SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**

UFC UNIVERSITÉ DE FRANCHE-COMTÉ

Contribution à la vérification de programmes C par combinaison de tests et de preuves

GUILLAUME PETIOT

cea list

femto-st SCIENCES & TECHNOLOGIES
DISC

SPIM

Thèse de Doctorat

UFC

école doctorale **sciences pour l'ingénieur et microtechniques**

UNIVERSITÉ DE FRANCHE-COMTÉ

THÈSE présentée par

GUILLAUME PETIOT

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Contribution à la vérification de programmes C par combinaison de tests et de preuves

Soutenue publiquement le 04 novembre 2015 devant le jury composé de :

CATHERINE DUBOIS	Président	Professeur à l'ENSIIE
JACQUES JULLIAND	Directeur	Professeur à l'Université de Franche-Comté
ALAIN GIORGETTI	Encadrant	MCF à l'Université de Franche-Comté
NIKOLAI KOSMATOV	Encadrant	Ingénieur-chercheur au CEA Saclay
PASCALE LE GALL	Rapporteur	Professeur à CentraleSupélec
VIRGINIE WIELS	Rapporteur	Maître de recherche à l'ONERA-Toulouse
DILLON PARIENTE	Examinateur	Ingénieur R&D à Dassault Aviation

REMERCIEMENTS

Je remercie tout d'abord Pascale Le Gall et Virginie Wiels d'avoir accepté de rapporter ma thèse, et Dillon Pariente pour d'avoir accepté d'examiner mes travaux. Merci également à Catherine Dubois d'avoir accepté de présider mon jury de thèse. Je remercie chaleureusement Nikolaï Kosmatov, Jacques Julliand et Alain Giorgetti pour leur patience, leur disponibilité et leurs conseils qui m'ont permis de réaliser cette thèse dans de bonnes conditions. Je remercie tous les membres du jury pour leurs commentaires et remarques qui ont contribué à améliorer la qualité de ce manuscrit.

Je remercie également tous les membres du LSL pour leur accueil et leur partage de connaissances. Merci à Julien Signoles pour le support moral et pour avoir partagé son bureau pendant ces longues années. Merci aux membres de l'équipe FRAMA-C et aux membres de l'équipe PATHCRAWLER pour l'assistance fournie sur ces outils. Merci à tous ceux qui animent les pauses café et qui contribuent à la bonne ambiance du laboratoire.

Je tiens finalement à remercier ma famille et mes amis pour m'avoir soutenu et encouragé pendant ces trois années.

SOMMAIRE

1

INTRODUCTION

Cette thèse se place dans le contexte de la vérification et de la validation des logiciels. Les principales approches de ce domaine sont présentées en partie 1.1. Nous y présentons notamment les outils que nous utilisons dans nos travaux : PATHCRAWLER en partie 1.1.3.1 et FRAMA-C en partie 1.1.5. Nous expliquons la problématique de nos travaux et nos motivations en partie 1.2. Les contributions de cette thèse sont présentées en partie 1.3 et le plan de la thèse est annoncé en partie 1.4.

1.1 VÉRIFICATION ET VALIDATION DE PROGRAMMES

Le domaine de la vérification et de la validation regroupe un ensemble de techniques du cycle de développement des logiciels qui ont pour objectif de s'assurer de leur bon fonctionnement (notamment de leur correction et de leur sûreté). Ces deux notions sont apparues dans les années soixante-dix avec les travaux de DIJKSTRA [Dijkstra, 1975], FLOYD [Floyd, 1963] et HOARE [Hoare, 1969].

[IEEE, 2011] définit la validation comme l'assurance qu'un produit, service ou système répond au besoin des utilisateurs extérieurs. La vérification est l'évaluation d'un produit, service ou système afin de déterminer s'il est conforme à des règles, des spécifications ou des contraintes. Contrairement à la validation, la vérification est le plus souvent un processus interne.

Nous présentons d'abord en partie 1.1.1 une technique de vérification appelée *model-checking* applicable aux systèmes à nombre fini d'états et qui consiste à examiner toutes les exécutions du système. En partie 1.1.2 nous présentons des méthodes de vérification par analyse statique du code, c'est-à-dire ne nécessitant pas d'exécuter le programme. En partie 1.1.3 nous présentons des méthodes de validation par analyse dynamique du code, c'est-à-dire nécessitant d'exécuter le programme. Celles-ci permettent d'effectuer la vérification du programme dans le cas particulier où le nombre d'exécutions est fini et suffisamment petit. En partie 1.1.4 nous présentons les langages de spécification, qui permettent d'encoder de manière formelle les propriétés du programme à vérifier. Enfin, nous présentons en partie 1.1.5 la plateforme de vérification FRAMA-C qui sert de base à nos travaux.

1.1.1 MODEL-CHECKING

Le model-checking [Queille et al., 1982, Clarke et al., 1986, Henzinger et al., 1994, Clarke et al., 2009] permet de vérifier algorithmiquement si un modèle donné (le système ou une abstraction de ce système) satisfait une spécification, formulée en logique temporelle [Clarke et al., 1982]. Un modèle est un ensemble d'états, de propriétés que vérifie chaque état, et de transitions entre ces états qui décrivent l'évolution du système.

Le model-checking couvre l'ensemble des états et des transitions du système afin d'analyser toutes les exécutions possibles du système. Sur de grands systèmes, cette méthode est pénalisée par l'explosion combinatoire du nombre des états (et la complexité en temps ou en espace qui en résulte). Il est néanmoins possible de modéliser des algorithmes asynchrones répartis et des automates non déterministes, comme le permet notamment l'outil SPIN [Holzmann, 1997].

1.1.2 ANALYSE STATIQUE

L'analyse statique [Nielson et al., 1999] examine le code source du programme sans l'exécuter. Elle raisonne sur tous les comportements qui pourraient survenir lors de l'exécution et permet donc de déduire des propriétés logiques devant être vérifiées pour toutes ces exécutions, dans le but de prouver la correction du programme.

En revanche, la vérification de programme étant en général indécidable [Landi, 1992], il est souvent nécessaire d'utiliser des sur-approximations, ce qui implique que les résultats peuvent être moins précis que ce que l'on souhaite mais qu'ils sont garantis pour toutes les exécutions. Ainsi, on peut établir des propriétés de sûreté (*safety*), où l'on cherche des invariants sur les valeurs des variables du programme (une plage de valeurs par exemple), afin d'exclure certains risques d'erreurs à l'exécution.

Nous présentons les principes de l'analyse statique par interprétation abstraite en partie 1.1.2.1. Nous présentons en partie 1.1.2.2 les principes de l'abstraction à base de prédicats permettant d'obtenir une abstraction du système à vérifier. Puis nous présentons l'analyse statique par preuve (ou vérification déductive) en partie 1.1.2.3.

1.1.2.1 INTERPRÉTATION ABSTRAITE

L'interprétation abstraite [Cousot et al., 1992] s'appuie sur les théories du point fixe et des domaines pour introduire des sur-approximations des comportements d'un programme. Elle consiste à abstraire les domaines des variables par d'autres domaines qui peuvent éventuellement être plus simples. Par exemple, le domaine des entiers pourrait être abstrait par un domaine de trois valeurs $(-, 0, +)$ symbolisant respectivement les entiers négatifs, l'entier nul et les entiers positifs. Une analyse de valeurs par interprétation abstraite consiste à calculer à chaque ligne du code une sur-approximation de l'ensemble des valeurs prises par chaque variable en cette ligne lors de toutes les exécutions du programme, permettant ainsi de détecter certaines erreurs comme les divisions par zéro ou les accès en dehors des bornes des tableaux.

Pour contourner le problème d'indécidabilité, la théorie de l'interprétation abstraite construit une méthode qui, à la même question, répondra “oui”, “non” ou “peut-être”. Si la méthode répond “peut-être”, c'est qu'on n'a pu prouver ni l'un ni l'autre des deux pre-

miers cas. C'est ce qu'on appelle une alarme : il est possible qu'une des exécutions du programme produise une erreur donnée, mais la méthode n'est capable ni de le confirmer ni de l'infirmer. L'erreur signalée par une alarme peut ne jamais apparaître à l'exécution, dans ce cas on l'appelle fausse alarme. On ne calcule donc pas la propriété exacte mais une approximation de cette propriété, en imposant la contrainte de sûreté suivante : "la propriété abstraite calculée ne doit oublier aucune exécution concrète". L'abstraction est effectuée à base de prédicats atomiques définissant des abstractions des domaines des variables.

POLYSPACE [Zitser et al., 2004] utilise l'interprétation abstraite pour détecter les erreurs à l'exécution dans les programmes en C, C++ et Ada mais signale beaucoup de fausses alarmes. L'ENS a développé ASTRÉE [Cousot et al., 2007], spécifique au langage C et aux logiciels critiques. FLUCTUAT [Goubault et al., 2011] mesure précisément les approximations faites à l'exécution d'un programme C. FRAMA-C intègre un greffon d'interprétation abstraite nommé VALUE [Canet et al., 2009, Kirchner et al., 2015].

1.1.2.2 ABSTRACTION À BASE DE PRÉDICATS

L'abstraction à base de prédicats [Schiller et al., 2012, Ball, 2005] est une technique permettant de générer automatiquement des abstractions ayant un nombre d'états fini. Pour un programme P au nombre d'états infini, un ensemble fini de prédicats $E = \{f_1, ..., f_n\}$ est défini. Ces prédicats sont des expressions booléennes sur les variables de P et les constantes du langage.

Chaque état concret de P est mis en correspondance avec un état abstrait de l'abstraction de P, après évaluation par les prédicats de E. Un état abstrait est un n-uplet de valeurs booléennes des n prédicats. Par exemple, si 3 prédicats f_1, f_2, f_3 sont définis et si la valeur de ces prédicats à l'état concret e est évaluée respectivement à $false, true, true$, alors l'état e correspond à l'état abstrait $(\neg f_1, f_2, f_3)$ dans l'abstraction générée.

L'abstraction générée comporte un nombre fini d'états (au plus 2^n) car il n'y a qu'un nombre fini de prédicats. Le model-checking peut donc être appliqué à cette abstraction. Si une propriété de sûreté est vérifiée dans l'abstraction, elle l'est également dans le système concret.

Cette technique est notamment utilisée par SLAM [Ball et al., 2011].

1.1.2.3 PREUVE DE PROGRAMMES

La vérification déductive, ou "preuve de programmes", utilise des fondements mathématiques et logiques pour prouver des propriétés de programmes [Hoare, 1969]. Cette technique nécessite de donner au moins une postcondition à prouver, exprimée dans un langage de spécification formelle (voir 1.1.4). Le calcul de la plus faible précondition [Dijkstra, 1975] génère des formules appelées obligations de preuve, qui sont soumises à un prouveur de théorèmes, qui applique différentes techniques de résolution.

Contrairement au model-checking, la preuve a l'avantage d'être indépendante de la taille de l'espace des états, et peut donc s'appliquer sur des systèmes de grande taille et de taille infinie. En contre-partie, cette technique requiert une expertise de l'utilisateur pour adapter le programme à la preuve (en l'annotant par exemple) et guider le prouveur si

nécessaire.

Il existe des prouveurs automatiques tels que SIMPLIFY [Detlefs et al., 2005], ERGO [Conchon et al., 2007], ALT-ERGO [Conchon, 2012], CVC3 [Barrett et al., 2007], CVC4 [Barrett et al., 2011] et Z3 [De Moura et al., 2008] ; et des prouveurs interactifs, où la preuve est guidée par l'utilisateur, tels que COQ [Castéran et al., 2004], ISABELLE [Wenzel et al., 2008], et HOL [Gordon et al., 1993]. Certains de ces prouveurs ou assistants de preuve sont intégrés à d'autres outils tels que BOOGIE [Barnett et al., 2006] ou ESC-JAVA [Flanagan et al., 2002]. FRAMA-C intègre le greffon de preuve WP [Correnson, 2014] qui génère des formules logiques à prouver à partir de la spécification d'un programme, et invoque différents prouveurs afin de vérifier ces formules.

1.1.3 ANALYSE DYNAMIQUE

L'analyse dynamique est basée sur des techniques d'exécution du programme, de simulation d'un modèle [Whitner et al., 1989] ou d'exécution symbolique [Clarke, 1976, King, 1976].

Ces techniques permettent notamment de réaliser de la génération de tests pour un système. Le test peut être utilisé tout au long du cycle de développement d'un logiciel. Les tests unitaires vérifient le bon fonctionnement des différentes entités d'un système, indépendamment les unes des autres. Les tests d'intégration vérifient la bonne communication entre ces entités. Les tests de validation s'assurent que les fonctionnalités correspondent au besoin de l'utilisateur final. Enfin, les tests de non-régression vérifient que l'ajout de nouvelles fonctionnalités ne détériore pas les anciennes fonctionnalités.

En général, les techniques de test ne sont pas exhaustives et n'explorent qu'un sous-ensemble des chemins d'exécution du programme. En conséquence, l'absence d'échecs lors du passage des tests n'est pas une garantie absolue de bon fonctionnement du système. Néanmoins, selon les critères utilisés pour la génération des tests, et selon le niveau de couverture fourni par les tests, on peut acquérir un certain niveau de confiance dans un système ainsi validé.

Les méthodes de test peuvent être classées en trois catégories : le test aléatoire, le test structurel et le test fonctionnel. Le test aléatoire consiste à générer des valeurs d'entrée du programme au hasard. Il ne sera pas détaillé dans cette thèse. Nous présentons la génération de tests structurels en partie 1.1.3.1 et la génération de tests fonctionnels en partie 1.1.3.2. Leur l'objectif est de produire des entrées pour le programme sous vérification. Nous présentons en partie 1.1.3.3 la validation à l'exécution, qui opère sur des programmes dont les entrées sont données.

1.1.3.1 TEST STRUCTUREL

Le test structurel, ou test “boîte blanche”, est une technique de test qui fonde la détermination des différents cas de test sur une analyse de la structure du code source du programme étudié. On distingue deux types de tests structurels : le test orienté flot de contrôle et le test orienté flot de données.

Le test orienté flot de données cherche à couvrir certaines relations entre la définition

d'une variable et son utilisation, par exemple, on peut souhaiter couvrir toutes les lectures d'une variable suivant une écriture.

Le test orienté flot de contrôle s'intéresse quant à lui à la structure du programme : l'ordre dans lequel les instructions sont exécutées. Il se base sur le graphe de flot de contrôle du programme : c'est un graphe connexe orienté avec un unique nœud d'entrée et un unique nœud de sortie, dont les nœuds sont les blocs de base du programme et les arcs représentent les branchements (conditions). Une couverture structurelle de ce graphe est recherchée, selon un critère qui peut être par exemple "toutes les instructions", "toutes les branches" (toutes les décisions) ou "tous les chemins".

PATHCRAWLER [Williams et al., 2004, Botella et al., 2009] est un outil de génération de tests structurels pour les programmes C. C'est le générateur de tests que nous utilisons dans nos travaux. Il est possible de l'utiliser sous la forme d'un service web : PATHCRAWLER ONLINE [Pat, 2011].

PATHCRAWLER utilise la technique d'exécution symbolique dynamique, ou exécution "concolique", qui associe l'exécution concrète du programme et l'exécution symbolique afin d'explorer les chemins du programme. L'exécution concrète permet de confirmer que le chemin parcouru lors de l'exécution d'un test est bien celui pour lequel le cas de test exécuté a été généré. L'exécution symbolique permet de mettre en relation les chemins et les sorties du programme avec les entrées, produisant des "prédicats de chemin". Un prédicat de chemin est un ensemble de contraintes sur les entrées qui caractérisent un chemin d'exécution du programme. Le principe de l'exécution concolique est de produire progressivement de nouvelles entrées pour le programme sous test, dans le but d'exécuter d'autres branches du programme. Ce processus est répété jusqu'à ce que le code du programme soit entièrement couvert par rapport au critère de couverture sélectionné.

Plusieurs outils se basent sur l'exécution concolique pour explorer un programme sous test, dont SMART [Godefroid, 2007], PEX [Tillmann et al., 2008], SAGE [Godefroid et al., 2008], CUTE [Sen et al., 2006], KLEE [Cadar et al., 2008a], EXE [Cadar et al., 2008b], et PATHCRAWLER [Williams et al., 2004, Botella et al., 2009]. Ces outils utilisent des solveurs de contraintes pour générer des cas de test permettant d'aboutir à une couverture souhaitée du programme par les tests.

1.1.3.2 TEST FONCTIONNEL

Le test fonctionnel, ou test "boîte noire", génère des jeux de test en fonction du comportement attendu du programme, qui peut être exprimé sous la forme d'une spécification formelle (un contrat) ou d'informations de conception informelles (documentation, cahier des charges). Le test fonctionnel est utilisé pour vérifier la conformité des réactions du logiciel avec les attentes de l'utilisateur, sans connaissance du code source. Il existe de nombreuses techniques qui se différencient par la manière de choisir les données de test, parmi lesquelles :

- **le test de partition :**
  les valeurs d'entrées du logiciel sont regroupées en classes d'équivalence, sur lesquelles le logiciel doit avoir le même comportement (*domain splitting*), une seule valeur aléatoire est choisie dans chaque classe de la partition ;
- **le test aux limites :**
  les données de test sont choisies aux bornes des domaines de définition des variables.

GATEL [Marre et al., 2000] est un générateur de tests fonctionnels qui se base sur une représentation symbolique des états du système : le programme, les invariants et les contraintes décrivant l'objectif de test sont exprimés dans le langage LUSTRE [Halbwachs et al., 1991]. Cet outil offre la possibilité de réaliser des partitions de domaines.

1.1.3.3 VALIDATION À L'EXÉCUTION

La validation à l'exécution ou *runtime assertion checking* [Clarke et al., 2006] est l'évaluation de la spécification formelle d'un programme lors de son exécution. Cette méthode opère généralement sur une version modifiée (on dira *instrumentée*) du programme. Ainsi, si une annotation du programme initial est invalide, l'exécution du programme instrumenté provoque une erreur, sinon l'exécution est identique à celle du programme initial.

Contrairement au test structurel ou au test fonctionnel, la validation à l'exécution opère sur un programme dont les entrées sont données.

JMLC [Leavens et al., 1999], JAHOB [Zee et al., 2007], SLICK [Nguyen et al., 2008], APP [Rosenblum, 1992], EIFFEL [Meyer, 1988], SPARK [Dross et al., 2014] et le greffon E-ACSL2C [Kosmatov et al., 2013] de FRAMA-C sont des outils de validation des assertions à l'exécution.

1.1.4 LANGAGES DE SPÉCIFICATION

Les propriétés d'un programme à vérifier sont le plus souvent exprimées dans un langage de spécification formel [Hatcliff et al., 2012]. Le langage EIFFEL [Meyer, 1988] embarque son propre langage de spécification. SPARK [Dross et al., 2014] est un langage de spécification pour le langage Ada. JML (Java Modeling Language) [Leavens et al., 1999] est un langage de spécification pour le langage Java et ACSL (ANSI/ISO C Specification Language) [Baudin et al., 2015] est un langage de spécification pour le langage C fourni par la plateforme FRAMA-C [Kirchner et al., 2015].

La plupart de ces langages conviennent aussi bien aux analyses statiques comme la preuve qu'aux analyses dynamiques comme le *runtime checking* [Leavens et al., 2005, Cheon et al., 2005, Delahaye et al., 2013, Ahrendt et al., 2015].

Nous présentons plus en détail le langage ACSL et son intégration dans FRAMA-C en partie 1.1.5.

1.1.5 LA PLATE-FORME FRAMA-C

FRAMA-C [Kirchner et al., 2015] est une plate-forme dédiée à l'analyse des programmes C, conjointement dévelopée par le CEA LIST et Inria. Son architecture comporte un noyau et un écosystème de greffons, rendant l'outil extensible. Les greffons peuvent échanger des informations et utiliser les services fournis par le noyau, permettant ainsi une collaboration entre différentes analyses.

FRAMA-C est basé sur CIL [Necula et al., 2002], une bibliothèque qui normalise des programmes C (ISO C99) en opérant des modifications syntaxiques : normalisation des

boucles en utilisant la structure `while`, unique `return` pour chaque fonction, etc. FRAMA-C étend CIL pour supporter des annotations dédiées portant sur le code source, exprimées dans le langage ACSL. ACSL [Baudin et al., 2015] est un langage formel de spécification comportementale [Hatcliff et al., 2012], inspiré de JML [Leavens et al., 1999], pouvant exprimer des propriétés fonctionnelles de programmes C : préconditions, postconditions, invariants, etc. Les annotations du langage ACSL sont écrites en utilisant la logique du premier ordre, et il est possible de définir ses propres fonctions et prédicats.

La spécification d'une fonction comprend les préconditions (exprimées par une clause `requires`) requises lors de l'appel et les postconditions (clause `ensures`) assurées lors du retour. Parmi ces postconditions, une clause `assigns` indique quels sont les emplacements mémoire qui doivent avoir la même valeur à la fin de la fonction.

```
1 /*@ requires \valid(a) && \valid(b);
2     requires \separated(a,b);
3     assigns *a, *b;
4     ensures *a == \at(*b,Pre);
5     ensures *b == \at(*a,Pre); */
6 void swap(int* a, int* b);
```

Listing 1.1 – Exemple de spécification ACSL

Considérons par exemple une spécification fournie pour une fonction `swap` (listing 1.1). La première précondition établit que les deux arguments doivent être des pointeurs valides, autrement dit, le déréférencement de `a` ou de `b` pour la lecture ou l'écriture des valeurs pointées ne produira pas d'erreur à l'exécution. La seconde précondition (ligne 2) impose que les emplacements mémoire occupés par chacune de ces variables soient disjoints et donc que leurs plages d'adresses soient différentes. La clause `assigns` de la ligne 3 indique que seules les valeurs pointées par les pointeurs `a` et `b` peuvent être modifiées par la fonction. En plus de `\valid` et `\separated`, ACSL fournit de nombreux prédicats et fonctions afin de décrire les états de la mémoire. L'expression `\at(e,l)` fait référence à la valeur de l'expression `e` dans l'état de la mémoire au label `l`. `Pre` est un label prédéfini qui fait référence à l'état de la mémoire avant l'exécution de la fonction. Ainsi, les postconditions (lignes 4–5) du listing 1.1 signifient qu'à la fin de la fonction, `*a` aura la valeur que `*b` avait au début de la fonction, et réciproquement.

Le langage ACSL offre aussi la possibilité d'écrire des annotations dans le code source, permettant d'exprimer des propriétés devant être vraies à un point donné du programme : les assertions (clause `assert`). Il est également possible d'exprimer des propriétés devant être vraies avant une boucle et après chaque itération de cette boucle : les invariants de boucle (clause `loop invariant`).

Les greffons peuvent valider ou invalider les propriétés ACSL et générer des annotations ACSL. Les annotations sont donc un moyen d'échanger des informations entre les différentes analyses opérées par les greffons. Par exemple, quand l'analyse de valeurs détecte une menace de division par zéro, elle peut ajouter l'annotation `/*@assert x != 0; */` avant l'opération de division par `x`, laissant à d'autres greffons la charge de valider ou invalider cette assertion.

Parmi les greffons de FRAMA-C, nous pouvons citer VALUE (analyse de valeurs par interprétation abstraite), WP (vérification déductive), PATHCRAWLER (génération de tests structurels), SLICING (simplification syntaxique), E-ACSL2C (validation d'assertions à l'exécution) et RTE (détection d'erreurs à l'exécution potentielles).

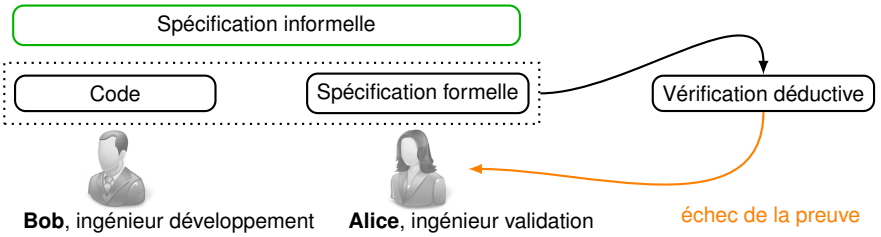

FIGURE 1.1 – Processus de vérification par la preuve

1.2 PROBLÉMATIQUE ET MOTIVATIONS

La figure 1.1 présente un schéma classique de vérification des logiciels. Le code du logiciel est écrit par un ingénieur développement en se basant sur une spécification informelle dont la correction n'est pas remise en cause. La vérification de logiciels repose le plus souvent sur une spécification formelle encodant les propriétés du programme à vérifier. La spécification formelle est écrite par un ingénieur validation en se basant sur la spécification informelle. La tâche de spécification et de vérification déductive (définie en 1.1.2.3) des programmes effectuée par l'ingénieur validation est longue et difficile et nécessite une connaissance des outils de preuve de programmes. En effet, un échec de preuve de programme peut être dû :

– à une non-conformité du code par rapport à sa spécification ;
– à un contrat de boucle ou de fonction appelée trop faible pour prouver une autre propriété ;
– ou à une incapacité du prouveur.

Il est souvent difficile pour l'utilisateur de décider laquelle de ces trois raisons est la cause de l'échec de la preuve, car cette information n'est pas (en général) donnée par le prouveur et requiert donc une revue approfondie du code et de la spécification.

L'objectif de cette thèse est de fournir une méthode de diagnostic automatique des échecs de preuve afin de décider si la cause d'un échec de preuve est une non-conformité entre le code et la spécification, ou la faiblesse d'un ou plusieurs sous-contrats de la spécification, ou enfin une incapacité de l'outil de vérification déductive. Nos travaux ont donc pour but d'améliorer et de faciliter le processus de spécification et de preuve des programmes.

1.3 CONTRIBUTIONS

Nous proposons une méthode originale d'aide à la vérification déductive, par génération de tests structurels sur une version instrumentée du programme d'origine. Cette méthode consiste en deux étapes.

Premièrement, nous essayons de détecter les non-conformités du code par rapport à la spécification. Cette phase nécessite une traduction en C des annotations ACSL afin d'expliciter les violations potentielles d'annotations par la création de nouvelles branches dans le programme. La préservation de la sémantique par cette traduction nous assure de trouver les erreurs de conformité entre le code et la spécification par génération de tests structurels à condition de couvrir tous les chemins d'exécution faisables du programme.

L'aspect méthodologique de cette première étape est détaillé dans [Petiot et al., 2014b] et la traduction des annotations est présentée dans [Petiot et al., 2014a].

Deuxièmement, si le programme ne contient pas de non-conformité, nous essayons de détecter les faiblesses de sous-contrats (contrats de boucle ou de fonction appelée). Cette seconde phase nécessite une traduction en C des annotations ACSL différente de la première, car nous voulons pouvoir "exécuter" certains contrats au lieu du code qu'ils spécifient. Si aucune de ces deux phases n'a fourni de raison quant à l'échec de la preuve et si la génération de tests a couvert tous les chemins d'exécution faisables du programme, alors l'échec de la preuve est dû à une incapacité du prouveur, sinon le problème reste non résolu. La combinaison de ces deux phases de détection est présentée dans [Petiot et al., 2015].

Cette méthode de diagnostic des échecs de preuve par génération de tests structurels est notre principale contribution. Elle inclut la traduction en C des annotations ACSL pour la détection de non-conformités, la traduction en C des annotations ACSL pour la détection de faiblesses de sous-contrats, la justification de correction de ces traductions, l'implémentation de notre méthode de diagnostic des échecs de preuve par le test sous la forme d'un greffon FRAMA-C appelé STADY et des expérimentations évaluant l'impact de la méthode sur le diagnostic des échecs de preuve. L'outil STADY nous a notamment permis de faciliter la vérification déductive de certains programmes [Genestier et al., 2015].

Une autre contribution est la validation à l'exécution des annotations ACSL liées au modèle mémoire, ainsi que l'implémentation d'une bibliothèque permettant de valider ces annotations à l'exécution. Cette contribution est détaillée dans [Kosmatov et al., 2013].

1.4 PLAN DE LA THÈSE

Nous avons présenté le contexte de nos travaux ainsi que la problématique, nos motivations et nos contributions. Nous annonçons maintenant le plan de la thèse.

Un état de l'art détaillé du domaine, centré sur les combinaisons d'analyses statiques et d'analyses dynamiques ainsi que sur l'aide à la preuve est donné dans le chapitre 2. Le chapitre 3 définit la syntaxe et la sémantique des langages de programmation et de spécification considérés dans nos travaux : respectivement un sous-ensemble du langage C et un sous-ensemble du langage ACSL. Ce chapitre définit également la sémantique de ces langages.

Les chapitres suivants détaillent nos contributions. Le chapitre 4 présente une traduction des annotations ACSL en C appropriée pour une détection (par génération de tests) des non-conformités entre le code et la spécification. Les contributions de ce chapitre ont été publiées dans l'article [Petiot et al., 2014a]. Le chapitre 5 aborde la validation d'annotations ACSL liées au modèle mémoire. Ce sont des annotations particulières que nous validons à l'exécution et non par génération automatique de tests structurels. Les contributions de ce chapitre ont été publiées dans l'article [Kosmatov et al., 2013].

Nous présentons notre méthode de détection des non-conformités entre le code et la spécification au chapitre 6, notre méthode de détection des faiblesses de contrats au chapitre 7. Puis le chapitre 8 présente une méthode plus générale combinant ces deux méthodes de détection afin de diagnostiquer les échecs de preuve de programmes. Les contributions du chapitre 6 ont été publiées dans l'article [Petiot et al., 2014b].

Les contributions des chapitres 7 et 8 sont présentées dans le rapport de recherche [Petiot et al., 2015].

Le chapitre 9 présente notre implémentation d'une bibliothèque C permettant de valider à l'exécution les annotations ACSL liées au modèle mémoire, ainsi que les résultats de nos expérimentations. L'implémentation de notre méthode de diagnostic des échecs de preuve ainsi que les résultats de nos expérimentations sont présentés au chapitre 10.

Enfin, le chapitre 11 présente le bilan de nos travaux et les différentes perspectives envisagées.

2

COMBINAISONS D'ANALYSES STATIQUES ET DYNAMIQUES POUR L'AIDE À LA PREUVE

Dans ce chapitre nous passons en revue les travaux existants combinant analyses statiques et analyses dynamiques, et en particulier ceux qui ont pour objectif d'aider la vérification déductive. Nous présentons leurs forces et leurs faiblesses puis concluons en mettant en avant la contribution que nous souhaitons apporter à ces domaines. La partie 2.1 présente les travaux combinant analyses statiques et dynamiques et la partie 2.2 recense les travaux visant à aider la vérification déductive de programmes.

2.1 COMBINAISONS D'ANALYSES STATIQUES ET DYNAMIQUES

Les méthodes statiques et les méthodes dynamiques ont des avantages et des inconvénients complémentaires : l'analyse statique est complète mais imprécise, l'analyse dynamique est précise mais incomplète. L'idée de les combiner pour associer leurs avantages et combattre leurs inconvénients [Ernst, 2003] est une voie de recherche active et fructueuse dans le domaine de la vérification de programmes.

VÉRIFICATION FORMELLE ET TEST AVEC SPARK

L'intégration de la vérification formelle dans un processus de validation à base de tests (comme c'est le cas dans l'avionique) est présenté dans [Comar et al., 2012] dans le cadre de la plate-forme SPARK. SPARK [Dross et al., 2014] est un sous-ensemble du langage Ada orienté pour la conception d'applications critiques sûres et sécurisées. C'est aussi un écosystème d'outils contenant entre autres l'outil de preuve SPARK-PROVE et l'outil de test SPARKTEST. [Comar et al., 2012] présente la vérification des contrats SPARK implicites et des contrats définis par l'utilisateur, ainsi que la définition de cas de tests formels dans les contrats SPARK. La vérification des hypothèses utilisées par la preuve ou le test d'une partie d'un programme SPARK est présentée dans [Kanig et al., 2014]. La méthodologie utilisée permet uniquement d'utiliser le test ou la preuve sur une fonction, contrairement à FRAMA-C où il est possible d'utiliser différentes techniques de vérification (interprétation abstraite, calcul de plus faible précondition, etc.) sur différentes propriétés d'une même fonction.

DÉTECTION DES ERREURS À L'EXÉCUTION DANS SANTE

La méthode SANTE (STATIC ANALYSIS AND TEST) [Chebaro, 2011, Chebaro et al., 2012b] pour la détection des erreurs à l'exécution, mise en œuvre au sein de FRAMA-C, combine l'interprétation abstraite, le *slicing* et la génération de tests structurels avec PATHCRAWLER.

L'analyse de valeurs (par interprétation abstraite) sélectionne les instructions pour lesquelles le risque d'une erreur à l'exécution n'est pas écarté (nous les appellerons "alarmes" par la suite), par exemple une division par zéro ou un accès invalide à la mémoire. Ces alarmes vont être confirmées ou non par des tests. L'outil de génération de tests structurels PATHCRAWLER [Botella et al., 2009] est utilisé pour tenter de mettre en évidence un cas de test pour lequel le chemin d'exécution passe par cette alarme et provoque une erreur.

Pour diminuer le coût de la génération de tests, SANTE simplifie syntaxiquement les programmes qui lui seront soumis pour ne garder que les instructions dont dépend l'alarme considérée, on appelle cette opération le *slicing*. Des programmes plus simples sont ainsi obtenus, tels que si on constate une erreur dans un programme simplifié, alors cette erreur existe dans le programme d'origine.

La première phase de la méthode (l'interprétation abstraite) peut donc assurer que certaines instructions ne provoqueront pas d'erreur à l'exécution, tandis que la seconde phase permet de confirmer que certaines alarmes produisent effectivement une erreur à l'exécution. La mise en commun des résultats des deux phases permet de classifier plus d'alarmes que chacune des deux méthodes prise séparément.

DÉTECTION D'ERREURS GUIDÉE PAR ANALYSE STATIQUE AVEC DYTA

DYTA [Ge et al., 2011] est un outil combinant une phase d'analyse statique et une phase d'analyse dynamique. CODECONTRACTS [Logozzo, 2011] est utilisé pour spécifier des pré/post-conditions et des invariants de programmes C#, il identifie également les bugs potentiels (violations de contrat) par interprétation abstraite. La deuxième étape utilise la génération de tests par exécution symbolique dynamique avec PEX [Tillmann et al., 2008].

DYTA instrumente le programme pour ajouter des instructions assurant le rôle de préconditions, pour ne pas générer de cas de test qui n'ont aucune chance de produire d'erreur à l'exécution. L'instrumentation ajoute également des points de contrôle à l'endroit des instructions signalées par l'analyse statique, afin de guider l'exécution symbolique dynamique du programme par PEX. Cette étape est semblable à l'instrumentation opérée par SANTE pour rajouter des points de contrôle afin de guider l'exécution concolique du programme par PATHCRAWLER.

L'analyse statique du graphe du flot de contrôle du programme permet à DYTA de calculer les points de contrôle à partir desquels les instructions potentiellement dangereuses sont inatteignables. La génération de tests réduit le nombre de faux positifs de l'analyse statique, et cette dernière guide l'exploration pour la génération de tests.

VÉRIFICATION COLLABORATIVE ET TEST

La méthode de [Christakis et al., 2012] part du constat que la plupart des outils de vérification statique (outils utilisant simplement des heuristiques, outils d'interprétation abstraite, model-checkers, ou outils de preuve) font des compromis afin d'améliorer les performances, de réduire le nombre de faux positifs ou de limiter l'effort à fournir pour annoter le programme. Cela se traduit par la supposition d'une propriété tout au long de l'analyse du programme (par exemple qu'un certain type d'erreurs ne peuvent pas se produire). Cette propriété n'est pas vérifiée par l'analyseur. Ceci implique que de tels analyseurs ne peuvent garantir l'absence d'erreurs dans un programme. Si un analyseur fait un compromis en supposant une propriété, il faut utiliser un autre analyseur capable de valider cette propriété.

Cette méthode propose d'exprimer les compromis dans un langage de contrats, afin de faciliter la collaboration entre plusieurs outils d'analyse statique et permettant de décrire le fait qu'une assertion a été complètement vérifiée par un analyseur ou partiellement vérifiée sous certaines hypothèses.

Pour faciliter la vérification statique du programme, l'utilisateur doit au préalable ajouter des annotations (invariants de boucle par exemple). Les propriétés qui n'ont pas été vérifiées statiquement sont validées par des tests unitaires. Le programme est instrumenté afin d'ajouter des vérifications à l'exécution pour guider la génération de contre-exemples. PEX [Tillmann et al., 2008] est utilisé pour générer des cas de test par exécution concolique, mettant en évidence des contre-exemples. L'utilisateur peut décider de privilégier l'analyse statique ou le test selon qu'il spécifie ou non son programme.

VÉRIFICATION DE PROPRIÉTÉS DÉCRITES PAR DES AUTOMATES FINIS

La méthode de [Slabý et al., 2012] combine une instrumentation du code source, une étape de *slicing* et une exécution symbolique. L'instrumentation ajoute des instructions simulant le comportement de l'automate fini correspondant au programme, dont les états correspondent aux propriétés du programme à vérifier. Le *slicing* est appliqué pour réduire la taille du programme afin de ne conserver que le code relatif aux états d'erreur de l'automate. Le programme simplifié doit être équivalent au programme instrumenté en ce qui concerne l'atteignabilité des états d'erreur de l'automate. L'exécution symbolique du programme par KLEE [Cadar et al., 2008a] permet de mettre en évidence des contre-exemples pour ces propriétés.

LOCALISATION D'ERREURS PAR SLICING GUIDÉ PAR UNE TRACE D'EXÉCUTION

La méthode de [Jiang et al., 2012] utilise un *slicing* arrière à partir d'une instruction de déréférencement produisant une *Null Pointer Exception* (en Java). Le *slicing* est guidé par une trace du programme fournissant la pile des appels de méthodes n'ayant pas terminé. Une analyse statique des pointeurs est ensuite opérée sur le programme slicé afin de déterminer si chacun des pointeurs peut être `null`. Puis une analyse d'alias est opérée afin d'augmenter la précision de l'analyse statique.

CEGAR : RAFFINEMENT D'ABSTRACTION GUIDÉ PAR DES CONTRE-EXEMPLES

Le raffinement d'abstraction guidé par des contre-exemples [Clarke et al., 2003, Pasareanu et al., 2007] associe l'abstraction par prédicats [Ball, 2005] et le model-checking : une abstraction du programme est générée à partir d'un ensemble de prédicats et invariants. Si le model-checking prouve la non-accessibilité des états d'erreurs de l'automate, alors le modèle concret est correct. Si le model-checking a trouvé un contre-exemple pour le modèle abstrait il faut déterminer s'il correspond à un contre-exemple réel dans le système concret. Pour cela, l'algorithme détermine s'il existe une trace d'exécution concrète correspondant à la trace d'exécution abstraite aboutissant au contre-exemple. Si une telle trace existe, un bug a été trouvé. Sinon, de nouveaux prédicats sont créés pour raffiner l'abstraction afin que ce contre-exemple en soit absent à la prochaine itération. Et ainsi de suite. Ce processus peut ne pas terminer. Plusieurs outils de vérification se basent sur cette méthode, parmi lesquels BLAST [Beyer et al., 2007], MAGIC [Chaki et al., 2003] et SLAM [Ball et al., 2011].

Cette approche élimine les contre-exemples un par un, et peut donc ne pas converger. Une amélioration a été proposée afin d'accélérer la convergence du raffinement de l'abstraction : CEGAAR [Hojjat et al., 2012]. Cette technique élimine une infinité de contre-exemples (traces infaisables) de la forme $\alpha.\lambda^*.\beta$ en une seule étape, où λ correspond à un nombre quelconque d'exécutions d'une boucle.

VÉRIFICATION DES PROPRIÉTÉS TEMPORELLES DE SÛRETÉ AVEC BLAST

BLAST [Beyer et al., 2007] vérifie les propriétés temporelles de sûreté d'un programme C, ou met en évidence un chemin d'exécution violant une propriété. Le raffinement des abstractions du programme est basé sur une abstraction par prédicats et la découverte des prédicats se fait par interpolation. C'est une implémentation de la méthode CEGAR [Clarke et al., 2003] tout comme SLAM [Ball et al., 2011] et MAGIC [Chaki et al., 2003]. La génération des cas de test se fait par exécution symbolique.

BLAST ne traite ni les débordements arithmétiques ni les opérations bit-à-bit, et considère que toutes les opérations arithmétiques sur les pointeurs sont sûres. Le langage d'invariants utilisé pour décrire les propriétés ne contient pas de quantificateurs.

GÉNÉRATION DE TESTS AVEC RÉSUMÉS : SMART

SMART (*Systematic Modular Automated Random Testing*) [Godefroid, 2007], basé sur son prédécesseur DART (*Directed Automated Random Testing*) [Godefroid et al., 2005] génère des tests par exécution concolique. Pour résoudre le problème de l'explosion du nombre de chemins, il va calculer à la demande des résumés de fonction qui sont des contrats (pré-conditions et post-conditions) pour chaque fonction (contraintes sur les variables en entrée et en sortie). Ces résumés vont être réutilisés si possible afin d'éviter de ré-exécuter la fonction correspondante. La génération automatique de ces résumés se base sur une exécution symbolique et une résolution de contraintes. SMART exécute une analyse dynamique compositionnelle, c'est-à-dire que les résultats intermédiaires sont mémorisés sous la forme de résumés réutilisables. C'est une extension de l'algorithme d'analyse dynamique non compositionnelle de DART.

COLLABORATION DE SUR-APPROXIMATION ET SOUS-APPROXIMATION

SYNERGY [Gulavani et al., 2006] est un algorithme combinant du test (essayer d'atteindre un état d'erreur) et une abstraction (trouver une abstraction suffisamment précise montrant qu'aucun chemin ne peut atteindre un état d'erreur). La sous-approximation du test et la sur-approximation de l'abstraction sont raffinées de manière itérative. L'abstraction est utilisée pour guider la génération de tests. Les tests sont utilisés pour décider *où* raffiner l'abstraction. Les états de l'abstraction, les régions, sont des classes d'équivalence des états du programme concret. S'il n'existe aucun chemin de la région initiale vers une région d'erreur, alors il n'existe aucune suite de transitions concrètes menant d'un état initial concret à un état d'erreur concret. SYNERGY combine SLAM et DART (CEGAR compositionnel et test non compositionnel). L'algorithme est intra-procédural.

DASH [Beckman et al., 2008] est une évolution de SYNERGY, prenant en compte les appels de procédure et les pointeurs (contrairement à SYNERGY). DASH est inter-procédural mais non compositionnel. DASH raffine l'abstraction en utilisant uniquement les relations d'alias mises en évidence par les tests. La génération de tests détermine non seulement *où* raffiner l'abstraction, mais aussi *comment* la raffiner. Cet algorithme est implémenté dans YOGI [Nori et al., 2009].

SMASH est la version compositionnelle de DASH [Beckman et al., 2008]. SMASH [Godefroid et al., 2010] combine une abstraction par prédicats et une génération dynamique de tests (par exécution concolique). Pour chaque fonction, un résumé est calculé par analyse statique (vrai pour toutes les exécutions, permettant de prouver l'absence d'erreurs, c'est une sur-approximation) et un autre résumé est calculé par analyse dynamique (vrai pour quelques exécutions uniquement, permettant de montrer l'existence d'erreurs, c'est une sous-approximation). Ces résumés sont calculés à la demande et seront utilisés aussi bien par l'analyse statique que par l'analyse dynamique (les deux analyses s'exécutent simultanément). Ces résumés sont progressivement raffinés pour chaque fonction, afin de prouver qu'une propriété n'est jamais violée (si un résumé statique est applicable), ou de mettre en évidence une exécution violant une propriété (si un résumé dynamique est applicable). Par construction, il n'est pas possible que les deux résumés soient applicables.

ANALYSES STATIQUE ET DYNAMIQUE POUR LA GÉNÉRATION D'INVARIANTS

[Gupta et al., 2009] propose une solution au problème du passage à l'échelle de la génération d'invariants (de programmes impératifs) par résolution de contraintes. Les invariants arithmétiques linéaires sont générés d'après les informations obtenues par interprétation abstraite du programme, par exécution concrète et par exécution symbolique du programme. Ces informations permettent de générer des contraintes qui vont permettre au solveur de contraintes de simplifier le système de contraintes et de réduire l'espace de recherche.

RECHERCHE D'ERREURS PAR DÉTECTION D'INVARIANTS AVEC DSD CRASHER ET CHECK 'N' CRASH

L'outil DSD CRASHER [Csallner et al., 2008] combine une première analyse dynamique, une analyse statique et une seconde analyse dynamique. La première analyse dyna-

mique utilise une génération de tests et des techniques d'apprentissage pour générer des invariants probables (inférence de la spécification par DAIKON [Ernst et al., 2001]).

Les deux dernières étapes sont celles de l'outil CHECK 'N' CRASH [Csallner et al., 2005]. L'analyse statique (ESC-JAVA2 [Cok et al., 2005]) va émettre des alarmes concernant le non-respect des invariants, et la seconde analyse dynamique va tenter de confirmer ces alarmes par résolution de contraintes et génération de tests (JCRASHER [Csallner et al., 2004]). La classification des alarmes et la génération de tests sont très dépendantes de la qualité des invariants générés par DAIKON.

GÉNÉRATION DE DONNÉES DE TEST PAR ALGORITHME GÉNÉTIQUE

[Romano et al., 2011] propose une méthode de génération de données de test mettant en évidence des *Null Pointer Exceptions* de Java. Tout d'abord une analyse inter-procédurale du flot de contrôle et du flot de données collecte les chemins menant aux exceptions. Cette analyse se fait en arrière, en partant des exceptions, propageant les contraintes sur les entrées dans le graphe de flot de contrôle. Les entrées de test sont ensuite générées par un algorithme génétique, dans le but de couvrir ces chemins. Les individus (des entrées potentielles), dont le type de donnée peut être complexe, sont encodés sous forme XML.

