d'exécution à l'intérieur d'un sous-programme. Ces outils génèrent en effet des contrats partiels pour chaque sous-programme, sous forme d'implications gardées par des conditions, sans garantie de couvrir l'ensemble des comportements. Avec CodePeer, nous voulons aller plus loin et générer des contrats conditionnels qui couvrent l'ensemble des comportements.

Utilisation des informations de non-partage mémoire

En présence de références (ou de pointeurs) du même type, CodePeer considère, à juste titre, que les références (ou les pointeurs) peuvent éventuellement être égales, ce qui limite la précision des résultats de l'analyse.

Une évolution actuellement proposée pour CodePeer serait de générer des préconditions de non-partage mémoire à chaque fois qu'une postcondition générée dépend justement de cette information de non-partage mémoire.

[10] *A static analyzer for finding dynamic programming errors*, William R. Bush, Jonathan D. Pincus, David J. Sielaff, Software, Pract. Exper., 2000

Dans une perspective plus avancée, il serait aussi possible d'étendre la propriété garantie par SPARK de non-partage des variables accédées dans un sous-programme, aux programmes manipulant des pointeurs. Cela nécessiterait l'utilisation d'annotations de non-partage sur les pointeurs, telles qu'utilisées dans Splint[11] ou dans les systèmes de types pour l'*ownership*[12].

Prise en compte des contextes d'appel

Une difficulté de la génération modulaire d'annotations est l'absence d'informations sur les contextes d'appels réels d'un sous-programme. Cela conduit à la génération de préconditions trop fortes, qui n'autorisent pas certains appels valides du programme.

Dans le cadre d'une analyse globale, quand tout le programme est connu, une solution à ce problème consiste à propager les informations globalement à travers le graphe d'appel. Cela permet de calculer une sur-approximation des contextes d'appel pour chaque sous-programme. L'analyse modulaire n'est en général plus autorisée par la suite à restreindre ce contexte d'appel. Dans le cas des programmes Ada, il est possible de connaître tous les appels d'une fonction définie dans une portée locale, ou non exportée par la spécification du paquetage englobant. Dans ces cas précis, il est possible de propager les informations de contexte des sites d'appels aux sous-programmes appelés.

Une autre solution envisageable dans le cadre d'Hi-Lite est la prise en compte des tests unitaires. En effet, l'expression des tests unitaires comme contrats partiels telle que proposée dans Hi-Lite permet d'accéder aux préconditions de ces tests, qui fournissent des *oracles*, c'est-à-dire une définition par l'utilisateur des contextes d'appels autorisés. Si une précondition générée par CodePeer est incompatible avec la précondition d'un des tests, alors la précondition générée est trop forte. L'utilisation des test unitaires comme oracles permet donc d'éliminer au plus tôt les préconditions trop fortes.

3.2 Preuve de propriétés quantifiées sur des conteneurs

Bibliothèque standard de conteneurs

Afin d'obtenir des garanties fortes par simple typage et des résultats de vérification satisfaisants[13], SPARK impose des restrictions très fortes par rapport à Ada. En particulier, l'utilisation d'allocations dynamiques et de pointeurs est interdite, ce qui force l'utilisateur à travailler avec les tableaux statiques comme uniques structures de données.

Afin de permettre l'expression et la vérification propriétés plus riches, nous fournirons une spécification de la bibliothèque standard de conteneurs d'Ada, qui définit les tableaux

[11] *Static Detection of Dynamic Memory Errors*, David Evans, PLDI, 1996
[12] *Universes : Lightweight Ownership for JML*, Werner Dietl and Peter Müller, JOT, 2002
[13] Typiquement, 95% des obligations de preuve sont prouvées automatiquement.

dynamiques, les listes doublement chaînées, les relations et les ensembles. Ces conteneurs se présentent sous la forme de paquetages génériques en Ada, utilisant de façon intensive des allocations dynamiques et des pointeurs. Nous fournirons une ou plusieurs interface (taille bornée ou non) pour ces conteneurs, annotées en ALFA, qui donneront à l'utilisateur une spécification exécutable des types abstraits algébriques usuels. De façon interne, les conteneurs seront traités spécialement, afin de pouvoir prouver également les propriétés classiques qui leur sont associées, en tant que types abstraits algébriques. Cela nécessitera d'ajouter les paquetages génériques (c'est-à-dire paramétrés par des types, des valeurs, *etc.*) définis en Ada au langage SPARK, une facilité envisagée depuis longtemps et demandée par de nombreux utilisateurs industriels de SPARK.