Cette méthode a été comparée avec d'autres façons de générer des données de test [Ahn et al., 2010]. Les expérimentations présentées dans [Romano et al., 2011] ont montré que la méthode était plus efficace que d'autres stratégies de recherche optimale comme le *hill climbing* et le recuit simulé, mais est moins efficace que la programmation par contraintes (en terme de temps d'exécution).

PREUVE ET VALIDATION À L'EXÉCUTION

La plateforme STARVOORS (*Static and Runtime Verification of Object-Oriented Software*) [Ahrendt et al., 2015] propose une vérification en deux étapes : vérification déductive du programme avec KEY afin de prouver les propriétés portant sur les données, puis une validation à l'exécution avec LARVA (*Logical Automata for Runtime Verification and Analysis*) afin de vérifier les propriétés portant sur le flot de contrôle. Cette méthode repose sur PPDATE, un langage de spécification encodant les propriétés du flot de contrôle par des automates, où les états de l'automate sont augmentés par des pré- et postconditions (à la JML) afin d'encoder les propriétés fonctionnelles du programme.

2.2 AIDE À LA PREUVE DE PROGRAMMES

ANALYSE DES ARBRES DE PREUVE

Un diagnostic plus précis en cas d'échec de preuve peut être obtenu en analysant statiquement les branches d'un arbre de preuve. La méthode proposée par [Gladisch, 2009] dans le contexte du prouveur KEY applique des règles de déduction à une formule mêlant le programme et sa spécification. Cette méthode propose une vérification de la préservation de falsifiabilité (*falsifiability preservation checking*) qui permet de décider si l'échec

de preuve d'une branche provient d'une erreur ou d'une faiblesse de contrat. Cependant, cette technique ne peut détecter les erreurs que si les contrats sont suffisamment forts et elle n'est automatique que si le prouveur peut décider la (non-)satisfaisabilité d'une formule logique de premier ordre exprimant la condition de la préservation de falsifiabilité.

[Engel et al., 2007] présente une méthode de génération automatique de tests unitaires pour JAVA CARD à partir d'une tentative de preuve par le prouveur KEY. Ici aussi, les arbres de preuve générés par KEY sont exploités. L'information contenue dans la preuve (même partielle) est utilisée pour extraire des données de test à partir des conditions de chemins et les oracles sont générés à partir des postconditions. En revanche, la pertinence des tests générés dépend de la qualité de la spécification écrite par l'utilisateur, et ne permet pas de distinguer les échecs de preuve dus à une non-conformité des échecs de preuve dus à une faiblesse de spécification.

COMPRÉHENSION DES ÉCHECS

[Tschannen et al., 2014] propose une vérification en deux étapes qui compare les échecs de preuve d'un programme EIFFEL avec et sans déroulage des boucles et *inlining* des fonctions. Cette méthode suggère des modifications à apporter au code et à la spécification après comparaison des échecs de preuve. L'*inlining* des fonctions et le déroulage des boucles sont limités respectivement à un nombre donné d'appels imbriqués et à un nombre donné d'itérations. Si ce nombre est insuffisant, la sémantique du programme initial est perdue et la fiabilité des résultats en est affectée.

[Claessen et al., 2008] propose une méthode de génération de contre-exemples montrant que les invariants inductifs d'un système de transitions sont trop forts ou trop faibles. Leur méthode utilise le test aléatoire (avec QUICKCHECK) et cible la vérification des systèmes de transitions et des protocoles, ce qui la rend difficilement applicable à la vérification de programmes.

VÉRIFICATION DES HYPOTHÈSES

Les axiomes sont des propriétés logiques utilisées comme hypothèses par les prouveurs et sont donc rarement vérifiés. Néanmoins, les vérifier permet de s'assurer de la cohérence d'une axiomatisation. [Ahn et al., 2010] propose d'appliquer du test à base de modèles (*model-based testing*) sur un modèle défini à partir d'un axiome afin de détecter les erreurs éventuelles dans les axiomes. Ce travail s'intéresse au cas où la vérification déductive réussit à cause d'une axiomatisation invalide (incohérence), ce qui est complémentaire à nos motivations qui visent à traiter le cas où la vérification déductive échoue.

[Christakis et al., 2012] propose de compléter les résultats des analyses statiques avec une exécution symbolique dynamique utilisant PEX. Les hypothèses explicites utilisées par la preuve (absence de débordement arithmétique, absence d'*aliasing*, etc.) créent de nouvelles branches dans le graphe de flot de contrôle du programme, que PEX essaiera ensuite d'explorer. Cette approche permet de détecter des erreurs hors de portée des analyseurs statiques considérés, mais ne génère pas de contre-exemples en cas de spécification trop faible.

Exécution des contre-exemples

[Müller et al., 2011] propose d'étendre un vérificateur d'assertions à l'exécution (*runtime assertion checker*) afin de l'utiliser comme débogueur (*debugger*) pour comprendre les contre-exemples d'échecs de preuve complexes.

L'environnement de développement Dafny [Leino et al., 2014] fournit un retour à l'utilisateur pendant la phase de développement. Dafny intègre le Boogie *Verification Debugger* [Le Goues et al., 2011] qui aide l'utilisateur à comprendre les contre-exemples produits par des outils de vérification tels que Boogie. Actuellement, Dafny ne supporte que les contre-exemples fournis par le solveur et ne produit pas d'informations supplémentaires quand la vérification échoue ou atteint un *timeout*.

Conclusion du chapitre

Nous avons présenté dans ce chapitre les différents travaux de vérification existants qui combinent analyses statiques et analyses dynamiques ou qui visent à aider la vérification déductive. Parmi eux, nous poursuivons des travaux précédents [Chebaro et al., 2012b] afin d'aider la vérification déductive par la génération de contre-exemples par exécution concolique. Nous présentons dans les chapitres suivants une technique originale de diagnostic des échecs de preuve. En particulier, la thèse que nous défendons est que les contre-exemples générés par notre méthode par analyse dynamique permettent de mieux identifier la raison des échecs de preuve qu'un contre-exemple extrait à partir d'un prouveur. Nous affirmons par la même occasion l'indépendance de notre méthode vis-à-vis des prouveurs sous-jacents utilisés. À notre connaissance, une telle méthode combinant le test et la preuve, fournissant automatiquement un retour aussi précis à l'utilisateur quant à la raison des échecs de preuve, n'a pas été étudiée, implémentée et expérimentée auparavant.

3

PROGRAMMES C ANNOTÉS AVEC E-ACSL

Ce chapitre présente le sous-ensemble du langage C et du langage de spécification E-ACSL que nous traitons. Dans la partie 3.1 nous présentons la syntaxe du langage illustrée par un exemple. C'est la syntaxe normalisée de programmes C annotés par des clauses E-ACSL. Dans la partie 3.2 nous définissons la notion de mémoire ainsi que les fonctions permettant de décrire la sémantique dénotationnelle de chaque élément du langage.

3.1 SYNTAXE DES PROGRAMMES ANNOTÉS

Le langage de spécification E-ACSL [Signoles, 2015] est un sous-langage exécutable d'ACSL, qui est un langage de spécification comportementale implémenté dans FRAMA-C. Premièrement, étant un sous-ensemble d'ACSL, E-ACSL préserve la sémantique d'ACSL, ce qui implique que les greffons de FRAMA-C existants supportant ACSL peuvent être utilisés avec E-ACSL sans modification. Deuxièmement, le langage E-ACSL est *exécutable*, ce qui veut dire que toutes les annotations peuvent être traduites en C et leur traduction peut être exécutée pour l'analyse dynamique et le *monitoring* (surveillance d'exécution).

3.1.1 EXEMPLE DE FONCTION C NORMALISÉE

Nous présentons dans le listing 3.1 un exemple de fonction normalisée en suivant la grammaire de la figure 3.1. Le terme `\old` fait référence à la valeur du terme au début de la fonction courante. `\result` fait référence à la valeur retournée par la fonction. La clause `ensures` aux lignes 4–5 indique qu'elle retourne une valeur différente de zéro quand la valeur `v` en paramètre est présente dans le tableau `t` de longueur `n` en paramètre, et 0 sinon.

La clause `requires` de la ligne 1 indique que la taille `n` du tableau est positive ou nulle. La clause `requires` de la ligne 2 indique que `(t+0)`, ..., `(t+(n-1))` sont des pointeurs valides. La clause `typically` de la ligne 3 est une extension de la syntaxe d'E-ACSL définissant une précondition pour le test. Elle renforce la précondition pour limiter l'explosion combinatoire du nombre de chemins à explorer par la génération de tests. Elle permet d'obtenir une couverture partielle des chemins : seuls les chemins satisfaisant cette clause

```
1  /*@ requires 0 <= n
2               && \valid(t+(0..n-1));
3      typically n <= 6;
4      ensures \result != 0 <==> \exists integer i; 0<=i<=\old(n)-1
5                                  && \old(t[i])==\old(v); */
6  int is_present(int* t, int n, int v) {
7    Beg_is_present : int res_is_present; int i; res_is_present = 0; i = 0;
8   l_0:
9    /*@ loop invariant 0 <= i && i <= n;
10       loop variant n - i; */
11   while (i < n && t[i] != v) {
12     BegIter_l_0 :
13     i = i+1;
14     EndIter_l_0 : ;
15   }
16   if(i < n) { res_is_present = 1; }
17  End_is_present : return res_is_present;
18 }
```

Listing 3.1 – Fonction C annotée décidant si `v` est présent dans le tableau `t` de taille `n`

sont considérés. Ici elle limite l'espace des valeurs de `n` (et donc la taille de `t`) à $[0, 6]$. Cette réduction de l'espace des états d'entrée peut être vue comme une "finitization" [Boyapati et al., 2002]).

L'invariant de boucle de la ligne 9 spécifie des bornes pour `i` sur l'ensemble des itérations de la boucle. Un variant de boucle permet de s'assurer de la terminaison d'une boucle : le terme du variant doit être positif ou nul avant chaque itération et doit décroître strictement entre chaque itération. Ainsi, le variant de boucle de la ligne 10 spécifie que la boucle pourra s'exécuter au plus `n-i` fois. Notons que pour prouver formellement la postcondition, l'invariant de boucle suivant est nécessaire, mais n'est pas inclus volontairement dans cet exemple simplifié : `\forall integer k; 0<=k<=i-1 ==> \old(t[k])!=v;`. Les clauses **`assigns`** et **`loop assigns`** ont volontairement été omises de cet exemple simplifié.

3.1.2 SYNTAXE DES INSTRUCTIONS ET FONCTIONS

La figure 3.1 définit la grammaire des instructions et fonctions C considérées ici. Les symboles terminaux sont soulignés. Les symboles non-terminaux sont en *italique*.

left-value est toute expression pouvant se trouver en partie gauche d'une affectation (variable simple ou déréférencée). Les accès aux pointeurs et aux tableaux sont notés `t[i]` (ou `*(t+i)`). Pour simplifier, *basic-type* se limite aux types C **`int`**, **`char`** et **`long`**, et *type* se limite aux types simples (non pointeur) et aux pointeurs sur types simples. *id* fait référence aux identificateurs de variable (et de fonctions) autorisés dans le langage C : suite de caractères alphanumériques pouvant contenir des "_" mais dont le premier caractère ne peut pas être numérique.

Les instructions de la forme *id* (*exp**); (respectivement *left-value* = *id* (*exp**);) correspondent respectivement aux appels de fonction sans (resp. avec) affectation de la valeur de retour. La notation e_s^* signifie que le motif e est répété 0, 1 ou plusieurs fois, et les occurrences du motif sont séparées par le symbole s. De manière similaire, la notation e_s^+ signifie que le motif e est répété une ou plusieurs fois. *label* respecte les mêmes contraintes que *id* mais fait référence aux labels d'un programme C. Pour des raisons de commodité, nous supposons qu'un label précède chaque annotation (assertion ou contrat de boucle) à l'intérieur d'une fonction. Nous considérons aussi qu'un label est présent au

basic-type	: :=	`int` | `char` | `long`
type	: :=	*basic-type* | *basic-type* `*`
left-value	: :=	*id* | *id* `[` *exp* `]`
instr	: :=	*type id* `;`
	|	*left-value* `=` *exp* `;`
	|	*id* `(` $exp^{*}_{,}$ `);`
	|	*left-value* `=` *id* `(` $exp^{*}_{,}$ `);`
	|	`if` `(` *exp* `)` `{` $instr^{*}$ `}` `else` `{` $instr^{*}$ `}`
	|	*label* `:` `/*@` `assert` *pred* `;` `*/`
	|	$label^{as\ l}$ `:` `/*@` `loop invariant` *pred* `;`
		`loop assigns` $term^{+}_{,}$ `;`
		`loop variant` *term* `;` `*/`
		`while` `(` *exp* `)` `{` $BegIter_{l}$ `:` `;` $instr^{*}$ $EndIter_{l}$ `:` `;` `}`
function	: :=	`/*@` `requires` *pred* `;`
		`typically` *pred* `;`
		`assigns` $term^{+}_{,}$ `;`
		`ensures` *pred* `;` `*/`
		$type^{as\ T}$ $id^{as\ f}$ `(` $(type\ id)^{*}_{,}$ `)` `{` Beg_{f} `:` T $\texttt{res}_{f}$`;` $instr^{*}$ End_{f} `:` `return` $\texttt{res}_{f}$`;` `}`

FIGURE 3.1 – Grammaire des instructions et fonctions

début et à la fin de chaque corps de boucle. Les assertions peuvent être associées à des instructions quelconques. Un invariant de boucle, un variant et une clause `loop assigns` peuvent être associés aux boucles. Nous considérons la sémantique des *assigns* définie dans [Baudin et al., 2015] selon laquelle les *left-values* hors de la clause `assigns` ou `loop assigns` ont la même valeur avant et après la fonction ou boucle correspondante. Certains greffons de FRAMA-C peuvent considérer une sémantique plus forte interdisant toute affectation d'une *left-value* hors de la clause même si sa valeur n'est pas modifiée par la fonction ou la boucle correspondante.

Nous supposons que les fonctions respectent la forme normale définie par l'entité syntaxique *function* de la grammaire définie dans la figure 3.1. Dans la figure 3.1, la notation "*as* X" en exposant d'un non-terminal signifie que chaque occurrence de X dans la règle courante doit être remplacée par la chaîne à laquelle fait référence ce non-terminal. Par exemple, si `foo` est le nom (*id*) d'une fonction, la règle *function* indique que les labels $\texttt{Beg}_{f}$ et $\texttt{End}_{f}$ dans le corps de cette fonction doivent être $\texttt{Beg}_{\texttt{foo}}$ et $\texttt{End}_{\texttt{foo}}$, ce qui assure des labels uniques au début et à la fin du corps de la fonction. Nous supposons qu'il existe une unique instruction `return` à la fin de chaque fonction, qu'elle retourne une variable appelée $\texttt{res}_{f}$ et que cette variable n'entre pas en conflit avec une autre variable du programme.

Un contrat de fonction à la EIFFEL [Meyer, 1988] peut être associé à chaque fonction pour spécifier les préconditions et postconditions. La clause `requires` d'un contrat de fonction désigne la précondition. La clause `typically` désigne la précondition pour le test. Elle est spécifique au test et permet de limiter l'explosion combinatoire du nombre de chemins. La clause `assigns` liste les termes que la fonction a le droit de modifier. La clause `ensures` désigne la postcondition de la fonction.

Les symboles non-terminaux *exp*, *term* et *pred* sont définis dans les parties 3.1.3 et 3.1.4.

3.1.3 SYNTAXE DES EXPRESSIONS C ET DES TERMES E-ACSL

unop : := `-` | `~`
binop : := `+` | `-` | `*` | `/` | `%`
exp : := *cst*
| *left-value*
| *unop exp*
| *exp binop exp*
term : := *cst*
| *left-value*
| *unop term*
| *term binop term*
| `\abs` `(` *term* `)`
| `\sum` `(` *term* `,` *term* `,` `\lambda integer` *id* `;` *term* `)`
| `\product` `(` *term* `,` *term* `,` `\lambda integer` *id* `;` *term* `)`
| `\numof` `(` *term* `,` *term* `,` `\lambda integer` *id* `;` *term* `)`
| *term* `?` *term* `:` *term*
| `\old` `(` *term* `)`
| `\null`
| `\result`

FIGURE 3.2 – Grammaire des expressions C et termes E-ACSL

La figure 3.2 définit la grammaire des termes E-ACSL et des expressions C du langage. Le symbole `~` est l'opérateur de négation bit-à-bit. Le non-terminal *cst* est une constante entière ou une chaîne de caractère littérale. Les constantes flottantes ne sont pas considérées. Les termes E-ACSL peuvent être vus comme une extension des expressions C. Ils peuvent être d'un type C (comme une expression C) ou d'un type logique (nous nous limitons à `integer` ($\mathbb{Z}$) dans nos travaux). Les termes sont "purs", c'est-à-dire qu'ils sont sans effet de bord. De même, nous nous limitons aux expressions C "pures". Pour cette raison, nous ne considérons pas les expressions comme `x++` ou `++x`. Pour cette même raison, les appels de fonction – pouvant produire des effets de bord – sont considérés non pas comme des expressions mais comme des instructions. L'hypothèse d'avoir des expressions pures n'est pas restrictive puisqu'une transformation de programme peut permettre de sortir les effets de bord des expressions. Dans FRAMA-C, cette transformation est faite par la bibliothèque CIL [Necula et al., 2002].

Le terme `\abs` fait référence à la valeur absolue d'un terme. Le terme `\sum(t1, t2, \lambda integer k; t3)` désigne la somme généralisée $\sum_{k=t1}^{t2} t3$. La notation `\product(t1, t2, \lambda integer k; t3)` désigne le produit généralisé $\prod_{k=t1}^{t2} t3$. Enfin, `\numof(t1, t2, \lambda integer k; t3)` fait référence au nombre de valeurs de `k` telles que $t1 \leq k \leq t2$ et $t3 \neq 0$. `\null` est le terme correspondant au pointeur `NULL`. L'opérateur ternaire `? :` signifie que le terme `x ? y : z` a la valeur de `y` si le terme `x` est non nul, ou la valeur de `z` si `x` est nul.

ENTIERS MATHÉMATIQUES

En plus des entiers machine, les termes E-ACSL peuvent être des entiers mathématiques de type `integer` dans l'ensemble $\mathbb{Z}$. Dans un terme E-ACSL, les constantes entières, les

opérations sur les entiers, ainsi que les variables logiques sont de type `integer`. L'arithmétique entière est non bornée et donc les débordements d'entier (*overflows*) sont impossibles. E-ACSL dispose d'un système de sous-typage pour convertir automatiquement les types entiers bornés du C vers les entiers mathématiques [Baudin et al., 2015]. Par exemple, si `x` est une variable C de type `int`, les termes E-ACSL `1` et `x+1` sont de type `integer` et une conversion implicite d'`int` vers `integer` est introduite dans ce contexte quand la variable `x` est typée.

Ce choix a été fait pour plusieurs raisons. Premièrement, un des buts principaux de FRAMA-C est la preuve de programmes par appel à des prouveurs externes. Or, la plupart de ces prouveurs fonctionnent mieux avec l'arithmétique entière qu'avec l'arithmétique bornée (ou modulaire). Deuxièmement, les spécifications sont habituellement écrites sans se préoccuper des détails d'implémentation, et les débordements d'entiers sont des détails d'implémentation. Troisièmement, il reste possible d'utiliser l'arithmétique bornée, si besoin est, en utilisant des conversions explicites (*casts*). Par exemple, `(int)(INT_MAX + 1)` est égal à `INT_MIN`, la plus petite valeur représentable de type `int`. Quatrièmement, ce choix facilite l'expression des débordements potentiels dans les spécifications : par exemple, grâce à l'arithmétique entière, `/*@assert INT_MIN <= x+y <= INT_MAX;*/` est le moyen le plus simple de spécifier que `x+y` ne déborde pas. Dans la suite, sauf mention contraire, "entier" fait référence à "entier non borné".

3.1.4 SYNTAXE DES PRÉDICATS E-ACSL

relation : := `>` | `>=` | `<` | `<=` | `==` | `!=`

pred : :=
- `\true`
- `\false`
- `\valid` `(` *id* `)`
- `\valid` `(` *id* `+` `(` *term* `..` *term* `)` `)`
- `\forall integer` *id* `;` *term* `<=` *id* `<=` *term* `==>` *pred*
- `\exists integer` *id* `;` *term* `<=` *id* `<=` *term* `&&` *pred*
- `!` *pred*
- *pred* `&&` *pred*
- *pred* `||` *pred*
- *pred* `==>` *pred*
- *pred* `<==>` *pred*
- *term* `?` *pred* `:` *pred*
- *term relation term*

FIGURE 3.3 – Grammaire des prédicats E-ACSL

La figure 3.3 définit la grammaire des prédicats du langage E-ACSL. Un prédicat a une valeur de vérité vraie ou fausse. Le prédicat `\valid` renvoie vrai si son argument est un pointeur valide : c'est-à-dire `\valid(ptr)` est vrai si et seulement si on peut déréférencer `ptr` et donc écrire `*ptr`. Dans le cas où son argument est de la forme `ptr+(x..y)`, le prédicat est valide si et seulement si `ptr+x`, `ptr+(x+1)`, ..., `ptr+y` sont valides. Dans E-ACSL, les quantifications existentielles (`\exists`) et universelles (`\forall`) doivent porter sur des domaines finis d'entiers.

Le symbole `!` est l'opérateur de négation logique, le symbole `&&` est l'opérateur de conjonction logique et le symbole `||` est l'opérateur de disjonction logique. Le symbole

`==>` encode l'implication logique et le symbole `<==>` encode l'équivalence logique de deux prédicats. L'opérateur ternaire `? :` signifie que le prédicat `x ? y : z` a la valeur de `y` si le terme `x` est non nul, ou la valeur de `z` si `x` est nul.

3.1.5 FONCTION SOUS VÉRIFICATION ET FONCTIONS APPELÉES

Dans ce document nous distinguons la "fonction sous vérification" et les "fonctions appelées". La fonction sous vérification est unique, et n'est appelée par aucune fonction : nous excluons toute récursivité (directe ou indirecte) dans le langage considéré. La précondition de la fonction sous vérification est supposée vraie.

Les fonctions appelées sont les fonctions atteignables à partir de la fonction sous vérification. Leurs préconditions doivent être vérifiées.

3.2 SÉMANTIQUE DES PROGRAMMES ANNOTÉS

Nous définissons dans les parties 3.2.1 et 3.2.2 les notions d'environnement et de *store*, qui nous permettent de définir la notion de mémoire en partie 3.2.3. Un environnement associe à chaque désignation (appelée *left-value* dans la grammaire de la figure 3.1) une adresse en mémoire. La notion de *store* associe aux adresses en mémoire des valeurs. Le tout constitue une mémoire. Cette notion de mémoire nous permet de définir la sémantique dénotationnelle des instructions et fonctions d'un programme annoté comme des modifications de mémoires. Dans la suite du document, nous utilisons les notations suivantes :

- lv, lv_1 et lv_2 sont des left-values ;
- x, x_1, ..., x_N sont des identificateurs de variable ;
- f est un identificateur de fonction ;
- v et v_i sont des valeurs ;
- cst est une constante ;
- G et X sont des listes de left-values E-ACSL ;
- i, i_1, i_2 sont des instructions C ;
- A, A_1, A_2 et B sont des séquences d'instructions C ;
- e, e_1, ..., e_N sont des expressions C ;
- t, t_1, t_2 et t_3 sont des termes E-ACSL ;
- h, h_1, h_2, h_3 sont des expressions C ou des termes E-ACSL ;
- p, p_1, p_2 sont des prédicats E-ACSL ;
- ρ, ρ_1, ρ_2 sont des environnements (voir définition en partie 3.2.1) ;
- σ, σ_1, σ_2 sont des stores (voir définition en partie 3.2.2) ;
- γ est une mémoire (voir définition en partie 3.2.3) ;
- δ, δ_1, ..., δ_N sont des adresses en mémoire (voir définition en partie 3.2.3) ;
- T est un type C ;
- $unop$ est un opérateur unaire ;
- $binop$ est un opérateur binaire ;
- rel est un opérateur binaire relationnel.

Pour des raisons de commodité, nous supposons que toutes les variables logiques liées dans les annotations et toutes les variables du programme sont différentes, ceci nous permet en particulier de traduire les variables logiques en C sans les renommer.

3.2.1 ENVIRONNEMENTS

Le domaine LV correspond à l'entité syntaxique *left-value*. LOC est le domaine des adresses (positions) en mémoire, incluant l'adresse nulle, notée $\texttt{NULL} \in LOC$. $\bot$ dénote une adresse indéfinie et donc invalide. Un environnement est une fonction $\rho \in LV \to LOC \cup \{\bot\}$ qui associe une adresse ou $\bot$ (adresse indéfinie) à chaque left-value définie dans le fragment de programme considéré. Nous notons $ENV = LV \to LOC \cup \{\bot\}$ le domaine des environnements. La notation $\rho[lv \mapsto \delta]$ désigne un nouvel environnement, correspondant à l'environnement ρ dans lequel l'adresse δ est associée à la left-value lv. L'environnement $\rho[lv \mapsto \delta]$ est défini ainsi :

$$\begin{array}{rcll} (\rho[lv \mapsto \delta])(lv) & = & \delta & \text{ENV-GET-1} \\ (\rho[lv_1 \mapsto \delta])(lv_2) & = & \rho(lv_2) \text{ si } lv_1 \neq lv_2 & \text{ENV-GET-2} \end{array}$$

L'adresse d'une left-value ne peut pas être $\texttt{NULL}$. Dans la suite, nous utiliserons la notation $\rho[lv_1 \mapsto \delta_1, lv_2 \mapsto \delta_2]$ (en supposant que $lv_1 \neq lv_2$) comme un raccourci pour $\rho[lv_1 \mapsto \delta_1][lv_2 \mapsto \delta_2]$. La notation $\rho - \{lv\}$ signifie que la left-value lv et l'adresse associée sont enlevées de l'environnement ρ :

$$\begin{array}{rcll} (\rho - \{lv\})(lv) & = & \bot & \text{ENV-DEL-1} \\ (\rho - \{lv_1\})(lv_2) & = & \rho(lv_2) \text{ si } lv_1 \neq lv_2 & \text{ENV-DEL-2} \end{array}$$

Nous définissons également une relation d'inclusion sur les environnements : ρ_1 est inclus dans ρ_2 si et seulement si, pour chaque paire (lv, δ), si $\delta \neq \bot$ est l'adresse de lv dans ρ_1, alors δ est aussi l'adresse de lv dans ρ_2.

$$\rho_1 \subseteq \rho_2 \equiv \forall lv \in LV.\ \forall \delta \in LOC.\ (\rho_1(lv) = \delta \Rightarrow \rho_2(lv) = \delta)$$

3.2.2 STORES

Le domaine des valeurs VAL est défini comme l'union des valeurs des termes et prédicats E-ACSL et des expressions C : $VAL = \mathbb{Z} \cup LOC \cup \{\bot\}$. Le symbole $\bot$ dénote une valeur indéfinie qui peut être soit une adresse indéfinie soit une valeur entière indéfinie. LOC est inclus dans VAL car la valeur d'un pointeur est une adresse. Un *store* est une fonction $\sigma \in LOC \to VAL$ qui associe une valeur à chaque adresse. Nous notons $STORE = LOC \to VAL$ le domaine des *stores*. La notation $\sigma[\delta \mapsto v]$ désigne un nouveau *store*, correspondant au *store* σ dans lequel la valeur v est stockée à l'adresse δ. Le *store* $\sigma[\delta \mapsto v]$ est défini ainsi :

$$\begin{array}{rcll} (\sigma[\delta \mapsto v])(\delta) & = & v & \text{STORE-GET-1} \\ (\sigma[\delta_1 \mapsto v])(\delta_2) & = & \sigma(\delta_2) \text{ si } \delta_1 \neq \delta_2 & \text{STORE-GET-2} \\ \sigma(\texttt{NULL}) & = & \bot & \text{STORE-GET-3} \end{array}$$

Dans la suite, nous utiliserons la notation $\sigma[\delta_1 \mapsto v_1, \delta_2 \mapsto v_2]$ (en supposant que $\delta_1 \neq \delta_2$) comme un raccourci pour $\sigma[\delta_1 \mapsto v_1][\delta_2 \mapsto v_2]$. La notation $\sigma - \{\delta\}$ signifie que l'adresse δ et la valeur associée sont enlevées du store σ :

$$
\begin{array}{rcll}
(\sigma - \{\delta\})(\delta) & = & \bot & \text{STORE-DEL-1} \\
(\sigma - \{\delta_1\})(\delta_2) & = & \sigma(\delta_2) \text{ si } \delta_1 \neq \delta_2 & \text{STORE-DEL-2}
\end{array}
$$

Nous définissons également une relation d'inclusion sur les stores : σ_1 est inclus dans σ_2 si et seulement si, pour chaque paire (δ, v), si $v \neq \bot$ est la valeur à l'adresse δ dans σ_1, alors v est également la valeur à l'adresse δ dans σ_2.

$$\sigma_1 \subseteq \sigma_2 \equiv \forall \delta \in LOC.\ \forall v \in VAL - \{\bot\}.\ (\sigma_1(\delta) = v \Rightarrow \sigma_2(\delta) = v)$$

3.2.3 MÉMOIRES

Nous définissons la notion de mémoire, correspondant à un couple environnement, store : $MEM = ENV \times STORE \cup \{\gamma_\lightning\}$. Une mémoire $\gamma \in MEM$ est un paramètre et un résultat des fonctions sémantiques définies dans la suite du chapitre. Cette mémoire sera notée :
– (ρ, σ) si nous devons accéder à l'environnement ρ ou au store σ ;
– γ sinon.

Nous définissons une mémoire spécifique d'erreur, notée $\gamma_\lightning \in MEM$. Une telle mémoire est le résultat de l'évaluation soit d'une instruction provoquant une erreur soit d'une annotation invalide (par exemple, `assert` ou `loop invariant` faux).

Une variable simple x de type C scalaire (`int`, ...) est représentée dans une mémoire $\gamma = (\rho, \sigma)$ par :

– une association d'une adresse δ à x dans l'environnement ρ notée $x \mapsto \delta$
– une association de sa valeur v à l'adresse mémoire δ dans le *store* σ notée $\delta \mapsto v$.

Donc la sémantique d'une variable simple x est sa valeur définie par $\sigma(\rho(x))$.

Un tableau x de taille n est représenté dans une mémoire (ρ, σ) par n adresses auxquelles sont associées respectivement ses n valeurs désignées syntaxiquement par $x_{[0]}$, $x_{[1]}$, ..., $x_{[n-1]}$. À la notation syntaxique $x_{[i]}$ est associée une *left-value* ($\in LV$) également dénotée $x_{[i]}$. À cette *left-value* est associée une adresse mémoire δ_i dans l'environnement ρ. À cette adresse mémoire est associée une valeur v_i dans le *store*. Donc la sémantique d'un accès à un élément d'un tableau $x_{[i]}$ est sa valeur définie par $\sigma(\rho(x_{[}\mathcal{E}[\![\ i\ ]\!]\ (\rho, \sigma)_{]}))$ où $\mathcal{E}[\![\ i\ ]\!]\ (\rho, \sigma)$ est la sémantique de i dans la mémoire, c'est-à-dire la valeur de i dans la mémoire (ρ, σ). Définissons maintenant la fonction sémantique $\mathcal{E}$.

3.2.4 SÉMANTIQUE DÉNOTATIONNELLE DES TERMES, PRÉDICATS ET EXPRESSIONS

La sémantique dénotationnelle des expressions C, des termes et des prédicats E-ACSL est définie par la fonction $\mathcal{E} : (EXP \cup TERM \cup PRED) \to MEM \to VAL$. Les domaines de syntaxe abstraite EXP, $TERM$ et $PRED$ correspondent respectivement aux entités syntaxiques *exp*, *term* et *pred* définies dans les figures 3.2 et 3.3. Cette fonction associe une valeur à une expression C en fonction d'une mémoire $\gamma = (\rho, \sigma)$. Ici on utilise l'hypothèse selon laquelle les expressions du langage considéré ne génèrent pas d'effets de bord. L'absence d'effet de bord permet à la fonction $\mathcal{E}$ de ne pas devoir retourner une nouvelle mémoire.

$$\mathcal{E}[\![\ cst\ ]\!]\ \gamma = cst \quad \text{E-CST}$$

$$\mathcal{E}[\![\ x\ ]\!]\ (\rho,\sigma) = \sigma(\rho(x)) \quad \text{E-LVAL}$$

$$\mathcal{E}[\![\ x\texttt{[}h\texttt{]}\ ]\!]\ (\rho,\sigma) = \sigma(\rho(x\ \texttt{[}\mathcal{E}[\![\ h\ ]\!]\ (\rho,\sigma)\texttt{]})) \quad \text{E-DEREF}$$

$$\mathcal{E}[\![\ unop\ h\ ]\!]\ \gamma = unop\ (\mathcal{E}[\![\ h\ ]\!]\ \gamma) \quad \text{E-UNOP}$$

$$\mathcal{E}[\![\ h_1\ binop\ h_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ h_1\ ]\!]\ \gamma)\ binop\ (\mathcal{E}[\![\ h_2\ ]\!]\ \gamma) \quad \text{E-BINOP}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash abs}(t)\ ]\!]\ \gamma = |\ \mathcal{E}[\![\ t\ ]\!]\ \gamma\ | \quad \text{E-ABS}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash sum}(t_1, t_2, \texttt{\textbackslash lambda integer}\ k; t_3\ )\ ]\!]\ \gamma$$
$$= \Sigma'(\mathcal{E}[\![\ t_1\ ]\!]\ \gamma, \mathcal{E}[\![\ t_2\ ]\!]\ \gamma, (\lambda\ k.\mathcal{E}[\![\ t_3\ ]\!]\ \gamma)) \quad \text{E-SUM}$$
$$\text{où } \Sigma'(v_1, v_2, f) = \sum_{i=v_1}^{v_2} f(i)$$

$$\mathcal{E}[\![\ \texttt{\textbackslash product}(t_1, t_2, \texttt{\textbackslash lambda integer}\ k; t_3\ )\ ]\!]\ \gamma$$
$$= \Pi'(\mathcal{E}[\![\ t_1\ ]\!]\ \gamma, \mathcal{E}[\![\ t_2\ ]\!]\ \gamma, (\lambda\ k.\mathcal{E}[\![\ t_3\ ]\!]\ \gamma)) \quad \text{E-PROD}$$
$$\text{où } \Pi'(v_1, v_2, f) = \prod_{i=v_1}^{v_2} f(i)$$

$$\mathcal{E}[\![\ \texttt{\textbackslash numof}(t_1, t_2, \texttt{\textbackslash lambda integer}\ k; t_3\ )\ ]\!]\ \gamma$$
$$= N'(\mathcal{E}[\![\ t_1\ ]\!]\ \gamma, \mathcal{E}[\![\ t_2\ ]\!]\ \gamma, (\lambda\ k.\mathcal{E}[\![\ t_3\ ]\!]\ \gamma)) \quad \text{E-NUM}$$
$$\text{où } N'(v_1, v_2, f) = \sum_{i=v_1}^{v_2} (f(i)?1 : 0)$$

$$\mathcal{E}[\![\ h_1\ ?\ h_2\ :\ h_3\ ]\!]\ \gamma = \mathcal{E}[\![\ h_2\ ]\!]\ \gamma \text{ si } (\mathcal{E}[\![\ h_1\ ]\!]\ \gamma) \neq 0, \quad \text{E-TIF}$$
$$\mathcal{E}[\![\ h_3\ ]\!]\ \gamma \text{ sinon}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash old}(t)\ ]\!]\ \gamma = \mathcal{E}[\![\ t\ ]\!]\ \gamma_{Beg_f} \quad \text{E-OLD}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash null}\ ]\!]\ \gamma = \texttt{NULL} \quad \text{E-NULL}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash result}\ ]\!]\ \gamma = \mathcal{E}[\![\ \texttt{res}\ ]\!]\ \gamma \quad \text{E-RES}$$

FIGURE 3.4 – Sémantique dénotationnelle des termes et expressions

La figure 3.4 donne la sémantique des expressions et des termes. Ces deux éléments du langage se retrouvent sur la même figure car les termes peuvent être vus comme une extension des expressions C. Les cinq premières règles de la figure 3.4 sont communes aux termes et aux expressions, les autres sont spécifiques aux termes. La règle E-LVAL traite le cas où une left-value est un identificateur de variable x. $\rho(x)$ est l'adresse de x, la valeur à cette adresse est $\sigma(\rho(x))$. La règle E-DEREF traite le cas où une left-value est un identificateur de variable x, accédé à l'offset de valeur h. L'unique différence avec la règle E-LVAL est l'évaluation de h. Dans les règles E-UNOP et E-BINOP, *unop* et *binop* correspondent à l'application de l'opérateur unaire ou binaire correspondant, parmi ceux autorisés par la grammaire de la figure 3.2. Dans la règle E-ABS, $|\ .\ | : VAL \rightarrow VAL$ est la fonction valeur absolue.

Dans les règles E-SUM, E-PROD et E-NUM, la variable logique k liée dans le terme $\texttt{\textbackslash lambda integer}\ k;\ t_3$ est supposée différente de toute autre variable du programme (lo-

p_1	p_2	! p_1	p_1 && p_2	p_1 || p_2	p_1 ==> p_2	p_1 <==> p_2
0	0	1	0	0	1	1
0	1	1	0	1	1	0
1	0	0	0	1	0	0
1	1	0	1	1	1	1

FIGURE 3.5 – Table de vérité des prédicats

gique ou C). Pour la règle E-SUM nous définissons une fonction $\Sigma'(v_1, v_2, f)$ pour tous v_1 et v_2 de type VAL et pour toute fonction $f : VAL \to VAL$. La fonction Σ' est définie en fonction de la somme mathématique sur les entiers en figure 3.4. Lors de l'application de la fonction Σ', les deux premiers paramètres (v_1 et v_2) sont évalués avant le troisième. Ainsi, la variable k a toujours une valeur définie lors l'évaluation du terme t_3. Les règles E-PROD et E-NUM sont définies de manière similaire.

Dans la règle E-OLD, γ_{Beg_f} désigne la mémoire au début de l'exécution de la fonction courante, nous supposons qu'une telle mémoire est conservée à chaque appel. Notons que nous n'avons pas formalisé la mémorisation des états mémoires au début de la fonction en cours d'évaluation car cela alourdirait considérablement la définition de la sémantique.

La figure 3.6 donne la sémantique des prédicats. Les valeurs de vérité des prédicats sont encodées par les entiers 0 et 1 comme en C. La règle P-VALID énonce qu'un identificateur x est valide si l'adresse associée à x dans l'environnement est définie (différente de $\perp$). Un pointeur est valide en mémoire si on peut le déréférencer afin de pouvoir accéder ou modifier l'emplacement mémoire qu'il référence (c'est-à-dire vers lequel il pointe). Le pointeur égal à l'adresse d'une variable locale est valide dans la portée de cette variable. De la même manière, `\valid(x+(t1..t2))` est valide si les pointeurs `(x+t1)`, ..., `(x+t2)` peuvent être déréférencés.

Dans la règle P-FORALL nous définissons une fonction $F'(v_1, v_2, f)$ pour tous v_1 et v_2 de type VAL et pour toute fonction $f : VAL \to VAL$. La fonction F' est définie en fonction du quantificateur universel $\forall$ en figure 3.6. Lors de l'application de la fonction F' dans la règle P-FORALL, les deux premiers paramètres (v_1 et v_2) sont évalués avant le troisième. Ainsi, la variable k a toujours une valeur définie lors l'évaluation du prédicat p. La règle P-EXISTS est définie de manière similaire.

La figure 3.5 donne la table de vérité des opérations de négation, de conjonction, de disjonction, d'implication et d'équivalence sur les prédicats E-ACSL.

3.2.5 SÉMANTIQUE DÉNOTATIONNELLE DES INSTRUCTIONS

La sémantique dénotationnelle des instructions C est exprimée avec la fonction $C : INSTR \to MEM \to MEM$. Cette fonction associe à une instruction i, dans un contexte mémoire γ, une nouvelle mémoire γ' qui tient compte des modifications effectuées par i dans γ. La sémantique d'une séquence d'instructions est exprimée avec la fonction $C^* : INSTR^* \to MEM \to MEM$. Cette fonction définie en figure 3.7 revient à appeler successivement C sur chaque instruction dans l'ordre de la séquence.

La sémantique des instructions C est donnée en figure 3.8. La règle C-ERR indique que toute évaluation d'instruction à partir de la mémoire d'erreur $\gamma_\lightning$ produit $\gamma_\lightning$. Pour toutes

$$\mathcal{E}[\![\ \texttt{\textbackslash true}\ ]\!]\ \gamma = 1 \quad \text{P-TRUE}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash false}\ ]\!]\ \gamma = 0 \quad \text{P-FALSE}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash valid(}x\texttt{)}\ ]\!]\ (\rho,\sigma) = 0 \text{ si } \rho(x) = \bot,\ 1 \text{ sinon} \quad \text{P-VALID}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash valid(}x\texttt{+(}t_1\ \texttt{..}t_2\texttt{))}\ ]\!]\ \gamma = \mathcal{E}[\![\ \texttt{\textbackslash valid(}x\texttt{+}t_1\texttt{)}\ \texttt{\&\&}\ \ldots\ \texttt{\&\&}\ \texttt{\textbackslash valid(}x\texttt{+}t_2\texttt{)}\ ]\!]\ \gamma \quad \text{P-VALIDR}$$

$$\mathcal{E}[\![\ \texttt{\textbackslash forall integer}\ k\texttt{;}t_1\ \texttt{<=}\ k\ \texttt{<=}\ t_2\ \texttt{==>}\ p\ ]\!]\ \gamma = F'(\mathcal{E}[\![\ t_1\ ]\!]\ \gamma, \mathcal{E}[\![\ t_2\ ]\!]\ \gamma, (\lambda\ k.\mathcal{E}[\![\ p\ ]\!]\ \gamma)) \quad \text{P-FORALL}$$

où $F'(v_1, v_2, f) = \forall i.(v_1 \leq i \leq v_2 \Rightarrow f(i))$

$$\mathcal{E}[\![\ \texttt{\textbackslash exists integer}\ k\texttt{;}t_1\ \texttt{<=}\ k\ \texttt{<=}\ t_2\ \texttt{\&\&}\ p\ ]\!]\ \gamma = E'(\mathcal{E}[\![\ t_1\ ]\!]\ \gamma, \mathcal{E}[\![\ t_2\ ]\!]\ \gamma, (\lambda\ k.\mathcal{E}[\![\ p\ ]\!]\ \gamma)) \quad \text{P-EXISTS}$$

où $E'(v_1, v_2, f) = \exists i.(v_1 \leq i \leq v_2 \wedge f(i))$

$$\mathcal{E}[\![\ \texttt{!}p\ ]\!]\ \gamma = \neg\ (\mathcal{E}[\![\ p\ ]\!]\ \gamma) \quad \text{P-NOT}$$

$$\mathcal{E}[\![\ p_1\ \texttt{\&\&}\ p_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ p_1\ ]\!]\ \gamma) \wedge (\mathcal{E}[\![\ p_2\ ]\!]\ \gamma) \quad \text{P-AND}$$

$$\mathcal{E}[\![\ p_1\ \texttt{||}\ p_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ p_1\ ]\!]\ \gamma) \vee (\mathcal{E}[\![\ p_2\ ]\!]\ \gamma) \quad \text{P-OR}$$

$$\mathcal{E}[\![\ p_1\ \texttt{==>}\ p_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ p_1\ ]\!]\ \gamma) \Rightarrow (\mathcal{E}[\![\ p_2\ ]\!]\ \gamma) \quad \text{P-IMPL}$$

$$\mathcal{E}[\![\ p_1\ \texttt{<==>}\ p_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ p_1\ ]\!]\ \gamma) \Leftrightarrow (\mathcal{E}[\![\ p_2\ ]\!]\ \gamma) \quad \text{P-EQ}$$

$$\mathcal{E}[\![\ t\ \texttt{?}\ p_1\ \ \texttt{:}\ p_2\ ]\!]\ \gamma = \mathcal{E}[\![\ p_1\ ]\!]\ \gamma \text{ si } (\mathcal{E}[\![\ t\ ]\!]\ \gamma) \neq 0, \mathcal{E}[\![\ p_2\ ]\!]\ \gamma \text{ sinon} \quad \text{P-PIF}$$

$$\mathcal{E}[\![\ t_1\ rel\ t_2\ ]\!]\ \gamma = (\mathcal{E}[\![\ t_1\ ]\!]\ \gamma)\ rel\ (\mathcal{E}[\![\ t_2\ ]\!]\ \gamma) \quad \text{P-REL}$$

FIGURE 3.6 – Sémantique dénotationnelle des prédicats

les règles suivantes, nous supposons que l'évaluation se fait à partir d'une mémoire différente de $\gamma_{\lightning}$. La règle C-DECL alloue l'adresse δ à la variable x (de type *basic-type* ou *basic-type*$_*$) dans l'environnement ρ, où δ est une adresse fraîche (non utilisée). Les règles C-SET et C-SET-2 associent la valeur de e à l'adresse de la left-value correspondante dans le store.

Dans les règles C-Z-SET et C-Z-UNSET nous utilisons la notation abrégée $\square_{\texttt{var}}$ pour indiquer que la variable entière `var` doit être déclarée et allouée au début de la portion de code insérée, et la notation $\texttt{var}^{\boxtimes}$ pour indiquer que la variable `var` doit être désallouée à la fin du code inséré. Cette notation est utilisée dans les instructions manipulant des données entières non bornées. La partie 4.1.2 illustre la détection de débordements d'entiers par GMP [Granlund et al., 2014]. Dans la règle C-Z-SET, la variable entière (non bornée) x est allouée et on lui affecte la valeur de e dans σ. Dans la règle C-Z-UNSET, la variable entière (non bornée) x et son adresse sont enlevées de la mémoire. Les règles C-Z-SET et C-Z-UNSET traitent uniquement le cas où une left-value est un identificateur et pas un accès à un tableau car le langage E-ACSL ne supporte pas les tableaux d'entiers non bornés.

La sémantique de la fonction `fassert` donnée par la règle C-FASSERT énonce que si l'expression e s'évalue en une expression vraie (non nulle), alors la mémoire γ est inchangée,

$$
\begin{array}{rcll}
C^*[\![\ A_1\ A_2\ ]\!]\ \gamma & = & C^*[\![\ A_2\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma) & \text{C-SEQ-1} \\
C^*[\![\ i\ A\ ]\!]\ \gamma & = & C^*[\![\ A\ ]\!]\ (C[\![\ i\ ]\!]\ \gamma) & \text{C-SEQ-2} \\
C^*[\![\ ]\!]\ \gamma & = & \gamma & \text{C-SEQ-3}
\end{array}
$$

FIGURE 3.7 – Sémantique dénotationnelle des séquences d'instructions

sinon on obtient la mémoire d'erreur $\gamma_{\lightning}$. La sémantique de la fonction `fassume` donnée par la règle C-FASSUME énonce que l'expression e est supposée vraie et doit donc s'évaluer en une expression non nulle. La mémoire est inchangée après cet appel de fonction. e ne peut pas s'évaluer en faux (0) par définition de la fonction `fassume`, qui peut être vue comme un moyen de rajouter une précondition à l'intérieur d'une fonction.

La sémantique des fonctions `fvalid` et `fvalidr` est donnée par les règles C-FVALID et C-FVALIDR. Elles énoncent que dans la nouvelle mémoire, la left-value lv a la valeur vrai (1) si les pointeurs en argument de la fonction sont valides, et faux (0) sinon.

La sémantique de la fonction `malloc` donnée par la règle C-MALLOC énonce que de nouvelles adresses (fraîches) sont associées aux left-values allouées pour le tableau x, et que la valeur associée à l'adresse de x dans le store est l'adresse du premier élément alloué ($\rho(x_{[0]})$), comme c'est le cas dans le langage C. Nous utilisons une fonction `malloc` prenant deux paramètres : e, le nombre d'éléments à allouer, et s, la taille (en octets) des éléments. Un pointeur alloué dynamiquement à l'aide de la fonction `malloc` est valide tant qu'il n'est pas libéré par la fonction `free`. La sémantique de la fonction `free` donnée par la règle C-FREE est l'inverse de la sémantique de `malloc` : les left-values allouées et l'adresse de la variable x sont enlevées de la mémoire. Dans ces deux règles, x est une variable de type pointeur, la valeur associée à son adresse dans la règle C-MALLOC est donc une adresse. La variable N dans la règle C-FREE est le nombre d'éléments alloués dynamiquement dans le tableau x par un appel à la fonction `malloc`.

La sémantique des appels de fonction est donnée par les règles C-FCT1 et C-FCT2. Dans ces règles, la sémantique de la fonction f est calculée à l'aide de la fonction $\mathcal{F}$ donnée en figure 3.9. $(\rho[x_1 \mapsto \delta_1, ..., x_N \mapsto \delta_N], \sigma[\delta_1 \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ (\rho,\sigma), ..., \delta_N \mapsto \mathcal{E}[\![\ e_N\ ]\!]\ (\rho,\sigma)\ ])$ est la mémoire dans laquelle les paramètres effectifs (les expressions $e_1, ..., e_N$) sont affectés aux paramètres formels (les identificateurs $x_1, ..., x_N$).

La règle C-RETURN énonce qu'une instruction **`return`** ne modifie pas la mémoire. En effet, le programme est dans une forme normalisée où la valeur de retour d'une fonction est stockée dans une variable $\texttt{res}_f$, où f est le nom de la fonction courante. Cette variable est lue après l'appel de la fonction comme l'énonce la règle C-FCT2.

La sémantique des assertions E-ACSL donnée par la règle C-ASSERT est similaire à la sémantique de la fonction `fassert`. Si le prédicat p s'évalue en une expression non nulle, la mémoire γ est inchangée, sinon la mémoire d'erreur $\gamma_{\lightning}$ est obtenue.

La sémantique des boucles annotées par un contrat contenant un invariant p, une liste de termes X et un variant t est donnée par la règle C-WHILE. Nous appelons G l'ensemble des left-values dans la portée courante. G contient donc les left-values des paramètres formels de la fonction courante et des variables locales définies avant la boucle. Les différents cas possibles sont notés C-WHILE-1 à C-WHILE-7 et sont ordonnés de telle sorte que la sémantique est définie par le premier cas qui s'applique dans l'ordre 1 à 7. Tout d'abord, si l'invariant de boucle n'est pas établi avant l'exécution du corps de la boucle, alors la mémoire d'erreur $\gamma_{\lightning}$ est obtenue (C-WHILE-1). Si la condition de boucle e s'évalue à 0 (la

$$C[\![\ i\ ]\!]\ \gamma_{\lightning} = \gamma_{\lightning} \quad \text{C-ERR}$$

$$C[\![\ T\ x;\ ]\!]\ (\rho,\sigma) = (\rho[x \mapsto \delta], \sigma[\delta \mapsto \bot]) \text{ où } \delta \text{ est une adresse fraîche} \quad \text{C-DECL}$$

$$C[\![\ x = e;\ ]\!]\ (\rho,\sigma) = (\rho, \sigma[\rho(x) \mapsto \mathcal{E}[\![\ e\ ]\!]\ (\rho,\sigma)]) \quad \text{C-SET}$$

$$C[\![\ x\texttt{[}e\texttt{]} = e_2;\ ]\!]\ (\rho,\sigma) = (\rho, \sigma[\rho(x\ \texttt{[}\mathcal{E}[\![\ e\ ]\!]\ (\rho,\sigma)\texttt{]}) \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ (\rho,\sigma)]) \quad \text{C-SET-2}$$

$$C[\![\ ^{\square}\ x = e\ ;\ ]\!]\ (\rho,\sigma) = (\rho[x \mapsto \delta], \sigma[\delta \mapsto \mathcal{E}[\![\ e\ ]\!]\ (\rho,\sigma)]) \quad \text{C-Z-SET}$$
où δ est une adresse fraîche

$$C[\![\ x^{\boxtimes};\ ]\!]\ (\rho,\sigma) = (\rho - \{x\}, \sigma - \{\rho(x)\}) \quad \text{C-Z-UNSET}$$

$$C[\![\ \texttt{fassert(}e\texttt{);}\ ]\!]\ \gamma = \gamma \text{ si } (\mathcal{E}[\![\ e\ ]\!]\ \gamma) \neq 0,\ \gamma_{\lightning} \text{ sinon} \quad \text{C-FASSERT}$$

$$C[\![\ \texttt{fassume(}e\texttt{);}\ ]\!]\ \gamma = \gamma \text{ (et on suppose dans la suite } (\mathcal{E}[\![\ e\ ]\!]\ \gamma) \neq 0) \quad \text{C-FASSUME}$$

$$C[\![\ lv = \texttt{fvalid(}x\texttt{);}\ ]\!]\ (\rho,\sigma) = (\rho, \sigma[\rho(lv) \mapsto 0]) \text{ si } \rho(x) = \bot, \quad \text{C-FVALID}$$
$(\rho, \sigma[\rho(lv) \mapsto 1])$ sinon

$$C[\![\ lv = \texttt{fvalidr(}e_1\texttt{,}e_2\texttt{,}e_3\ \texttt{);}\ ]\!]\ (\rho,\sigma) = (\rho, \sigma[\rho(lv) \mapsto 0]) \text{ si } (\rho(e_1 + \mathcal{E}[\![\ e_2\ ]\!]\ (\rho,\sigma)) = \bot) \quad \text{C-FVALIDR}$$
$\vee \cdots \vee\ (\rho(e_1 + \mathcal{E}[\![\ e_3\ ]\!]\ (\rho,\sigma)) = \bot)$,
$(\rho, \sigma[\rho(lv) \mapsto 1])$ sinon

$$C[\![\ x = \texttt{malloc(}e,s\texttt{);}\ ]\!]\ (\rho,\sigma) = (\rho[x\texttt{[0]} \mapsto \delta_1, \ldots, x\texttt{[}N-1\texttt{]} \mapsto \delta_N], \quad \text{C-MALLOC}$$
$\sigma[\rho(x) \mapsto \delta_1, \delta_1 \mapsto \bot, \ldots, \delta_N \mapsto \bot])$
où $N = \mathcal{E}[\![\ e\ ]\!]\ (\rho,\sigma)$
et $\delta_1, \ldots, \delta_N$ sont des adresses fraîches

$$C[\![\ \texttt{free(}x\texttt{)}\ ]\!]\ (\rho,\sigma) = (\rho - \{x\texttt{[0]}\} - \ldots - \{x\texttt{[}N-1\texttt{]}\}, \quad \text{C-FREE}$$
$\sigma - \{\rho(x)\} - \{\rho(x\texttt{[0]})\} - \ldots - \{\rho(x\texttt{[}N-1\texttt{]})\})$
où N est le nombre d'éléments alloués par le `malloc` correspondant
si x est un pointeur alloué et non libéré, $\gamma_{\lightning}$ sinon

$$C[\![\ f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_N\texttt{);}\ ]\!]\ (\rho,\sigma) = \mathcal{F}[\![\ \texttt{/*@... */ }f\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_N\ \texttt{) \{...\}}\ ]\!] \quad \text{C-FCT1}$$
$(\rho[x_1 \mapsto \delta_1, \ldots, x_N \mapsto \delta_N],$
$\sigma[\delta_1 \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ (\rho,\sigma), \ldots, \delta_N \mapsto \mathcal{E}[\![\ e_N\ ]\!]\ (\rho,\sigma)\ ])$
où $\delta_1, \ldots, \delta_N$ sont des adresses fraîches

$$C[\![\ lv = f\texttt{(}e_1\texttt{,}\ldots\texttt{,}e_N\ \texttt{);}\ ]\!]\ (\rho,\sigma) = (\rho, \sigma[\rho(lv) \mapsto \mathcal{E}[\![\ \texttt{res}_f\ ]\!]\ (\rho',\sigma')]) \quad \text{C-FCT2}$$
où $(\rho', \sigma') = \mathcal{F}[\![\ \texttt{/*@... */ }f\texttt{(}x_1\texttt{,}\ldots\texttt{,}x_N\ \texttt{) \{...\}}\ ]\!]$
$(\rho[x_1 \mapsto \delta_1, \ldots, x_N \mapsto \delta_N],$
$\sigma[\delta_1 \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ (\rho,\sigma), \ldots, \delta_N \mapsto \mathcal{E}[\![\ e_N\ ]\!]\ (\rho,\sigma)\ ])$
où $\delta_1, \ldots, \delta_N$ sont des adresses fraîches

$$C[\![\ \texttt{return res}_f\texttt{;}\ ]\!]\ \gamma = \gamma \quad \text{C-RETURN}$$

$$C[\![\ \texttt{if(}e\texttt{) \{}A\texttt{\} else \{}B\texttt{\}}\ ]\!]\ \gamma = C^*[\![\ A\ ]\!]\ \gamma \text{ si } (\mathcal{E}[\![\ e\ ]\!]\ \gamma) \neq 0,\ C^*[\![\ B\ ]\!]\ \gamma \text{ sinon} \quad \text{C-IF}$$

$$C[\![\ \texttt{/*@assert }p\texttt{; */}\ ]\!]\ \gamma = \gamma \text{ si } (\mathcal{E}[\![\ p\ ]\!]\ \gamma) \neq 0,\ \gamma_{\lightning} \text{ sinon} \quad \text{C-ASSERT}$$

$$C[\![\ \texttt{/*@loop invariant }p\texttt{; loop assigns }X\texttt{; loop variant }t\texttt{; */ while(}e\texttt{) \{}A\texttt{\}}\ ]\!]\ \gamma \quad \text{C-WHILE}$$

$$= \gamma_{\lightning} \text{ si } \mathcal{E}[\![\ p\ ]\!]\ \gamma = 0 \quad \text{C-WHILE-1}$$

$$= \gamma \text{ si } (\mathcal{E}[\![\ e\ ]\!]\ \gamma) = 0 \quad \text{C-WHILE-2}$$

$$= \gamma_{\lightning} \text{ si } (\mathcal{E}[\![\ t\ ]\!]\ \gamma) < 0 \quad \text{C-WHILE-3}$$

$$= \gamma_{\lightning} \text{ si } \mathcal{E}[\![\ p\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma) = 0 \quad \text{C-WHILE-4}$$

$$= \gamma_{\lightning} \text{ si } (\mathcal{E}[\![\ t\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma)) \geq (\mathcal{E}[\![\ t\ ]\!]\ \gamma) \quad \text{C-WHILE-5}$$

$$= \gamma_{\lightning} \text{ si } \exists x.(x \in G - X.\ (\mathcal{E}[\![\ x\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma)) \neq (\mathcal{E}[\![\ x\ ]\!]\ \gamma)\ ) \quad \text{C-WHILE-6}$$

$$= C[\![\ \texttt{/*@... */while(}e\texttt{) \{}A\texttt{\}}\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma) \text{ sinon} \quad \text{C-WHILE-7}$$

FIGURE 3.8 – Sémantique dénotationnelle des instructions

boucle n'est pas exécutée), alors la mémoire reste inchangée (C-WHILE-2). Si le variant de boucle n'est pas positif ou nul avant l'exécution de la boucle, alors $\gamma_{\lightning}$ est obtenue (C-WHILE-3). Si l'invariant de boucle n'est pas maintenu après l'exécution du corps de boucle, alors $\gamma_{\lightning}$ est obtenue (C-WHILE-4). Si le variant de boucle ne décroît pas strictement entre le début et la fin de l'exécution du corps de boucle, alors $\gamma_{\lightning}$ est obtenue (C-WHILE-5). Si l'évaluation d'un élément appartenant à $G - X$ (ensemble des left-values n'appartenant pas à la clause `loop assigns` de la boucle) est différente entre le début et la fin de l'exécution du corps de la boucle, alors $\gamma_{\lightning}$ est obtenue (C-WHILE-6). Si aucun des six cas précédents ne s'applique, alors il faut ré-évaluer la sémantique de la boucle dans la mémoire obtenue après exécution du corps de la boucle dans γ (C-WHILE-7). Nous supposons pour simplifier que la non-terminaison éventuelle d'une boucle est toujours évitée grâce au variant.

$$
\begin{array}{lll}
\mathcal{F}[\![\ \texttt{/*@requires } p_1\texttt{; assigns } X\texttt{; ensures } p_2\texttt{; */ } f\texttt{(}id_1\texttt{, ..., }id_N\texttt{) \{}A\texttt{\}}\ ]\!]\ \gamma & & \text{F} \\
= \quad \gamma_{\lightning} \text{ si } \mathcal{E}[\![\ p_1\ ]\!]\ \gamma = 0 \text{ et } f \text{ est une fonction appelée} & & \text{F-1} \\
\quad \gamma_{\lightning} \text{ si } \exists x.(x \in G - X.\ (\mathcal{E}[\![\ x\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma)) \neq (\mathcal{E}[\![\ x\ ]\!]\ \gamma)\ ) & & \text{F-2} \\
\quad \gamma_{\lightning} \text{ si } \mathcal{E}[\![\ p_2\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma) = 0 & & \text{F-3} \\
\quad C^*[\![\ A\ ]\!]\ \gamma \text{ sinon} & & \text{F-4}
\end{array}
$$

FIGURE 3.9 – Sémantique dénotationnelle des fonctions

La sémantique dénotationnelle des fonctions est exprimée avec la fonction $\mathcal{F} : FCT \rightarrow MEM \rightarrow MEM$ où FCT correspond à l'entité syntaxique *function* définie par la figure 3.1. FCT est l'ensemble des déclarations de fonctions du programme. Cette fonction calcule une nouvelle mémoire à partir d'une fonction annotée et d'une mémoire γ, ce qui permet de traiter les appels de fonction (règles C-FCT1 et C-FCT2 de la figure 3.8) et la fonction sous vérification.

La sémantique des fonctions est donnée en figure 3.9 par la règle F. Les différents cas possibles sont notés F-1 à F-4 et sont ordonnés comme pour le WHILE. Si f est la fonction sous vérification, alors sa précondition p_1 est supposée vraie. Si f n'est pas la fonction sous vérification, alors la mémoire d'erreur $\gamma_{\lightning}$ est obtenue si le prédicat p_1 s'évalue à faux (F-1). Rappelons que nous appelons G l'ensemble des left-values dans la portée courante. G contient donc les left-values des paramètres formels de la fonction courante. Si l'évaluation d'un élément appartenant à $G - X$ (ensemble des left-values n'appartenant pas à la clause `assigns` de la fonction) est différente entre le début et la fin de l'exécution du corps de la fonction, alors $\gamma_{\lightning}$ est obtenue (F-2). Si le prédicat de la postcondition p_2 est faux à la fin de la fonction, alors $\gamma_{\lightning}$ est obtenue (F-3). Enfin, si aucun des trois cas précédents ne s'est produit, le résultat est la mémoire obtenue après exécution des instructions du corps de la fonction (F-4).

CONCLUSION DU CHAPITRE

Dans ce chapitre nous avons présenté un sous-ensemble du langage C et du langage E-ACSL normalisé que nous traitons. Nous avons défini la sémantique dénotationnelle de chaque élément du langage pour pouvoir justifier de la correction de la traduction des annotations en C dans le but de générer des tests. Le chapitre 4 présente la traduction de ces annotations E-ACSL en code C et justifie qu'ils sont sémantiquement équivalents, en utilisant les fonctions définissant la sémantique dénotationnelle.

4

TRADUCTION EN C DES ANNOTATIONS POUR LE TEST

Ce chapitre présente une transformation d'un programme C annoté avec E-ACSL en un programme C, appelé programme instrumenté, dans lequel les annotations E-ACSL du programme initial ont été traduites en fragments de programme C ajoutés au programme initial. Ces fragments mettent en œuvre les annotations E-ACSL afin de générer des tests satisfaisant ou violant ces annotations.

Le programme instrumenté est analysé dynamiquement par un générateur de tests structurels qui prend en entrée des programmes C. L'instrumentation permet de l'utiliser pour détecter des erreurs dans les programmes C annotés sans que cet outil n'ait à traiter les annotations dans leur format d'origine. Il détecte ces erreurs par analyse du code C produit par l'instrumentation.

Dans la partie 4.1 nous présentons les principes de la traduction des programmes annotés définis par la syntaxe présentée au chapitre précédent. Ensuite nous détaillons les règles de traduction des annotations E-ACSL en code C. Les règles de traduction pour les annotations sont décrites dans la partie 4.2. Elles reposent sur les règles de traduction des termes et des prédicats, qui sont détaillées dans les parties 4.3 et 4.4. Nous discutons dans la partie 4.5 les différences et les similitudes entre l'instrumentation pour la génération de tests et l'instrumentation pour la validation à l'exécution. Nous donnons une justification de correction des règles de traduction en partie 4.6.

4.1 PROCESSUS DE TRANSFORMATION DE PROGRAMMES

Cette partie présente le processus d'instrumentation pour la génération de tests des fonctions C annotées avec le langage d'annotations E-ACSL. Elle l'illustre sur l'exemple donné dans le listing 3.1 au chapitre précédent.

4.1.1 PRINCIPES GÉNÉRAUX DE LA TRADUCTION

Présentons maintenant les principes de la traduction d'une fonction annotée respectant la grammaire de la figure 3.1 du chapitre 3.

Le programme instrumenté obtenu après traduction des annotations de la fonction du listing 3.1 est présenté dans le listing 4.1. Les principes de la traduction sont illustrés par

des extraits de ce listing, au fur et à mesure de leur présentation.

Rappelons que pour simplifier la lecture des insertions de code générées par la traduction, nous utilisons la notation abrégée $\square_{\texttt{var}}$ pour indiquer que la variable `var` doit être déclarée et allouée au début de la portion de code insérée, et la notation $\texttt{var}^{\boxtimes}$ pour indiquer que la variable `var` doit être désallouée à la fin du code inséré. Nous soulignons les opérations (affectation, comparaison, ...) qui utilisent l'arithmétique non bornée (par opposition à l'arithmétique modulaire), et doivent être traduites en utilisant une bibliothèque comme GMP [Granlund et al., 2014] afin de conserver la sémantique de l'arithmétique entière non bornée d'E-ACSL et de pouvoir raisonner sur les débordements d'entiers. La partie 4.1.2 illustre la détection de débordements d'entiers par GMP.

```
1  int is_present(int* t, int n, int v) {
2   Beg_is_present :
3    int *old_t = t; int *old_val_t = malloc(((n-1)+1)*sizeof(int)); int old_n = n;
4    int old_v = v; int res_is_present; int i; int iter_t; res_is_present = 0; i = 0;
5    for(iter_t = 0; iter_t < n; iter_t++) old_val_t[iter_t] = t[iter_t];
6    □var_38 = 0; □var_39 = n; int var_40 = var_38⊠ <= var_39⊠; fassume(var_40);
7    fassume(fvalidr(t,0,(n-1)));
8    □var_41 = n; □var_42 = 6; int var_43 = var_41⊠ <= var_42⊠; fassume(var_43);
9   l0 :
10   □var_0 = 0; □var_1 = i; int var_2 = var_0⊠ <= var_1⊠; int var_3 = var_2;
11   if(var_3) { □var_4 = i; □var_5 = n; int var_6 = var_4⊠ <= var_5⊠; var_3 = var_6; }
12   fassert(var_3);
13   while(i < n && t[i] != v) {
14    BegIter_l0 :
15     □var_7 = n; □var_8 = i; □var_9 = var_7⊠ - var_8⊠;
16     int var_10 = 0 <= var_9; fassert(var_10);
17     i = i+1;
18    EndIter_l0 :
19     □var_14 = 0; □var_15 = i; int var_16 = var_14⊠ <= var_15⊠; int var_17 = var_16;
20     if(var_17) {
21       □var_18 = i; □var_19 = n; int var_20 = var_18⊠ <= var_19⊠; var_17 = var_20;
22     }
23     fassert(var_17);
24     □var_26 = n; □var_27 = i; □var_28 = var_26⊠ - var_27⊠;
25     int var_29 = var_28⊠ < var_9⊠; fassert(var_29);
26   }
27   if(i < n) { res_is_present = 1; }
28  End_is_present :
29   □var_30 = 0; □var_31 = res_is_present; int var_32 = var_30⊠ != var_31⊠;
30   □var_33 = 0; □var_34 = old_n;
31   int var_35 = 0;
32   for(□i_0 = var_33⊠; i_0 < var_34⊠ && !var_35; i_0++⊠) {
33     □var_36 = i_0; int var_37 = var_36⊠; var_35 = old_val_t[var_37] == old_v; }
34   fassert((!var_32 || var_35) && (!var_35 || var_32));
35   free(old_val_t); return res_is_present; }
```

Listing 4.1 – Version instrumentée du programme du listing 3.1

Durant le processus d'instrumentation, chaque fonction C est considérée indépendamment des autres. Les fonctions sont traduites dans l'ordre dans lequel elles apparaissent dans le programme. Pour chacune de ces fonctions, les annotations E-ACSL sont également considérées dans l'ordre dans lequel elles apparaissent.

Nous associons un label à chaque annotation – qui caractérise sa position dans le programme initial et l'endroit où elle doit être évaluée – ce qui nous donne une paire $(label, annotation)$. Chacune de ces paires est traduite en une séquence d'insertions de code dénotée par : $(l_1, c_1) \cdot (l_2, c_2) \cdot \ldots \cdot (l_n, c_n)$. Elle représente une liste de fragments de programmes C $c_1, c_2, \ldots, c_n$ où le fragment c_i sera inséré dans le programme instrumenté au label l_i.

Les fragments de programmes – ou insertions de code – sont les instructions nécessaires

a)

```
1  /*@ requires p;
2      typically q;
3      assigns X;
4      ensures r; */
5  T f(T1 i1, ..., Tn in) {
6    Beg_f : T res_f;
7    l :
8     /*@ loop invariant i;
9         loop assigns Y;
10        loop variant v; */
11    while(...) { ... }
12   m : /*@ assert a; */
13   End_f : return res_f;
14 }
```

b)

- $(Beg_f, \texttt{requires}\ p;) \overset{\alpha}{\mapsto} I_p$
- $(Beg_f, \texttt{typically}\ q;) \overset{\alpha}{\mapsto} I_q$
- $(End_f, \texttt{assigns}\ X;) \overset{\alpha}{\mapsto} I_X$
- $(End_f, \texttt{ensures}\ r;) \overset{\alpha}{\mapsto} I_r$
- $(l, \texttt{loop invariant}\ i;) \overset{\alpha}{\mapsto} I_i$
- $(l, \texttt{loop assigns}\ Y;) \overset{\alpha}{\mapsto} I_Y$
- $(l, \texttt{loop variant}\ v;) \overset{\alpha}{\mapsto} I_v$
- $(l, \texttt{assert}\ a;) \overset{\alpha}{\mapsto} I_a$

FIGURE 4.1 – Fonction f de P **(a)** et insertions de code générées par sa traduction **(b)**

à l'évaluation d'une annotation, d'un prédicat ou d'un terme E-ACSL. Ces insertions de code générées à partir des annotations sont placées dans une séquence d'insertions de code globale. Si plusieurs fragments ont le même label (et doivent donc être insérés au même endroit), ils sont insérés dans le programme instrumenté en respectant leur ordre dans la séquence.

Ainsi, le programme P réduit à la déclaration de la fonction f de la figure 4.1 **(a)** est transformé en programme P' obtenu à partir de P par les insertions de code I_p, I_q, I_X, I_r, I_i, I_Y, I_v et I_a, définies sur la figure 4.1 **(b)**, où la fonction de traduction $\overset{\alpha}{\mapsto}$ est définie dans la partie 4.2.

Sauvegarde des valeurs au début de la fonction. Indépendamment de la traduction des annotations, un traitement particulier est effectué pour chaque fonction, nous permettant d'obtenir la valeur du terme `\old(x)` si `x` est une variable globale ou un paramètre formel de la fonction. Chaque valeur d'entrée `x` (un paramètre formel de la fonction ou une variable globale) de type T est mémorisée par une instruction `T old_x = x;` au début de la fonction instrumentée, c'est-à-dire au label Beg_f. Pour chaque tableau ou pointeur `x`, les valeurs pointées sont mémorisées dans un tableau alloué dynamiquement `old_val_x` dont la taille est inférée à partir du prédicat `\valid` dont on suppose la présence en précondition de la fonction à traduire. Dans l'exemple du listing 4.1, les valeurs des paramètres formels `t`, `n` et `v` de la fonction `is_present` sont mémorisées lignes 3–4, un nouveau tableau `old_val_t` stocke les anciennes valeurs contenues dans `t`, il est alloué ligne 3, rempli ligne 5 et désalloué ligne 35.

Traduction des annotations. Une fois ce pré-traitement effectué, toute annotation E-ACSL de la forme `kwd w` (où `kwd` ∈ { `assert`, `requires`, `typically`, `ensures`, `loop variant`, `loop invariant`, `assigns`, `loop assigns` } et `w` est un terme ou un prédicat) est traduite.

Les figures 4.2 et 4.3 illustrent la traduction de l'invariant de boucle `loop invariant x<y` au label de boucle l. Puisque tout entier d'un type C est implicitement converti en un entier non borné de type `integer` en E-ACSL, cet invariant peut être réécrit `loop invariant` $(\mathbb{Z})$`x < ` $(\mathbb{Z})$`y;` pour expliciter cette conversion. La figure 4.2 montre la séquence d'insertions, étiquetées par des labels, obtenue après traduction complète de l'invariant de boucle. Les séquences d'instructions numérotées ⓝ désignent les fragments de code générés. Cette numérotation est reprise dans la figure 4.3. La figure 4.3 illustre le principe d'insertion des fragments de code de la figure 4.2, qui produit le programme instrumenté en appliquant les principes de traduction décrites ci-dessus.

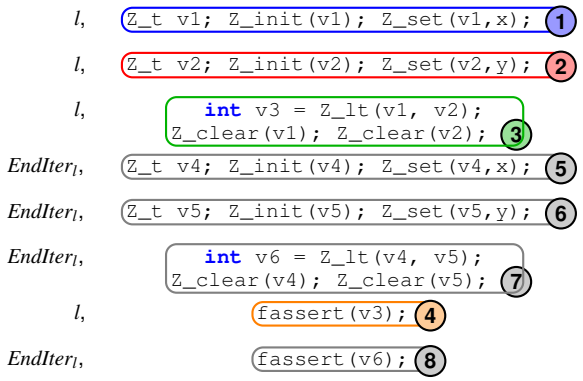

FIGURE 4.2 – Génération d'insertions de code à partir d'un invariant de boucle

Maintenant que le principe général de la traduction a été présenté, revenons sur celle de la fonction du listing 3.1. Chaque annotation du listing 3.1 est traduite en une séquence d'insertions de code. La première précondition à la ligne 1 (listing 3.1) est traduite à la ligne 6 du listing 4.1. La seconde précondition à la ligne 2 (listing 3.1) est traduite à la ligne 7 du listing 4.1. La clause `typically` de la ligne 3 du listing 3.1 est traduite ligne 8 du listing 4.1. L'invariant de boucle de la ligne 9 du listing 3.1 est traduit aux lignes 10–12 du listing 4.1 pour vérifier que l'invariant est vrai avant la première itération (établissement), et aux lignes 19–23 du listing 4.1 pour vérifier la préservation de l'invariant après chaque itération. Le variant de boucle ligne 10 du listing 3.1 est traduit aux lignes 15–16 du listing 4.1 pour vérifier que le variant est positif ou nul avant chaque itération de la boucle et aux lignes 24–25 du listing 4.1 pour vérifier que le variant décroît strictement après chaque itération, assurant ainsi la terminaison de la boucle. La postcondition des lignes 4–5 du listing 3.1 est traduite aux lignes 29–34 du listing 4.1.

4.1.2 ARITHMÉTIQUE NON BORNÉE

```
1 //@ assert x+1 <= INT_MAX; // fails with x = INT_MAX
2 // naive instrumentation: never fails in modular arithmetics
3 fassert(x+1 <= 2147483647);
4
5 // correct instrumentation with unbounded integers:
6 Z_t var_0; Z_init(var_0); Z_set(var_0, x); Z_t var_1; Z_init(var_1); Z_set(var_1, 1);
7 Z_t var_2; Z_init(var_2); Z_add(var_2, var_0, var_1); Z_clear(var_0); Z_clear(var_1);
8 Z_t var_3; Z_init(var_3); Z_set(var_3, 2147483647); int var_4 = Z_le(var_2, var_3);
9 Z_clear(var_2); Z_clear(var_3); fassert(var_4);
10
11 // correct instrumentation in abbreviated notation:
12 □var_0 = x; □var_1 = 1; □var_2 = var_0⊠ + var_1⊠; □var_3 = 2147483647;
13 int var_4 = var_2⊠ <= var_3⊠; fassert(var_4);
```

Listing 4.2 – Propriétés des entiers : faux négatif dû à une traduction naïve

La figure 4.4 présente la correspondance entre les deux notations utilisées pour les opérations sur les entiers non bornés dans la suite du document. Nous nous permettons d'abréger les opérations de création et initialisation (resp. de libération) d'une variable v de type $\mathbb{Z}$ effectuées avant (resp. après) une autre instruction en accollant $\square$ (resp. $\boxtimes$) à une occurrence de v dans cette instruction. Dans ce cas, les opérations de création et initialisation (resp. de libération) sont effectuées avant (resp. après) l'instruction complète. Par exemple, cette convention nous permet de réécrire $\square$ `x; x = y; y`$\boxtimes$`;` en $\square$ `x = y`$\boxtimes$`;`.

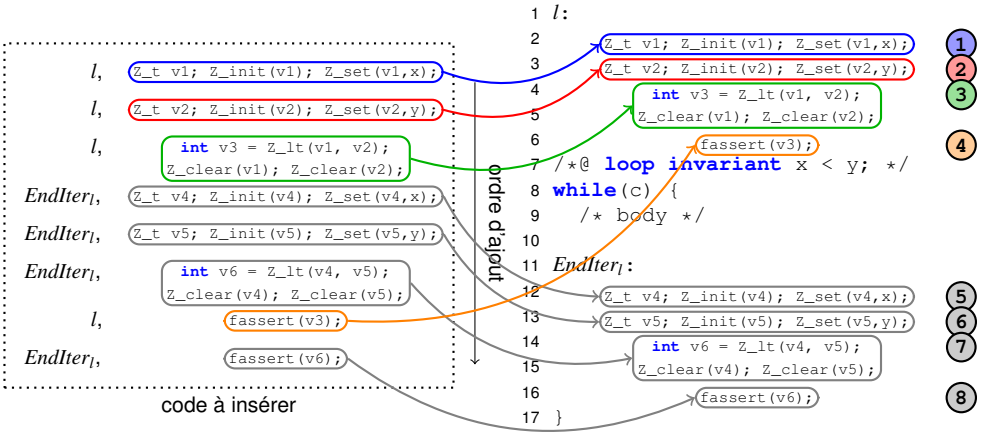

FIGURE 4.3 – Insertion du code généré pour l'invariant de la figure 4.2

Les listings 4.2 et 4.3 montrent deux exemples où une traduction naïve des annotations contenant des entiers mathématiques mène à un résultat incorrect. Soit `x` de type `int`. L'assertion ligne 1 du listing 4.2 est invalide lorsque `x = INT_MAX` car `INT_MAX+1` ne déborde pas en E-ACSL. La traduction naïve de cette assertion (ligne 3) utilise les entiers machine avec arithmétique modulaire (supposons une architecture 32 bits), donc `x+1` reste inférieur ou égal à `INT_MAX` pour toute valeur de `x`, rendant tout échec de l'assertion impossible. La traduction correcte pour cette annotation (lignes 6–9) conserve la sémantique de l'arithmétique entière non bornée en utilisant une bibliothèque externe (nous avons choisi GMP, la bibliothèque GNU Multi-Precision) pour représenter les valeurs pouvant provoquer un débordement d'entier (ici `INT_MAX+1`). Les variables nécessaires pour la gestion des entiers non bornés sont créées et initialisées, puis le résultat est calculé et les comparaisons sont faites en tant qu'entier non borné.

```
1 if( x > 0 ) {
2   /*@ assert x+1 > 0; */ // never fails in unbounded arithmetics
3   // naive instrumentation may fail in modular arithmetics
4   fassert(x+1 > 0);
5 }
```

Listing 4.3 – Propriétés des entiers : faux positif dû à une traduction naïve

Le second exemple (listing 4.3) définit une assertion E-ACSL (ligne 2) qui est toujours correcte : pour tout entier positif `x`, son successeur est aussi positif. Une traduction naïve de cette annotation (ligne 4) génèrera une erreur à l'exécution pour `x = INT_MAX` : en arithmétique modulaire `x+1` déborde et devient négatif, violant l'assertion ligne 2. Une traduction correcte utilisant les entiers non bornés (non présentée ici) conserve le comportement attendu : l'assertion est vraie pour tout entier positif `x`. D'une traduction naïve des annotations dans ces deux exemples résulterait un faux négatif dans le premier cas et un faux positif dans le deuxième cas, ce qui porterait atteinte à la correction et à la précision de notre méthode. Les règles de traduction pour les constructions E-ACSL que nous présentons respectent la sémantique des entiers non bornés d'E-ACSL et supposent l'utilisation

opération	notation abrégée	notation abstraite
création et initialisation	□ x;	Z_t x; Z_init(x);
affectation	x = y;	Z_set(x,y);
addition	x = y+z;	Z_add(x,y,z);
comparaison	x < y x <= y	Z_lt(x,y) Z_le(x,y)
libération	x⊠;	Z_clear(x);

FIGURE 4.4 – Exemples de notations pour l'arithmétique non bornée

d'une bibliothèque externe telle GMP, comme c'est illustré dans les listings 4.1 et 4.2.

4.2 TRADUCTION DES ANNOTATIONS E-ACSL

La traduction des annotations est définie dans cette partie par des règles. Cela requiert de traduire les termes et les prédicats E-ACSL. Leur traduction est décrite par les règles des parties 4.3 et 4.4. Dans ces règles, les notations suivantes s'ajoutent aux notations utilisées au chapitre précédent :

- c_1, c_2, ..., c_i, ..., c_n sont des fragments de programmes C ;
- l, l_1, l_2, ..., l_i, ..., l_n, Beg_f, End_f, $BegIter_l$ et $EndIter_l$ sont des labels du programme ;
- j et k sont des identificateurs de variables liées dans un prédicat E-ACSL et un compteur d'itérations dans un programme C ;
- x est un identificateur de variable C (donc une left-value) ;
- A_1 et A_2 sont des blocs d'instructions C ;
- I, I_1, I_2 et I_3 sont des listes d'insertions de code $(l_i, c_i)^*$;
- y et z sont des left-values quelconques ;
- P est un programme C annoté et P' est le programme instrumenté obtenu après traduction du programme P.

Les variables `res`$_f$, `var_`n, `var_`n_1, ..., `var_`n_m, `j_`n, `old_`x et `old_val_`x sont "fraîches", c'est-à-dire que ces identificateurs sont supposés différents de tous les autres identifiants de variables du programme instrumenté : quand on utilise plusieurs fois une règle introduisant une variable fraîche, toutes les occurrences de la variable créée doivent être différentes. Ceci peut être implémenté en incrémentant le compteur n à chaque occurrence dans `var_`n et `j_`n, et en utilisant le nom de la variable d'origine à la place de x dans `old_`x et `old_val_`x.

Définissons maintenant les règles de traduction des annotations, qui permettent de mettre en évidence les échecs d'annotation. Pour chaque annotation, un fragment de programme C se terminant par le test d'une expression par la fonction `fassert` est inséré dans le programme. Le générateur de tests essaiera de couvrir tous les chemins d'exécution faisables, donc s'il existe des entrées permettant de rendre fausse l'expression de `fassert`, une session complète de génération de tests permettra de trouver une erreur et de donner un cas de test la produisant.

La fonction de traduction des annotations, notée α, est définie comme une fonction partielle qui, à un label l et une annotation a, associe une séquence d'insertions de code $(l_i, c_i)^*$:

$$(l, a) \overset{\alpha}{\mapsto} \underbrace{(l_1, c_1) \cdot (l_2, c_2) \cdot \ \ldots \cdot (l_n, c_n)}_{I\text{: liste d'insertions de code}}$$

Cette fonction est définie dans les figures 4.5 et 4.6. Ces figures présentent les règles permettant de traduire les annotations du langage E-ACSL considéré.

$$\alpha\text{-ASSERT} \quad \frac{(l, p) \overset{\pi}{\mapsto} (I, e)}{(l, \texttt{assert}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (l, \texttt{fassert}\,(e)\,\texttt{;})}$$

$$\alpha\text{-CHECK-PRE} \quad \frac{(Beg_f, p) \overset{\pi}{\mapsto} (I, e)}{(Beg_f, \texttt{requires}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (Beg_f, \texttt{fassert}\,(e)\,\texttt{;})}$$

$$\alpha\text{-ASSUME-PRE} \quad \frac{(Beg_f, p) \overset{\pi}{\mapsto} (I, e)}{(Beg_f, \begin{cases}\texttt{typically} \\ \texttt{requires}\end{cases}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (Beg_f, \texttt{fassume}\,(e)\,\texttt{;})}$$

$$\alpha\text{-CHECK-ASSIGNS} \quad \frac{(Beg_f, x_1) \overset{\tau}{\mapsto} (I_1^1, e_1^1) \quad (End_f, x_1) \overset{\tau}{\mapsto} (I_1^2, e_1^2) \quad \ldots \quad (Beg_f, x_m) \overset{\tau}{\mapsto} (I_m^1, e_m^1) \quad (End_f, x_m) \overset{\tau}{\mapsto} (I_m^2, e_m^2)}{\begin{array}{l}(End_f, \texttt{assigns}\ X\texttt{;}) \overset{\alpha}{\mapsto} \\ \quad I_1^1 \cdot I_1^2 \cdot (Beg_f, ctype\ \texttt{var\_}n_1 = e_1^1\texttt{;}) \cdot (End_f, \texttt{fassert}\,(e_1^2 \texttt{ == var\_}n_1)\,\texttt{;}) \cdot \ldots \\ \quad\quad \cdot I_m^1 \cdot I_m^2 \cdot (Beg_f, ctype\ \texttt{var\_}n_m = e_m^1\texttt{;}) \cdot (End_f, \texttt{fassert}\,(e_m^2 \texttt{ == var\_}n_m)\,\texttt{;})\end{array}}$$

$$\alpha\text{-CHECK-POST} \quad \frac{(End_f, p) \overset{\pi}{\mapsto} (I, e)}{(End_f, \texttt{ensures}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (End_f, \texttt{fassert}\,(e)\,\texttt{;})}$$

FIGURE 4.5 – Règles de traduction pour les assertions, pré-/postconditions et assigns

La figure 4.5 décrit les règles de traduction pour les assertions (α-ASSERT), les pré-conditions (α-CHECK-PRE et α-ASSUME-PRE), les clauses **assigns** (α-CHECK-ASSIGNS) et les postconditions (α-CHECK-POST). Ces règles utilisent la traduction des termes notée $(l, t) \overset{\tau}{\mapsto} (I, e)$ et la traduction des prédicats notée $(l, p) \overset{\pi}{\mapsto} (I, e)$, respectivement définies dans les parties 4.3 et 4.4, où I est la liste d'insertions de code traduisant p ou t et e est une expression qui s'évalue en la valeur de p ou t.

La règle α-ASSERT vérifie simplement avec la fonction `fassert` la valeur e de la traduction du prédicat p au label l. La règle α-CHECK-PRE vérifie avec la fonction `fassert` la valeur e de la traduction du prédicat p au début d'une fonction appelée. La règle α-ASSUME-PRE traite la précondition de la fonction sous test, qui est supposée vraie par définition. Cette même règle permet de traduire les clauses **typically** (précondition pour le test qui permet de limiter l'explosion combinatoire du nombre de chemins du programme) et les clauses **requires**, elle suppose avec la fonction `fassume` que la valeur e de la traduction du prédicat p au début de la fonction est vraie. La règle α-CHECK-ASSIGNS vérifie que les left-values qui ne sont pas dans une clause **assigns** ne sont pas modifiées par la fonction. L'ensemble des left-values considérées est la différence $G - X = \{x_1, ..., x_m\}$, où X est l'ensemble des left-values de la clause **assigns** et G est l'ensemble des left-values dans le contexte courant qui ne sont en alias avec aucune left-value de X[1]. Considérons l'exemple du listing 4.4 où une variable `x` a pour alias `*p`. La clause **assigns** ne nécessitant pas de citer tous les alias, **assigns** `x` suffit, et il est inutile de vérifier que la valeur de `*p` n'a pas été modifiée par la fonction.

```
1 int x = 1;
```

1. Nous supposons l'absence des alias des éléments de X dans G pour simplifier la présentation des règles et des exemples. En pratique, il est immédiat de vérifier à l'exécution si une left-value donnée est un alias d'un élément de X en comparant les adresses.

```
2 int *p = &x;
3
4 /*@ assigns x; */
5 void f() {
6   *p = 2;
7 }
```

Listing 4.4 – Alias de `x` et `*p`

La règle α-CHECK-POST vérifie la validité de la postcondition à la fin de la fonction instrumentée.

$$\alpha\text{-CHECK-LOOP-ASSIGNS}\quad \frac{(BegIter_l, x_1) \overset{\tau}{\mapsto} (I_1^1, e_1^1) \quad (EndIter_l, x_1) \overset{\tau}{\mapsto} (I_1^2, e_1^2) \quad \dots \quad (BegIter_l, x_m) \overset{\tau}{\mapsto} (I_m^1, e_m^1) \quad (EndIter_l, x_m) \overset{\tau}{\mapsto} (I_m^2, e_m^2)}{\begin{array}{l}(l, \texttt{loop assigns } X\texttt{;}) \overset{\alpha}{\mapsto} \\ \quad I_1^1 \cdot I_1^2 \cdot (BegIter_l, ctype\ \texttt{var\_}n_1 = e_1^1\texttt{;}) \cdot (EndIter_l, \texttt{fassert}\,(e_1^2 \texttt{ == var\_}n_1)\texttt{;}) \cdot \dots \\ \quad\quad \cdot I_m^1 \cdot I_m^2 \cdot (BegIter_l, ctype\ \texttt{var\_}n_m = e_m^1\texttt{;}) \cdot (EndIter_l, \texttt{fassert}\,(e_m^2 \texttt{ == var\_}n_m)\texttt{;})\end{array}}$$

$$\alpha\text{-CHECK-INVARIANT}\quad \frac{(l, p) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (EndIter_l, p) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, \texttt{loop invariant } p\texttt{;}) \overset{\alpha}{\mapsto} I_1 \cdot (l, \texttt{fassert}\,(e_1)\texttt{;}) \cdot I_2 \cdot (EndIter_l, \texttt{fassert}\,(e_2)\texttt{;})}$$

$$\alpha\text{-VARIANT}\quad \frac{(BegIter_l, t) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (EndIter_l, t) \overset{\tau}{\mapsto} (I_2, e_2)}{\begin{array}{l}(l, \texttt{loop variant } t\texttt{;}) \overset{\alpha}{\mapsto} \\ \quad I_1 \cdot (BegIter_l, \texttt{fassert}\,(\underline{\texttt{0 <= }e_1})\texttt{;}) \cdot I_2 \cdot (EndIter_l, \texttt{fassert}\,(\underline{e_2{}^{\boxtimes} < e_1{}^{\boxtimes}})\texttt{;})\end{array}}$$

FIGURE 4.6 – Règles de traduction pour les annotations de boucle : invariants, variant et assigns

La figure 4.6 présente les règles de traduction pour les invariants de boucle (α-CHECK-INVARIANT), les variants de boucle (α-VARIANT) et les loop assigns (α-CHECK-LOOP-ASSIGNS) pour une boucle au label l. La règle α-CHECK-LOOP-ASSIGNS vérifie que les left-values qui ne sont pas dans la clause `loop assigns` ne sont pas modifiées par la boucle. Pour chacune de ces left-values de l'ensemble $G - X = \{x_1, ..., x_m\}$, où X est l'ensemble des left-values de la clause `loop assigns` et G est l'ensemble des left-values dans le contexte courant qui ne sont en alias avec aucune left-value de X, une sauvegarde de leur valeur courante est faite au début de chaque itération, puis on vérifie que leur valeur n'a pas été modifiée en fin de boucle. La règle α-CHECK-INVARIANT vérifie le prédicat de l'invariant avant la boucle et après chaque itération de la boucle. La règle α-VARIANT vérifie que le terme variant est non-négatif au début de chaque itération. Enfin, on vérifie à la fin de chaque itération que le variant décroît strictement par rapport à sa valeur au début de l'itération.