Conteneurs et quantification

La possibilité de désigner des ensembles n'est vraiment intéressante qu'à partir du moment où on autorise également l'expression de propriétés sur ces ensembles, à l'aide de quantificateurs universels et existentiels. Alors que de telles propriétés sur les tableaux s'expriment facilement en SPARK en quantifiant sur un indice entier du tableau, il n'existe pas de tel mécanisme pour quantifier sur un conteneur, où l'équivalent de l'indice serait un itérateur. Nous fournirons donc un mécanisme de quantification pour les conteneurs. D'autre part, l'utilisation généralisée de quantificateurs, y compris telle qu'elle est autorisée en SPARK (et en général en logique du premier ordre), conduit à des spécifications non exécutables. Pour éviter ce problème, nous restreindrons l'usage des quantificateurs qui ne pourrons porter que sur des conteneurs, à la manière de ce qui est fait dans le langage SETL ou dans les *Code Contracts* de la plate-forme .NET, et non sur des types, à l'exception des types discrets bornés (qui représentent effectivement un ensemble fini d'entiers).

Preuve automatique sur les conteneurs

Les conteneurs permettent de manipuler de façon abstraite des types de données algébriques tels que les ensembles et les relations. Cependant, il existe peu de travaux de recherche sur la preuve automatique de programmes manipulant de tels types algébriques de façon abstraite[14]. Le défi sera donc d'adapter les travaux existant sur la preuve automatique de modèles manipulant des ensembles et des relations de façon abstraite à la preuve de programmes.

[14]Un rare exemple de tels travaux : *On Decision Procedures for Set-Valued Fields*, Viktor Kuncak et Martin Rinard, MIT Technical Report, 2004

3.3 Interactions modulaires avec l'utilisateur

Interface modulaire

Bien que toutes les techniques sur lesquelles repose le projet soient modulaires, donc applicables à un sous-programme isolé étant donné les informations relatives aux sous-programmes appelés, aucun des outils correspondants ne permet à ce jour une utilisation modulaire. Cette situation ne permet pas l'utilisation des outils par le programmeur lui-même en cours de développement, ce qui est l'objectif du projet. Il sera donc nécessaire de développer une interface d'interaction modulaire avec chacun des outils.

Ce fonctionnement modulaire devra prendre en compte les contrats des sous-programmes appelés, idéalement à travers une interaction avec l'*Evidential Tool Bus*. D'autre part, il devra être possible de minimiser le nombre de propriétés à reprouver après une modification du code, ce qui implique de pouvoir contrôler finement les modifications apportées au code et leur impact sur les propriétés à vérifier.

Affichage des chemins d'exécution

Le retour au source classique consistant en une localisation ponctuelle, c'est-à-dire une ligne de code du programme, un point de programme (ligne + colonne) ou un segment de programme (ligne + colonne début + colonne fin) n'est pas suffisant en général pour comprendre un avertissement donné par un outil ou l'échec d'une vérification. La notion plus précise de chemin, qui consiste en une liste d'éléments de localisation ponctuels, est plus adaptée. Elle permet d'afficher à l'utilisateur un chemin d'exécution dans le code amenant à l'opération erronée ou qui ne peut pas être prouvée. Ce type d'interactions est déjà largement utilisé dans les outils d'analyse statique (Coverity Prevent, Grammatech CodeSonar, PREfix, *etc.*) pour afficher à l'utilisateur des avertissements. Idéalement, l'affichage dans un navigateur internet (utilisé localement) permet de parcourir un chemin à travers les fonctions appelées à l'aide d'hyperliens.

Dans le cadre d'Hi-Lite, il serait désirable d'afficher sous forme de chemin d'une part les avertissements donnés par CodePeer et d'autre part les obligations de preuve générées par l'Examiner ou par Why, en particulier quand le prouveur automatique échoue. Enfin, il serait particulièrement intéressant d'ajouter à Alt-Ergo la possibilité de donner un contre-exemple traduisible en chemin, quand un tel contre-exemple est trouvé de façon interne. Les environnements de développement GPS et GNATbench seront chargés d'afficher le chemin.

Traçabilité des résultats de vérification

Afin de faciliter l'utilisation des outils développés dans le projet dans un contexte de certification, tel que celui prévu dans la norme DO-178C pour l'avionique, nous aurons comme objectif d'assurer la traçabilité des vérifications effectuées.

Hi-Lite

Notre approche comportant l'utilisation de plusieurs techniques de preuves, à travers des outils différents, il sera déterminant de pouvoir identifier précisément un objectif de preuve particulier, grâce à l'utilisation de l'*Evidential Tool Bus*.