4.3 TRADUCTION DES TERMES E-ACSL

La fonction de traduction des termes, notée τ, est définie comme une fonction partielle qui, à un label l et un terme t, associe une séquence d'insertions de code $(l_i, c_i)^*$ et une expression C dénotée e qui s'évalue en la valeur du terme t au label l :

$$(l, t) \overset{\tau}{\mapsto} (\underbrace{(l_1, c_1) \cdot (l_2, c_2) \cdot \ \dots \cdot (l_n, c_n)}_{I\text{: liste d'insertions de code}}, e)$$

Le fragment de code c_i sera à insérer au label l_i. Cette fonction est définie dans les figures 4.7, 4.8, 4.9 et 4.10. Ces figures présentent les règles permettant de traduire les

termes du langage E-ACSL considéré. Le type du terme peut être $\mathbb{Z}$ (`integer`), un type pointeur quelconque (`int*`, `char*`, etc.) noté *ptr* ou un type scalaire du C (`char`, `int`, `long`, etc.) noté *ctype*. *ctype* correspond à l'entité syntaxique *basic-type* et *ptr* correspond à l'entité syntaxique *basic-type*$_*$ de la figure 3.1 du chapitre précédent. Les insertions de code sont les instructions C nécessaires à l'évaluation du terme. Elles permettent notamment d'évaluer les sous-termes dont est composé le terme t. L'expression e est pure (elle ne provoque pas d'effet de bord) et représente le résultat de l'évaluation du terme t au point de programme l. L'évaluation de e requiert le plus souvent plusieurs opérations, qui sont effectuées par les fragments de programme qui résultent de la traduction du terme et seront insérés aux endroits adéquats dans le programme instrumenté. Par exemple, un terme quantifié (comme une somme : `\sum`) a besoin de plusieurs instructions pour être évalué, et notamment d'une boucle (voir la figure 4.10). Dans ce cas, le second élément e de la paire retournée est la variable calculée par la boucle. Quand un terme t peut être traduit directement sans nécessiter de code C supplémentaire, la séquence d'insertions de code est vide et notée ε. Pour un terme $t : \mathbb{Z}$, le résultat de traduction e est toujours une variable de type entier GMP (noté `z_t` dans le listing 4.2).

$$\tau\text{-LV-SIMPLE}\ \frac{}{(l, x) \overset{\tau}{\mapsto} (\varepsilon, x)} \qquad \tau\text{-LV-ARRAY}\ \frac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I, e)}{(l, x\texttt{[}t\texttt{]} : ctype) \overset{\tau}{\mapsto} (I \cdot (l, ctype\ \texttt{var\_}n = x\texttt{[}e\texttt{];}), \texttt{var\_}n)}$$

$$\tau\text{-OLD-SIMPLE}\ \frac{}{(l, \texttt{\textbackslash old}\,(x)) \overset{\tau}{\mapsto} (\varepsilon, \texttt{old\_}x)}$$

$$\tau\text{-OLD-ARRAY}\ \frac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash old}\,(x\texttt{[}t\texttt{]}) : ctype) \overset{\tau}{\mapsto} (I \cdot (l, ctype\ \texttt{var\_}n = \texttt{ old\_val\_}x\texttt{[}e\texttt{];}), \texttt{var\_}n)}$$

$$\tau\text{-RES}\ \frac{}{(l, \texttt{\textbackslash result}) \overset{\tau}{\mapsto} (\varepsilon, \texttt{res}_f)} \qquad \tau\text{-CONST}\ \frac{}{(l, cst : \mathbb{Z}) \overset{\tau}{\mapsto} ((l, \underline{\square\texttt{var\_}n = cst\texttt{;}}), \texttt{var\_}n)}$$

FIGURE 4.7 – Règles de traduction pour les termes simples

La figure 4.7 présente les règles de traduction pour les left-values (τ-LV-SIMPLE pour les identificateurs de variable et τ-LV-ARRAY pour les tableaux), pour le terme `\result` (τ-RES) et pour les constantes (τ-CONST). Deux règles sont nécessaires pour traduire la construction E-ACSL `\old`. Appliquée à un identificateur x (règle τ-OLD-SIMPLE), le terme est traduit en variable `old_`x qui contient la valeur de x au début de la fonction. Appliquée à un accès mémoire (règle τ-OLD-ARRAY), on utilise le tableau alloué dynamiquement `old_val_`x qui contient les valeurs du tableau ou pointeur x au début de la fonction (se référer à la partie 4.1.1). Le terme `\result` est traduit par la règle τ-RES, il fait référence à la valeur de retour de la fonction en E-ACSL qui, par la normalisation, est désignée par la variable `res`$_f$ où f est le nom de la fonction courante. Cette variable est unique dans chaque fonction après normalisation de l'arbre de syntaxe abstraite par FRAMA-C. Notons que l'accès au nom de la fonction courante n'est pas formalisé pour simplifier les règles. La règle τ-CONST établit qu'une variable entière GMP est définie pour prendre la valeur d'une contante entière non bornée.

La figure 4.8 détaille les règles de conversions implicites, de `integer` vers le type C `int` et du type C `int` vers `integer`. Ces règles peuvent être adaptées pour les autres types entiers du C (`char`, `long`, etc.). Ces conversions étant implicites, nous n'avons jamais à construire un terme de la forme $(\mathbb{Z})t$ ou $(\texttt{int})t$ et ces constructions n'apparaissent donc pas dans la syntaxe des termes définie au chapitre précédent. Par exemple, la règle τ-COERCE$_1$ s'applique dans la règle τ-LV-ARRAY puisque le terme t (de type $\mathbb{Z}$) doit être traduit en

$$\tau\text{-COERCE}_1 \quad \frac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I, e)}{(l, (\texttt{int})(t : \mathbb{Z})) \overset{\tau}{\mapsto} (I \cdot (l, \texttt{int var\_n} = \underline{e^{\boxtimes}\texttt{;}}), \texttt{var\_n})}$$

$$\tau\text{-COERCE}_2 \quad \frac{(l, t : \texttt{int}) \overset{\tau}{\mapsto} (I, e)}{(l, (\mathbb{Z})(t : \texttt{int})) \overset{\tau}{\mapsto} (I \cdot (l, \underline{^{\square}\texttt{var\_n} = e\texttt{;}}), \texttt{var\_n})}$$

FIGURE 4.8 – Règles de traduction pour les conversions

une expression e (de type $ctype$). Nous nous limitons aux conversions entre entiers et nous supposons que ces conversions ne provoquent pas de débordement d'entiers, ainsi l'application d'une conversion de type ne change pas la sémantique du terme.

$$\tau\text{-UNOP} \quad \frac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I, e)}{(l, (unop\ t) : \mathbb{Z}) \overset{\tau}{\mapsto} (I \cdot (l, \underline{^{\square}\texttt{var\_n} = unop\ e^{\boxtimes}\texttt{;}}), \texttt{var\_n})} \quad unop \in \{-, \sim\}$$

$$\tau\text{-BINOP}_1 \quad \frac{(l, t_1 : ptr) \overset{\tau}{\mapsto} (I_1, e_1) \quad (l, t_2 : \texttt{int}) \overset{\tau}{\mapsto} (I_2, e_2)}{(l, (t_1\ binop\ t_2) : ptr) \overset{\tau}{\mapsto} (I_1 \cdot I_2 \cdot (l, ptr\ \texttt{var\_n} = e_1\ binop\ e_2\texttt{;}), \texttt{var\_n})} \quad binop \in \{+, -\}$$

$$\tau\text{-BINOP}_2 \quad \frac{(l, t_1 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \quad (l, t_2 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_2, e_2)}{(l, (t_1\ binop\ t_2) : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1 \cdot I_2 \cdot (l, \underline{^{\square}\texttt{var\_n} = e_1^{\boxtimes}\ binop\ e_2^{\boxtimes}\texttt{;}}), \texttt{var\_n})} \quad binop \in \{+, -, *, /, \%\}$$

$$\tau\text{-ABS} \quad \frac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I, \mathrm{e})}{(l, \texttt{\textbackslash abs}\ (t)\ : \mathbb{Z}) \overset{\tau}{\mapsto} (I \cdot (l, \underline{^{\square}\texttt{var\_n} = |e^{\boxtimes}|\texttt{;}}), \texttt{var\_n})}$$

$$\tau\text{-IF} \quad \frac{(l, t_1 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (l, t_2 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_2, e_2) \qquad (l, t_3 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_3, e_3)}{(l, (t_1\ \texttt{?}\ t_2\ \texttt{:}\ t_3) : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1 \cdot (l, \underline{^{\square}\texttt{var\_n}}\texttt{;}\ \texttt{if}\ (\underline{e_1^{\boxtimes}\ \texttt{!=}\ 0})\ \{I_2 \cdot (l, \underline{\texttt{var\_n} = e_2^{\boxtimes}\texttt{;}})\}\texttt{else}\{I_3 \cdot (l, \underline{\texttt{var\_n} = e_3^{\boxtimes}\texttt{;}})\}), \texttt{var\_n})}$$

FIGURE 4.9 – Règles de traduction pour les opérations unaires et binaires

La figure 4.9 détaille les règles pour les opérations unaires (τ-UNOP), les opérations binaires (τ-BINOP$_{\{1,2\}}$), la valeur absolue (τ-ABS) et la condition ternaire sur les termes (τ-IF). τ-UNOP traite le cas du moins unaire et du complément bit-à-bit ($\sim$) pour lesquels on suppose que ces opérations prennent un `integer` en entrée et en sortie.

τ-BINOP$_1$ traite l'arithmétique de pointeur, l'opérande gauche est un pointeur, $binop$ doit être `+` ou `-`, et l'opérande droit est de type `int`. τ-BINOP$_2$ traite les opérations arithmétiques entières (les opérandes et le résultat sont des entiers), $binop$ est un des opérateurs arithmétiques suivants : `+`, `-`, `*`, `/`, `%`. Dans la règle τ-IF, les évaluations de t_2 et t_3 sont dans des branches conditionnelles : on calcule paresseusement l'un des deux, en fonction de l'évaluation de t_1. La notation suivante :

$$(l, \underline{^{\square}\texttt{var\_n}}\texttt{;}\ \texttt{if}\ (\underline{e_1^{\boxtimes}\ \texttt{!=}\ 0})\ \{I_2 \cdot (l, \underline{\texttt{var\_n} = e_2^{\boxtimes}\texttt{;}})\}\texttt{else}\{I_3 \cdot (l, \underline{\texttt{var\_n} = e_3^{\boxtimes}\texttt{;}})\})$$

signifie que les blocs du `if` et du `else` contiendront, après l'étape d'insertion (voir partie 4.1.1), les instructions à insérer au label l obtenues à partir des fragments de code de I_2 et I_3 respectivement. Dans la suite cette notation sera utilisée pour chaque conditionnelle `if` et chaque boucle `while` ou `for` générée par les règles de traduction.

La figure 4.10 présente les règles de traduction pour les fonctions logiques `\sum` (τ-SUM) et `\numof` (τ-NUMOF). Le terme `\sum`$(t_1, t_2,$ `\lambda integer` $k; t_3)$ calcule la somme généralisée $\sum_{k=t_1}^{t_2} t_3$. La règle τ-SUM permet de calculer cette somme. Elle initialise une variable entière fraîche `var_n` à 0 et l'incrémente avec la valeur du `\lambda`-terme t_3 à chaque itération. Le terme `\product`$(t_1, t_2,$ `\lambda integer` $k\ ; t_3)$ calcule le produit généralisé $\Pi_{k=t_1}^{t_2} t_3$. La règle

$$\tau\text{-SUM}\quad \frac{(l,t_1:\mathbb{Z})\overset{\tau}{\mapsto}(I_1,e_1)\qquad (l,t_2:\mathbb{Z})\overset{\tau}{\mapsto}(I_2,e_2)\qquad (l,t_3:\mathbb{Z})\overset{\tau}{\mapsto}(I_3,e_3)}{\begin{array}{l}(l,(\texttt{\textbackslash sum}\,(t_1,t_2,\texttt{\textbackslash lambda integer}\ j;\ t_3)):\mathbb{Z})\overset{\tau}{\mapsto}\\ \quad (I_1\cdot I_2\cdot(l,\overline{\overline{\texttt{var\_n = 0;}}}^{\square})\\ \qquad\cdot(l,\texttt{for}\,(\overline{\overline{\texttt{j = e1;}}}^{\boxtimes}\ \overline{\texttt{j <= e2;}}\ \overline{\texttt{j++}})\ \{I_3\cdot(l,\overline{\texttt{var\_n += e3;}}^{\boxtimes})\}\ \overline{\overline{\texttt{e2;}}}^{\boxtimes}\ \overline{\overline{\texttt{j;}}}^{\boxtimes}),\texttt{var\_n})\end{array}}$$

$$\tau\text{-NUMOF}\quad \frac{(l,t_1:\mathbb{Z})\overset{\tau}{\mapsto}(I_1,e_1)\qquad (l,t_2:\mathbb{Z})\overset{\tau}{\mapsto}(I_2,e_2)\qquad (l,t_3:\texttt{int})\overset{\tau}{\mapsto}(I_3,e_3)}{\begin{array}{l}(l,(\texttt{\textbackslash numof}\,(t_1,t_2,\texttt{\textbackslash lambda integer}\ j;\ t_3)):\mathbb{Z})\overset{\tau}{\mapsto}\\ \quad (I_1\cdot I_2\cdot(l,\overline{\overline{\texttt{var\_n = 0;}}}^{\square})\\ \qquad\cdot(l,\texttt{for}\,(\overline{\overline{\texttt{j = e1;}}}^{\boxtimes}\ \overline{\texttt{j <= e2;}}\ \overline{\texttt{j++}})\ \{I_3\cdot(l,\texttt{if}\,(e_3)\ \{(l,\overline{\texttt{var\_n++;}})\})\}\ \overline{\overline{\texttt{e2;}}}^{\boxtimes}\ \overline{\overline{\texttt{j;}}}^{\boxtimes}),\texttt{var\_n})\end{array}}$$

FIGURE 4.10 – Règles de traduction pour les fonctions logiques `sum` et `numof`

permettant de calculer ce produit n'est pas donnée mais est similaire à la règle τ-SUM et peut être facilement dérivée à partir de celle-ci. Le terme `\numof`$(t_1, t_2,$ `\lambda integer` k $;t_3)$ désigne le nombre d'entiers k tel que $t_1 \leq k \leq t_2$ et $t_3 \neq 0$. La règle τ-NUMOF permet de traduire ce terme. Elle initialise une variable entière fraîche `var_`n à 0 et incrémente sa valeur quand le `\lambda`-terme t_3 s'évalue en une expression non nulle.

4.4 TRADUCTION DES PRÉDICATS E-ACSL

De manière similaire à la fonction de traduction τ pour les termes (partie 4.3), la fonction de traduction des prédicats, notée π, est définie comme une fonction partielle qui, à un label et un prédicat E-ACSL p, associe une séquence d'insertions de code $(l_i, c_i)^*$ et une expression $e \in \{0, 1\}$ du langage C (que nous encoderons dans un `int`) :

$$(l,p)\overset{\pi}{\mapsto}\underbrace{((l_1,c_1)\cdot(l_2,c_2)\cdot\ \ldots\cdot(l_n,c_n)}_{I\text{: liste d'insertions de code}},e)$$

Les insertions de code sont les instructions C nécessaires à l'évaluation du prédicat. Elles permettent notamment d'évaluer les termes et prédicats auxiliaires dont est composé le prédicat p. Quand un prédicat p peut être traduit directement et ne nécessite pas la création d'instructions C, la séquence d'insertions est vide et notée ε.

Nous définissons dans les figures 4.11, 4.12 et 4.13 les règles de traduction pour les principaux prédicats E-ACSL que nous traitons. Pour chacune de ces règles, une variable fraîche `var_`n est générée (même quand la traduction du prédicat est triviale) afin de standardiser les règles et ainsi de faciliter la justification de leur correction.

La figure 4.11 détaille les règles de traduction pour les prédicats les plus simples du langage E-ACSL. Les règles π-TRUE et π-FALSE indiquent que `\true` (respectivement `\false`) se traduit en 1 (respectivement 0). Les autres règles sont compositionnelles : les prédicats fils sont traduits récursivement, puis le résultat de la traduction est construit à partir des valeurs des prédicats fils. La règle π-EQUIV utilise l'équivalence suivante $p_1 \Leftrightarrow p_2 \equiv p_1 \Rightarrow p_2 \wedge p_2 \Rightarrow p_1$ sous la forme $(\neg p_1 \vee p_2) \wedge (\neg p_2 \vee p_1)$. Les règles π-AND, π-OR et π-IMPL reflètent l'évaluation paresseuse de ces opérateurs selon la sémantique d'E-ACSL : le premier opérande est toujours évalué (en I_1) mais le deuxième n'est évalué (en I_2) que si nécessaire. La règle π-IF est similaire à τ-IF (voir partie 4.3). Dans l'exemple du listing 3.1, la traduction de la postcondition `\result != 0 <==> \exists integer j; 0<=j<n && t[j]==v` requiert la traduction des prédicats `\result != 0` et `\exists(...)` qui sont respectivement traduits en `var_32` et `var_35`. Ces deux prédicats sont assemblés pour construire

π-TRUE $\quad \dfrac{}{(l, \texttt{\textbackslash true}) \overset{\pi}{\mapsto} ((l, \texttt{int var\_}n\texttt{ = 1;}), \texttt{var\_}n)}$

π-FALSE $\quad \dfrac{}{(l, \texttt{\textbackslash false}) \overset{\pi}{\mapsto} ((l, \texttt{int var\_}n\texttt{ = 0;}), \texttt{var\_}n)}$

π-NOT $\quad \dfrac{(l, p) \overset{\pi}{\mapsto} (I, e)}{(l, \texttt{!}p) \overset{\pi}{\mapsto} (I \cdot (l, \texttt{int var\_}n\texttt{ = !}e\texttt{;}), \texttt{var\_}n)}$

π-AND $\quad \dfrac{(l, p_1) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, p_1 \texttt{\&\&} p_2) \overset{\pi}{\mapsto} (I_1 \cdot (l, \texttt{int var\_}n = e_1\texttt{;}) \cdot (l, \texttt{if (var\_}n\texttt{)} \{I_2 \cdot (l, \texttt{var\_}n = e_2\texttt{;})\}), \texttt{var\_}n)}$

π-OR $\quad \dfrac{(l, p_1) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, p_1 \texttt{ || } p_2) \overset{\pi}{\mapsto} (I_1 \cdot (l, \texttt{int var\_}n = e_1\texttt{;}) \cdot (l, \texttt{if (!var\_}n\texttt{)} \{I_2 \cdot (l, \texttt{var\_}n = e_2\texttt{;})\}), \texttt{var\_}n)}$

π-IMPL $\quad \dfrac{(l, p_1) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, p_1 \texttt{ ==> } p_2) \overset{\pi}{\mapsto} (I_1 \cdot (l, \texttt{int var\_}n\texttt{ = 1;}) \cdot (l, \texttt{if (}e_1\texttt{)} \{I_2 \cdot (l, \texttt{var\_}n = e_2\texttt{;})\}), \texttt{var\_}n)}$

π-EQUIV $\quad \dfrac{(l, p_1) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, p_1 \texttt{ <==> } p_2) \overset{\pi}{\mapsto} (I_1 \cdot I_2 \cdot (l, \texttt{int var\_}n\texttt{ = (!}e_1\texttt{ || }e_2\texttt{) \&\& (!}e_2\texttt{ || }e_1\texttt{);}), \texttt{var\_}n)}$

π-IF $\quad \dfrac{(l, t : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2) \qquad (l, p_3) \overset{\pi}{\mapsto} (I_3, e_3)}{(l, t \texttt{ ? } p_2 \texttt{ : } p_3) \overset{\pi}{\mapsto} (I_1 \cdot (l, \texttt{int var\_}n\texttt{;}) \cdot (l, \texttt{if (}\underline{e_1^{\boxtimes}\texttt{ != 0}}\texttt{)} \{I_2 \cdot (l, \texttt{var\_}n = e_2\texttt{;})\}\texttt{else}\{I_3 \cdot (l, \texttt{var\_}n = e_3\texttt{;})\}), \texttt{var\_}n)}$

π-REL $\quad \dfrac{(l, t_1 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \quad (l, t_2 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_2, e_2)}{(l, t_1 \ rel \ t_2) \overset{\pi}{\mapsto} (I_1 \cdot I_2 \cdot (l, \texttt{int var\_}n = \underline{e_1^{\boxtimes} \ rel \ e_2^{\boxtimes}}\texttt{;}), \texttt{var\_}n)} \quad rel \in \{<, <=, >, >=, ==, !=\}$

FIGURE 4.11 – Règles de traduction pour les prédicats simples

la traduction du prédicat composé, d'après la règle π-EQUIV, : `(!var_32 || var_35)&& (!var_35 || var_32)` (line 33 du listing 4.1).

π-VALID $\quad \dfrac{}{(l, \texttt{\textbackslash valid}(t)) \overset{\pi}{\mapsto} ((l, \texttt{int var\_}n\texttt{ = fvalid}(t)\texttt{;}), \texttt{var\_}n)}$

π-VALID-RANGE $\quad \dfrac{}{(l, \texttt{\textbackslash valid}(t_1\texttt{+}(t_2\texttt{..}t_3))) \overset{\pi}{\mapsto} ((l, \texttt{int var\_}n\texttt{ = fvalidr}(t_1, t_2, t_3)\texttt{;}), \texttt{var\_}n)}$

FIGURE 4.12 – Règles de traduction pour les prédicats de validité mémoire

La figure 4.12 montre les règles de traduction pour le prédicat `\valid`, qui est vrai si le pointeur en argument pointe vers une case mémoire valide et peut être déréférencé. `\valid(t)` exprime la validité d'un seul pointeur et `\valid(t1+(t2..t3))` exprime la validité du pointeur `t1` dans l'intervalle d'offsets `t2..t3`, ce qui veut dire que `(t1+t2)`, `(t1+t2+1)` ..., `(t1+t3)` sont valides. Les règles π-VALID et π-VALID-RANGE couvrent les deux cas. Le support de ces prédicats nécessite une représentation précise de la mémoire du programme, la réalisation de telles fonctions est abordée dans le chapitre 5. Dans notre exemple du listing 3.1, le prédicat `\valid(t+(0..n-1))` de la clause `requires` de la ligne 2 est traduit en `fvalidr(t,0,(n-1))` (ligne 7 du listing 4.1).

Dans la figure 4.13, la règle π-FORALL traite de la traduction du prédicat universel `\forall`, quantifié sur les entiers. Le code généré met à jour une variable `var_`n (initialisée à *true*) à chaque itération de boucle, jusqu'à ce que toutes les valeurs de k soient considérées ou que `var_`n soit évaluée à $false$. La règle π-EXISTS (figure 4.13) aborde la traduction du prédicat existentiel `\exists`, quantifié sur les entiers. Le code généré met à jour une variable `var_`n (initialement $false$) à chaque itération de la boucle, jusqu'à ce que toutes les valeurs de k soient considérées ou que `var_`n soit évaluée à *true*. Dans le listing 3.1,

$$\pi\text{-FORALL}\quad \frac{(l, t_1 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (l, t_2 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_2, e_2) \qquad (l, p) \overset{\pi}{\mapsto} (I_3, e_3)}{\begin{array}{l}(l, \texttt{\textbackslash forall integer } k\texttt{;}\ t_1 \texttt{ <= } k \texttt{ <= } t_2 \texttt{ ==> } p) \overset{\pi}{\mapsto} \\ \quad (I_1 \cdot I_2 \cdot (l, \texttt{int var\_}n \texttt{ = 1;} \overset{\square}{\underline{k = e_1\texttt{;}}}) \\ \qquad \cdot (l, \texttt{while (}\underline{k \texttt{ <= } e_2}\texttt{ \&\& var\_}n\texttt{)}\ \{I_3 \cdot (l, \texttt{var\_}n = e_3\texttt{;}\ \underline{k\texttt{++;}})\}) \cdot (l, \underline{k}^{\boxtimes}\texttt{;}\ \underline{e_1}^{\boxtimes}\texttt{;}\ \underline{e_2}^{\boxtimes}\texttt{;}), \texttt{var\_}n)\end{array}}$$

$$\pi\text{-EXISTS}\quad \frac{(l, t_1 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (l, t_2 : \mathbb{Z}) \overset{\tau}{\mapsto} (I_2, e_2) \qquad (l, p) \overset{\pi}{\mapsto} (I_3, e_3)}{\begin{array}{l}(l, \texttt{\textbackslash exists integer } k\texttt{;}\ t_1 \texttt{ <= } k \texttt{ <= } t_2 \texttt{ \&\& } p) \overset{\pi}{\mapsto} \\ \quad (I_1 \cdot I_2 \cdot (l, \texttt{int var\_}n \texttt{ = 0;} \overset{\square}{\underline{k = e_1\texttt{;}}}) \\ \qquad \cdot (l, \texttt{while (}\underline{k \texttt{ <= } e_2}\texttt{ \&\& !var\_}n\texttt{)}\ \{I_3 \cdot (l, \texttt{var\_}n = e_3\texttt{;}\ \underline{k\texttt{++;}})\}) \cdot (l, \underline{k}^{\boxtimes}\texttt{;}\ \underline{e_1}^{\boxtimes}\texttt{;}\ \underline{e_2}^{\boxtimes}\texttt{;}), \texttt{var\_}n)\end{array}}$$

FIGURE 4.13 – Règles de traduction pour les prédicats quantifiés

le prédicat `\exists integer i; 0<=i<n && t[i]==v` de la ligne 4 est traduit aux lignes 29–33 du listing 4.1 et la variable fraîche contenant la valeur du prédicat après évaluation est `var_35` (déclarée à la ligne 31).

4.5 SIMILARITÉS ET DIFFÉRENCES SELON L'OBJECTIF, TEST VS. VALIDATION À L'EXÉCUTION

La traduction d'annotations E-ACSL en code C a été implémentée dans deux greffons FRAMA-C : E-ACSL2C [Delahaye et al., 2013, Kosmatov et al., 2013] et STADY [Petiot et al., 2014b, Petiot et al., 2014a]. Le premier génère un programme instrumenté pour la validation d'assertions à l'exécution, le second génère un programme instrumenté pour la génération de tests avec PATHCRAWLER [Botella et al., 2009]. Cette partie discute les similarités et les différences entre ces deux façons d'instrumenter.

La traduction pour la validation à l'exécution et la traduction pour la génération de tests ont toutes deux besoin de générer du code exécutable, c'est pourquoi un sous-ensemble exécutable du langage de spécification est considéré dans les deux cas. La plupart des règles définies dans les parties 4.2, 4.3 et 4.4 sont autant valables pour la validation à l'exécution que pour le test, le code généré devant être correct vis-à-vis de la sémantique des annotations. Cependant, il y a plusieurs différences liées à la différence d'objectif. Les trois différences sont :
– la manière de considérer la précondition de la fonction sous test ;
– la manière de traiter les annotations liées au modèle mémoire ;
– et la manière de traiter les fonctions sur les entiers mathématiques.

Précondition de la fonction sous test. Une première différence est le traitement de la précondition de la fonction sous test. En vérification à l'exécution, elle est vérifiée comme n'importe quelle autre annotation. En génération de tests, la précondition est utilisée pour éviter de tester le programme avec des valeurs d'entrées incorrectes, pour lesquelles le bon fonctionnement du programme n'est pas garanti. La précondition de la fonction sous test est donc supposée vraie pour la génération de tests, afin de s'assurer que toutes les entrées générées par le test satisfont cette précondition.

Annotations liées au modèle mémoire. La validation à l'exécution nécessite une instrumentation complexe pour traiter les constructions E-ACSL liées au modèle mémoire comme `\valid`, `\block_length`, `\base_addr`, etc. Chaque opération affectant la mémoire doit être instrumentée et les informations de chaque case mémoire doivent être enregistrées afin de pouvoir évaluer ces annotations [Kosmatov et al., 2013]. Nous aborderons les

spécificités d'une telle instrumentation dans le chapitre 5. Certaines de ces constructions peuvent être traitées symboliquement par le test concolique et ne nécessitent donc pas de traduction spécifique en C. C'est le cas avec `\valid` : nous supposons la définition de fonctions C `fvalid` et `fvalidr`, supportées par le générateur de tests, qui retournent la valeur de validité du pointeur en argument si celui-ci est une variable globale ou un paramètre formel de la fonction sous test [Chebaro et al., 2012a].

Arithmétique non bornée. Tandis que l'utilisation d'une bibliothèque externe d'arithmétique non bornée (comme GMP) pour traduire les entiers d'E-ACSL est appropriée pour la validation à l'exécution, il n'en est pas de même pour la génération de tests qui traite tous les chemins d'exécution : si le code des fonctions de la bibliothèque est analysé par le générateur de tests, cette solution se révèlera inefficace à cause de l'explosion du nombre de chemins d'exécution. En effet, la génération de tests à partir du programme instrumenté aura à traiter un code beaucoup plus compliqué, avec de nombreux appels de fonctions, des allocations/désallocations dynamiques, etc. Ceci peut être évité en exécutant symboliquement les fonctions de la bibliothèque : PATHCRAWLER offre un support dédié aux fonctions de la bibliothèque GMP, dont les opérations sont traduites efficacement en contraintes sur les entiers non bornés, que le solveur de contraintes sous-jacent peut traiter.

4.6 JUSTIFICATION DE CORRECTION DE LA TRADUCTION

Nous donnons une justification de correction des règles de traduction. Nous voulons notamment nous assurer que si une erreur est trouvée par le générateur de tests alors il y a une inconsistance entre le code et sa spécification. Pour atteindre cet objectif, nous montrons que l'exécution du code inséré par la traduction d'un terme, d'un prédicat ou d'une annotation a pour effet de calculer la même valeur que la valeur de ce terme, ce prédicat ou cette annotation évalués à leur place dans le programme initial. Pour cela, nous nous basons sur la sémantique dénotationnelle des instructions définie au chapitre précédent.

Nous ne détaillons pas les justifications de toutes les constructions. Nous avons choisi un échantillon représentatif du langage composé des constantes pour les termes, des conjonctions pour les prédicats, des assertions, des contrats de boucle et des contrats de fonction. Les propriétés que nous vérifions sont définies en partie 4.6.1. Les parties 4.6.2, 4.6.3 et 4.6.4 justifient la correction de ces propriétés respectivement sur les termes, sur les prédicats et sur les annotations.

4.6.1 PROPRIÉTÉS DE L'INSTRUMENTATION

Définissons maintenant les propriétés exprimant la correction de la traduction, c'est-à-dire assurant que le code instrumenté a la même sémantique que le code annoté dont il est issu.

Ces propriétés se décomposent en trois parties qui sont la préservation des erreurs, l'inclusion des mémoires et l'absence d'erreur à l'exécution due à l'instrumentation.

Propriété 1 : Préservation des erreurs

À partir de toute mémoire $\neq \gamma_{\lightning}$, si l'exécution des instructions obtenues par traduction d'une annotation a du programme P provoque une erreur dans le programme instrumenté P' par cette traduction de a, alors l'annotation a est invalide.

La préservation des erreurs (propriété 1) assure l'absence de faux positifs lors de la génération de tests sur le programme instrumenté, c'est-à-dire que toute erreur détectée par la génération de tests correspond à une erreur du programme annoté.

Propriété 2 : Inclusion des mémoires

Soit A la séquence d'instructions obtenue à partir d'une liste d'insertions de code I selon les règles d'insertion énoncées dans la partie 4.1.1. Soit $(\rho, \sigma) \neq \gamma_{\lightning}$ une mémoire quelconque. Pour tout préfixe B de A, si $(\rho', \sigma') = C^*[\![B]\!] (\rho, \sigma)$, alors $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$.

L'inclusion des mémoires (propriété 2) est un lemme nécessaire à la justification de la propriété 1. Il énonce que les seules variables modifiées dans les instructions ajoutées dans le programme instrumenté sont les variables fraîchement créées par la traduction. Les variables existantes dans le programme d'origine ne sont pas modifiées par la traduction.

Propriété 3 : Absence d'erreur

Soit A la séquence d'instructions obtenue à partir d'une liste d'insertions de code I issue de la traduction d'un terme ou d'un prédicat. Soit $\gamma \neq \gamma_{\lightning}$ une mémoire quelconque. Pour tout préfixe B de A, $C^*[\![B]\!] \gamma \neq \gamma_{\lightning}$.

L'absence d'erreur (propriété 3) est un lemme nécessaire à la justification des propriétés 1 et 2. Il énonce que la traduction d'un terme ou d'un prédicat E-ACSL quelconque ne provoque pas d'erreur à l'exécution, pas de division par zéro, pas d'accès invalide, etc.

Dans les parties 4.6.2 et 4.6.3 nous déclinons ces propriétés sur les termes et sur les prédicats. Dans la partie 4.6.4 nous justifions les propriétés 1 et 2 à partir des propriétés prouvées dans les parties 4.6.2 et 4.6.3.

4.6.2 CORRECTION DE LA TRADUCTION DES TERMES

Afin de justifier la correction de la traduction des prédicats et des annotations, nous devons prouver la correction de la traduction des termes.

Définition 1 : Préservation de la sémantique des termes

Soit $t : T$ un terme de type T au label l qui est traduit par le couple (I, e) où I est une liste d'insertions de code et e est la variable donnant la valeur du terme t. On dit que la sémantique des termes est préservée si le fragment de programme A obtenu à partir de I et exécuté à partir d'une mémoire quelconque $\gamma \neq \gamma_{\lightning}$ donne une mémoire $C^*[\![A]\!] \gamma$ telle que $\mathcal{E}[\![t]\!] \gamma = \mathcal{E}[\![e]\!] (C^*[\![A]\!] \gamma)$.

Propriété 4 : Préservation de la sémantique des termes

Pour tout programme annoté P, les règles de traduction préservent la sémantique de tout terme E-ACSL dans P.

L'évaluation des insertions de code I générées lors de la traduction d'un terme t ajoute une nouvelle variable e (fraîche) à la mémoire courante. La propriété 4 énonce que sa valeur correspond à l'évaluation de t dans la mémoire γ. Il est facile de vérifier que les variables qui sont déjà dans la mémoire initiale ne sont pas impactées dans la mémoire résultante (propriété 2), et la mémoire résultant de la traduction du terme n'est pas la mémoire d'erreur $\gamma_\lightning$ (propriété 3). Ces propriétés sont vérifiées par la traduction de chaque terme. Elles nous permettent de prouver la propriété 4. À titre d'exemple, la preuve est fournie pour la règle τ-CONST de traduction des constantes.

CORRECTION DE LA TRADUCTION DES CONSTANTES

Rappelons la règle de traduction des constantes :

$$\tau\text{-CONST}\quad \frac{}{(l, cst : \mathbb{Z}) \overset{\tau}{\mapsto} ((l, \square_{\underline{\texttt{var\_}n = cst;}}), \texttt{var\_}n)}$$

Les propriétés 2, 3 et 4 sont déclinées ainsi sur le cas de la constante :

Propriété 4. $\mathcal{E}[\![\ cst\ ]\!]\ \gamma = \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \square_{\underline{\texttt{var\_}n = cst;}}\ ]\!]\ \gamma)$.

Propriété 2. $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$ où $(\rho', \sigma') = C^*[\![\ \square_{\underline{\texttt{var\_}n = cst;}}\ ]\!]\ (\rho, \sigma)$.

Propriété 3. $C^*[\![\ \square_{\underline{\texttt{var\_}n = cst;}}\ ]\!]\ \gamma \neq \gamma_\lightning$.

Nous les justifions une à une :

Justification de la propriété 4.
Nous transformons le membre droit en une expression identique au membre gauche en appliquant les règles de sémantique dénotationnelle (indiquées à chaque étape au dessus du symbole =) et de définition des mémoires :

$$\begin{aligned}
&\mathcal{E}[\![\ \texttt{var_}n\ ]\!]\ (C^*[\![\ \square_{\underline{\texttt{var_}n = cst}\ ;}\ ]\!]\ (\rho, \sigma)) \\
&\overset{\text{C-Z-SET}}{=} \mathcal{E}[\![\ \texttt{var_}n\ ]\!]\ (\rho[\texttt{var_}n \mapsto \delta], \sigma[\delta \mapsto \mathcal{E}[\![\ cst\ ]\!]\ (\rho, \sigma)]) \\
&\qquad \text{où } \delta \text{ est une adresse fraîche} \\
&\overset{\text{E-LVAL}}{=} \sigma[\delta \mapsto \mathcal{E}[\![\ cst\ ]\!]\ (\rho, \sigma)]\ (\ \rho[\texttt{var_}n \mapsto \delta]\ (\texttt{var_}n)\ ) \\
&\overset{\text{ENV-GET-1}}{=} \sigma[\delta \mapsto \mathcal{E}[\![\ cst\ ]\!]\ (\rho, \sigma)]\ (\delta) \\
&\overset{\text{STORE-GET-1}}{=} \mathcal{E}[\![\ cst\ ]\!]\ (\rho, \sigma)
\end{aligned}$$

On peut appliquer la règle C-Z-SET car la mémoire $C^*[\![\ \square_{\underline{\texttt{var\_}n = cst}\ ;}\ ]\!]\ (\rho, \sigma) \neq \gamma_\lightning$ d'après la propriété 3.

□

Justification de la propriété 2.
L'instruction générée $\square_{\underline{\texttt{var\_}n = cst;}}$ ne modifie que la variable fraîche $\texttt{var\_}n$. Le nouvel environnement est $\rho' = \rho[\texttt{var\_}n \mapsto \delta]$ et le nouveau store est $\sigma' = \sigma[\delta \mapsto \mathcal{E}[\![\ cst\ ]\!]\ (\rho, \sigma)]$. On peut donc en conclure $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$.

□

Justification de la propriété 3.
La nouvelle mémoire est $(\rho[\texttt{var\_}n \mapsto \delta], \sigma[\delta \mapsto \mathcal{E}[\![\ cst\ ]\!]\ (\rho,\sigma)])$. Ce n'est pas une mémoire d'erreur car l'évaluation d'une constante ne produit pas d'erreur. On a donc bien $C^*[\![\ \square_{\underline{\texttt{var\_}n\ =\ cst\texttt{;}}}\ ]\!]\ \gamma \neq \gamma_{\lightning}$.

□

4.6.3 CORRECTION DE LA TRADUCTION DES PRÉDICATS

Afin de prouver la correction de la traduction des annotations, nous devons prouver la correction de la traduction des prédicats.

Définition 2 : Préservation de la sémantique des prédicats

Soit p un prédicat au label l qui est traduit par le couple (I, e) où I est une liste d'insertions de code et e est la variable donnant la valeur du prédicat p. On dit que la sémantique des prédicats est préservée si le fragment de programme A obtenu à partir de I et exécuté à partir d'une mémoire quelconque $\gamma \neq \gamma_{\lightning}$ donne une mémoire $C^*[\![\ A\ ]\!]\ \gamma$ telle que $\mathcal{E}[\![\ p\ ]\!]\ \gamma = \mathcal{E}[\![\ e\ ]\!]\ (C^*[\![\ A\ ]\!]\ \gamma)$.

Propriété 5 : Préservation de la sémantique des prédicats

Pour tout programme annoté P, les règles de traduction préservent la sémantique de tout prédicat E-ACSL dans P.

L'évaluation des insertions de code I générées lors de la traduction d'un prédicat p ajoute une nouvelle variable e (fraîche) à la mémoire courante. La propriété 5 énonce que sa valeur correspond à l'évaluation de p dans la mémoire γ. Il est facile de vérifier que les variables qui sont déjà dans la mémoire initiale ne sont pas impactées dans la mémoire résultante (propriété 2), et la mémoire résultant de la traduction du prédicat n'est pas la mémoire d'erreur $\gamma_{\lightning}$ (propriété 3). Ces propriétés sont vérifiées par la traduction de chaque prédicat. Elles nous permettent de prouver la propriété 5. À titre d'exemple, la preuve est fournie pour la règle π-AND de traduction des conjonctions.

CORRECTION DE LA TRADUCTION DES CONJONCTIONS

La règle de traduction du prédicat de conjonction π-AND est définie ainsi :

$$\pi\text{-AND}\quad \frac{(l, p_1) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (l, p_2) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, p_1 \texttt{\&\&}\ p_2) \overset{\pi}{\mapsto} (I_1 \cdot (l, \texttt{int var\_}n = e_1\texttt{;}) \cdot (l, \texttt{if(var\_}n\texttt{)\{}I_2 \cdot (l, \texttt{var\_}n = e_2\texttt{;})\texttt{\}}), \texttt{var\_}n)}$$

Les propriétés 2, 3 et 5 sont déclinées ainsi sur le cas de la conjonction :

Propriété 5.
$\mathcal{E}[\![\ \texttt{p1 \&\& p2}\ ]\!]\ \gamma = \mathcal{E}[\![\ \texttt{var\_n}\ ]\!]\ (\ C^*[\![\ A_1\texttt{; int var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma\ )$ en admettant que les fragments de programme A_1 et A_2 sont obtenus à partir des listes d'insertions I_1 et I_2 selon les règles d'insertion énoncées dans la partie 4.1.1.

Propriété 2.
$\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$ où $(\rho', \sigma') = C^*[\![\ A_1\texttt{; int var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ (\rho, \sigma)$.

Propriété 3.
$C^*[\![A_1\texttt{; int var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}]\!]\ \gamma \neq \gamma_\lightning$.

Nous les justifions une à une en raisonnant par induction :

Justification de la propriété 5.
Par hypothèse, les traductions de p_1 et p_2 sont correctes, c'est-à-dire :

$$\begin{array}{lcll} \mathcal{E}[\![\ p_1\ ]\!]\ \gamma & = & \mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma) & \text{H1} \\ \mathcal{E}[\![\ p_2\ ]\!]\ \gamma & = & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ \gamma) & \text{H2} \end{array}$$

Nous transformons le membre droit en une expression identique au membre gauche en appliquant les règles de sémantique dénotationnelle et de définition des mémoires :

$$\begin{array}{ll} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (\ C^*[\![\ A_1\texttt{; int var\_}n\texttt{; var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma\ ) \\ \overset{\text{C-SEQ-1}}{=} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{int var\_}n\texttt{; var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)) \\ = & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{int var\_}n\texttt{; var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma') \\ \overset{\text{C-SEQ-2}}{=} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ (C[\![\ \texttt{int var\_}n\texttt{;}\ ]\!]\ \gamma')\ ) \\ \overset{\text{C-DECL}}{=} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ (\rho'[\texttt{var\_}n \mapsto \delta], \sigma'[\delta \mapsto \bot])\ ) \\ = & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{var\_}n = e_1\texttt{; if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma''\ ) \\ \overset{\text{C-SEQ-2}}{=} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ C[\![\ \texttt{var\_}n = e_1\texttt{;}\ ]\!]\ \gamma''\ ) \\ \overset{\text{C-SET}}{=} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (C^*[\![\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{; var\_}n = e_2\texttt{;\}}\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'])\ ) \\ & \text{où } (\rho', \sigma') = \gamma' = C^*[\![\ A_1\ ]\!]\ \gamma, \\ & (\rho'', \sigma'') = \gamma'' = (\rho'[\texttt{var\_}n \mapsto \delta], \sigma'[\delta \mapsto \bot]) \\ & \text{et } \delta \text{ est une adresse fraîche.} \end{array}$$

Justifions que $\mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) = \mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)$.

Développons le membre gauche :

$$\begin{array}{ll} & \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) \\ \overset{\text{E-LVAL}}{=} & \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''](\rho''(\texttt{var\_}n)) \\ = & \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''](\rho'[\texttt{var\_}n \mapsto \delta](\texttt{var\_}n)) \text{ par définition de } \rho'' \\ \overset{\text{ENV-GET-1}}{=} & \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''](\delta) \\ \overset{\text{STORE-GET-1}}{=} & \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'' \\ = & \mathcal{E}[\![\ e_1\ ]\!]\ (\rho'[\texttt{var\_}n \mapsto \delta], \sigma'[\delta \mapsto \bot]) \text{ par définition de } \gamma'' \\ \overset{\text{E-LVAL}}{=} & \sigma'[\delta \mapsto \bot](\rho'[\texttt{var\_}n \mapsto \delta](e_1)) \\ \overset{\text{ENV-GET-2}}{=} & \sigma'[\delta \mapsto \bot](\rho'(e_1)) \text{ car } \texttt{var\_}n \neq e_1 \\ \overset{\text{STORE-GET-2}}{=} & \sigma'(\rho'(e_1)) \text{ car } \delta \text{ est une adresse fraîche (donc différente de l'adresse de } e_1) \\ \overset{\text{E-LVAL}}{=} & \mathcal{E}[\![\ e_1\ ]\!]\ (\rho', \sigma') \\ = & \mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma) \text{ par définition de } \gamma' \end{array}$$

On a donc $\mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) = \mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)$.

(a) Cas où la valeur de l'expression conditionnelle $\texttt{var\_}n$ est vraie, soit $\mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)$ = true, c'est-à-dire par hypothèse H1 : $\mathcal{E}[\![\ p_1\ ]\!]\ \gamma$ = true. Dans ce cas, par définition de C-IF on exécute le bloc $A_2\texttt{; var\_}n = e_2\texttt{;}$:

$$
\begin{array}{ll}
C^*[\![\ \texttt{if(var\_n)}\ \{A_2;\ \texttt{var\_n = } e_2;\}\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) & \\
\overset{\text{C-IF}}{=} & C^*[\![\ A_2;\ \texttt{var\_n = } e_2;\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) \\
\overset{\text{C-SEQ-1}}{=} & C[\![\ \texttt{var\_n = } e_2;\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']))\ ) \\
= & C[\![\ \texttt{var\_n = } e_2;\ ]\!]\ \gamma''' \\
\overset{\text{C-SET}}{=} & (\rho''', \sigma'''[\rho'''(\texttt{var\_n}) \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ \gamma''']) \\
\overset{\text{ENV-GET-1}}{=} & (\rho''', \sigma'''[\delta \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ \gamma''']) \text{ car } \rho'''(\texttt{var\_n}) = \delta, \\
& \text{où } (\rho''', \sigma''') = \gamma''' = (C^*[\![\ A_2\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']))\ ).
\end{array}
$$

Donc :

$$
\begin{array}{ll}
\mathcal{E}[\![\ \texttt{var\_n}\ ]\!]\ (\rho''', \sigma'''[\delta \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ \gamma''']) & \\
\overset{\text{E-LVAL}}{=} & \sigma'''[\delta \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ \gamma'''](\rho'''(\texttt{var\_n})) \\
\overset{\text{ENV-GET-1}}{=} & \sigma'''[\delta \mapsto \mathcal{E}[\![\ e_2\ ]\!]\ \gamma'''](\delta) \\
\overset{\text{STORE-GET-1}}{=} & \mathcal{E}[\![\ e_2\ ]\!]\ \gamma''' \\
= & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''])\ )\ \text{par définition de } \gamma''' \\
= & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ (\rho'[\texttt{var\_n} \mapsto \delta], \sigma'[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'])\ )\ \text{par définition de } \rho'' \text{ et } \sigma'' \\
& \text{or } \texttt{var\_n} \text{ est une variable fraîche introduite par la traduction qui n'intervient} \\
& \text{pas dans } p_2, \text{ on peut donc remplacer } \rho'[\texttt{var\_n} \mapsto \delta] \text{ et } \sigma'[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''] \\
& \text{par } \rho' \text{ et } \sigma'. \\
= & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ (\rho', \sigma')) \\
& \text{comme la propriété 2 est satisfaite } (\rho \subseteq \rho' \text{ et } \sigma \subseteq \sigma') \text{ et comme } p_2 \text{ n'utilise} \\
& \text{aucune des variables ajoutée à } \rho \text{ pour constituer } \rho', \text{ on peut remplacer } (\rho', \sigma') \\
& \text{par } (\rho, \sigma). \\
= & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ (\rho, \sigma)) \\
= & \mathcal{E}[\![\ e_2\ ]\!]\ (C^*[\![\ A_2\ ]\!]\ \gamma) \text{ par définition de } \gamma \\
\overset{\text{H2}}{=} & \mathcal{E}[\![\ p_2\ ]\!]\ \gamma \\
= & \mathcal{E}[\![\ p_1\ ]\!]\ \gamma \wedge \mathcal{E}[\![\ p_2\ ]\!]\ \gamma \text{ (car on est dans le cas } \mathcal{E}[\![\ p_1\ ]\!]\ \gamma = \text{true)} \\
\overset{\text{P-AND}}{=} & \mathcal{E}[\![\ p_1\ \texttt{\&\&}\ p_2\ ]\!]\ \gamma
\end{array}
$$

(b) Cas où la valeur de l'expression conditionnelle $\texttt{var\_n}$ est fausse, soit $\mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)$ = false, c'est-à-dire par hypothèse H1 : $\mathcal{E}[\![\ p_1\ ]\!]\ \gamma$ = false. Dans ce cas, par définition de C-IF on exécute le bloc du **else** (qui ici est vide et n'apparaît pas car facultatif) :

$$
\begin{array}{ll}
C^*[\![\ \texttt{if(var\_n)}\ \{A_2;\ \texttt{var\_n = } e_2;\}\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) & \\
\overset{\text{C-IF}}{=} & (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''])
\end{array}
$$

Donc :

$$
\begin{array}{ll}
\mathcal{E}[\![\ \texttt{var\_n}\ ]\!]\ (\rho'', \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'']) & \\
\overset{\text{E-LVAL}}{=} & \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''\ ]\ (\rho''(\texttt{var\_n})) \\
\overset{\text{ENV-GET-1}}{=} & \sigma''[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma''\ ](\delta) \\
\overset{\text{STORE-GET-1}}{=} & \mathcal{E}[\![\ e_1\ ]\!]\ \gamma'' \\
= & \mathcal{E}[\![\ e_1\ ]\!]\ (\rho'[\texttt{var\_n} \mapsto \delta], \sigma'[\delta \mapsto \bot])\ \text{par définition de } \gamma'' \\
& \text{or } \texttt{var\_n} \text{ est une variable fraîche introduite par la traduction qui n'intervient} \\
& \text{pas dans } p_1, \text{ on peut donc remplacer } \rho'[\texttt{var\_n} \mapsto \delta] \text{ et } \sigma'[\delta \mapsto \bot] \\
& \text{par } \rho' \text{ et } \sigma'. \\
= & \mathcal{E}[\![\ e_1\ ]\!]\ (\rho', \sigma') \\
= & \mathcal{E}[\![\ e_1\ ]\!]\ (C^*[\![\ A_1\ ]\!]\ \gamma)\ \text{par définition de } \gamma' \\
\overset{\text{H1}}{=} & \mathcal{E}[\![\ p_1\ ]\!]\ \gamma \\
= & \mathcal{E}[\![\ p_1\ ]\!]\ \gamma \wedge \mathcal{E}[\![\ p_2\ ]\!]\ \gamma\ \ \text{(car on est dans le cas } \mathcal{E}[\![\ p_1\ ]\!]\ \gamma = \text{false)} \\
\overset{\text{P-AND}}{=} & \mathcal{E}[\![\ p_1\ \texttt{\&\&}\ p_2\ ]\!]\ \gamma
\end{array}
$$

On a donc bien :

$\mathcal{E}[\![\ p_1\ \texttt{\&\&}\ p_2\ ]\!]\ \gamma = \mathcal{E}[\![\ \texttt{var\_}n\ ]\!]\ (\ C^*[\![\ A_1\texttt{;}\ \texttt{int}\ \texttt{var\_}n = e_1\texttt{;}\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{;}\ \texttt{var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma\ ).$

□

Justification de la propriété 2.
Par hypothèse, les traductions de p_1 et p_2 sont correctes, c'est-à-dire :

$$\begin{array}{ll}
\rho \subseteq \rho_1 & \text{H3}\\
\sigma \subseteq \sigma_1 \text{ où } (\rho_1, \sigma_1) = C^*[\![\ A_1\ ]\!]\ (\rho, \sigma) & \text{H4}\\
\rho \subseteq \rho_2 & \text{H5}\\
\sigma \subseteq \sigma_2 \text{ où } (\rho_2, \sigma_2) = C^*[\![\ A_2\ ]\!]\ (\rho'_1, \sigma'_1) & \text{H6}\\
\text{où } \rho'_1 = \rho_1[\texttt{var_}n \mapsto \delta] \text{ et } \sigma'_1 = \sigma_1[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma] &
\end{array}$$

De manière informelle, les instructions générées $\texttt{int}\ \texttt{var\_}n = e_1\texttt{;}$ et $\texttt{var\_}n = e_2\texttt{;}$ ne modifient que la variable fraîche $\texttt{var\_}n$ et les hypothèses H3 à H6 nous assurent que les blocs A_1 et A_2 ne modifient pas les variables de la mémoire $\gamma = (\rho, \sigma)$. On en conclut donc que le fragment de programme instrumenté $A_1\texttt{;}\ \texttt{int}\ \texttt{var\_}n = e_1\texttt{;}\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{;}\ \texttt{var\_}n = e_2\texttt{;\}}$ ne modifie pas la mémoire γ. On a donc bien :

$\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$ où $(\rho', \sigma') = C^*[\![\ A_1\texttt{;}\ \texttt{int}\ \texttt{var\_}n = e_1\texttt{;}\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{;}\ \texttt{var\_}n = e_2\texttt{;\}}\ ]\!]\ (\rho, \sigma).$

□

Justification de la propriété 3.
Par hypothèse, les traductions de p_1 et p_2 sont correctes, c'est-à-dire :

$$\begin{array}{rcll}
C^*[\![\ A_1\ ]\!]\ \gamma & \neq & \gamma_{\lightning} & \text{H7}\\
C^*[\![\ A_2\ ]\!]\ \gamma'_1 & \neq & \gamma_{\lightning} & \text{H8}\\
 & & \text{où } \gamma_1 = (\rho_1, \sigma_1) = C^*[\![\ A_1\ ]\!]\ \gamma & \\
 & & \text{et } \gamma'_1 = (\rho_1[\texttt{var_}n \mapsto \delta], \sigma_1[\delta \mapsto \mathcal{E}[\![\ e_1\ ]\!]\ \gamma]) &
\end{array}$$

Les hypothèses H7 et H8 et le fait que les fragments de code générés ne produisent pas d'erreur (seule la variable fraîche $\texttt{var\_}n$ est affectée) nous assurent que si γ n'est pas une mémoire d'erreur (γ ne peut pas être $\gamma_{\lightning}$ puisque la traduction se serait arrêtée dès l'obtention de l'erreur). On a donc bien :

$C^*[\![\ A_1\texttt{;}\ \texttt{int}\ \texttt{var\_}n = e_1\texttt{;}\ \texttt{if(var\_}n\texttt{)\{}A_2\texttt{;}\ \texttt{var\_}n = e_2\texttt{;\}}\ ]\!]\ \gamma \neq \gamma_{\lightning}.$

□

4.6.4 Correction de la traduction des annotations

Nous allons maintenant prouver les propriétés 1 et 2 pour chaque classe d'annotations : les assertions, les contrats de boucle et les contrats de fonction. Ces justifications s'appuient sur la correction de la traduction des termes et des prédicats justifiée dans les parties précédentes.

4.6.4.1 Correction de la traduction des assertions

La règle de traduction d'une annotation `assert` est la suivante :

$$\alpha\text{-ASSERT} \quad \frac{(l,p) \overset{\pi}{\mapsto} (I,e)}{(l,\texttt{assert}\ p;) \overset{\alpha}{\mapsto} I \cdot (l,\texttt{fassert}(e);)}$$

Les propriétés 1 et 2 sont déclinées sur le cas de l'assertion :

Propriété 1.
Prouvons que pour toute mémoire γ on a $C[\![\ \texttt{/*@assert}\ p;\texttt{*/}\ ;\ ]\!]\,\gamma = \gamma_{\lightning}$ si $C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma = \gamma_{\lightning}$, où le fragment de programme A est obtenu à partir de la séquence d'insertions de code I selon les règles d'insertion énoncées dans la partie 4.1.1.

Propriété 2.
Prouvons que si l'`assert` est valide, alors $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$, où $(\rho',\sigma') = C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,(\rho,\sigma)$.

Nous les justifions une à une :

Justification de la propriété 1.
Par hypothèse, la traduction de p est correcte, c'est-à-dire :

$$\begin{array}{rcll} \mathcal{E}[\![\ e\ ]\!]\,(C^*[\![\ A\ ]\!]\,\gamma) & = & \mathcal{E}[\![\ p\ ]\!]\,\gamma & \text{H1} \\ C^*[\![\ A\ ]\!]\,\gamma & \neq & \gamma_{\lightning} & \text{H2} \end{array}$$

Supposons que le code généré à partir de l'`assert` produit une erreur :

$$C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma \quad = \quad \gamma_{\lightning} \quad \text{H3}$$

et prouvons que l'`assert` est invalide : $C[\![\ \texttt{/*@assert}\ p;\texttt{*/}\ ;\ ]\!]\,\gamma = \gamma_{\lightning}$.

Nous distinguons deux cas quant à la valeur de e.

(a) Cas où $\mathcal{E}[\![\ e\ ]\!]\,(C^*[\![\ A\ ]\!]\,\gamma) = 1$, c'est-à-dire par hypothèse H1 : $\mathcal{E}[\![\ p\ ]\!]\,\gamma = 1$. Prouvons que l'exécution du code généré ne produit pas $\gamma_{\lightning}$. Développons $C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma$:

$$C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma \overset{\substack{\text{C-SEQ-1}\\ =\\ \text{C-FASSERT}\\ =}}{} \begin{array}{l} C[\![\ \texttt{fassert}(e);\ ]\!]\,(C^*[\![\ A\ ]\!]\,\gamma) \\ C^*[\![\ A\ ]\!]\,\gamma\ (\text{car } \mathcal{E}[\![\ e\ ]\!]\,(C^*[\![\ A\ ]\!]\,\gamma) \neq 0) \end{array}$$

D'autre part, l'hypothèse H3 nous donne : $C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma = \gamma_{\lightning}$. On obtient par transitivité de l'égalité : $C^*[\![\ A\ ]\!]\,\gamma = \gamma_{\lightning}$, ce qui contredit l'hypothèse H2 selon laquelle $C^*[\![\ A\ ]\!]\,\gamma \neq \gamma_{\lightning}$. Cette contradiction montre que ce cas n'existe pas.

(b) Cas où $\mathcal{E}[\![\ e\ ]\!]\,(C^*[\![\ A\ ]\!]\,\gamma) = 0$, c'est-à-dire par hypothèse H1 : $\mathcal{E}[\![\ p\ ]\!]\,\gamma = 0$. Développons $C[\![\ \texttt{/*@assert}\ p;\texttt{*/}\ ;\ ]\!]\,\gamma$:

$$C[\![\ \texttt{/*@assert}\ p;\texttt{*/}\ ;\ ]\!]\,\gamma \overset{\substack{\text{C-ASSERT}\\ =}}{} \gamma_{\lightning}\ (\text{car } \mathcal{E}[\![\ p\ ]\!]\,\gamma = 0)$$

Nous avons ainsi prouvé que si le code traduisant une assertion provoque une erreur alors l'assertion en question est invalide :

$$C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\,\gamma = \gamma_{\lightning} \Rightarrow C[\![\ \texttt{/*@assert}\ p;\texttt{*/}\ ;\ ]\!]\,\gamma = \gamma_{\lightning}.$$

□

Justification de la propriété 2.
Par hypothèse, la traduction de p est correcte, c'est-à-dire :

$$\begin{array}{ll} \rho \subseteq \rho'' & \text{H4} \\ \sigma \subseteq \sigma'' & \text{H5} \\ \text{où } (\rho'', \sigma'') = C^*[\![\ A\ ]\!]\ (\rho, \sigma) & \end{array}$$

On doit prouver $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$, avec $(\rho', \sigma') = C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\ (\rho, \sigma)$.

Or, nous avons montré (voir justification de la propriété 1) que si l'assertion est valide, alors $C^*[\![\ A;\ \texttt{fassert}(e);\ ]\!]\ \gamma = C^*[\![\ A\ ]\!]\ \gamma$. Cette égalité nous donne également $(\rho', \sigma') = (\rho'', \sigma'')$. En remplaçant ρ'' par ρ' et σ'' par σ' dans les hypothèses H4 et H5, on obtient $\rho \subseteq \rho'$ et $\sigma \subseteq \sigma'$.

□

4.6.4.2 CORRECTION DE LA TRADUCTION DES CONTRATS DE BOUCLES

Donnons une intuition de la preuve des propriétés 1 et 2 dans le cas des contrats de boucles. Nous considérons une boucle **`while`** annotée dont le contrat est composé d'un **`loop invariant`**, d'un **`loop variant`** et d'une clause **`loop assigns`**. La traduction utilise les trois règles suivantes :

α-CHECK-LOOP-ASSIGNS
$$\frac{(BegIter_l, x_1) \overset{\tau}{\mapsto} (I_1^1, e_1^1) \quad (EndIter_l, x_1) \overset{\tau}{\mapsto} (I_1^2, e_1^2) \quad \ldots \quad (BegIter_l, x_m) \overset{\tau}{\mapsto} (I_m^1, e_m^1) \quad (EndIter_l, x_m) \overset{\tau}{\mapsto} (I_m^2, e_m^2)}{\begin{array}{l}(l, \texttt{loop assigns } X;) \overset{\alpha}{\mapsto} \\ \quad I_1^1 \cdot I_1^2 \cdot (BegIter_l, ctype\ \texttt{var\_}n_1 = e_1^1;) \cdot (EndIter_l, \texttt{fassert}(e_1^2 == \texttt{var\_}n_1);) \cdot \ldots \\ \quad\quad \cdot I_m^1 \cdot I_m^2 \cdot (BegIter_l, ctype\ \texttt{var\_}n_m = e_m^1;) \cdot (EndIter_l, \texttt{fassert}(e_m^2 == \texttt{var\_}n_m);)\end{array}}$$

α-CHECK-INVARIANT
$$\frac{(l, p) \overset{\pi}{\mapsto} (I_1, e_1) \qquad (EndIter_l, p) \overset{\pi}{\mapsto} (I_2, e_2)}{(l, \texttt{loop invariant } p;) \overset{\alpha}{\mapsto} I_1 \cdot (l, \texttt{fassert}(e_1);) \cdot I_2 \cdot (EndIter_l, \texttt{fassert}(e_2);)}$$

α-VARIANT
$$\frac{(BegIter_l, t) \overset{\tau}{\mapsto} (I_1, e_1) \qquad (EndIter_l, t) \overset{\tau}{\mapsto} (I_2, e_2)}{\begin{array}{l}(l, \texttt{loop variant } t;) \overset{\alpha}{\mapsto} \\ \quad I_1 \cdot (BegIter_l, \texttt{fassert}(\underline{0 <= e_1});) \cdot I_2 \cdot (EndIter_l, \texttt{fassert}(\underline{e_2^{\boxtimes} < e_1^{\boxtimes}});)\end{array}}$$

La figure 4.14 illustre la démarche à partir de ces règles. La partie gauche de la figure montre une boucle disposant d'un contrat de boucle (**`loop invariant`**, **`loop assigns`** et **`loop variant`**) au label l, avec un corps quelconque A. En partie droite est affiché le code résultant de la traduction de cette boucle, où chaque ensemble d'instructions (regroupées avec une accolade) est mis en relation avec l'annotation du programme original dont il est la traduction. $Spec2Code(p, e_1)$ est une notation signifiant que p se traduit en e_1, en accord avec la règle α-CHECK-INVARIANT.

Si l'invariant (ligne 2) n'est pas établi avant la boucle, alors on obtient la mémoire d'erreur (cas C-WHILE-1), ce qui correspond au cas où le `fassert` des lignes 2–3 du code généré renvoie faux. Si la condition de boucle (ligne 5) est fausse, la mémoire est inchangée (cas C-WHILE-2) car le code que nous générons n'impacte pas les variables du programme original, donc la condition de boucle e a la même valeur en chaque point dans le programme original et le programme instrumenté. Si le variant de la boucle (ligne 4) est négatif au début de la boucle, on obtient la mémoire d'erreur (cas C-WHILE-3), ce qui correspond au cas où le `fassert` des lignes 9–10 du code généré renvoie faux. Si l'invariant de boucle n'est pas préservé à la fin de l'itération de boucle, la mémoire d'erreur est obtenue (cas C-WHILE-4), ce qui correspond au cas où le `fassert` des lignes 13–14 du code généré renvoie faux. Si le variant ne décroît pas strictement entre deux itérations de boucle, alors la mémoire d'erreur est obtenue (cas C-WHILE-5), ce qui correspond au cas où le `fassert`

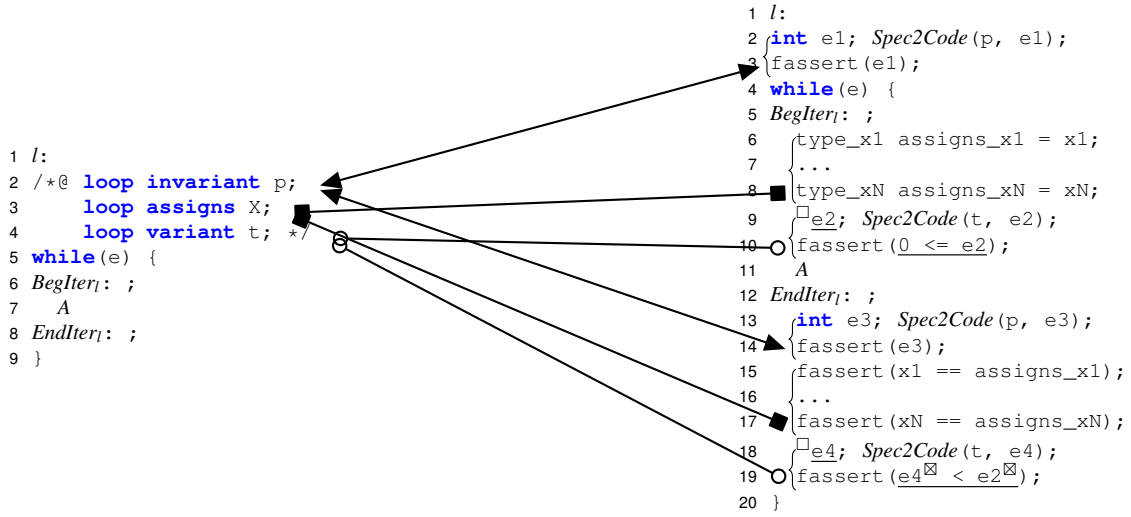

FIGURE 4.14 – Schéma de preuve de correction de la traduction des contrats de boucle

des lignes 18–19 du code généré renvoie faux (la comparaison est faite avec la valeur du variant mémorisée à la ligne 9). Si une left-value n'appartenant pas à la clause assigns (ligne 3) a été modifiée par la boucle, alors la mémoire d'erreur est obtenue (cas C-WHILE-6), ce qui correspond au cas où l'un des `fassert` des lignes 15–17 du code généré renvoie faux. Ces comparaisons de valeurs sont faites avec les valeurs mémorisées aux lignes 6–8 du code généré.

4.6.4.3 CORRECTION DE LA TRADUCTION DES CONTRATS DE FONCTIONS

Donnons maintenant une intuition de la preuve des propriétés 1 et 2 dans le cas des contrats de fonctions. Il nous faut distinguer deux cas : le cas où la fonction est "sous vérification" – c'est-à-dire où c'est la fonction de plus haut niveau et sa précondition est supposée vraie – et le cas où la fonction est appelée – et sa précondition doit être vérifiée au point d'appel. Nous traitons les deux cas de manière séparée dans les deux sous-parties suivantes.

Fonction sous vérification

Nous considérons une fonction f sous vérification annotée, dont le contrat est composé d'une clause **`requires`**, d'une clause **`assigns`** et d'une clause **`ensures`**. La traduction utilise les trois règles suivantes :

$$\alpha\text{-ASSUME-PRE}\quad \frac{(Beg_f, p) \overset{\pi}{\mapsto} (I, e)}{(Beg_f, \left\{\begin{array}{l}\texttt{typically}\\ \texttt{requires}\end{array}\right. p\,;) \overset{\alpha}{\mapsto} I \cdot (Beg_f, \texttt{fassume}\,(e)\,;)}$$

$$\alpha\text{-CHECK-ASSIGNS}\quad \frac{(Beg_f, x_1) \overset{\tau}{\mapsto} (I_1^1, e_1^1) \quad (End_f, x_1) \overset{\tau}{\mapsto} (I_1^2, e_1^2) \quad \ldots \quad (Beg_f, x_m) \overset{\tau}{\mapsto} (I_m^1, e_m^1) \quad (End_f, x_m) \overset{\tau}{\mapsto} (I_m^2, e_m^2)}{\begin{array}{l}(End_f, \texttt{assigns}\ X\,;) \overset{\alpha}{\mapsto}\\ \quad I_1^1 \cdot I_1^2 \cdot (Beg_f, ctype\ \texttt{var\_}n_1 = e_1^1\,;) \cdot (End_f, \texttt{fassert}\,(e_1^2 == \texttt{var\_}n_1)\,;) \cdot \ldots\\ \quad \cdot I_m^1 \cdot I_m^2 \cdot (Beg_f, ctype\ \texttt{var\_}n_m = e_m^1\,;) \cdot (End_f, \texttt{fassert}\,(e_m^2 == \texttt{var\_}n_m)\,;)\end{array}}$$

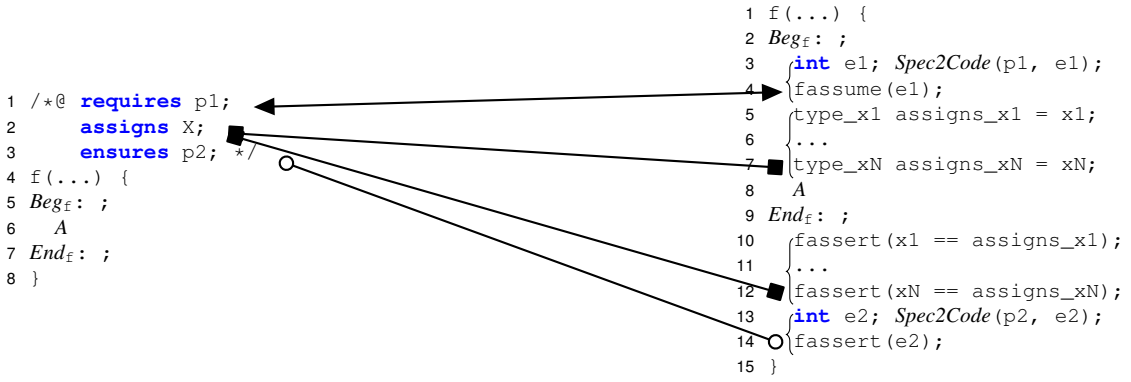

FIGURE 4.15 – Schéma de preuve de correction de la traduction des contrats de fonction (fonction sous vérification)

$$\alpha\text{-CHECK-POST}\quad \frac{(End_f, p) \overset{\pi}{\mapsto} (I, e)}{(End_f, \texttt{ensures}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (End_f, \texttt{fassert}\,(e)\,\texttt{;})}$$

La figure 4.15 illustre la justification de correction de la traduction des contrats pour la fonction sous vérification sur un schéma générique normalisé de fonction. La partie gauche de la figure montre une fonction f disposant d'un contrat, avec un corps quelconque A. En partie droite est affiché le code résultant de la traduction de cette fonction, où chaque ensemble d'instructions (regroupées avec une accolade) est mis en relation avec l'annotation du programme original dont il est la traduction.

Par hypothèse, la précondition de la fonction sous vérification est vraie, ce qui correspond aux lignes 3–4 du code généré. Si la fonction modifie une des left-values n'appartenant pas à la clause `assigns`, alors la mémoire d'erreur est obtenue (cas F-2), ce qui correspond au cas où l'un des `fassert` des lignes 10–12 du code généré renvoie faux. Si la postcondition n'est pas vérifiée à la fin de la fonction, on obtient la mémoire d'erreur (cas F-3), ce qui correspond au cas où le `fassert` des lignes 13–14 du code généré renvoie faux.

Fonctions appelées

Considérons enfin une fonction f appelée, dont le contrat est composé d'une clause `requires`, d'une clause `assigns` et d'une clause `ensures`. La traduction des contrats d'une fonction appelée réutilise les règles α-CHECK-ASSIGNS et α-CHECK-POST de la partie précédente. En revanche, afin de vérifier la précondition d'une fonction appelée et non la supposer vraie, la règle α-ASSUME-PRE est remplacée par la règle α-CHECK-PRE :

$$\alpha\text{-CHECK-PRE}\quad \frac{(Beg_f, p) \overset{\pi}{\mapsto} (I, e)}{(Beg_f, \texttt{requires}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (Beg_f, \texttt{fassert}\,(e)\,\texttt{;})}$$

La figure 4.16 illustre la justification de correction de la traduction des contrats pour les fonctions appelées. La partie gauche de la figure montre une fonction f disposant d'un contrat, avec un corps quelconque A. En partie droite est affiché le code résultant de la traduction de cette fonction, où chaque ensemble d'instructions (regroupées avec une accolade) est mis en relation avec l'annotation du programme original dont il est la traduction.

Si la précondition n'est pas vérifiée au début de la fonction, alors la mémoire d'erreur est obtenue (cas F-1), ce qui correspond aux lignes 3–4 du code généré. Si la fonction modifie

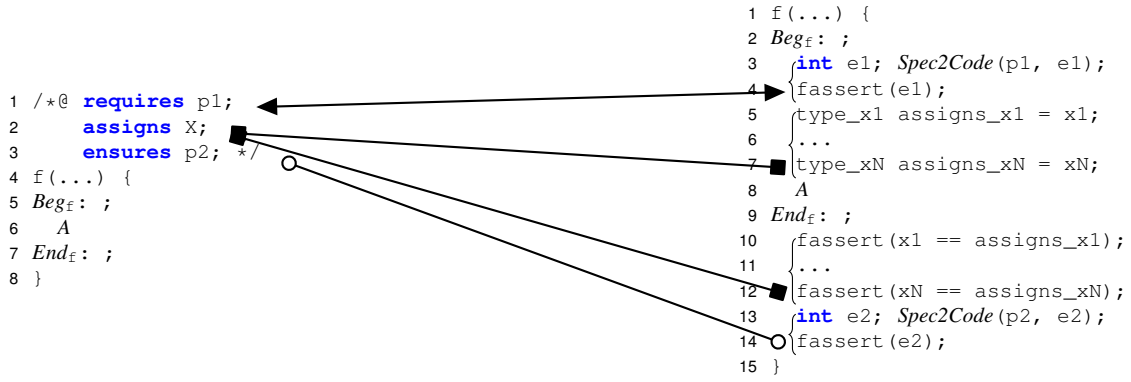

FIGURE 4.16 – Schéma de preuve de correction de la traduction des contrats de fonction (fonction appelée)

une des left-values n'appartenant pas à la clause `assigns`, alors la mémoire d'erreur est obtenue (cas F-2), ce qui correspond au cas où l'un des `fassert` des lignes 10–12 du code généré renvoie faux. Si la postcondition n'est pas vérifiée à la fin de la fonction, on obtient la mémoire d'erreur (cas F-3), ce qui correspond au cas où le `fassert` des lignes 13–14 du code généré renvoie faux.

CONCLUSION DU CHAPITRE

L'objectif de ce chapitre était de définir une traduction en C des annotations E-ACSL pour la génération de tests et de fournir une justification de la correction de cette traduction.

Nous avons tout d'abord présenté le processus de la traduction dans sa généralité. Puis nous avons défini les règles permettant pour chaque terme et prédicat E-ACSL d'obtenir un code C calculant cette expression tout en préservant la sémantique d'E-ACSL. Ces règles nous ont permis de définir les règles de traduction des annotations E-ACSL : `assert`, `requires`, `typically`, `assigns`, `ensures`, `loop invariant`, `loop assigns` et `loop variant`. Ces différentes instrumentations des annotations nous permettront par la suite de détecter différentes contradictions entre le code et la spécification.

Grâce à notre justification de correction, nous pouvons assurer l'absence de faux positifs lors de la génération de tests sur le programme instrumenté : si une erreur est trouvée par la génération de tests alors il y a une erreur dans le programme d'origine. Cette propriété est énoncée comme une implication si l'on considère le langage C et le langage E-ACSL dans leur ensemble, mais si on se restreint au sous-ensemble du langage décrit par la grammaire du chapitre précédent il y a probablement une équivalence, c'est-à-dire que s'il y a une erreur dans le programme original et si la traduction des annotations E-ACSL est correcte, alors il y a une erreur dans le programme instrumenté et elle est révélée par la génération de tests.

La justification de cette propriété est fournie de manière formelle pour les assertions et de manière informelle mais intuitive et rigoureuse pour les contrats de boucle et les contrats de fonction.

5

VÉRIFICATION À L'EXÉCUTION DES ANNOTATIONS LIÉES AU MODÈLE MÉMOIRE

Dans ce chapitre nous abordons la vérification dynamique des annotations liées au modèle mémoire. Ceci requiert de pouvoir simuler un modèle mémoire bas niveau afin de pouvoir surveiller les opérations bas niveau des programmes C (allocations, initialisations, etc.). En revanche, le modèle mémoire utilisé par le générateur de tests structurels que nous utilisons n'est pas assez bas niveau et ne permet donc pas l'exécution symbolique de ces constructions. En conséquence, nous changeons d'approche pour ces propriétés. En effet, contrairement au chapitre 4, ces annotations ne seront pas vérifiées par test structurel mais par validation à l'exécution (*runtime assertion checking*).

Dans un premier temps nous présentons les différentes annotations que nous ne pouvons pas traiter par génération de tests et que nous qualifions d'opérations de bas niveau (partie 5.1). Puis nous présentons quelques détails de conception de notre modèle mémoire (partie 5.2). Enfin nous abordons les principes de l'instrumentation nécessaire pour la vérification de ces annotations (partie 5.3).

5.1 ANNOTATIONS LIÉES AU MODÈLE MÉMOIRE

Définissons d'abord quelques notions. Les objets de la mémoire sont des blocs. Chaque bloc est caractérisé par une adresse de base, une taille (en nombre d'octets) et un contenu (qui peut être initialisé ou non). Tout ceci constitue le modèle mémoire.

Les annotations E-ACSL que nous considérons sont décrites dans la figure 5.1. Cette figure est une extension de la grammaire des termes et des prédicats E-ACSL définie au chapitre 3.

term : :=	...	*pred* : :=	...
	| `\base_addr` (*term*)		| `\valid` (*term*)
	| `\block_length` (*term*)		| `\valid_read` (*term*)
	| `\offset` (*term*)		| `\initialized` (*term*)

FIGURE 5.1 – Extension de la grammaire des termes et des prédicats

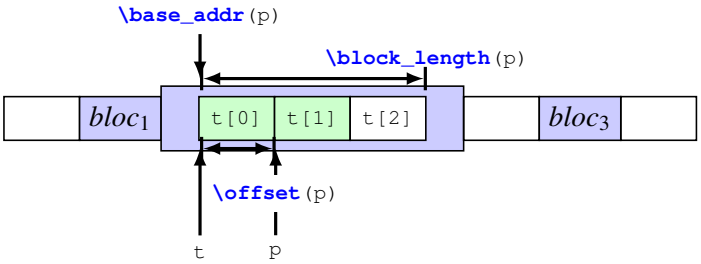

FIGURE 5.2 – Adresse de base, longueur du bloc et offset dans un bloc

La figure 5.2 illustre les annotations `\base_addr`, `\block_length` et `\offset`. On y voit un modèle mémoire simplifié contenant 3 blocs. Le deuxième bloc est constitué de trois cases dénotées `t[0]`, `t[1]` et `t[2]`, pouvant contenir chacune un entier. Le pointeur `p` pointe vers la deuxième case du deuxième bloc. L'adresse de base de `p` (`\base_addr(p)`) désigne le début du bloc. La longueur du bloc de `p` (`\block_length(p)`) est la taille du bloc à partir de son adresse de base (trois fois la taille d'un entier sur cet exemple). L'offset de `p` (`\offset(p)`) est le décalage de `p` par rapport à son adresse de base (la taille d'un entier sur cet exemple).

Donnons la sémantique informelle de chacune de ces annotations. `\base_addr(`t`)` retourne l'adresse de base du bloc alloué dans lequel pointe le pointeur t. `\block_length(`t`)` retourne la longueur (en octets) du bloc alloué dans lequel pointe le pointeur t. `\offset(`t`)` retourne le décalage (en octets) entre t et l'adresse de base du bloc dans lequel pointe t. `\valid(`t`)`, respectivement `\valid_read(`t`)`, est vrai si le déréférencement de t est autorisé en lecture et en écriture, respectivement au moins en lecture. `\valid(`t`)` implique `\valid_read(`t`)` mais l'inverse n'est pas vrai. `\initialized(`t`)` est un prédicat prenant en paramètre un pointeur t sur une left-value lv, il est vrai si lv est initialisée. Soit une mémoire $\gamma = (\rho, \sigma)$ en accord avec la formalisation des mémoires du chapitre 3. `\initialized(`t`)` est vrai s'il existe une left-value lv et une adresse δ telles que $\rho(lv) = \delta$, le pointeur t a pour valeur δ et $\sigma(\delta) \neq \bot$.

```
1  void main() {
2    int * t = malloc(3*sizeof(int));
3    t[0] = 1;
4    t[1] = 1;
5    int * p = t+1;
6    //@ assert \base_addr(p) == (char*)t;
7    //@ assert \block_length(p) == 3 * sizeof(int);
8    //@ assert \offset(p) == 1*sizeof(int);
9    //@ assert \valid(p+0) && \valid(p+1);
10   //@ assert \initialized(p) && !\initialized(p+1);
11   free(t);
12 }
```

Listing 5.1 – Annotations mémoire – Exemple en C

Le listing 5.1 illustre l'utilisation de toutes ces annotations sur un programme C. Un tableau `t` de trois entiers est alloué ligné 2. Les deux premières cases du tableau sont initialisées à 1 aux lignes 3–4. Un pointeur `p` est ensuite initialisé à `t+1` ligne 5, c'est-à-dire pointe la deuxième case du tableau. L'assertion ligne 6 affirme que `p` et `t` ont la même adresse de base. L'assertion ligne 7 affirme que le bloc contenant `p` a une taille de trois entiers. L'assertion ligne 8 affirme que `p` a un offset de la taille d'un entier dans son bloc. L'assertion ligne 9 affirme que `p` peut être déréférencé sur les offsets 0 et 1 (`*(p+0)` et `*(p+1)`). L'assertion ligne 10 affirme que `*p` est initialisé mais pas `*(p+1)`. Les annotations

```
1  void main(void) {
2    int *t;
3    int *p;
4    __store_block((void *)(& p),4U);
5    __store_block((void *)(& t),4U);
6    __full_init((void *)(& t));
7    t = (int *)__e_acsl_malloc((unsigned long)((unsigned int)3 * sizeof(int)));
8    __initialize((void *)(t + 0),sizeof(int));
9    *(t + 0) = 1;
10   __initialize((void *)(t + 1),sizeof(int));
11   *(t + 1) = 1;
12   __full_init((void *)(& p));
13   p = t + 1;
14   /*@ assert \base_addr(p) == (char *)t; */
15   void *__e_acsl_base_addr;
16   __e_acsl_base_addr = __base_addr((void *)p);
17   e_acsl_assert(__e_acsl_base_addr == (char *)t);
18   /*@ assert \block_length(p) == 3*sizeof(int); */
19   unsigned long __e_acsl_block_length;
20   __e_acsl_block_length = (unsigned long)__block_length((void *)p);
21   e_acsl_assert((unsigned long long)__e_acsl_block_length ==
22                 (unsigned long long)(3 * 4));
23   /*@ assert \offset(p) == 1*sizeof(int); */
24   int __e_acsl_offset;
25   __e_acsl_offset = __offset((void *)p);
26   e_acsl_assert((unsigned int)__e_acsl_offset == (unsigned int)(1 * 4));
27   /*@ assert \valid(p+0) && \valid(p+1); */
28   int __e_acsl_valid;
29   int __e_acsl_and;
30   __e_acsl_valid = __valid((void *)(p + 0),sizeof(int));
31   if (__e_acsl_valid) {
32     int __e_acsl_valid_2;
33     __e_acsl_valid_2 = __valid((void *)(p + 1),sizeof(int));
34     __e_acsl_and = __e_acsl_valid_2;
35   }
36   else __e_acsl_and = 0;
37   e_acsl_assert(__e_acsl_and);
38   /*@ assert \initialized(p) && !\initialized(p+1); */
39   int __e_acsl_initialized;
40   int __e_acsl_and_2;
41   __e_acsl_initialized = __initialized((void *)p,sizeof(int));
42   if (__e_acsl_initialized) {
43     int __e_acsl_initialized_2;
44     __e_acsl_initialized_2 = __initialized((void *)(p + 1),sizeof(int));
45     __e_acsl_and_2 = ! __e_acsl_initialized_2;
46   }
47   else __e_acsl_and_2 = 0;
48   e_acsl_assert(__e_acsl_and_2);
49   __e_acsl_free((void *)t);
50   __delete_block((void *)(& p));
51   __delete_block((void *)(& t));
52   __e_acsl_memory_clean();
53   return;
54 }
```

Listing 5.2 – Version instrumentée du programme du listing 5.1

du programme sont satisfaites : leur vérification à l'exécution ne produit pas d'erreur.

Le listing 5.2 présente le programme du listing 5.1 instrumenté afin de vérifier les annotations à l'exécution. Les lignes 15–17 du listing 5.2 permettent d'exécuter l'assertion ligne 6 du listing 5.1. Les lignes 19–22 du listing 5.2 permettent d'exécuter l'assertion ligne 7 du listing 5.1. Les lignes 24–26 du listing 5.2 permettent d'exécuter l'assertion ligne 8 du listing 5.1. Les lignes 28–37 du listing 5.2 permettent d'exécuter l'assertion ligne 9 du listing 5.1. Les lignes 39–48 du listing 5.2 permettent d'exécuter l'assertion ligne 10

du listing 5.1. Les appels de la fonction `__store_block` (lignes 4–5) servent à enregistrer les blocs des nouvelles variables. Les appels de la fonction `__delete_block` (lignes 50–51) permettent de les désinscrire à la fin de leur existence.

5.2 MODÈLE MÉMOIRE POUR LA VÉRIFICATION À L'EXÉCUTION

Dans cette partie, nous présentons les détails de conception du modèle mémoire proposé. Nous justifions nos choix d'implémentation puis présentons les algorithmes de calcul du plus grand préfixe commun, de recherche, d'ajout et de suppression d'un bloc dans le modèle mémoire.

5.2.1 STRUCTURE DE DONNÉES "STORE"

Au chapitre 3, nous avons défini un *store* comme étant une fonction associant une valeur à une adresse. Dans ce chapitre, ce terme fait référence à une structure de données qui implémente efficacement cette fonction, donc contenant des adresses et des valeurs associées. C'est dans le *store* que les informations sur la validité et l'initialisation des blocs mémoire seront enregistrées et recherchées par la suite.

Le code instrumenté pouvant accéder et modifier fréquemment les données du *store*, une implémentation efficace requiert une structure de données offrant une bonne complexité en temps et en espace. Cette structure de données doit être triée : on peut avoir besoin d'accéder directement à un bloc à partir de son adresse de base, mais aussi à partir de n'importe quelle adresse contenue dans le bloc (c'est-à-dire entre l'adresse de base et l'adresse de fin du bloc). Par exemple, la fonction `__base_addr` utilisée pour le traitement de la construction `\base_addr`(t) cherche l'adresse de base la plus proche et inférieure à t. Cette contrainte ne nous permet pas d'utiliser une table de hachage. Les listes chaînées ne sont pas assez efficaces à cause de la complexité linéaire dans le pire cas. Les arbres binaires de recherche non équilibrés ont aussi une complexité linéaire dans le pire des cas quand les données sont insérées dans un ordre strictement croissant ou strictement décroissant. Certes, le coût du rééquilibrage de l'arbre (pour un arbre binaire de recherche équilibré) serait amorti dans le cas où les modifications de la structure de l'arbre sont moins nombreuses que les accès simples ; mais ce n'est pas nécessairement vrai sur les exemples de code que nous avons instrumentés comme nous le verrons au chapitre 9.

Notre choix s'est donc porté sur la structure de *Patricia trie* [Szpankowski, 1990] (appelé aussi *radix tree* ou "arbre à préfixe compact"). Cette structure est efficace même si l'arbre n'est pas équilibré. Les clés sont les adresses de base des blocs (c'est-à-dire des mots de 32 ou 64 bits) ou des préfixes d'adresses. Chaque feuille contient les données relatives à un bloc en mémoire (voir partie 5.1 pour le détail des informations stockées). Le routage de la racine jusqu'à une feuille particulière se fait grâce aux nœuds internes. Chacun d'eux contient le plus grand préfixe de l'adresse de base commun entre ses deux fils.

La figure 5.3 illustre un Patricia trie sur des adresses 8 bits, limite choisie pour des raisons de simplicité. Il contient trois blocs représentés par trois feuilles. Seules les adresses de base apparaissent sur le schéma. Il contient deux préfixes stockés dans les nœuds qui ne sont pas des feuilles. Le symbole "$\star$" signifie que la valeur du bit à cette position n'a pas d'importance.

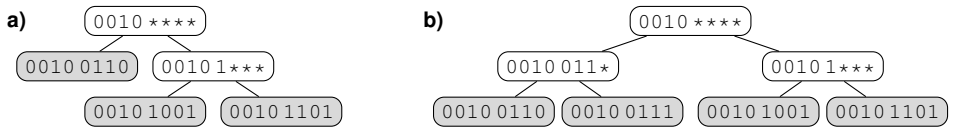

FIGURE 5.3 – Exemple de Patricia trie avant **(a)** et après **(b)** insertion de 0010 0111

La complexité théorique dans le pire des cas d'un accès dans un Patricia trie dans notre cas est en $O(k)$ où k est la longueur d'un mot (c'est-à-dire 32 ou 64 bits). En pratique, un programme ne pouvant allouer des blocs que dans un espace mémoire limité, la profondeur de l'arbre est inférieure à cette borne. De plus, contrairement aux chaînes de caractères (la première application des Patricia tries), la comparaison des mots de bits peut être implémentée très efficacement par des opérations bit-à-bit.

Les données de chaque bloc n'occupent que quelques octets en mémoire, exception faite des données d'initialisation du bloc. Le statut d'initialisation de chaque octet est monitoré séparément (lignes 8 et 10 du listing 5.2) et les champs de bits ne sont pas supportés. Dans le pire cas (bloc partiellement initialisé), chaque octet utilise un bit supplémentaire portant l'information sur son initialisation. Nous conservons le nombre d'octets initialisés de chaque bloc dans un compteur. Dans le cas où tous les octets (ou aucun) sont initialisés, le tableau censé contenir les bits portant l'information d'initialisation est libéré et nous utilisons alors la valeur du compteur pour avoir cette information. De plus, nous utilisons une fonction spécifique (ligne 12 du listing 5.2) dans le cas où tous les octets d'un bloc sont initialisés d'un coup, au lieu d'invoquer une fonction d'initialisation sur chaque octet du bloc.

5.2.2 CALCUL DU MASQUE DU PLUS GRAND PRÉFIXE COMMUN (MPGPC)

Définissons les types *word* et *Mask* utilisés dans ce chapitre. Un *word* de taille n est une suite de n bits (0 ou 1). La taille d'un word est généralement un multiple de 2 et dépend de l'architecture matérielle. Nous utiliserons la constante `NULL` pour dénoter l'adresse nulle et nous noterons $word$ le type des words. Nous utilisons une constante $WLEN$ pour faire référence à la taille des words. Un *masque* est un word dont les $k \leq WLEN$ premiers bits sont à 1 et les $WLEN - k$ bits suivants sont à 0. Nous noterons $Mask$ le type des masques. Les types de données *word* et *Mask* seront utilisés dans les algorithmes présentés dans ce chapitre. Nous définissons un ordre lexicographique sur les masques. Soient m et m' deux masques, $m \leq m'$ si et seulement si m' contient au moins autant de 1 que m.

Le *masque du plus grand préfixe commun* m de deux adresses a et b est le plus grand des masques (suivant l'ordre lexicographique) tel que, si n est le nombre de 1 de m, alors les n premiers bits de a et b sont identiques. On dira aussi que n est la taille du préfixe commun à a et b.

Par exemple, le masque du plus grand préfixe commun de a = 0110 0111 et b = 0111 1111 est m = 1110 0000. En effet, seuls les trois premiers bits de a sont identiques au trois premiers bits de b, les words divergent à partir du quatrième bit. On notera p = 011⋆ ⋆⋆⋆⋆ le plus grand préfixe commun à a et b, où le symbole "$\star$" signifie que la valeur du bit à cette position ne fait pas partie du préfixe commun.

Algorithm 1 Recherche du masque du plus grand préfixe commun de a et b

$array[0..WLEN]$ of Mask $masks$ ▷ masques dans l'ordre lexicographique
$array[0..WLEN]$ of int $longer$ ▷ indices des masques plus longs à tester
$array[0..WLEN]$ of int $shorter$ ▷ indices des masques plus courts à tester

Ensure: $(a \ \& \ \texttt{\textbackslash result}) == (b \ \& \ \texttt{\textbackslash result})$
Ensure: $\forall i.(0 \leq i \leq WLEN \Rightarrow (masks[i] \ \& \ a) == (masks[i] \ \& \ b) \Rightarrow \texttt{\textbackslash result} \geq masks[i])$
Ensure: $\exists i.(0 \leq i \leq WLEN \wedge \texttt{\textbackslash result} == masks[i])$

1: **function** MPGPC(word a, b)
2: int i, word $nxor$
3: $nxor \leftarrow \ \sim (a \ \hat{} \ b)$ ▷ $\hat{}$: "ou" exclusif bit-à-bit, $\sim$: "non" bit-à-bit
4: $i \leftarrow WLEN/2$
5: **while** $i > 0$ **do**
6: **if** $nxor \geq masks[i]$ **then**
7: $i \leftarrow longer[i]$
8: **else**
9: $i \leftarrow shorter[i]$
10: **end if**
11: **end while**
12: **return** $masks[-i]$
13: **end function**

Définition 3 : Masque du plus grand préfixe commun (MPGPC)

Le MPGPC de a et b est le plus grand masque inférieur ou égal à la négation du ou exclusif de a et de b.

Nous présentons maintenant un algorithme efficace de recherche du masque du plus grand préfixe commun de deux adresses. Une version naïve et inefficace de cet algorithme consiste en un parcours linéaire des mots mémoire de gauche à droite jusqu'à trouver des bits différents, de la même manière qu'on pourrait le faire sur des chaînes de caractères. La complexité de la version naïve est en $O(WLEN)$.

La version optimisée de cet algorithme consiste en une recherche dichotomique du masque dans un tableau pré-calculé qui contient tous les masques de préfixes possibles. L'algorithme 1 présente cette méthode de recherche. La complexité de cet algorithme est en $O(log(WLEN))$ (soit ≤ 6 itérations pour des words de 32 bits et ≤ 7 itérations pour des words de 64 bits). Les transitions entre les étapes de la recherche se font en utilisant des indices pré-calculés, de manière à obtenir le prochain masque à comparer avec $nxor$. La variable $nxor$ contient la négation du ou exclusif de a et b. Le tableau $masks$ contient tous les $WLEN + 1$ masques possibles sur $WLEN$ bits. Les tableaux $longer$ et $shorter$ contiennent des indices du tableau $masks$. Si $nxor$ est plus grand que le masque courant à l'indice i ($masks[i]$), le prochain masque à comparer avec $nxor$ sera celui à l'indice $longer[i]$ ($masks[longer[i]]$). Si $nxor$ est plus petit que le masque courant, le prochain masque à comparer avec $nxor$ sera à l'indice $shorter[i]$. Si l'indice courant k est négatif, alors $-k$ est l'indice du masque du plus grand préfixe commun.

Sur des words de longueur 8 bits, les tableaux $masks$, $longer$ et $shorter$ sont initialisés de cette manière :

```
// index            0    1    2    3    4    5    6    7    8
byte  masks[] = {0x00,0x80,0xC0,0xE0,0xF0,0xF8,0xFC,0xFE,0xFF};
int longer [] = {   0,  -1,   3,  -3,   6,  -5,   7,   8,  -8};
int shorter[] = {   0,   0,   1,  -2,   2,  -4,   5,  -6,  -7};
```

Considérons deux words de 8 bits a = 0110 0111 et b = 0111 1111. $nxor$ prend la valeur 1110 0111. L'algorithme essaie $i = 4$ d'abord, puis $i = shorter[4] = 2$, puis $i = longer[2] = 3$, puis $i = longer[3] = -3$, pour finalement retourner $masks[3] = 0xE0$, qui est 1110 0000, soit précisément le masque du plus grand préfixe commun de a et b.

5.2.3 RECHERCHE

Présentons maintenant les algorithmes de recherche d'un bloc dans le *store* à partir d'une adresse a. Ils sont de deux sortes. La recherche exacte cherche un bloc dont l'adresse de base est égale à a, elle est notamment utilisée dans l'instrumentation des fonctions `free` et `realloc`. La recherche du contenant cherche un bloc qui contient a, c'est-à-dire que a est incluse entre le début (adresse de base) et la fin du bloc. Elle est notamment utilisée dans l'instrumentation des annotations **`\valid`** et **`\base_addr`**.

Dans ces algorithmes, nous noterons $Ptree$ le type de données des Patricia tries et $Block$ le type des blocs. Voici une définition simplifiée de ces types :

```
Block = { word ptr; int size; };
Ptree = {
  bool isLeaf;
  word addr;
  Mask mask;
  Ptree left, right, father;
  Block leaf;
};
```

Un objet de type $Block$ est caractérisé par un word et une taille. Un objet de type $Ptree$ est quant à lui caractérisé par un booléen indiquant si l'arbre est une feuille, un word encodant la valeur et le masque du plus grand préfixe commun à ses deux fils, le sous-arbre gauche, le sous-arbre droit, le nœud "père" et le bloc porté par le nœud si c'est une feuille. Par exemple pour la racine du Patricia trie de la figure 5.3 **(a)**, les deux champs `addr = 0010 0000` et `mask = 1111 0000` encodent le préfixe 0010 ****. Nous utiliserons la constante $emptyPtree$ pour dénoter un Patricia trie vide.

RECHERCHE EXACTE

L'algorithme 2 retourne le bloc B dont l'adresse de base est égale à l'adresse a passée en paramètre si un tel bloc est dans le store, ou la valeur $noBlock$ sinon. Nous notons $root$ le *store* global représenté par un Patricia trie, dans lequel tous les blocs sont enregistrés.

L'algorithme procède comme suit : tant que x n'est pas une feuille – c'est donc un nœud interne ayant deux fils – on se dirige vers le fils ayant le plus grand préfixe commun avec l'adresse a passée en paramètre de l'algorithme. Quand on arrive sur une feuille, cette dernière contient l'adresse que l'on cherchait.

Algorithm 2 Recherche du bloc d'adresse de base a, dans Block ∪ {noBlock}

Require: $root \neq emptyPtree \wedge a \neq \text{NULL}$
Ensure: `\result` $\neq noBlock \Rightarrow$ `\result`$.ptr == a$

```
1:  function FIND-EXACT(Ptree root, word a)
2:      Ptree x
3:      x ← root
4:      while ¬x.isLeaf do
5:          if (x.addr & x.mask) ≠ (a & x.mask) then return noBlock
6:          end if
7:          if (x.right.addr & x.right.mask) == (a & x.right.mask) then
8:              x ← x.right
9:          else if (x.left.addr & x.left.mask) == (a & x.left.mask) then
10:             x ← x.left
11:         else return noBlock
12:         end if
13:     end while
14:     if a ≠ x.leaf.ptr then return noBlock
15:     else return x.leaf
16:     end if
17: end function
```

RECHERCHE DU "CONTENANT"

L'algorithme 3 retourne le bloc B contenant l'adresse a passée en paramètre, c'est-à-dire que l'adresse a est plus grande que l'adresse de base de B et plus petite que l'adresse de fin de B (adresse de base du bloc + taille du bloc). Si un tel bloc n'existe pas, la valeur $noBlock$ est renvoyée. Cet algorithme est utilisé pour des requêtes du type `\valid` ou `\initialized` où l'adresse passée en paramètre ne correspond pas nécessairement à l'adresse de base d'un bloc.

Cet algorithme est similaire à celui de la recherche exacte, au détail près qu'il faut vérifier que a est comprise entre l'adresse de base et l'adresse de fin de B quand on arrive sur une feuille. On utilise une pile $stack$ contenant les nœuds à partir desquels il faut reprendre le parcours si la recherche n'aboutit pas. Nous notons $Stack$ le type de la pile, $emptyStack$ une pile vide, $push$ la fonction d'empilement et pop la fonction de dépilement. Quand on explore le fils droit, on empile le fils gauche. Quand on arrive sur une feuille, on vérifie que a est comprise entre l'adresse de base et l'adresse de fin du bloc. Si c'est le cas alors on a trouvé le bloc que l'on cherchait. Sinon on utilise le dernier nœud empilé (et on le dépile) s'il existe, sinon on retourne $noBlock$. L'instruction $continue$ aux lignes 12 et 18 de l'algorithme 3 indique que l'exécution ignore les instructions suivantes et reprend au début de la boucle à la ligne 6.

5.2.4 AJOUT D'UN BLOC

La figure 5.4 illustre l'insertion de l'adresse `0010 0111` dans un arbre. L'algorithme 4 a déterminé que le nœud le plus similaire est `0010 0110`, et crée le père correspondant : `0010 011*`.

Algorithm 3 Recherche du bloc contenant une adresse a, dans Block ∪ {noBlock}

```
1: function FIND-CONT(Ptree root, word a)
   Ptree x, Stack stack
2:     if (root == emptyPtree) ∨ (a == NULL) then return noBlock
3:     else
4:         x ← root
5:         stack ← emptyStack
6:         while true do
7:             if x.isLeaf then
8:                 if x.addr > a then
9:                     if stack = emptyStack then return noBlock
10:                    else
11:                        x ← pop(stack)
12:                        continue
13:                    end if
14:                else if a < x.leaf.size + x.addr then return x.leaf
15:                else if x.leaf.size == 0 ∧ a == x.leaf.ptr then return x.leaf
16:                else if stack = emptyStack then return noBlock
17:                else
18:                    x ← pop(stack)
19:                    continue
20:                end if
21:            end if
22:            if x.right.addr & x.right.mask ≤ a & x.right.mask then
23:                stack ← push(x.left)
24:                x ← x.right
25:            else if x.left.addr & x.left.mask ≤ a & x.left.mask then
26:                x ← x.left
27:            else if stack == emptyStack then return noBlock
28:            else
29:                x ← pop(stack)
30:            end if
31:        end while
32:    end if
33: end function
```

L'algorithme 4 présente l'ajout d'un nouvel élément a à l'arbre du *store*. Si l'arbre est vide, l'élément est ajouté à la racine. Sinon, on recherche le nœud b (qui n'est pas nécessairement une feuille) le plus similaire au nœud à insérer. Celui-ci sera son frère dans la nouvelle configuration de l'arbre. Nous créons ensuite le nœud correspondant au père. Le plus grand préfixe commun des deux fils est calculé pour le père, et les fils sont ordonnés en fonction de leur adresse (le plus petit à gauche). Puis le père est inséré dans l'arbre à l'ancien emplacement de b. Les champs de b et de l'ancien père de b (s'il existe) sont également mis à jour pour maintenir la cohérence de l'arbre.

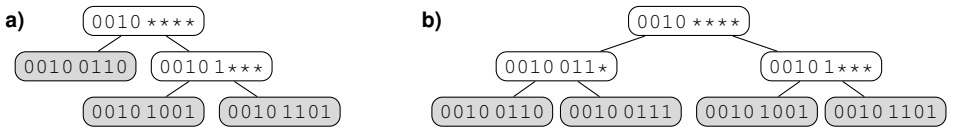

FIGURE 5.4 – Exemple de Patricia trie avant **(a)** et après **(b)** insertion de `0010 0111`

5.2.5 SUPPRESSION D'UN BLOC

La figure 5.5 illustre la suppression de l'adresse `0010 0111` dans un Patricia trie. Ce nœud ainsi que son père `0010 011*` sont supprimés, et `0010 0110` est remonté d'un niveau.

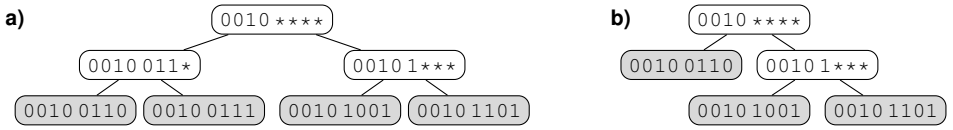

FIGURE 5.5 – Exemple de Patricia trie avant **(a)** et après **(b)** suppression de `0010 0111`

La procédure de suppression d'un bloc B porté par une feuille du Patricia trie est présentée par l'algorithme 5. L'algorithme commence par rechercher la feuille x qui porte le bloc B. Si cet élément est la racine, l'arbre devient vide. Sinon, Cette feuille a un frère b et un père f. B et son père f sont supprimés, et b remonte d'un niveau (et prend donc la place de f). Enfin, les champs du nouveau père de b (g) sont mis à jour pour maintenir la cohérence de l'arbre (le masque du plus grand préfixe commun est recalculé).

5.3 INSTRUMENTATION POUR LA VÉRIFICATION À L'EXÉCUTION

Nous présentons l'instrumentation d'un programme C annoté dans une optique de vérification à l'exécution des annotations liées au modèle mémoire. Certaines instructions C doivent être instrumentées afin d'enregistrer à l'exécution des informations sur la mémoire, et les annotations sont instrumentées afin de pouvoir utiliser ces informations.

5.3.1 INSTRUMENTATION DES ALLOCATIONS ET AFFECTATIONS

Dans cette partie, nous définissons de manière informelle l'instrumentation à opérer sur le programme annoté afin de pouvoir observer les allocations et les affectations en mémoire.

Comme nous l'avons vu, pour traiter ces annotations E-ACSL, nous devons conserver pour chaque bloc les informations suivantes : l'adresse de base, le nombre d'octets occupés en mémoire, l'initialisation de chaque octet et un booléen indiquant si le bloc est en lecture seule (par exemple si c'est une chaîne littérale).

Afin d'enregistrer ces informations nous définissons les fonctions C suivantes. La fonction `__store_block`, qui prend en paramètre un pointeur et une taille en octets, ajoute un bloc dans le *store* (présenté en partie 5.2). La fonction `__store_block` est invoquée sur chaque variable locale et paramètre formel de fonction (au début de leur portée), ainsi que sur chaque variable globale (au début de la fonction `main`). Elle utilise l'algorithme 4. La fonction `__delete_block` permet d'enlever du *store* un bloc reçu en paramètre. Elle est appelée en fin de portée d'une variable et utilise l'algorithme 5.

Nous définissons également des "surcouches" aux fonctions de la bibliothèque standard du C : `malloc`, `realloc`, `calloc` et `free`. L'instrumentation va substituer les appels à ces fonctions par un appel à nos fonctions : `__malloc`, `__realloc`, `__calloc` et `__free`. Ces fonctions permettent d'ajouter ou de supprimer automatiquement les blocs au *store*, ainsi que de transférer les informations d'initialisation d'un bloc à un autre en cas d'application de la fonction `realloc`.

Enfin, nous définissons une fonction `__initialize` qui permet de marquer un certain nombre d'octets d'un bloc comme étant initialisés. L'instrumentation ajoute des appels à cette fonction pour chaque affectation du programme original, ainsi que pour chaque variable globale (initialisée à 0 par défaut en C).

5.3.2 INSTRUMENTATION DES ANNOTATIONS

Afin de pouvoir exploiter les informations sur les blocs stockés dans le *store*, nous définissons les fonctions `__base_addr`, `__block_length`, `__offset`, `__valid`, `__valid_read` et `__initialized`. La sémantique de ces fonctions est conforme à la sémantique des annotations E-ACSL donnée en partie 5.1. La fonction `__base_addr` retourne l'adresse de début d'un bloc dans lequel pointe le pointeur qu'elle reçoit en argument. La fonction `__block_length` retourne le nombre d'octets occupés par le bloc dans lequel pointe le pointeur qu'elle reçoit en argument. La fonction `__offset` retourne le décalage d'un pointeur par rapport à son adresse de base dans le bloc dans lequel pointe le pointeur qu'elle reçoit en argument. Les fonctions `__valid` et `__valid_read` retournent le statut de validité à partir d'un pointeur et d'un nombre d'octets. La fonction `__initialized` retourne le statut d'initialisation d'un nombre d'octets d'un pointeur en argument. Ces fonctions utilisent les algorithmes 2 et 3 afin de récupérer les informations du *store*.

La figure 5.6 présente les règles de traduction des annotations en utilisant le même formalisme qu'au chapitre 4. Nous rappelons que τ est la fonction de traduction des termes E-ACSL et π est la fonction de traduction des prédicats E-ACSL. Dans les règles π-VALID, π-VALID-READ et π-INITIALIZED, `typeof(`$\star e$`)` désigne le type de l'expression $\star e$, qui est un

$$\tau\text{-BASE-ADDR}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash base\_addr}\,(t)\,) \overset{\tau}{\mapsto} (I \cdot (l, \texttt{void}\star\ \texttt{var\_}n = \texttt{\_\_base\_addr}\,(e)\,\texttt{;}), \texttt{var\_}n)}$$

$$\tau\text{-BLOCK-LENGTH}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash block\_length}\,(t)\,) \overset{\tau}{\mapsto} (I \cdot (l, \texttt{unsigned long}\ \texttt{var\_}n = \texttt{\_\_block\_length}\,(e)\,\texttt{;}), \texttt{var\_}n)}$$

$$\tau\text{-OFFSET}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash offset}\,(t)\,) \overset{\tau}{\mapsto} (I \cdot (l, \texttt{int}\ \texttt{var\_}n = \texttt{\_\_offset}\,(e)\,\texttt{;}), \texttt{var\_}n)}$$

$$\pi\text{-VALID}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash valid}\,(t)\,) \overset{\pi}{\mapsto} (I \cdot (l, \texttt{int}\ \texttt{var\_}n = \texttt{\_\_valid}\,(e, \texttt{sizeof}\,(\texttt{typeof}\,(\star e)\,))\,\texttt{;}), \texttt{var\_}n)}$$

$$\pi\text{-VALID-READ}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash valid\_read}\,(t)\,) \overset{\pi}{\mapsto} (I \cdot (l, \texttt{int}\ \texttt{var\_}n = \texttt{\_\_valid\_read}\,(e, \texttt{sizeof}\,(\texttt{typeof}\,(\star e)\,))\,\texttt{;}), \texttt{var\_}n)}$$

$$\pi\text{-INITIALIZED}\quad \frac{(l, t : ptr) \overset{\tau}{\mapsto} (I, e)}{(l, \texttt{\textbackslash initialized}\,(t)\,) \overset{\pi}{\mapsto} (I \cdot (l, \texttt{int}\ \texttt{var\_}n = \texttt{\_\_initialized}\,(e, \texttt{sizeof}\,(\texttt{typeof}\,(\star e)\,))\,\texttt{;}), \texttt{var\_}n)}$$

FIGURE 5.6 – Règles de traduction pour les annotations liées au modèle mémoire

type C tel que `int` ou `char`. `sizeof(typeof(*`e`))` est le nombre d'octets occupés par un objet de ce type en mémoire.

CONCLUSION DU CHAPITRE

Nous avons présenté dans ce chapitre une manière de vérifier dynamiquement les annotations E-ACSL qui concernent la mémoire : `\base_addr`, `\block_length`, `\offset`, `\valid`, `\valid_read` et `\initialized`. Nous avons présenté ces annotations, un modèle mémoire permettant de décider de leur satisfaction ou non à l'exécution, ainsi qu'une manière d'instrumenter le programme afin de les vérifier à l'exécution.

Rappelons que nous avons choisi de vérifier ces annotations à l'exécution plutôt que d'utiliser la génération de tests structurels car le modèle mémoire utilisé par le générateur de tests structurels que nous utilisons n'est pas assez bas niveau et ne permet donc pas l'exécution symbolique de ces constructions.

Notons que nous avons choisi d'implémenter la notion de *store* par des Patricia tries pour des raisons d'efficacité. Le chapitre 9 donne plus de détails concernant cette implémentation du modèle mémoire ainsi qu'une évaluation de ses performances.

Algorithm 4 Ajout d'un bloc a, modifie le Ptree $root$

```
1: procedure ADD(Ptree root, Block a)
   Ptree x, b, f ; Mask lprefix, rprefix
2:     x.addr ← a.ptr, x.mask ← masks[WLEN], x.left ← emptyPtree,
3:     x.right ← emptyPtree, x.father ← emptyPtree, x.leaf ← a
4:     if root == emptyPtree then
5:         root ← x
6:     else
7:         b ← root
8:         while true do
9:             if b.isLeaf then
10:                break
11:            end if
12:            lprefix ← MPGPC(b.left.addr & b.left.mask, a.ptr)
13:            rprefix ← MPGPC(b.right.addr & b.right.mask, a.ptr)
14:            if lprefix > rprefix then
15:                b ← b.left
16:            else if rprefix > lprefix then
17:                b ← b.right
18:            else
19:                break
20:            end if
21:        end while
22:        f.addr ← b.addr & x.addr
23:        f.leaf ← noBlock
24:        f.left ← (x.addr ≤ b.addr) ? x : b
25:        f.right ← (x.addr ≤ b.addr) ? b : x
26:        x.father ← f
27:        if b == root then
28:            f.father ← emptyPtree
29:            f.mask ← MPGPC(b.addr & b.mask, a.ptr)
30:            root ← f
31:        else
32:            if b.father.left == b then
33:                b.father.left ← f
34:            else
35:                b.father.right ← f
36:            end if
37:            f.father ← b.father
38:            f.mask ← MPGPC(f.left.addr & f.left.mask, f.right.addr & f.right.mask)
39:        end if
40:        b.father ← f
41:        if ¬b.isLeaf then
42:            b.mask ← MPGPC(b.left.addr & b.left.mask, b.right.addr & b.right.mask)
43:        end if
44:    end if
45: end procedure
```

Algorithm 5 Suppression d'un bloc B, modifie le Ptree $root$

Require: $root \neq emptyPtree \wedge B \neq noBlock$

```
1:  procedure REM(Ptree root, Block B)
    Ptree x, b, f, g
2:      x ← root
3:      while ¬x.isLeaf do
4:          if x.right.addr & x.right.mask == B.ptr & x.right.mask then
5:              x ← x.right
6:          else
7:              x ← x.left
8:          end if
9:      end while
10:     f ← x.father
11:     if f == emptyPtree then
12:         root ← emptyPtree
13:     else
14:         b ← (x == f.left) ? f.right : f.left
15:         f ← b
16:         if ¬b.isLeaf then
17:             b.left.father ← f
18:             b.right.father ← f
19:         end if
20:         g ← f.father
21:         if g ≠ emptyPtree then
22:             g.mask ← MPGPC(g.left.addr & g.left.mask, g.right.addr & g.right.mask)
23:         end if
24:     end if
25: end procedure
```

6

DÉCOUVERTE DE NON-CONFORMITÉS

Il est souvent difficile de déterminer la raison de l'échec de la vérification déductive d'un programme C annoté en E-ACSL. Dans ce chapitre et le suivant, on suppose que l'échec de vérification ne vient pas des limitations du prouveur. L'échec vient donc de problèmes dans le code et/ou dans sa spécification. On distingue deux sortes de problèmes, de nature très différente : la non-conformité entre un code et sa spécification, approfondie dans ce chapitre, et la faiblesse de spécification, étudiée dans le chapitre suivant. La non-conformité entre un code et sa spécification est une notion de correction relative entre code et spécification. Ce n'est que lorsqu'une spécification et un code sont corrects, c'est-à-dire conformes, qu'il est pertinent d'étudier le second problème, celui de la précision suffisante de certaines annotations pour permettre d'en démontrer d'autres. C'est pourquoi cette seconde question – de complétude de la spécification – est étudiée après la question de non-conformité étudiée ici.

Il n'est pas assez précis de parler d'erreur dans le code et/ou dans sa spécification. En effet, on ne s'intéresse pas ici directement à la détection dans le code d'erreurs dites "à l'exécution" (*runtime error*), telles que les divisions par zéro ou les accès à un tableau hors de ses bornes. On ne s'y intéresse qu'indirectement, par vérification déductive d'assertions précédant immédiatement un code susceptible de générer une telle erreur. Une telle assertion doit être fausse lorsque les conditions de déclenchement de l'erreur sont réunies. On ne considère donc que des erreurs concernant le code **et** sa spécification. De plus, parmi ces erreurs, on ne considère que celles que nous pouvons détecter par analyse dynamique d'une instrumentation du programme annoté, c'est-à-dire qu'elles correspondent à la violation d'une annotation du fragment d'E-ACSL défini au chapitre 3. Ce sont ces erreurs que nous appelons "non-conformités".

Nous proposons plusieurs scénarios de détection de non-conformités entre un code et sa spécification, par instrumentation du programme annoté et génération de tests structurels pour cette instrumentation. La partie 6.1 présente la méthode de génération de tests utilisée par PATHCRAWLER, que nous utilisons dans nos travaux mais qui n'est pas une de nos contributions. La partie 6.2 définit la notion de non-conformité. La partie 6.3 présente l'exemple que nous utilisons tout au long de ce chapitre pour illustrer nos propos. La partie 6.4 illustre la détection de non-conformités dans deux scénarios construits sur l'exemple présenté en partie 6.3.

6.1 GÉNÉRATION DE TESTS AVEC PATHCRAWLER

PATHCRAWLER [Williams et al., 2004, Botella et al., 2009] est un outil de génération de tests structurels pour les programmes C. PATHCRAWLER utilise la technique d'exécution symbolique dynamique, ou exécution "concolique", qui associe l'exécution concrète du programme et l'exécution symbolique afin d'explorer les chemins du programme.

Étant donné un programme C sous test P et une précondition sur ses entrées, PATHCRAWLER génère des cas de test respectant un critère de couverture de test. Le critère "tous les chemins" impose une couverture de tous les chemins faisables de P. L'exploration exhaustive de tous les chemins étant en pratique irréalisable sur des programmes réels, le critère "tous les k-chemins" a été défini. Il limite l'exploration aux chemins qui ont au plus k itérations consécutives de chaque boucle.

PATHCRAWLER commence par construire une version instrumentée de P permettant de tracer l'exécution de chaque cas de test. Puis il génère les contraintes représentant la sémantique de chaque instruction de P. L'étape suivante est la génération et la résolution de contraintes pour produire les cas de test pour un ensemble de chemins satisfaisant le critère de couverture. La résolution de contraintes s'effectue à l'aide d'ECLIPSE PROLOG [Schimpf et al., 2010], un environnement de programmation en logique par contraintes basé sur PROLOG. Étant donné un préfixe de chemin π, c'est-à-dire un chemin partiel de P, l'idée est de résoudre les contraintes correspondant à l'exécution symbolique de P en suivant le chemin π.

La méthode de génération de tests pour le critère "tous les chemins" est composée des étapes suivantes :

($\mathcal{G}_1$) Création d'une variable logique pour chaque entrée. Prise en compte des contraintes de la précondition. Le préfixe de chemin initial π est vide. Aller à ($\mathcal{G}_2$).

($\mathcal{G}_2$) Exécuter symboliquement le chemin π : ajout des contraintes et mise à jour de la mémoire en fonction des instructions de π. Si le solveur de contraintes détermine que certaines contraintes sont insatisfaisables, aller à ($\mathcal{G}_5$). Sinon, aller à ($\mathcal{G}_3$).

($\mathcal{G}_3$) Appeler le solveur de contraintes pour générer un cas de test t satisfaisant les contraintes du chemin partiel courant π. Si les contraintes sont insatisfaisables, aller à ($\mathcal{G}_5$). Sinon, aller à ($\mathcal{G}_4$).

($\mathcal{G}_4$) Exécuter le programme avec trace sur le cas de test t généré pour obtenir le chemin d'exécution, qui doit commencer par π. Aller à ($\mathcal{G}_5$).

($\mathcal{G}_5$) Calculer le prochain chemin partiel π à couvrir. Un parcours en profondeur détermine la dernière décision d pour laquelle il reste une branche à explorer. S'il n'existe pas une telle décision, l'algorithme s'arrête. Sinon, π est recalculé et contient maintenant le chemin partiel précédent dans lequel les contraintes correspondant à d ont été niées, et retour à l'étape ($\mathcal{G}_2$). Cela nous assure que tous les chemins faisables sont couverts (en considérant que le solveur de contraintes peut trouver une solution dans un temps raisonnable) et que seulement le plus court des préfixes infaisables de chaque chemin infaisable est exploré [Botella et al., 2009].

Cette méthode s'adapte facilement au critère de couverture "tous les k-chemins" en évitant d'explorer les chemins avec $\geq k+1$ itérations consécutives pour une boucle.

6.2 DÉFINITION DE LA NON-CONFORMITÉ

Soit P un programme C annoté avec E-ACSL, et f la fonction sous vérification de P. La fonction f est supposée non-récursive. Elle peut appeler d'autres fonctions, notons g l'une d'entre elles. Un cas de test V pour f est un vecteur de valeurs pour chaque variable d'entrée de f. Le chemin de programme activé par un cas de test V, noté π_V, est la séquence d'instructions exécutées par le programme P sur le cas de test V. Nous utilisons le terme générique de contrat pour désigner l'ensemble des annotations E-ACSL décrivant une boucle ou une fonction. Un contrat de fonction est composé de pré- et postconditions incluant les clauses `requires`, `assigns` et `ensures`. Un contrat de boucle est composé des clauses `loop invariant`, `loop variant` et `loop assigns`.

Le chapitre 4 a présenté la transformation d'un programme P annoté avec E-ACSL en un programme instrumenté (dénoté P^{NC} dans la suite) sur lequel nous pouvons appliquer la génération de tests pour produire des données de test violant des annotations à l'exécution. P^{NC} vérifie toutes les annotations de P aux endroits correspondants dans le programme et reporte toute violation d'annotation éventuelle. Par exemple, la postcondition $Post_f$ de f est évaluée par le code suivant, inséré à la fin de la fonction f dans P^{NC} :

```
int post_f;Spec2Code(Post_f, post_f); fassert(post_f);
```
(6.1)

Pour un prédicat E-ACSL `P`, nous notons par *Spec2Code*`(P, b)` le code C généré qui évalue le prédicat `P` et donne sa valeur de vérité à la variable booléenne `b`. De manière similaire, les préconditions et postconditions d'une fonction appelée g sont évaluées respectivement avant et après l'exécution de la fonction g. Un invariant de boucle est vérifié avant la boucle (on dit qu'il est établi) et après chaque pas de la boucle (on dit qu'il est préservé par l'itération). Une assertion est vérifiée à sa position courante. Afin de produire uniquement des données de test qui respectent la précondition Pre_f de f, cette précondition est testée au début de f de sorte à admettre la précondition :

```
int pre_f;Spec2Code(Pre_f, pre_f); fassume(pre_f);
```
(6.2)

Définition 4 : Non-conformité

Il y a une non-conformité entre le code d'une fonction f et sa spécification dans le programme P s'il existe un cas de test V pour f respectant sa précondition, tel que P^{NC} produit une violation d'annotation à partir de V. On dit que V est un contre-exemple de non-conformité (ou NCCE pour *non-compliance counter-example*).

La génération de tests peut être utilisée sur le programme instrumenté P^{NC} pour générer des NCCEs. Nous appelons cette technique Détection de Non-Conformité (ou NCD pour *Non-Compliance Detection*). Dans nos travaux, nous utilisons le générateur de tests PATHCRAWLER qui essaie de couvrir tous les chemins d'exécution faisables du programme. Étant donné que l'étape de traduction a ajouté des branches dans le programme afin d'expliciter la violation des annotations, PATHCRAWLER essaie de couvrir au moins un chemin d'exécution où l'annotation n'est pas vérifiée.

```
1  int delete_substr(char *str, int strlen, char *substr, int sublen, char *dest) {
2   int start = find_substr(str, strlen, substr, sublen), j, k;
3   if (start == -1) {
4    for (k = 0; k < strlen; k++) dest[k] = str[k];
5    return 0;
6   }
7   for (j = 0; j < start; j++) dest[j] = str[j];
8   for (j = start; j < strlen-sublen; j++) dest[j] = str[j+sublen];
9   return 1;
10 }
```

FIGURE 6.1 – Fonction `delete_substr` non spécifiée, appelant la fonction de la figure 6.2

```
1  /*@ requires 0 < sublen <= strlen;
2    @ requires \valid(str+(0..strlen-1)) && \valid(substr+(0..sublen-1));
3    @ assigns \nothing;
4    @ behavior found:
5    @   assumes \exists integer i; 0 <= i < strlen-sublen &&
6    @     (\forall integer j; 0 <= j < sublen ==> str[i+j] == substr[j]);
7    @   ensures 0 <= \result < strlen-sublen;
8    @   ensures \forall integer j; 0 <= j < sublen ==> str[\result+j] == substr[j];
9    @ behavior not_found:
10   @   assumes \forall integer i; 0 <= i < strlen-sublen ==>
11   @     (\exists integer j; 0 <= j < sublen && str[i+j] != substr[j]);
12   @   ensures \result == -1; */
13 int find_substr(char *str, int strlen, char *substr, int sublen);
```

FIGURE 6.2 – Contrat E-ACSL de la fonction vérifiée `find_substr`

L'étape NCD peut produire trois résultats. Elle retourne un triplet (nc, V, a) si un NCCE V a été trouvé, indiquant que le chemin de programme π_V activé par V sur P^{NC} provoque une violation de l'annotation a. Si NCD a exploré tous les chemins du programme sans trouver de NCCE, elle retourne "no". Sinon, si la couverture des chemins du programme par NCD est partielle (si un *timeout* a été atteint par exemple), elle retourne " ?", ce qui signifie qu'elle n'a pas été en mesure de conclure.

6.3 EXEMPLE ILLUSTRATIF

Supposons qu'Alice est une ingénieure validation en charge de la spécification et de la vérification déductive de la fonction `delete_substr` de la figure 6.1. Dans ce chapitre, nous suivons Alice pendant le processus de validation afin d'illustrer les problèmes et les différents choix possibles que rencontre un ingénieur validation en charge d'une preuve de programme.

Le comportement attendu de la fonction `delete_substr` est de supprimer une occurrence de la sous-chaîne `substr` de longueur `sublen` au sein d'une chaîne `str` de longueur `strlen`. Le résultat de cette opération est affecté à la chaîne `dest` de longueur `strlen`, allouée au préalable. Les chaînes `str` et `substr` ne doivent pas être modifiées. Pour des raisons de simplicité, nous utilisons des tableaux plutôt que des chaînes de caractères "à la C", terminées par un 0. La fonction `delete_substr` retourne 1 si une occurrence de la sous-chaîne a été trouvée et supprimée, et 0 sinon. La figure 6.2 présente le contrat de la fonction `find_substr`, appelée par la fonction `delete_substr`, qui retourne l'index d'une occurrence de `substr` dans `str` si la sous-chaîne est présente, et -1 sinon. Nous supposons qu'Alice a déjà prouvé avec succès la correction de la fonction `find_substr` vis-à-vis de son contrat.

6.4 SCÉNARIOS DE DÉTECTION DE NON-CONFORMITÉS

Pendant le processus de spécification et de vérification déductive, la génération de tests peut fournir automatiquement à l'ingénieur validation un retour rapide et pertinent afin de l'aider dans son travail. En effet, la génération de tests peut trouver un contre-exemple mettant en évidence le non respect de la spécification par le code ou l'inverse. Ceci est applicable dès le début du processus de spécification, lorsque la vérification déductive ne peut pas aboutir car le programme à vérifier n'est pas entièrement spécifié (il manque par exemple des contrats de boucle et de fonctions appelées). La génération de tests peut également être utile dans le cas d'un échec de preuve concernant une propriété de programme, lorsque l'ingénieur validation n'a pas d'autre alternative que d'analyser manuellement le programme afin de découvrir la raison de cet échec.

L'absence de NCCE après une exploration partielle (ou, si possible, complète) des chemins du programme donne à l'ingénieur validation une confiance accrue (respectivement une garantie) quant à la conformité du code vis-à-vis de la spécification. Cette information pousse l'ingénieur à penser que l'échec de la preuve est dû soit à un manque dans la spécification, soit à une annotation trop faible, plutôt qu'à une erreur, ce qui l'incite à annoter davantage le programme.

Dans le cas contraire, un contre-exemple met immédiatement en évidence le non-respect de la spécification par le programme (ou l'inverse), signifiant pour l'ingénieur validation que cette non-conformité doit être corrigée avant de poursuivre la preuve. Remarquons que l'objectif n'est certainement pas de faire correspondre à tout prix la spécification à un code (potentiellement erroné), mais d'aider l'ingénieur validation à identifier le problème (dans la spécification ou dans le code).

Nous illustrons maintenant nos propos sur des scénarios de vérification concrets que l'ingénieure validation Alice pourrait rencontrer.

6.4.1 VALIDATION DES CONTRATS DE LA FONCTION SOUS VÉRIFICATION

Supposons qu'Alice écrive tout d'abord la précondition suivante (ajoutée avant la ligne 1 de la figure 6.1) :

```
requires 0 < sublen ≤ strlen;
requires \valid(str+(0..strlen-1));
requires \valid(dest+(0..strlen-1));
requires \valid(substr+(0..sublen-1));
requires \separated(dest+(0..strlen-1), substr+(0..sublen-1));
requires \separated(dest+(0..strlen-1), str+(0..strlen-1));
typically strlen ≤ 5;
```

Nous rappelons que la clause `typically p;` étend E-ACSL et définit une précondition `p` qui n'est prise en compte que par la génération de tests. Cette clause permet à Alice de renforcer la précondition si elle souhaite restreindre l'exploration des chemins du programme, en diminuant leur nombre. La couverture des chemins du programme est alors partielle et contrôlée par l'utilisateur. Ici, la clause `typically strlen <= 5` contraint la génération de tests à ne considérer que les chemins faisables pour lesquels la chaîne `str` contient au plus cinq caractères. Étant ignorée par la vérification déductive, cette clause n'impacte pas la preuve du programme.

Puis Alice spécifie que la fonction peut uniquement modifier le contenu du tableau `dest`, et définit la postcondition dans le cas où la sous-chaîne n'est pas présente dans la chaîne `str`. Elle ajoute les clauses (erronées) suivantes au contrat de la fonction, après la précondition :

```
assigns dest[0..strlen-1];
behavior not_present:
 assumes !(\exists integer i; 0 ≤ i < strlen-sublen ∧
  (\forall integer j; 0 ≤ j < sublen⇒ str[i+j] ≠ substr[j]));
 ensures \forall integer k; 0 ≤ k < strlen⇒ \old(str[k]) == dest[k];
 ensures \result == 0;
```

Afin de valider cette postcondition avant d'aller plus loin, Alice applique la méthode NCD. La génération de tests reporte que les deux clauses **ensures** sont invalidées par le contre-exemple suivant : `strlen = 2`, `sublen = 1`, `str[0] = 'A'`, `str[1] = 'B'`, `substr[0] = 'A'`, `dest[0] = 'B'` and `\result = 0`. Alice remarque que dans ce cas, la chaîne `substr` est présente dans la chaîne `str`. Le comportement `not_present` ne devrait donc pas s'appliquer, ce qui indique que l'erreur vient de la clause **assumes**. Ceci aide Alice à corriger sa spécification en corrigeant cette clause en remplaçant ≠ par ==, pour obtenir :

```
 assumes !(\exists integer i; 0 ≤ i < strlen-sublen ∧
  (\forall integer j; 0 ≤ j < sublen⇒ str[i+j] == substr[j]));
```

Exécuter NCD une seconde fois informe Alice que tous les chemins faisables pour lesquels `strlen <= 5` ont été couverts (en 3.4 secondes), que 9442 cas de tests ont été générés et qu'aucune violation d'annotation n'a été observée. Alice a maintenant confiance que le comportement `not_present` est correctement défini.

Concernant le comportement complémentaire, `present`, Alice copie le comportement `not_present` et le modifie (avec erreur). Elle écrit par exemple ceci :

```
behavior present:
 assumes \exists integer i; 0 ≤ i < strlen-sublen ∧
  (\forall integer j; 0 ≤ j < sublen⇒ str[i+j] == substr[j]);
 ensures \exists integer i; 0 ≤ i < strlen-sublen ∧
  (\forall integer j; 0 ≤ j < sublen⇒ \old(str[i+j]) == \old(substr[j])) ∧
  (\forall integer k; 0 ≤ k < i⇒ \old(str[k]) == dest[k]) ∧
  (\forall integer l; i ≤ l < strlen⇒ \old(str[l+sublen]) == dest[l]);
 ensures \result == 1;
```

La détection de non-conformité rapporte une erreur à l'exécution lors de l'accès à l'élément de la chaîne `str` à l'indice `l+sublen` dans l'avant-dernière clause **ensures**. Cette information permet à Alice de comprendre que la borne supérieure de l'indice `l` devrait être `strlen-sublen` au lieu de `strlen`. Elle corrige cette erreur et ré-applique la NCD. La génération de tests rapporte que 13448 cas de tests couvrent sans erreur les chemins faisables pour lesquels `strlen <= 5`. Alice est maintenant satisfaite avec les comportements du programme ainsi définis.

6.4.2 VALIDATION DES SPÉCIFICATIONS DE BOUCLES

Alice spécifie maintenant la première boucle **for** de la ligne 4 de la figure 6.1 :

```
loop invariant \forall integer m; 0 ≤ m < k⇒ dest[m] == \at(str[m],Pre);
loop assigns k, dest[0..strlen-1];
loop variant strlen-k;
```

La vérification déductive du programme échoue à prouver la postcondition de la fonction `delete_substr`, notamment parce que les deux autres boucles du programme aux lignes 7

et 8 ne sont pas spécifiées. En revanche, Alice s'attend à ce que la vérification déductive de ce premier contrat de boucle réussisse. Ce n'est malheureusement pas le cas et Alice n'arrive pas à déterminer si cet échec de la preuve est dû à une non-conformité du code de la boucle vis-à-vis de son contrat, ou parce que le contrat de boucle n'est pas assez précis pour être vérifié automatiquement.

NCD ne détecte aucune erreur dans le contrat de boucle et la postcondition, couvrant le programme avec 15635 cas de tests. Ces résultats poussent Alice à penser que la spécification de la boucle est valide mais incomplète, ce qui la conduit à ajouter un invariant de boucle supplémentaire, définissant les bornes de k :

```
loop invariant 0 ≤ k < strlen;
```

Alice réessaie à présent de prouver la boucle, et la vérification déductive de WP échoue une fois encore. Elle ré-applique NCD et cette fois le nouvel invariant de boucle est invalidé. Après analyse de cet échec sur le contre-exemple produit, Alice comprend que l'invariant de boucle `k < strlen` n'est pas correct. En effet, `k` est égal à `strlen` après la dernière itération de la boucle. L'invariant de boucle devrait alors être `k <= strlen`. Après correction de l'erreur, le contrat de boucle est prouvé avec succès par WP. De manière similaire, Alice spécifie et vérifie de manière itérative les deux autres boucles du programme.

La fonction `delete_substr` étant maintenant entièrement spécifiée (figure 6.3), elle peut être prouvée par WP. Cependant le *timeout* par défaut du prouveur (10 secondes par propriété) doit être étendu de manière significative (par exemple 50 secondes par propriété). Lorsque la preuve du programme n'aboutit pas car un *timeout* a été atteint, l'ingénieur validation doit décider s'il souhaite retenter la preuve avec un *timeout* plus élevé ou s'il analyse manuellement le programme afin de trouver une non-conformité ou une faiblesse de contrat. Le premier choix peut causer une perte de temps importante à l'ingénieur qui peut être évitée si NCD parvient à produire un contre-exemple exhibant une non-conformité. Dans ce cas, NCD peut aider l'ingénieur validation à se décider. Le fait que la génération de tests parvienne (en 4 secondes) à couvrir une portion significative des chemins du programme (restreinte par la clause `typically`) sans qu'aucune non-conformité ne soit rapportée convainc Alice d'augmenter le *timeout*. Attention toutefois, car le risque que le contrat soit trop faible pour la vérification déductive du programme n'a pas été écarté par NCD.

CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons défini la notion de non-conformité entre un code et sa spécification. Nous avons montré l'intérêt de la détection de non-conformité pour effectuer des preuves de programmes. Nous avons montré que la détection de non-conformité par génération de tests rendue possible par la traduction des annotations permettait d'aborder la preuve de manière incrémentale et itérative. Nous avons illustré la démarche sur l'exemple de la figure 6.1. Cette démarche a abouti au code et à la spécification de la figure 6.3 qui est prouvée automatiquement avec le greffon WP de FRAMA-C [1]. En effet, il est possible de spécifier partiellement un programme puis de détecter si le code est conforme à cette partie avant de poursuivre le travail de spécification. La preuve peut

1. nécessite l'option `-wp-split`, FRAMA-C Sodium, ALT-ERGO 0.95.2 et CVC3 2.4.1

être décomposée en une succession d'étapes de ce type.

En revanche, ce chapitre n'aborde pas le cas où un contrat d'une boucle ou d'une fonction appelée (que nous appelerons sous-contrat) est trop faible pour prouver entièrement le programme. Cette difficulté du processus de spécification et de vérification est traitée au chapitre 7, qui présente la détection de faiblesses de sous-contrats. Le chapitre 8 montrera que la combinaison de ces deux méthodes peut améliorer l'aide au diagnostic des échecs de preuve lors de la vérification déductive de programmes.

```
/*@ requires 0 < sublen <= strlen;
  @ requires \valid(str+(0..strlen-1)) && \valid(substr+(0..sublen-1));
  @ assigns \nothing;
  @ behavior present:
  @  assumes \exists integer i; 0 <= i < strlen-sublen &&
  @    (\forall integer j; 0 <= j < sublen ==> str[i+j] == substr[j]);
  @  ensures 0 <= \result < strlen-sublen;
  @  ensures \forall integer j; 0 <= j < sublen ==> str[\result+j] == substr[j];
  @ behavior not_present:
  @  assumes \forall integer i; 0 <= i < strlen-sublen ==>
  @    (\exists integer j; 0 <= j < sublen && str[i+j] != substr[j]);
  @  ensures \result == -1; */
int find_substr(char *str, int strlen, char *substr, int sublen);

/*@ requires 0 < sublen <= strlen;
  @ requires \valid(str+(0..strlen-1));
  @ requires \valid(dest+(0..strlen-1));
  @ requires \valid(substr+(0..sublen-1));
  @ requires \separated(dest+(0..strlen-1), substr+(0..sublen-1));
  @ requires \separated(dest+(0..strlen-1), str+(0..strlen-1));
  @ assigns dest[0..strlen-1];
  @ behavior not_present:
  @  assumes !(\exists integer i; 0 <= i < strlen-sublen &&
  @    (\forall integer j; 0 <= j < sublen ==> str[i+j] == substr[j]));
  @  ensures (\forall integer k; 0 <= k < strlen ==> \old(str[k]) == dest[k]);
  @  ensures \result == 0;
  @ behavior present:
  @  assumes \exists integer i; 0 <= i < strlen-sublen &&
  @    (\forall integer j; 0 <= j < sublen ==> str[i+j] == substr[j]);
  @  ensures \exists integer i; 0 <= i < strlen-sublen &&
  @    (\forall integer j; 0 <= j < sublen ==> \old(str[i+j]) == \old(substr[j])) &&
  @    (\forall integer k; 0 <= k < i ==> \old(str[k]) == dest[k]) &&
  @    (\forall integer l; i <= l < strlen-sublen ==> \old(str[l+sublen]) == dest[l]);
  @  ensures \result == 1; */
int delete_substr(char *str, int strlen, char *substr, int sublen, char *dest) {
 int start = find_substr(str, strlen, substr, sublen), j, k;
 if(start == -1) {
  /*@ loop invariant \forall integer m; 0 <= m < k ==> dest[m] == \at(str[m],Pre);
    @ loop invariant 0 <= k <= strlen;
    @ loop assigns k, dest[0..strlen-1];
    @ loop variant strlen-k; */
  for(k = 0; k < strlen; k++) dest[k] = str[k];
  return 0;
 }
 /*@ loop invariant \forall integer m; 0 <= m < j ==> dest[m] == \at(str[m],Pre);
   @ loop invariant 0 <= j <= start;
   @ loop assigns j, dest[0..start-1];
   @ loop variant start-j; */
 for(j = 0; j < start; j++) dest[j] = str[j];
 /*@ loop invariant \forall integer m; start <= m < j ==>
   @                    \at(str[m+sublen],Pre) == dest[m];
   @ loop invariant \forall integer i; 0 <= i < strlen ==>
   @                   str[i] == \at(str[i], Pre);
   @ loop invariant start <= j <= strlen-sublen;
   @ loop assigns dest[start..strlen-sublen-1], j;
   @ loop variant strlen-sublen-j; */
 for(j = start; j < strlen-sublen; j++) dest[j] = str[j+sublen];
 return 1;
}
```

FIGURE 6.3 – Fonction `delete_substr` spécifiée et prouvée automatiquement

7

DÉCOUVERTE DE FAIBLESSES DE SOUS-CONTRATS

Nous avons vu au chapitre précédent que la méthode de détection de non-conformités par génération de tests peut n'en trouver aucune alors que la preuve échoue. Dans ce cas il est possible que la preuve échoue soit parce que le prouveur n'est pas assez puissant pour mener la preuve à terme, soit parce qu'il ne dispose pas de suffisamment d'hypothèses à partir des sous-contrats (contrats de boucles et contrats des fonctions appelées). En effet, il se peut par exemple qu'un invariant de boucle soit prouvé mais qu'il soit insuffisant pour prouver la postcondition de la fonction sous test. Dans ce cas, nous disons qu'il y a une faiblesse du sous-contrat. Nous utilisons la génération de tests pour exhiber ces faiblesses.

Dans ce chapitre, nous présentons la notion de faiblesse de l'ensemble des sous-contrats et de faiblesse d'un sous-contrat, pour les sous-contrats de la fonction sous vérification que sont les contrats de boucles et les contrats de fonctions appelées. Nous présentons également les spécificités de la traduction des annotations pour la détection des faiblesses de sous-contrats par rapport à la traduction pour la détection de non-conformités présentée au chapitre 4.

La partie 7.1 présente la notion de faiblesse de l'ensemble des sous-contrats et la partie 7.2 présente la notion de faiblesse d'un sous-contrat. La partie 7.3 continue le cas pratique du chapitre 6 pour montrer l'intérêt de la détection des faiblesses des sous-contrats.

7.1 DÉTECTION DE FAIBLESSE DE L'ENSEMBLE DES SOUS-CONTRATS

Avant de présenter nos définitions de faiblesses de sous-contrat, nous avons besoin de définir des concepts supplémentaires. Une boucle (respectivement une fonction, une assertion) non-imbriquée de f est une boucle (respectivement une fonction appelée, une assertion) de f en dehors de toute boucle de f. Un sous-contrat de f est le contrat d'une boucle ou d'une fonction non-imbriquée de f. Une annotation non-imbriquée de f est soit une assertion non-imbriquée soit une annotation d'un sous-contrat de f. Par exemple, la fonction `delete_substr` de la figure 6.3 du chapitre 6 a quatre sous-contrats : le contrat de la fonction appelée `find_substr` et le contrat de chacune des trois boucles de la fonction `delete_substr`. Les contrats éventuels des boucles de `find_substr` sont des sous-contrats

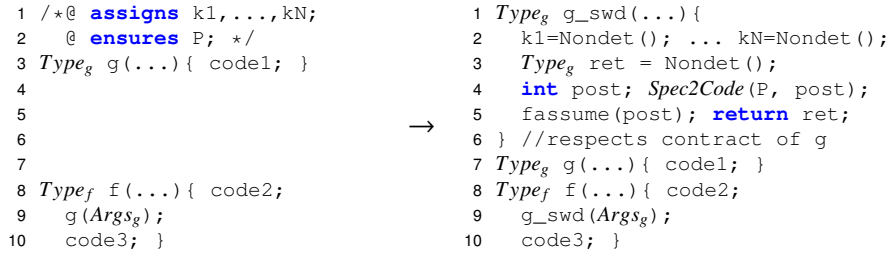

FIGURE 7.1 – (a) Un contrat $c \in C$ d'une fonction g appelée par f, et (b) sa traduction pour la détection des faiblesses de sous-contrats de fonction appelée

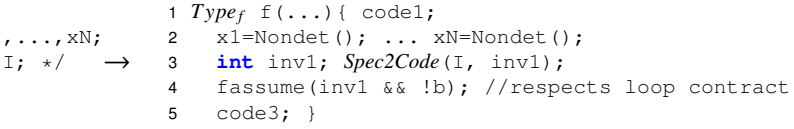

FIGURE 7.2 – (a) Un contrat $c \in C$ d'une boucle de f, et (b) sa traduction pour la détection des faiblesses de sous-contrats de boucle

de `find_substr` mais pas des sous-contrats de `delete_substr`.

Nous nous concentrons sur les annotations non-imbriquées de f et supposons que tous les sous-contrats de f sont respectés : les fonctions appelées par f respectent leur contrat et les boucles de f préservent leurs invariants et respectent les annotations imbriquées. Soit c_f le contrat de f, C l'ensemble des sous-contrats non-imbriqués de f, et $\mathcal{A}$ l'ensemble des annotations non-imbriquées de f et des annotations de c_f. En d'autres termes, $\mathcal{A}$ contient les annotations des contrats dans $C \cup \{c_f\}$, ainsi que les assertions non-imbriquées de f. Nous supposons également que chaque sous-contrat de f contient une clause **`assigns`** ou **`loop assigns`**. Cette hypothèse n'est pas limitative puisqu'une telle clause est de toute manière nécessaire à la preuve de toute fonction ou boucle non triviale.

Afin d'appliquer la génération de tests aux contrats des fonctions appelées et des boucles de C au lieu de leur code, nous utilisons une transformation de programme qui produit un nouveau programme P^{GSW} (pour *global subcontract weakness*) à partir de P. Le code de chaque appel de fonction et de chaque boucle de f est remplacé par un nouveau code dérivé de leur contrat de la manière suivante.

Pour un contrat $c \in C$ d'une fonction `g` appelée par `f`, la transformation de programme (illustrée par la figure 7.1) génère une nouvelle fonction `g_swd` ayant la même signature et dont le code simule tous les comportements possibles respectant la postcondition du contrat c. Chaque appel à `g` dans `f` est remplacé par un appel à `g_swd`. Tout d'abord, `g_swd` permet aux variables (et plus généralement aux left-values) présentes dans la clause **`assigns`** de c de changer de valeur (ligne 2 de la figure 7.1(b)). Ceci est possible en affectant une valeur non-déterministe du type adéquat, dont la génération est ici notée par `Nondet()` (ou simplement en utilisant un tableau contenant les variables d'entrée fraîches, dans lequel on va lire de nouvelles valeurs pour chaque appel de la fonction `g`). La règle α-ASSUME-ASSIGNS de la figure 7.3 formalise la traduction de la clause **`assigns`** où $X = \{x_1, ..., x_m\}$. Si le type de retour de `g` n'est pas **`void`**, une valeur non-déterministe supplémentaire est générée pour la valeur de retour `ret` (ligne 3 de la figure 7.1(b)). Enfin, la validité de la postcondition est évaluée (en prenant en compte ces nouvelles valeurs

$$\alpha\text{-ASSUME-ASSIGNS}\quad \frac{(End_f, x_1) \overset{\tau}{\mapsto} (I_1, e_1) \quad \dots \quad (End_f, x_m) \overset{\tau}{\mapsto} (I_m, e_m)}{(End_f, \texttt{assigns}\ X\texttt{;}) \overset{\alpha}{\mapsto} I_1 \cdot (End_f, e_1 \texttt{ = Nondet();}) \cdot \dots \cdot I_m \cdot (End_f, e_m \texttt{ = Nondet();})}$$

$$\alpha\text{-ASSUME-POST}\quad \frac{(End_f, p) \overset{\pi}{\mapsto} (I, e)}{(End_f, \texttt{ensures}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (End_f, \texttt{fassume(}e\texttt{);})}$$

FIGURE 7.3 – Règles de traduction pour la détection des faiblesses de sous-contrats des fonctions appeleés

$$\alpha\text{-ASSUME-LOOP-ASSIGNS}\quad \frac{(l, x_1) \overset{\tau}{\mapsto} (I_1, e_1) \quad \dots \quad (l, x_m) \overset{\tau}{\mapsto} (I_m, e_m)}{(l, \texttt{loop assigns}\ X\texttt{;}) \overset{\alpha}{\mapsto} I_1 \cdot (l, e_1 \texttt{ = Nondet();}) \cdot \dots \cdot I_m \cdot (l, e_m \texttt{ = Nondet();})}$$

$$\alpha\text{-ASSUME-INVARIANT}\quad \frac{(l, p) \overset{\pi}{\mapsto} (I, e)}{(l, \texttt{loop invariant}\ p\texttt{;}) \overset{\alpha}{\mapsto} I \cdot (l, \texttt{fassume(}e\ \texttt{\&\&}\ \texttt{!}loopcond\texttt{);})}$$

FIGURE 7.4 – Règles de traduction pour la détection des faiblesses de sous-contrats des boucles

non-déterministes) et admise afin de ne considérer que les exécutions de la fonction qui respectent cette postcondition, puis la fonction termine son exécution (lignes 4–5 de la figure 7.1(b)). La règle α-ASSUME-POST de la figure 7.3 formalise la traduction de la postcondition.

De même, pour un contrat $c \in C$ d'une boucle de `f`, la transformation de programme remplace le code de la boucle par un autre code qui simule tous les comportements possibles respectant c, c'est-à-dire, s'assurant de la validité de la "postcondition de boucle" $I \wedge \neg b$ après la boucle, comme illustré sur la figure 7.2. Les règles α-ASSUME-LOOP-ASSIGNS et α-ASSUME-INVARIANT de la figure 7.4 formalisent respectivement la traduction des clauses `loop assigns` et `loop invariant`. Dans la règle α-ASSUME-LOOP-ASSIGNS on a : $X = \{x_1, ..., x_m\}$. La règle α-ASSUME-INVARIANT génère un code qui calcule la valeur de l'invariant p dans la variable e et fait l'hypothèse que la condition $I \wedge \neg b$ codée par e `&&` $!loopcond$ où $loopcond$ désigne la condition de la boucle sur laquelle porte l'invariant p. La transformation de programme traite toute autre annotation de $\mathcal{A}$ de la même manière que P^{NC} : préconditions des fonctions appelées, vérifications initiales des invariants de boucle et pré- et postcondition de `f` (qui n'apparaissent pas dans les figures 7.1 et 7.2).

Définition 5 : Faiblesse de l'ensemble des sous-contrats

Soit P un programme et f une fonction de P. Le programme P a une faiblesse de l'ensemble des sous-contrats de f s'il existe un cas de test V de f respectant sa précondition, tel que P^{NC} ne rapporte aucune violation d'annotation sur V, alors que P^{GSW} rapporte une violation d'annotation sur V. Dans ce cas, on dit que V est un contre-exemple de faiblesse de l'ensemble des sous-contrats (ou GSWCE pour *global subcontract weakness counter-example*) pour l'ensemble des sous-contrats de f.

Nous ne considérons pas qu'un même contre-exemple puisse être à la fois un contre-exemple de non-conformité et un contre exemple de faiblesse de sous-contrat. En effet, même si certains contre-exemples peuvent illustrer à la fois une non-conformité et une faiblesse de sous-contrat, une non-conformité provient généralement d'un conflit direct entre le code et sa spécification et doit être corrigée en priorité, tandis qu'une faiblesse de contrat est souvent plus subtile et ne doit être corrigée qu'une fois toutes les non-

```
1 int x;
2 /*@ ensures x >= \old(x)+1; assigns x; */
3 void g1() { x=x+2; }
4 /*@ ensures x >= \old(x)+1; assigns x; */
5 void g2() { x=x+2; }
6 /*@ ensures x >= \old(x)+1; assigns x; */
7 void g3() { x=x+2; }
8 /*@ ensures x >= \old(x)+4; assigns x; */
9 void f() { g1(); g2(); g3(); }
```

Listing 7.1 – Faiblesse de l'ensemble des sous-contrats

```
1 int x;
2 /*@ ensures x >= \old(x)+1; assigns x; */
3 void g1() { x=x+1; }
4 /*@ ensures x >= \old(x)+1; assigns x; */
5 void g2() { x=x+1; }
6 /*@ ensures x >= \old(x)+1; assigns x; */
7 void g3() { x=x+2; }
8 /*@ ensures x >= \old(x)+4; assigns x; */
9 void f() { g1(); g2(); g3(); }
```

Listing 7.2 – Faiblesse d'un sous-contrat

conformités éliminées.

Encore une fois, la génération de tests peut être appliquée sur P^{GSW} pour générer des GSWCE potentiels. Lorsqu'un cas de test V a été trouvé tel que P^{GSW} échoue sur V, nous appliquons la validation à l'exécution : si P^{NC} échoue sur V, alors V est un NCCE, sinon V est un GSWCE. Nous appelons cette technique Détection de faiblesse de l'ensemble des sous-contrats, notée GSWD pour *Global Subcontract Weakness Detection*. GSWD peut produire quatre résultats : (nc, V, a), (sw, V, a, C), "no" ou " ?". Elle retourne (nc, V, a) si un NCCE V a été trouvé pour l'annotation a, et (sw, V, a, C) si V a été classifié en tant que GSWCE indiquant que l'ensemble des sous-contrats C est trop faible pour vérifier l'annotation a. Le chemin de programme π_V activé par le cas de test V et menant à l'erreur (sur P^{NC} ou P^{GSW}) est également enregistré. Si GSWD a réussi à explorer l'intégralité des chemins du programme sans trouver de GSWCE, elle retourne no. Sinon, si l'exploration des chemins du programme est seulement partielle, et qu'aucun contre-exemple n'a été trouvé, elle retourne ? (*unknown*).

7.2 DÉTECTION DE FAIBLESSE D'UN SOUS-CONTRAT

Un GSWCE indique une faiblesse de l'ensemble des sous-contrats d'une fonction mais n'identifie pas précisément quel contrat $c \in C$ est trop faible. Nous proposons pour cela une autre transformation du programme P en un programme instrumenté P_c^{SSW} (pour *single subcontract weakness*). Elle réalise le remplacement d'un seul appel de fonction non imbriquée ou de boucle par le code respectant la postcondition du sous-contrat c correspondant (comme le montrent les figures 7.1 et 7.2) et traduit les autres annotations de $\mathcal{A}$ de la même manière que pour P^{NC}.

Définition 6 : Faiblesse d'un sous-contrat

Soit c un sous-contrat de f. c est un sous-contrat trop faible (ou a une faiblesse de sous-contrat) de f s'il existe un cas de test V de f respectant sa précondition, tel que P^{NC} ne rapporte aucune violation d'annotation sur V, alors que P_c^{SSW} rapporte une violation d'annotation sur V. Dans ce cas, on dit que V est un contre-exemple de faiblesse d'un sous-contrat (ou SSWCE pour *single subcontract weakness counter-example*) pour le sous-contrat c de f.

Pour chaque sous-contrat $c \in C$, la génération de tests peut être appliquée de manière séparée sur P_c^{SSW} pour générer des SSWCE potentiels. Si un tel cas de test V est généré, il est exécuté sur P^{NC} afin de déterminer si c'est un NCCE ou un SSWCE. Nous appelons

cette technique, appliquée pour tous les sous-contrats l'un après l'autre jusqu'à ce qu'un contre-exemple V soit trouvé, Détection de faiblesse d'un sous-contrat, notée SSWD pour *Single Contract Weakness Detection*. SSWD peut produire trois résultats. Elle retourne (nc, V, a) si un NCCE V a été trouvé pour une annotation a, et (sw, $V, a, \{c\}$) si V a été classé comme étant un SSWCE indiquant que le sous-contrat c est trop faible pour l'annotation a. Le chemin de programme π_V activé par V et menant à l'erreur (sur P^{NC} ou P_c^{SSW}) est également enregistré. Sinon, elle retourne ? (*unknown*), puisque même en cas d'exploration complète des chemins du programme l'absence de SSWCE pour un unique sous-contrat c n'implique pas l'absence de GSWCE.

En effet, il arrive que SSWD ne puisse pas détecter de faiblesse pour un sous-contrat en particulier alors qu'il y a une faiblesse pour un ensemble de sous-contrats. Dans l'exemple du listing 7.1, si on applique SSWD à un sous-contrat quelconque, on obtient toujours `x >= \old(x)+5` à la fin de la fonction `f` (on ajoute 1 à `x` en exécutant le sous-contrat traduit et on ajoute 2 par deux fois en exécutant le code des deux autres fonctions), donc la postcondition de `f` est vérifiée et aucune faiblesse de sous-contrat n'est détectée. Si on applique GSWD pour considérer tous les sous-contrats en une seule fois, on obtient uniquement `x >= \old(x)+3` après exécution des trois sous-contrats, ce qui permet d'exhiber un contre-exemple pour cette faiblesse des sous-contrats.

Inversement, GSWD produit un GSWCE qui n'indique pas de manière précise lequel des sous-contrats est trop faible, alors que SSWD apporte cette précision. Dans le listing 7.2, puisque GSWD remplace le code des trois fonctions appelées par leur contrat, il est impossible de déterminer quel sous-contrat est trop faible. Les contre-exemples générés par un prouveur souffrent du même problème d'imprécision : prendre en compte tous les sous-contrats au lieu du code correspondant ne permet pas d'identifier la faiblesse d'un unique sous-contrat. Dans cet exemple nous pouvons être plus précis avec SSWD, puisque le remplacement de l'appel à `g3` par son contrat produit un SSWCE : on a `x >= \old(x)+3` en exécutant `g1`, `g2` et le sous-contrat de `g3`, exhibant la faiblesse du sous-contrat de `g3`. Ainsi, la technique SSWD fournit à l'ingénieur validation un diagnostic plus précis qu'un contre-exemple produit par GSWD ou par un prouveur.

Nous définissons une technique de détection combinée de faiblesses de sous-contrats, notée SWD, qui consiste à appliquer d'abord SSWD puis GSWD jusqu'à ce qu'un SWCE soit trouvé. SWD produit les mêmes quatre résultats que SSWD. Cette méthode nous permet d'être plus précis (et d'indiquer si possible que la faiblesse ne provient que d'un seul sous-contrat) et complet (capable de trouver un GSWCE même lorsqu'il n'y a pas de faiblesse de sous-contrat provenant d'un unique sous-contrat).

7.3 DÉTECTION DE FAIBLESSE DE SOUS-CONTRATS EN PRATIQUE

Retrouvons Alice là où nous l'avons laissée à la fin du chapitre 6. Alice spécifie la deuxième boucle de la fonction `delete_substr` (qui copie les éléments de la chaîne `str` dans la chaîne `dest` jusqu'à l'indice `start`) avec ce contrat de boucle :

```
loop invariant \forall integer m; 0 ≤ m < j⇒ dest[m] == \at(str[m],Pre);
loop invariant 0 ≤ j ≤ start;
loop assigns j, dest[0..start-1];
loop variant start-j;
```

Supposons maintenant qu'elle ait oublié de préciser dans le contrat de la fonction `find_substr` la valeur du résultat dans chacun des cas, par exemple elle oublie d'écrire :

```
ensures 0 ≤ \result < strlen-sublen;
```

ou

```
ensures \result == -1;
```

respectivement aux lignes 7 et 12 de la figure 6.2.

Cet oubli a un impact direct sur le contrat de la boucle qu'elle vient de spécifier, dont la preuve par WP échoue. Alice pense d'abord que cet échec de la preuve est causé par une non-conformité de la boucle par rapport à son contrat. Elle applique donc la méthode NCD dans l'espoir d'obtenir un NCCE. Bien qu'ayant couvert les chemins du programme respectant la clause `typically strlen <= 5` avec 12170 cas de tests, aucun NCCE n'est produit. Alice a maintenant le choix entre analyser manuellement le programme afin de localiser le problème ou relancer la preuve en augmentant le *timeout*, ce qui peut lui prendre beaucoup de temps. Elle décide d'analyser le programme car elle suspecte une faiblesse de spécification d'un des sous-contrats.

Elle applique alors SWD, plus précisément SSWD, qui détecte que le contrat de la fonction `find_substr` est trop faible pour l'invariant de boucle `loop invariant 0 <= j <= start` de la deuxième boucle de la fonction `delete_substr`. En effet, ayant enlevé les contraintes sur le résultat de la fonction `\result`, l'exécution du contrat de `find_substr` produit une valeur de retour qui ne permet pas de valider l'invariant. À partir du contre exemple produit par SSWD et de la nature de l'invariant non satisfait, Alice peut facilement localiser le problème dans le contrat de la fonction `find_substr`.

Alice poursuit son travail de spécification de la fonction `delete_substr` et annote la dernière boucle avec le contrat de boucle suivant :

```
loop invariant start ≤ j ≤ strlen-sublen;
loop assigns dest[start .. strlen-sublen-1], j;
loop variant strlen-sublen-j;
```

La vérification déductive de la fonction échoue, et NCD ne parvient pas à exhiber de contre-exemple car le programme ne comporte pas de non-conformité. En revanche, l'application de SWD met en évidence la faiblesse de ce dernier contrat de boucle, qui est insuffisant pour la preuve de la postcondition de la fonction. En effet, le contrat indique (par sa clause `loop assigns`) que la boucle peut modifier `dest[start .. strlen-sublen-1]` et `j`. Des cas de tests sont donc générés dans lesquels l'exécution du contrat modifie la valeur de `j` en respectant la contrainte de l'invariant `start <= j <= strlen-sublen` et modifie les valeurs du tableau, sans contrainte particulière, de manière à violer la postcondition de la fonction `delete_substr`. Un des contre-exemples produits est ⟨ `dest[0] = 23; dest[1] = -27;` $\text{nondet}_{\texttt{dest[0]}}$`= 11;` $\text{nondet}_{\texttt{j}}$`= 1; str[0] = 74; str[1] = 48; strlen = 2; sublen = 1; substr[0] = 74` ⟩ où $\text{nondet}_{\texttt{dest[0]}}$ et $\text{nondet}_{\texttt{j}}$ sont les valeurs non déterministes produites par la fonction `Nondet` afin de simuler la modification des variables `dest[0]` et `j` par la boucle.

Ici aussi, le résultat de SWD indique à Alice précisément quel sous-contrat est trop faible pour la propriété non prouvée par WP, lui facilitant son travail de revue du code et lui permettant de progresser rapidement vers une spécification correcte et complète de son programme.

CONCLUSION DU CHAPITRE

Dans ce chapitre, nous avons défini les notions de faiblesse de l'ensemble des sous-contrats et de faiblesse d'un sous-contrat. Notons qu'en l'absence de faiblesse de chacun des sous-contrats, il peut exister une faiblesse (globale) de l'ensemble des sous-contrats. L'exemple du listing 7.1 illustre cette subtilité.

Nous avons également présenté les spécificités de la traduction des annotations pour la détection de faiblesses de sous-contrats par rapport à leur traduction pour la détection de non-conformités présentée au chapitre 4. Enfin, nous avons continué le cas pratique du chapitre 6 pour illustrer et montrer l'intérêt de la détection des faiblesses des sous-contrats.

Le chapitre 8 présente une méthode qui combine la détection des non-conformités présentée au chapitre 6 et la détection des faiblesses de sous-contrats présentée dans ce chapitre pour aider l'utilisateur à diagnostiquer les échecs de preuve.

8

AIDE À LA PREUVE COMBINANT NCD ET SWD

Dans ce chapitre nous présentons notre méthode de diagnostic des échecs de preuve utilisant les techniques de détection présentées dans les chapitres 6 et 7. La partie 8.1 présente l'exemple sur lequel nous nous appuierons dans ce chapitre, la partie 8.2 rappelle les différentes causes possibles d'un échec de preuve, la partie 8.3 présente notre méthode de diagnostic des échecs de preuve et la partie 8.4 suggère les actions à effectuer après diagnostic d'un échec de preuve.

8.1 EXEMPLE ILLUSTRATIF

Nous illustrons sur le programme C donné en figure 8.1 les problèmes qui se posent lors de la vérification déductive d'un programme et les solutions que nous proposons. Cet exemple provient d'un travail de spécification formelle et de vérification déductive [Genestier et al., 2015] que j'ai réalisé avec R. Genestier et A. Giorgetti. Il implémente un algorithme proposé dans [Arndt, 2010, page 235]. L'exemple de la figure 8.1 concerne la génération de fonctions à croissance limitée (ou RGF pour *Restricted Growth Functions*). Une fonction à croissance limitée de taille n est une endofonction r de $\{0, ..., n-1\}$ telle que $r(0) = 0$ et $r(k) \leq r(k-1) + 1$ pour $1 \leq k \leq n-1$. Nous représentons une endofonction r de $\{0, ..., n-1\}$ par les valeurs d'un tableau C. Le prédicat `ACSL is_rgf` aux lignes 1–4 de la figure 8.1 exprime cette propriété sur un tableau `a` de taille `n`.

La figure 8.1 présente une fonction principale `f` et une fonction auxiliaire `g`. La précondition de `f` affirme que `a` est un tableau valide de taille `n>0` (lignes 27–28) qui représente une RGF (ligne 29). La postcondition affirme que la fonction `f` ne peut modifier que le contenu du tableau `a`, à l'exception du premier élément `a[0]` (ligne 30), et que le tableau `a` en sortie de la fonction est toujours une RGF (ligne 31). De plus, si la fonction retourne 1, alors la RGF générée `a` doit respecter une propriété d'ordre lexicographique selon laquelle il existe un indice i ($0 \leq i \leq n-1$) tel qu'en fin de fonction, $a[i]$ est supérieur à sa valeur en début de fonction et $a[j]$ est inchangé pour tout indice j inférieur à i (lignes 32–36). Ici `\at(a[j],Pre)` dénote la valeur de `a[j]` dans l'état `Pre` de la fonction, c'est-à-dire avant l'exécution de la fonction.

Observons maintenant le corps de la fonction `f` de la figure 8.1. La boucle aux lignes 42–43 parcourt le tableau de droite à gauche afin de trouver l'élément non-croissant le plus à droite, c'est-à-dire l'indice maximum `i` du tableau tel que `a[i] <= a[i-1]`. Si un tel

```
1  /*@ predicate is_rgf(int *a, integer n) =
2        a[0] == 0 &&
3          \forall integer i; 1 <= i < n ==>
4            (0 <= a[i] <= a[i-1]+1); */
5
6  /*@ lemma max_rgf:
7    \forall int* a; \forall integer n;
8      is_rgf(a, n) ==>
9        (\forall integer i; 0 <= i < n ==>
10         a[i] <= i); */
11
12 /*@ requires n > 0;
13     requires \valid(a+(0..n-1));
14     requires 1 <= i <= n-1;
15     requires is_rgf(a,i+1);
16     assigns a[i+1..n-1];
17     ensures is_rgf(a,n); */
18 void g(int a[], int n, int i) {
19  int k;
20  /*@ loop invariant i+1 <= k <= n;
21      loop invariant is_rgf(a,k);
22      loop assigns k, a[i+1..n-1];
23      loop variant n-k; */
24  for (k = i+1; k < n; k++) a[k] = 0;
25 }
26
```

```
27 /*@ requires n > 0;
28     requires \valid(a+(0..n-1));
29     requires is_rgf(a,n);
30     assigns a[1..n-1];
31     ensures is_rgf(a,n);
32     ensures \result == 1 ==>
33      \exists integer j; 0 <= j < n &&
34       (\at(a[j],Pre) < a[j] &&
35        \forall integer k; 0 <= k < j ==>
36          \at(a[k],Pre) == a[k]); */
37 int f(int a[], int n) {
38  int i,k;
39  /*@ loop invariant 0 <= i <= n-1;
40      loop assigns i;
41      loop variant i; */
42  for (i = n-1; i >= 1; i--)
43   if (a[i] <= a[i-1]) { break; }
44  if (i == 0) { return 0; } // Last RGF.
45  //@ assert a[i]+1 <= 2147483647;
46  a[i] = a[i] + 1;
47  g(a,n,i);
48  /*@ assert \forall integer l;
49        0 <= l < i ==>
50          \at(a[l],Pre) == a[l]; */
51  return 1;
52 }
```

FIGURE 8.1 – Fonction “successeur” d’une RGF

indice est trouvé, la fonction incrémente `a[i]` (ligne 46) et remplit le reste du tableau avec `0` (appel à la fonction `g`, line 47). Le contrat de boucle (lignes 39–41) spécifie l’intervalle de valeurs de la variable `i`, les variables que la boucle peut modifier, ainsi que le variant de boucle qui permet d’assurer la terminaison de la boucle. L’expression du variant de boucle doit être positive ou nulle au début de chaque itération et doit décroître strictement entre deux itérations.

La fonction `g` est utilisée pour remplir le tableau avec des zéros à droite de l’indice `i`. En plus des contraintes de taille et de validité du tableau (lignes 12–13), sa précondition demande que les éléments de `a` jusqu’à l’indice `i` forment une RGF (lignes 14–15). La fonction ne peut modifier que les éléments de `a` à partir de l’indice `i+1` (ligne 16) et produit une RGF (line 17). Les invariants de boucle précisent l’intervalle de valeurs de la variable de boucle `k` (ligne 20), et énoncent que la propriété `is_rgf` est satisfaite jusqu’à `k` (ligne 21). Cet invariant permet à un outil de vérification déductive de prouver la postcondition. L’annotation `loop assigns` (ligne 22) énonce que la boucle peut uniquement modifier les valeurs de `k` et des éléments de `a` à partir de l’indice `i+1`. Le terme `n-k` est le variant de la boucle (ligne 23).

Le lemme `max_rgf` aux lignes 6–10 affirme que si un tableau encode une RGF, alors chacun de ses éléments est au plus égal à son indice. Ce lemme n’est pas prouvé automatiquement par WP mais peut néanmoins être utilisé pour assurer l’absence de débordement d’entier signé à la ligne 46.

Les fonctions de la figure 8.1 peuvent être prouvées à l’aide de WP.

Supposons maintenant que cet exemple contient une des quatre erreurs suivantes :

1. soit l’ingénieur validation oublie la précondition de la ligne 29,
2. soit il se trompe dans l’affectation `a[i]=a[i]+2;` ligne 46,
3. soit il écrit une clause `loop assigns i,a[1..n-1];` trop imprécise ligne 40,

4. soit il oublie de fournir le lemme des lignes 6–10.

Dans chacun de ces scénarios, la preuve du programme échoue (sur la précondition de `g` lors de son appel ligne 47 et/ou sur l'assertion ligne 45).

Les deux premiers cas présentent une non-conformité entre le code et la spécification, dans le troisième cas l'échec de la preuve est dû à une faiblesse de sous-contrat, et le dernier cas est dû à une limitation ou une faiblesse du prouveur utilisé.

8.2 ÉCHECS DE PREUVE

Rappelons les différentes causes possibles d'un échec de la vérification déductive d'un programme.

Non-conformité. Un échec de preuve peut être dû à une non-conformité du code d'un programme par rapport à sa spécification formelle. Le chapitre 6 a défini la notion de non-conformité, de contre-exemple de non-conformité (NCCE) et a présenté une méthode de détection des non-conformités (NCD).

Faiblesse de sous-contrat. Un échec de preuve peut également être dû à une faiblesse d'un (ou de plusieurs) sous-contrat(s), ce qui cause un manque d'hypothèses et ne permet pas de prouver une propriété. Le chapitre 7 a défini la notion de sous-contrat, de faiblesse de sous-contrat, de contre-exemple pour la faiblesse de sous-contrats (SWCE) et a présenté une méthode de détection de ces faiblesses (SWD).

Incapacité de prouveur. Quand il n'existe ni non-conformité, ni faiblesse de sous-contrats, l'échec de preuve est dû aux limitations du prouveur. Nous introduisons la notion d'incapacité de prouveur.

> **Définition 7 : Incapacité de prouveur**
>
> On dit qu'un échec de preuve du programme P est lié à une incapacité de prouveur si pour tout cas de test V pour la fonction f respectant sa précondition, P^{NC} et P^{GSW} ne rapportent aucune violation d'annotation sur V. En d'autres termes, il n'y a ni NCCE, ni GSWCE pour P.

Cette catégorie contient entre autres les limitations techniques des prouveurs comme par exemple la difficulté que peuvent rencontrer les prouveurs à instancer les variables quantifiées dans un prédicat `\exists`. La nécessité d'écrire des lemmes pour aider le prouveur, voire d'avoir recours à un assistant de preuve comme Coq définit également une incapacité de prouveur.

8.3 PRÉSENTATION DE LA MÉTHODE

La méthode que nous proposons est illustrée par la figure 8.2. Supposons que la preuve du programme annoté P échoue pour une annotation non-imbriquée $a \in \mathcal{A}$ quelconque, où $\mathcal{A}$ est l'ensemble des annotations non-imbriquées de f et des annotations du contrat de f. La première étape est d'essayer de trouver une non-conformité entre le code et la spécification en utilisant NCD. Si une telle non-conformité a été trouvée, un NCCE est

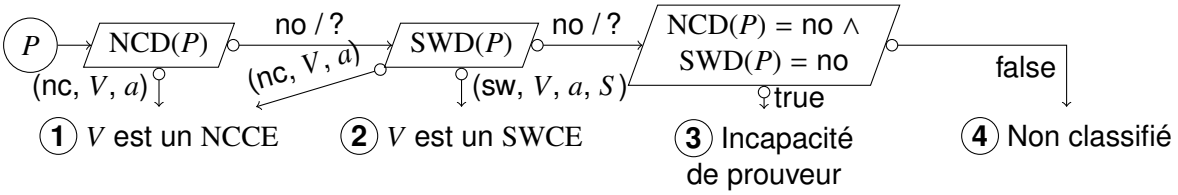

FIGURE 8.2 – Méthode de vérification combinant NCD et SWD en cas d'échec de preuve du programme P

#	Lignes modifiées		Résultats intermédiaires			Résultat final
	Ligne	Modifications	Preuve (annot. non prouvée)	NCD	SWD	
0	–	–	✓	–	–	Preuve
1	29	(supprimée)	? (l.45, 47, 31)	nc	–	V = ⟨`n=1; a[0]=-214739`⟩ est un NCCE
2	40	`loop assigns i,a[1..n-1];`	? (l.45, 47, 48–50, 31, 32–36)	?	sw pour l.39–40	V = ⟨ `n=2;a[0]=0;a[1]=0;` $\mathbf{nondet}_{\texttt{a[1]=97157;}}$ $\text{nondet}_{\texttt{i=0}}$ ⟩ est un SWCE
3	6–10 27	(supprimée) `requires n>0 && n<21;`	? (l.45)	no	no	Incapacité de prouveur
4	6–10	(supprimée)	? (l.45)	?	?	Non classifié

FIGURE 8.3 – Résultats de la méthode sur différentes versions de l'exemple de la figure 8.1.

généré (cas (1) sur la figure 8.2) et l'échec de la preuve est classifié comme une non-conformité. Si cette première étape n'a pas pu générer de contre-exemple, l'étape SWD combine SSWD et GSWD et essaie de générer des SWCEs pour les sous-contrats pris un-à-un, sinon des SWCEs pour l'ensemble des sous-contrats, jusqu'à ce que le premier contre-exemple soit généré et classifié (soit comme un NCCE (1), soit comme un SWCE (2)) Si aucun contre-exemple n'a été trouvé, nous analysons le résultat de NCD et SWD. Si NCD et SWD ont retourné no, c'est-à-dire que NCD et GSWD ont couvert tous les chemins du programme sans produire de contre-exemple, l'échec de preuve est dû à une incapacité de prouveur (3) (voir la définition 7). Sinon, l'échec de preuve reste non classifié (4). La figure 8.3 associe à chacun des quatre cas une version modifiée de l'exemple illustratif de la figure 8.1. Dans chaque cas, nous détaillons les lignes modifiées dans le programme de la figure 8.1 pour obtenir le nouveau programme, les résultats intermédiaires de la vérification déductive, de NCD et SWD, et le verdict final de la méthode (ainsi que le contre-exemple éventuellement généré).

La classification de l'échec de preuve, le contre-exemple V, le chemin d'exécution π_V activé par V, ainsi que l'annotation invalidée a et l'ensemble des sous-contrats trop faibles S peuvent s'avérer très utiles à l'ingénieur validation. Supposons que l'on essaie de prouver avec WP une version modifiée de la fonction f de la figure 8.1 où la précondition de la ligne 29 est manquante (#1 sur la figure 8.3). La preuve de la précondition de g (ligne 15) lors de l'appel de la fonction ligne 47 échoue sans indiquer de raison précise. L'étape NCD génère un NCCE (cas (1)) où la propriété `is_rgf(a,n)` est clairement fausse puisque `a[0]` est différent de zéro, et indique l'annotation non respectée (ligne 15). Ceci permet à l'ingénieur validation de comprendre et corriger le problème.

Supposons maintenant que la clause de la ligne 40 a été écrite de manière erronée de la

manière suivante : `loop assigns i, a[1..n-1];` (#2 sur la figure 8.3). La boucle aux lignes 42–43 préserve néanmoins son invariant. L'étape NCD ne trouve aucun NCCE, puisque cette modification n'a introduit aucune non-conformité entre le code et sa spécification. L'étape SSWD pour le contrat de cette boucle détecte une faiblesse de sous-contrat pour ce même contrat (cas (2)), et indique que la précondition de g (ligne 15) est fausse lors de l'appel de la fonction ligne 47. Avec l'information de la faiblesse du sous-contrat de la boucle, l'ingénieur validation essaiera de trouver la cause de la faiblesse et renforcera le contrat de boucle en conséquence.

Supposons maintenant que nous voulions prouver l'absence de débordement arithmétique à la ligne 46 de la figure 8.1, et que le lemme aux lignes 6–10 est manquant (#4 sur la figure 8.3). La preuve échoue sans donner de raison précise puisque le prouveur n'effectue pas l'induction nécessaire afin de déduire les bornes de `a[i]`. Ni NCD ni SWD n'ont réussi à produire de contre-exemple, et puisque le programme initial comporte trop de chemins d'exécution faisables pour pouvoir tous les couvrir dans le temps imparti, leur résultat est ? (non classifié) (cas (4)). Dans une telle situation, l'ingénieur validation peut décider de réduire le domaine des entrées du programme. Supposons dans ce cas qu'il choisisse de restreindre le domaine de `n` à l'aide de la précondition pour f suivante : `requires n>0 && n<21;` (#3 sur la figure 8.3). Avec cette nouvelle contrainte sur les entrées, l'application de la méthode permet de couvrir tous les chemins du programme (respectant la nouvelle précondition) avec NCD et SWD sans trouver de contre-exemple. La méthode classifie l'échec de preuve du programme avec le domaine restreint sur les variables comme une incapacité de prouveur (cas (3)). Ces résultats poussent l'ingénieur validation à penser que l'échec de la preuve sur la variante #4 du programme, pour un `n` plus grand, est causé par les mêmes raisons. Ainsi, l'ingénieur validation préfèrera essayer une preuve interactive, ou écrire des lemmes ou assertions supplémentaires, et ne perdra pas son temps à chercher une non-conformité ou une faiblesse de contrat.

8.4 SUGGESTIONS D'ACTIONS

À partir des différents résultats possibles de la méthode présentée dans la partie 8.3, nous pouvons suggérer à l'ingénieur validation les actions à effectuer une fois que l'échec de preuve a été diagnostiqué, afin de l'aider dans son processus de spécification et de vérification déductive. Nos suggestions sont résumées en figure 8.4. Nous les expliquons ci-dessous.

Une non-conformité du code vis-à-vis d'une annotation a signifie qu'il y a une incohérence entre la précondition, l'annotation a et le code du chemin de programme π_V menant à a. Grâce au contre-exemple V, les valeurs des variables à différents points du programme suivant le chemin π_V peuvent être tracées ou observées à l'aide d'un débogueur [Müller et al., 2011]. Dans FRAMA-C, l'exécution de V peut être observée à l'aide de VALUE ou PATHCRAWLER. Ceci aide l'ingénieur validation dans sa compréhension de l'erreur. En effet, si un NCCE est généré, l'ingénieur n'a pas besoin de tenter une preuve déductive ou de chercher une faiblesse de contrat dans la spécification, puisque la raison de l'échec de la preuve est nécessairement liée à une non-conformité entre le code et les annotations traversées par le chemin d'exécution π_V.

Une faiblesse d'un ensemble de sous-contrats S signifie qu'au moins un des sous-contrats de C doit être renforcé. Par les définitions 5 et 6, la non-conformité est exclue,

Cas	Verdict	Suggestions
①	Non-conformité du code vis-à-vis de l'annotation a : (nc, V, a)	corriger l'annotation a ou le code menant à a par le chemin d'exécution π_V, ou renforcer la précondition de la fonction sous vérification
②	Faiblesse de sous-contrats parmi ceux de S vis-à-vis de l'annotation a : (sw, V, a, S)	renforcer un ou plusieurs sous-contrats de S
③	Incapacité de prouveur (test complet)	ajouter des lemmes ou des assertions pour aider le prouveur, ou utiliser un autre prouveur plus approprié, ou utiliser un prouveur interactif (comme Coq)
④	Non-classifié (test incomplet)	renforcer la clause `typically` ou le critère de couverture (par exemple k-path), ou augmenter le temps alloué à la génération de tests (*timeout*)

FIGURE 8.4 – Suggestions d'actions pour chaque catégorie d'échec de preuve

ce qui veut dire que l'exécution de P^{NC} sur V respecte l'annotation a. Notre suggestion pour l'ingénieur validation est donc de renforcer les sous-contrats de S. Dans le cas d'une faiblesse d'un seul sous-contrat, S est un singleton donc la suggestion est précise et aide l'utilisateur. Ici encore, essayer une preuve interactive ou ajouter des assertions ou des lemmes supplémentaires sera inutile puisque le contre-exemple témoigne de l'impossibilité de prouver la propriété.

Pour une incapacité de prouveur, l'ingénieur validation peut écrire des lemmes ou des assertions, ou ajouter des hypothèses qui peuvent aider le prouveur. Il peut aussi essayer un autre prouveur automatique plus approprié au programme à vérifier, ou encore, utiliser un assistant de preuve comme Coq afin de ne pas souffrir des limitations des prouveurs automatiques, mais cette tâche peut s'avérer longue et complexe.

Enfin, lorsque le verdict de la méthode est non classifié, la génération de tests pour NCD et/ou SWD ne parvient pas à conclure dans le temps imparti (le *timeout* a été atteint), nous suggérons donc à l'ingénieur validation de renforcer la précondition du programme pour le test afin de réduire le domaine des entrées, ou de repousser le *timeout* afin de donner à la méthode plus de temps pour conclure.

CONCLUSION DU CHAPITRE

Dans ce chapitre nous avons rappelé les différentes causes possibles d'un échec de preuve. Nous avons présenté notre méthode de diagnostic des échecs de preuve utilisant les techniques de détection présentées dans les chapitres 6 et 7. Enfin, nous avons suggéré des actions qu'un ingénieur validation peut effectuer après qu'un échec de preuve ait été diagnostiqué et ce dans chacune des quatre situations identifiées : non-conformité, faiblesse de sous-contrat, incapacité du prouveur et test complet, et incapacité du prouveur et test incomplet (non-classifié). Ces suggestions d'actions sont une assistance à l'ingénieur validation dans son processus de spécification et de vérification déductive d'un programme.

9

BIBLIOTHÈQUE DE MONITORING DE LA MÉMOIRE D'E-ACSL2C

Dans ce chapitre nous présentons notre implémentation du modèle mémoire des programmes C qui permet d'exécuter les annotations E-ACSL présentées dans le chapitre 5. Notre bibliothèque permet au greffon E-ACSL2C de vérifier à l'exécution les annotations E-ACSL portant sur le modèle mémoire.

E-ACSL2C traduit automatiquement un programme C annoté en un autre programme C dont l'exécution échouera si une annotation n'est pas valide. Si aucune annotation n'est violée, le comportement du nouveau programme est exactement le même que celui du programme d'origine. Ce greffon utilise notre bibliothèque afin de vérifier à l'exécution les annotations E-ACSL portant sur la mémoire. Le greffon E-ACSL2C lui-même n'étant pas le fruit de nos travaux [Delahaye et al., 2013], il ne sera pas présenté dans ce chapitre.

Nous présentons la bibliothèque en partie 9.1. Au cours de nos travaux, nous avons mesuré la capacité de détection d'erreur de notre implémentation et nous avons comparé les performances à l'exécution de plusieurs choix d'implémentation. Nous présentons les résultats de nos expérimentations en partie 9.2.

9.1 ARCHITECTURE DE LA BIBLIOTHÈQUE DE MONITORING DE LA MÉMOIRE

Les annotations présentées au chapitre 5 prennent en paramètre des adresses mémoire, et non des left-values. La vérification de ces annotations nécessite donc une représentation du *store* tel qu'il a été présenté dans la partie 5.2, mais ne nécessite pas de représenter l'environnement.

La figure 9.1 présente l'architecture générale de la bibliothèque au moyen du graphe de dépendance des fonctions C les plus importantes. Une flèche $A \rightarrow B$ signifie que la fonction A appelle la fonction B.

L'architecture comporte quatre niveaux, numérotés de (1) à (4). Les fonctions du niveau (1) correspondent à l'implémentation de la structure de données *store* (présentée au chapitre 5) gardant les informations à propos des blocs alloués. Plusieurs implémentations sont fournies pour ces fonctions, correspondant à différentes structures de données génériques : listes chaînées, arbres binaires de recherche, Splay trees et Patricia tries. La fonction `get_exact` de la figure 9.1 implémente l'algorithme 2. La fonction `get_cont` im-

FIGURE 9.1 – Architecture de la bibliothèque de monitoring de la mémoire

plémente l'algorithme 3. La fonction `add_element` implémente l'algorithme 4. La fonction `remove_element` implémente l'algorithme 5. Les fonctions du niveau (2) servent à modifier le contenu du *store* (ajout ou suppression d'élément, initialisation des éléments, etc.). Les fonctions du niveau (3) doivent être appelées en lieu et place des fonctions de la bibliothèque standard associées (`malloc`, etc.). Les fonctions `e_acsl_malloc`, `e_acsl_calloc`, `e_acsl_realloc` et `e_acsl_free` de la figure 9.1 correspondent respectivement aux fonctions `__malloc`, `__calloc`, `__realloc` et `__free` présentées au chapitre 5. Les fonctions du niveau (4) permettent de calculer la valeur des annotations E-ACSL, elles retournent la valeur du terme ou du prédicat E-ACSL correspondant. Ces fonctions ne modifient pas le contenu du *store*. Les fonctions `offset`, `block_length`, `base_addr`, `initialized`, `valid` et `valid_read` de la figure 9.1 correspondent respectivement aux fonctions `__offset`, `__block_length`, `__base_addr`, `__initialized`, `__valid` et `__valid_read` présentées au chapitre 5.

Les fonctions des niveaux (2), (3) et (4) sont implémentées de manière indépendante à l'implémentation du *store* (niveau (1)). Cette architecture facilite la conception des algorithmes, ainsi que la comparaison d'efficacité des différentes implémentations du *store*, effectuée en partie 9.2.

	alarmes	mutants	équivalents	tués	% erronés tués
fibonacci	19	27	2	25	100%
bubbleSort	15	44	2	42	100%
insertionSort	10	39	3	36	100%
binarySearch	7	38	1	37	100%
merge	5	92	5	87	100%

FIGURE 9.2 – Capacité de détection d'erreurs

9.2 EXPÉRIMENTATIONS

Nous présentons maintenant les résultats de nos expérimentations visant à évaluer la capacité de détection d'erreurs de la bibliothèque et la performance à l'exécution des différents choix d'implémentation.

9.2.1 CAPACITÉ DE DÉTECTION D'ERREURS

Nous avons utilisé la technique de mutation pour évaluer la capacité de détection d'erreurs en utilisant la vérification d'assertion à l'exécution avec FRAMA-C. Nous avons considéré 5 programmes écrits et annotés par nos soins. Nous avons généré leurs *mutants* (en appliquant une *mutation* sur leur code source) et leur avons appliqué la vérification à l'exécution. Les mutations du programme incluent : modifications d'opérateurs arithmétiques numériques, modifications d'opérateurs arithmétiques sur les pointeurs, modifications d'opérateurs de comparaison et modifications d'opérateurs logiques (et et ou). L'outil de génération de test PATHCRAWLER [Botella et al., 2009] a été utilisé pour produire les cas de test. Chaque mutant a été instrumenté par E-ACSL2C et exécuté sur chaque cas de test pour vérifier que la spécification était satisfaite à l'exécution. Les programmes d'origine passent toutes les vérifications à l'exécution. Lorsqu'une violation d'une annotation a été reportée pour au moins un cas de test, le mutant est considéré comme étant *tué*. La figure 9.2 présente les résultats. La première colonne contient les exemples étudiés. La deuxième colonne contient le nombre d'annotations dans chaque programme. Les colonnes suivantes contiennent le nombre de mutants générés, le nombre de mutants équivalents à leur programme d'origine (décidé par analyse manuelle), le nombre de mutants non-équivalents tués, et le pourcentage des mutants non-équivalents tués lors de l'exécution. Lors de ces expérimentations, tous les mutants non-équivalents erronés ont été tués, ce qui témoigne de la précision et de la complétude de notre méthode.

9.2.2 PERFORMANCE DES CHOIX D'IMPLÉMENTATION

Pour évaluer la performance à l'exécution de notre implémentation de la bibliothèque, nous avons effectué plus de 300 exécutions sur plus de 30 programmes, obtenus à partir d'une dizaine de programmes écrits et annotés par nos soins. Nous avons volontairement gardé des exemples plutôt courts (moins de 200 lignes de code) car ils ont dû être annotés en E-ACSL manuellement.

Nous avons mesuré le temps d'exécution du programme d'origine et du code instrumenté par E-ACSL2C avec différentes options, afin d'évaluer les performances des différentes implémentations du *store* et des différentes optimisations.

Calcul du plus grand préfixe commun :
Nous avons comparé deux implémentations de ce calcul, qui est utilisé par l'implémentation du *store* utilisant des Patricia tries. La première implémentation utilise un parcours linéaire de l'adresse (bit-à-bit, de gauche à droite). La seconde implémentation est une recherche dichotomique du meilleur préfixe dans un tableau dont le contenu et les indices sont pré-calculés. Elle implémente l'algorithme 1 présenté en partie 5.2.2. Cette seconde implémentation s'est révélée en moyenne 2.7 fois plus rapide que la première sur nos exemples.

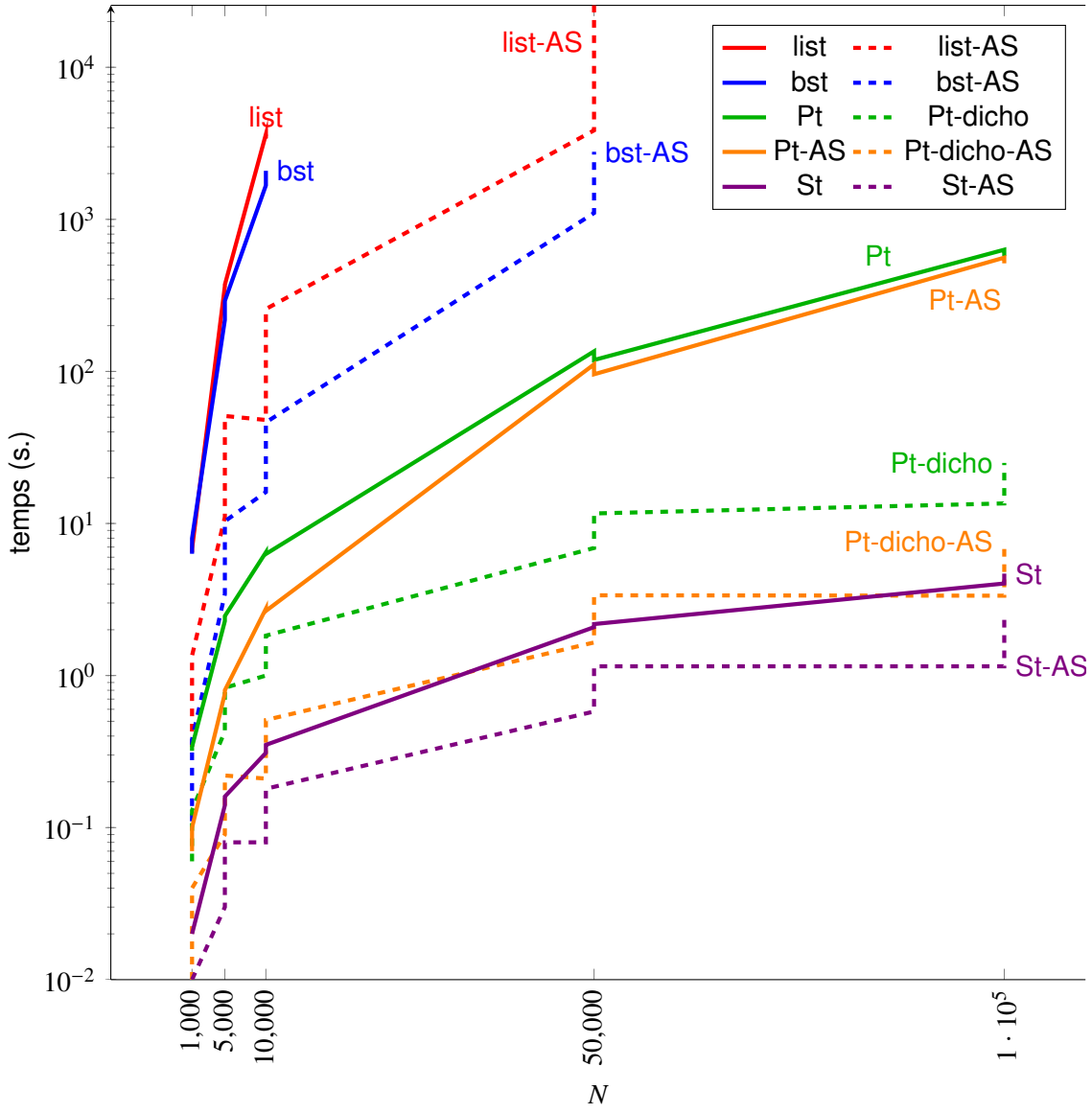

FIGURE 9.3 – Comparaison du temps d'exécution des différentes implémentations du *store* sur un tri fusion de N éléments

Implémentation du *store* :
Pour déterminer quelle implémentation du *store* est la plus appropriée, nous avons comparé quatre implémentations utilisant des chaînées (notées $list$ sur les figures 9.3 et 9.4), des arbres binaires de recherche non équilibrés (notés bst), des arbres binaires de recherche équilibrés (Splay trees [Sleator et al., 1985], notés St) et des Patricia tries. Nous distinguons deux implémentations à base de Patricia tries : la première (notée Pt) où

l'algorithme de recherche du masque du plus grand préfixe commun est implémenté de manière linéaire, et la seconde (notée $Pt{-}dicho$) où la recherche du masque du plus grand préfixe commun est implémentée en utilisant l'algorithme 1 présenté en partie 5.2.2.

Le greffon E-ACSL2C utilise une analyse statique qui permet de n'instrumenter que ce qui est nécessaire. Une analyse dataflow arrière est faite afin de déterminer une sur-approximation des left-values dont l'instrumentation est nécessaire pour chaque annotation, ce qui réduit le nombre de left-values surveillées à l'exécution. Cette analyse n'étant pas le fruit de nos travaux (section 6 de [Kosmatov et al., 2013]), nous ne la présentons pas ici. Nous en profitons néanmoins pour mesurer l'impact qu'a cette analyse pour chaque implémentation du *store* dans nos expérimentations. Ces mesures sont respectivement dénotées pour chaque implémentation $list\text{-}AS$, $bst\text{-}AS$, $St\text{-}AS$, $Pt\text{-}AS$ et $Pt\text{-}dicho\text{-}AS$.

La figure 9.3 représente graphiquement le temps (en secondes) passé par chaque implémentation du *store* sur un tri fusion pour N = 1000, 5000, 10000, 50000 et 100000 éléments. Les données chiffrées sont dans les cinq dernières lignes de la figure 9.4. Nous voyons sur la figure 9.3 que, sur cet exemple, les implémentations du *store* sont, de la plus efficace à la moins efficace : Splay tree, Patricia trie "dicho", Patricia trie "linéaire", arbre binaire de recherche et liste chaînée. Pour chacune de ces implémentations, l'utilisation de l'analyse statique dataflow apporte un gain d'efficacité. Les meilleures performances de l'implémentation à base de Splay tree sont dues à la nature de l'exemple (un tri fusion) où il y a de nombreux accès mémoire consécutifs sur même bloc, ce qui joue en faveur des Splay tree : le dernier élément accédé est remonté à la racine de l'arbre. La figure 9.4 prend en compte un plus large spectre de programmes, ce qui permet de nuancer ces résultats.

La figure 9.4 présente les résultats des expérimentations comparant les différentes implémentations du *store* et du calcul du plus grand préfixe commun sur de nombreux exemples. La première colonne du tableau contient les exemples étudiés. bS_{10000} est une recherche binaire dans un tableau de 10000 éléments. iS_{10000} est un tri par insertion d'un tableau de 10000 éléments. mM_{n^2} est une multiplication de matrices $n \times n$. mI_{n^2} contient des calculs matriciels (dont inversion et multiplication) sur des matrices $n \times n$. qS_n est un tri rapide sur un tableau de n éléments. bbS_{10000} est un tri à bulle sur un tableau à 10000 éléments. m_{30000} est une fusion de deux listes chaînées de 10000 et 20000 éléments. Rbt_{10000} est une insertion/suppression de 10000 éléments dans un arbre rouge et noir. mS_n est un tri fusion d'une liste chaînée de n éléments. La ligne supplémentaire "+ RTE" de chaque exemple correspond à l'exemple après application du greffon RTE qui génère des assertions qui sont vraies si le programme ne contient pas d'erreur à l'exécution. Nous appelons ces assertions à vérifier des "alarmes".

La colonne $\lightning$ contient le nombre d'annotations du programme (celles ajoutées manuellement et celles générées par le greffon RTE). La colonne $\emptyset$ contient le temps d'exécution du programme original. Tous les temps d'exécution sont mesurés en secondes. Un temps d'exécution est noté ∞ quand il dépasse 24 heures. Les colonnes $list$ à $St\text{-}AS$ contiennent le temps d'exécution du programme pour chaque implémentation du *store*. Le temps d'analyse du programme sans instrumentation avec le débogueur VALGRIND [Nethercote et al., 2007] est indiqué dans la dernière colonne.

Nous remarquons que le temps d'exécution de VALGRIND n'est pas comparable avec celui de notre bibliothèque, cela s'explique simplement par le fait que celui-ci ne prend pas en compte la spécification E-ACSL, et se contente de vérifier des propriétés comme l'absence

d'erreur de segmentation ou l'absence de fuite de mémoire. En effet, notre démarche vise à supporter au maximum les annotations E-ACSL, ce qui nécessite un monitoring plus lourd.

Les résultats de nos expérimentations confirment nos hypothèses, à savoir :

H1 le Patricia trie est la structure de données la plus appropriée pour l'implémentation du *store* ;
H2 notre optimisation de la recherche du masque du plus grand préfixe commun par recherche dichotomique et l'utilisation d'indices pré-calculés entraîne un vrai gain de performance ;
H3 l'utilisation d'une analyse statique visant à réduire l'instrumentation du programme permet de réduire le temps d'exécution de manière efficace.

Concernant **H1**, la figure 9.4 montre que l'implémentation du *store* par un Patricia trie est la plus efficace. La version utilisant les Splay trees offre des performances comparables (ou légèrement meilleures, jusqu'à 3 fois) sur les exemples contenant de fréquents accès mémoire consécutifs au même bloc dans le *store*, comme c'est le cas du tri fusion (voir la figure 9.3). En revanche, sur des exemples où les accès mémoire consécutifs ne se font pas sur le même bloc (une multiplication de matrices dans notre exemple), les performances sont beaucoup moins bonnes (jusqu'à 500 fois). Ceci est dû à la nature des Splay trees : le dernier élément accédé est remonté à la racine de l'arbre. En moyenne, l'implémentation à base de Patricia trie est 2500 fois plus rapide que l'implémentation à base de listes chaînées, 200 fois plus rapide que celle utilisant les arbres binaires de recherche, et 27 fois plus rapide que celle se basant sur les Splay trees.

Concernant **H2**, la figure 9.4 montre que la version dichotomique de la recherche du masque du plus grand préfixe commun est toujours plus efficace qu'une recherche linéaire. Elle est en moyenne 3 fois plus rapide que cette dernière.

Concernant **H3**, nos résultats montrent que l'utilisation d'une analyse statique visant à réduire l'instrumentation du programme entraîne un gain de performance. En effet, pour chacune des implémentations du *store*, l'analyse statique permet de réduire le temps d'exécution. L'exécution après analyse statique peut être jusqu'à 200 fois plus rapide sur certains exemples. Elle permet notamment de ne pas observer de *timeout* pour les listes chaînées et les arbres binaires de recherche sur l'exemple du tri fusion de 50000 éléments.

CONCLUSION DU CHAPITRE

Dans ce chapitre nous avons présenté l'architecture de l'implémentation de la bibliothèque de monitoring de la mémoire, dont les algorithmes et les principes théoriques ont été traités dans le chapitre 5.

Nous avons présenté les résultats de nos expérimentations visant à mesurer la capacité de détection d'erreurs et les performances à l'exécution de nos implémentations. Nous avons notamment comparé les performances de quatre implémentations du *store*. Nous avons également comparé deux implémentations de la recherche du masque du plus grand préfixe commun, dans le cas où le *store* est implémenté en utilisant un Patricia trie. Enfin, nous avons mesuré l'impact de l'utilisation d'une analyse statique visant à réduire l'instrumentation du programme (section 6 de [Kosmatov et al., 2013]), pour chacune des

implémentations du *store*. Nos expérimentations ont mis en avant l'efficacité d'une implémentation du *store* à base de Patricia trie. Elles ont également montré qu'une recherche du masque du plus grand préfixe commun par dichotomie (l'algorithme est présenté en partie 5.2.2) est plus efficace qu'une recherche linéaire et que l'analyse statique permet effectivement de réduire le temps de la vérification à l'exécution.

Le travail réalisé a permis de concevoir une méthode et un outil de vérification à l'exécution des annotations E-ACSL portant sur le modèle mémoire. Nos travaux ont permis de déterminer expérimentalement quelles implémentations du *store* et des différents algorithmes sont les plus efficaces pour notre outil de vérification. L'implémentation du *store* à base de Patricia trie et les différents algorithmes présentés dans ce chapitre et dans le chapitre 5 sont intégrés à l'outil E-ACSL2C [Kosmatov et al., 2013].

	↯	∅	list	list-AS	bst	bst-AS	Pt	Pt-dicho	Pt-AS	Pt-dicho-AS	St	St-AS	valgrind
bS_{10000}	22	.01	1.10	0.50	1.14	0.64	1.55	1.55	0.57	0.55	1.39	0.61	0.27
+ RTE	41		1.10	0.51	1.14	0.62	1.59	1.59	0.53	0.53	1.39	0.64	
iS_{10000}	5	.12	2.29	0.12	1.83	0.12	2.89	2.91	0.12	0.12	2.46	0.12	2.81
+ RTE	24		3.52	1.27	2.89	1.26	3.99	3.86	1.25	1.25	3.46	1.30	
mM_{100^2}	0	.01	2.29	0.73	2.94	1.15	0.14	0.14	0.10	0.09	1.07	0.98	0.34
+ RTE	82		13.17	10.66	21.92	17.39	2.78	3.00	2.64	2.82	75.97	73.62	
mM_{150^2}	0	.01	13.23	3.96	15.65	6.20	0.54	0.51	0.36	0.35	5.86	5.64	0.48
+ RTE	82		72.36	58.48	110.70	90.43	10.77	9.01	8.57	8.75	403.50	398.60	
mI_{100^2}	2	.01	22.51	0.10	7.74	0.13	0.09	0.08	0.01	0.01	0.19	0.10	0.35
+ RTE	155		28.96	4.22	13.67	5.48	0.54	0.55	0.53	0.47	26.37	26.16	
mI_{150^2}	2	.02	130.04	0.34	40.35	0.45	0.28	0.27	0.02	0.02	0.68	0.34	0.47
+ RTE	155		153.30	21.54	73.55	29.94	2.00	1.90	1.42	1.53	146.15	145.80	
qS_{1000}	15	.01	12.70	2.08	1.76	0.59	0.33	0.06	0.13	0.02	0.02	0.01	0.27
+ RTE	32		12.38	2.13	1.64	0.56	0.38	0.12	0.14	0.04	0.03	0.02	
qS_{2000}	15	.01	85.99	11.31	8.39	2.78	0.71	0.14	0.28	0.05	0.03	0.02	0.27
+ RTE	32		81.65	11.15	7.72	2.67	1.13	0.48	0.36	0.13	0.05	0.02	
bbS_{10000}	4	.22	13.78	1.02	16.84	1.67	117.47	22.36	1.57	1.54	8.80	1.67	3.36
+ RTE	16		23.08	4.64	30.69	7.16	107.05	32.58	7.26	6.90	17.29	7.21	
m_{30000}	2	.01	412.10	11.38	176.35	11.01	1.11	0.26	0.30	0.06	0.08	0.01	0.45
+ RTE	49		451.58	101.33	219.12	94.80	1.15	0.29	0.47	0.14	0.10	0.05	
Rbt_{10000}	0	.01	47.39	0.28	48.44	0.27	0.32	0.09	0.03	0.01	0.59	0.01	0.51
+ RTE	270		120.02	101.69	165.77	145.20	0.47	0.30	0.39	0.27	18.82	19.59	
mS_{1000}	7	.01	6.45	0.34	6.32	0.11	0.32	0.07	0.06	0.01	0.02	0.01	0.27
+ RTE	45		6.82	1.35	7.98	0.38	0.34	0.10	0.13	0.04	0.02	0.01	
mS_{5000}	7	.01	362.87	11.00	218.01	3.43	2.28	0.76	0.43	0.09	0.14	0.03	0.27
+ RTE	45		371.40	50.94	290.88	10.34	2.46	0.80	0.83	0.22	0.16	0.08	
mS_{10000}	7	.01	3624.01	47.94	1673.00	16.10	6.46	2.75	1.00	0.21	0.31	0.08	0.27
+ RTE	45		3406.43	257.18	2086.32	46.22	6.30	2.66	1.83	0.51	0.35	0.18	
mS_{50000}	7	.01	∞	3847.72	∞	1100.93	135.54	111.22	6.90	1.65	2.08	0.58	0.63
+ RTE	45		∞	25554.08	∞	2781.90	118.86	95.74	11.64	3.37	2.18	1.15	
mS_{100000}	7	.01	∞	∞	∞	∞	631.41	559.93	13.55	3.35	4.03	1.15	0.27
+ RTE	45		∞	∞	∞	∞	573.47	513.85	25.02	7.63	4.68	2.50	

FIGURE 9.4 – Comparaison des différentes implémentations du *store*

10

GREFFON D'ANALYSE STATIQUE-DYNAMIQUE (STADY)

Dans ce chapitre nous présentons l'architecture générale du greffon STADY, notre implémentation de la méthode présentée aux chapitres 6, 7 et 8. Nous présentons quelques éléments d'implémentation que nous avons jugé importants en partie 10.1. En partie 10.2 nous présentons les résultats de nos expérimentations visant à mesurer la capacité de diagnostic des échecs de preuve et les performances de STADY.

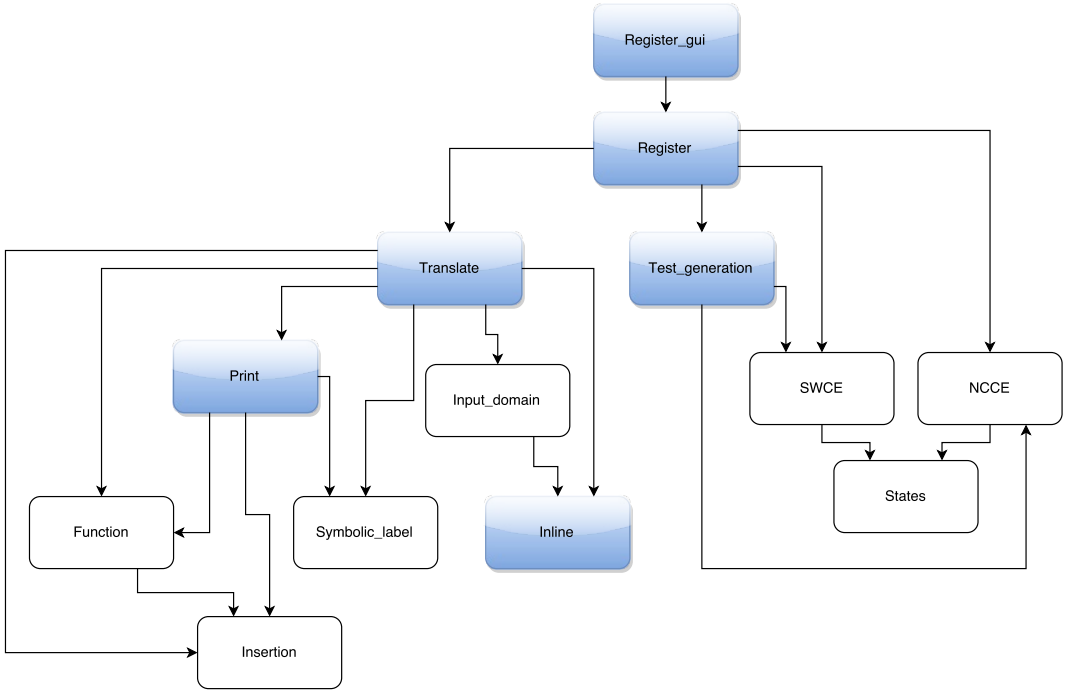

FIGURE 10.1 – Dépendances entre les modules de STADY

Module	Lignes de code	Lignes de commentaires
FUNCTION	33	0 (0%)
INLINE	168	19 (10%)
INPUT_DOMAIN	430	6 (1%)
INSERTION	55	0 (0%)
NCCE	55	0 (0%)
PRINT	108	3 (2%)
REGISTER_GUI	83	0 (0%)
REGISTER	186	1 (0%)
STATES	76	6 (7%)
SWCE	64	0 (0%)
SYMBOLIC_LABEL	21	0 (0%)
TEST_GENERATION	147	8 (5%)
TRANSLATE	1413	20 (1%)
Autres	827	58 (7%)
Total	3236	115 (3%)

FIGURE 10.2 – Lignes de code et de commentaires par module

10.1 IMPLÉMENTATION DU GREFFON STADY

La figure 10.1 présente l'architecture de STADY, elle offre une vue simplifiée des relations entre les différents modules OCaml (nous ferons l'amalgame entre les modules et les fichiers) qui composent l'outil. Chaque bloc représente un module et la relation $A \rightarrow B$ symbolise le fait que A fait appel à B. Les blocs dont le fond est blanc représentent des modules décrivant un ou plusieurs types de données et implémentant les opérations sur ce(s) type(s). Les blocs dont le fond est coloré représentent des modules implémentant une fonctionnalité importante de l'outil.

La figure 10.2 détaille le nombre de lignes de code et de commentaires (en ne comptant pas les lignes vides) de l'implémentation de STADY. Au total, STADY compte un peu plus de 3200 lignes de code OCaml, ce qui correspond à un greffon de FRAMA-C de taille moyenne. En effet, le greffon E-ACSL2C, de complexité similaire, compte environ 4200 lignes de code OCaml, ce qui reste loin derrière les greffons les plus importants et complexes de FRAMA-C, VALUE et WP, qui comptent chacun environ 30000 lignes de code OCaml.

Le module REGISTER pilote les autres modules de STADY, tandis que le module REGISTER_GUI implémente les comportements souhaités du greffon dans l'interface graphique de FRAMA-C. Les deux fonctionnalités centrales de l'outil sont, conformément à ce qui est décrit dans le chapitre 4, la traduction des annotations, effectuée par le module TRANSLATE, et la génération de tests, pilotée par le module TEST_GENERATION. Les parties 10.1.1 et 10.1.2 présentent ces opérations et les modules qui les composent.

10.1.1 TRADUCTION DES ANNOTATIONS

Afin de traduire les annotations E-ACSL en C nous avons besoin de définir des types de données spécifiques. Le module INSERTION définit le type des listes d'insertions de code telles qu'elles sont définies au chapitre 4. Le module SYMBOLIC_LABEL définit les labels symboliques qui permettent d'associer un endroit du programme aux insertions de code (début ou fin d'une fonction, début ou fin d'une boucle, avant ou après une instruction). Le module FUNCTION définit le type de données des fonctions générées de toute pièce par la traduction. Le module INPUT_DOMAIN définit des types de données représentant les domaines et les différentes contraintes sur les variables en entrée du programme (c'est-

à-dire les variables en entrée de la fonction sous vérification et les variables globales).

Les types définis par FUNCTION et INSERTION sont des extensions des types de l'arbre de syntaxe abstraite fournis par FRAMA-C, ils nous permettent de ne pas modifier l'arbre de syntaxe abstraite du programme et de stocker les insertions de code en dehors de l'arbre. En effet, la modification de l'arbre de syntaxe abstraite par les greffons de FRAMA-C est une opération complexe car il faut maintenir les relations des données entre l'état *avant* et *après* la modification, ce qui est source d'erreurs. Notre manière de faire a permis de réduire drastiquement le temps de développement de STADY.

Le module INLINE définit une transformation de programme dépliant les prédicats et fonctions logiques dans les annotations. En d'autres termes, cette transformation remplace un appel à un prédicat ou une fonction par sa définition et remplace les paramètres formels par les paramètres effectifs à l'appel. Ce module permet de traiter à moindre coût les annotations contenant des appels de fonctions logiques ou de prédicats définis par l'utilisateur. En revanche il ne permet pas de supporter les fonctions récursives ou les prédicats inductifs. Le module INLINE fournit la fonction pred qui produit la version dépliée du prédicat passé en paramètre. Son profil est le suivant :

```
val pred : Cil_types.predicate → Cil_types.predicate
```

Le module TRANSLATE définit les transformations de programme à opérer lors de la traduction des annotations E-ACSL en C. Ce module fournit une fonction translate dont le profil est le suivant :

```
val translate:
  Property.t list →
  int list →
  precond_fname:string →
  instru_fname:string →
  Property.t list
```

La fonction translate produit le fichier C (étiqueté instru_fname) contenant le programme instrumenté et le fichier Prolog (étiqueté precond_fname) contenant la précondition de la fonction sous vérification dans un format qui est plus efficace à traiter pour PATHCRAWLER. Elle retourne la liste de propriétés du programme qui ont été effectivement traduites, ce qui évitera de valider des propriétés qui n'ont pas été traduites. La traduction implémentée par ce module utilise un "visiteur en place" (par opposition au "visiteur de copie") de FRAMA-C [Signoles et al., 2015, section 4.16], ce choix n'est pas pénalisant puisque nous ne modifions pas l'arbre de syntaxe abstraite et nous stockons les insertions de code en dehors de l'arbre.

Le module PRINT est invoqué par le module TRANSLATE afin d'écrire dans les fichiers correspondants le résultat de la traduction qui incorpore les nouvelles fonctions ainsi que les insertions de code aux endroits adéquats. Ce module fournit la classe print_insertions suivante :

```
class print_insertions:
  (Symbolic_label.t, Insertion.t Queue.t) Hashtbl.t →
  Function.t list →
  int list →
  Printer.extensible_printer
```

Cette classe prend en argument les insertions de code étiquetées par les labels symboliques, les fonctions générées, les contrats paramétrant SWD et définit un objet de type

printer qui fera l'affichage et sera utilisé par la fonction translate .

Une fois les fichiers générés, la génération de tests débute.

10.1.2 GÉNÉRATION DE TESTS

Nous définissons maintenant les types de données qui nous permettent de manipuler les résultats de la génération de tests.

Le module NCCE implémente le type de données éponyme tel qu'il est défini au chapitre 6, ainsi que les fonctions d'enregistrement d'un NCCE et de récupération d'un ou plusieurs NCCEs pour une propriété donnée.

Le module SWCE implémente le type de données éponyme tel qu'il est défini au chapitre 7, ainsi que les fonctions d'enregistrement et de récupération d'un ou plusieurs SWCEs pour une propriété donnée.

Le module STATES définit l'état interne de STADY : des tables de hachage stockent notamment les contre-exemples NCCE et SWCE générés par PATHCRAWLER.

Le module TEST_GENERATION définit un protocole de communication pour communiquer avec PATHCRAWLER et ouvre un canal de communication (*socket*) entre STADY et PATHCRAWLER afin de récupérer les résultats du test. Ce module fournit la fonction run suivante :

```
val run: entry_point:string →
  precondition_filename:string →
  instrumented_filename:string →
  unit
```

La fonction run construit la ligne de commande qui permettra d'invoquer PATHCRAWLER avec les paramètres adéquats, démarre la communication avec PATHCRAWLER, enregistre les résultats de la génération de tests dans les états de STATES au fur et à mesure, puis rend la main à REGISTER à la fin de la communication.

Enfin, REGISTER met à jour le statut des propriétés du programme en fonction des résultats de la génération de tests.

10.2 EXPÉRIMENTATIONS

L'implémentation actuelle de STADY supporte une partie significative d'E-ACSL comprenant les assertions, les pré- et postconditions, les invariants et les variants de boucle, les prédicats quantifiés `\exists` et `\forall`, les fonctions logiques, les pointeurs, et une partie de l'arithmétique des pointeurs. Notre support du prédicat `\valid` ne prend en compte que les tableaux et pointeurs globaux ou en entrée du programme. Les clauses `assigns` sont uniquement traitées lors de la phase de SWD : nous n'avons pas pour but de trouver les éléments manquants d'une clause `assigns` (NCD) puisque les prouveurs donnent en général suffisamment d'indications dans ce cas, mais nous voulons trouver les éléments pouvant être enlevés d'une clause `assigns` (SWD). STADY traite partiellement les termes `\at` et ne traite pas les fonctions récursives, l'arithmétique réelle et les annotations liées au modèle mémoire (elles sont en revanche implémentées dans la bibliothèque de monitoring présentée au chapitre 9). Notre support de l'arithmétique des pointeurs ne com-

Exemple	Temps (s.)	Chemins explorés
array-unsafe	1.299	9
count-up-down-unsafe	1.285	3
eureka-01-unsafe	1.355	48
for-bounded-loop1-unsafe	1.320	11
insertion-sort-unsafe	16.530	730
invert-string-unsafe	1.359	48
linear-search-unsafe	3.624	2766
matrix-unsafe	1.367	22
nec20-unsafe	1.463	1035
string-unsafe	1.362	48
sum01-bug02-base-unsafe	1.335	26
sum01-bug02-unsafe	1.327	36
sum01-unsafe	1.312	56
sum03-unsafe	1.291	46
sum04-unsafe	1.310	22
sum-array-unsafe	1.358	14
trex03-unsafe	1.358	21
sendmail-unsafe	1.396	77
vogal-unsafe	1.349	341

FIGURE 10.3 – Détection de non-conformités : temps d'exécution

prend pas la différence de pointeurs (`p1-p2`) et les offsets négatifs (`*(p-i)`), ce qui est dû aux limitations du générateur de tests PATHCRAWLER.

Les hypothèses que nous souhaitons valider par nos expérimentations sont les suivantes.

H1 STADY est capable de diagnostiquer la plupart des échecs de preuve dans les programmes C.
H2 SWD apporte un avantage significatif par rapport à NCD en terme de capacité de diagnostic des échecs de preuve.
H3 STADY est capable de générer des NCCEs et des SWCEs même avec une couverture partielle des chemins du programme par le test.
H4 Le temps d'exécution de STADY est comparable au temps d'une preuve automatique.

Afin de vérifier ces hypothèses nous avons réalisé deux campagnes d'expérimentations. La première mesure la capacité de STADY à détecter les non-conformités dans les programmes avec NCD. La seconde campagne mesure la capacité de STADY à détecter les faiblesses de sous-contrats par SWD, ainsi que l'apport de SWD par rapport à NCD pour le diagnostic des échecs de preuve.

Les expériences ont été menées sur un processeur Intel Core i7-3520M 3.00GHz $\times$ 4 sous Linux Ubuntu 14.04.

10.2.1 DÉTECTION DE NON-CONFORMITÉS

Protocole expérimental. Afin d'évaluer la capacité de détection de non-conformités de NCD, nous l'avons appliqué sur des programmes corrects et incorrects de la compétition de vérification logicielle de TACAS 2014 [1].

Premièrement, nous avons exécuté STADY sur 20 programmes erronés manipulant des tableaux et des boucles. Les propriétés à invalider étaient à l'origine exprimées sous la forme d'assertions C, que nous avons manuellement remplacées par des assertions E-ACSL équivalentes. Des préconditions E-ACSL appropriées ont également été ajoutées.

1. https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/loops

Exemple	Mutants	Mutants non équivalents	Mutants tués	Succès
merge-sort	96	92	88	95.65%
merge-arrays	68	63	59	93.65%
quick-sort	130	130	130	100%
binary-search	40	40	39	97.5%
bubble-sort	52	49	42	85.71%
insertion-sort	39	37	36	97.3%
array-safe	18	16	15	93.75%
bubble-sort-safe	64	58	55	94.83%
count-up-down-safe	14	13	13	100%
eureka-01-safe	60	60	60	100%
eureka-05-safe	36	36	36	100%
insertion-sort-safe	43	41	40	97.56%
invert-string-safe	47	47	47	100%
linear-search-safe	19	17	16	94.12%
matrix-safe	30	27	25	92.59%
nc40-safe	20	20	20	100%
nec40-safe	20	20	20	100%
string-safe	65	65	65	100%
sum01-safe	14	14	13	92.86%
sum02-safe	14	14	11	78.57%
sum03-safe	10	10	10	100%
sum04-safe	14	14	10	71.43%
sum-array-safe	17	17	15	88.24%
trex03-safe	56	56	56	100%
sendmail-safe	31	31	31	100%
vogal-safe	71	68	67	98.53%
Total	1088	1054	1019	**96.68%**

FIGURE 10.4 – Détection de non-conformités : test mutationnel

Les programmes contenant des boucles infinies et des propriétés d'atteignabilité à invalider ne sont pas traités car il n'est pas possible de les exécuter avec PATHCRAWLER.

Deuxièmement, nous avons utilisé le test mutationnel afin d'évaluer la capacité de STADY à détecter les non-conformités dans des programmes incorrects. Nous avons considéré 20 programmes corrects du même *benchmark* et 6 programmes corrects supplémentaires classiques (fusion de tableaux, recherche par dichotomie et différents tris de tableaux). Nous avons annoté ces programmes avec E-ACSL afin d'exprimer les préconditions, les postconditions, la validité des pointeurs, les invariants et variants des boucles. Nous avons utilisé le test mutationnel sur ces programmes corrects afin de générer des programmes modifiés (appelés *mutants*) et de voir si STADY est capable de *tuer*, c'est-à-dire de trouver une erreur, dans ces mutants. Les mutations sont appliquées sur le code uniquement et simulent des erreurs de programmation classiques. Elles peuvent être une modification d'opérateur arithmétique (numérique ou sur les pointeurs), une modification d'opérateur de comparaison, la négation d'une condition ou la négation d'un opérateur logique (de conjonction ou de disjonction). Ces mutations peuvent générer des mutants équivalents au programme d'origine. Nous déterminons si un mutant est équivalent par analyse manuelle des mutants non tués.

Résultats expérimentaux. Pour chacun des programmes erronés considérés dans la première phase de cette expérimentation, STADY a détecté la non-conformité des annotations. La figure 10.3 présente ces exemples et le temps qui a été nécessaire pour invalider les propriétés et le nombre de chemins d'exécution explorés par la génération de tests. Les résultats de la deuxième phase de l'expérimentation sont présentés dans la figure 10.4. La figure 10.4 indique le nombre total de mutants, le nombre de mutants non équivalents, le nombre de mutants non-équivalents tués par NCD et la proportion de mutants non-équivalents tués par NCD. Pour cette campagne d'expérimentations, NCD

a détecté une non-conformité dans 1019 des 1054 mutants non-équivalents. Le taux de détection des non-conformités par StaDy est en moyenne de 96.68% et a atteint 100% sur de nombreux programmes. Ce taux n'est pas parfait car le langage E-ACSL n'est pas exhaustivement traité par StaDy et par le générateur de tests PathCrawler. Cette première campagne d'expérimentations permet donc de valider partiellement l'hypothèse **H1** puisque l'étape NCD de StaDy permet à elle seule de diagnostiquer la plupart des échecs de preuve dans les programmes C.

10.2.2 Détection de faiblesses de sous-contrats

Évaluons à présent la capacité de StaDy à détecter les faiblesses de sous-contrats par SWD et l'apport de SWD par rapport à NCD pour le diagnostic des échecs de preuve.

Protocole expérimental. Nous avons évalué StaDy sur 20 programmes annotés de [Burghardt et al., 2015], dont la taille varie de 35 à 100 lignes de code annoté. Ces programmes manipulent des tableaux, ils sont entièrement annotés avec E-ACSL et leurs spécifications expriment des propriétés non triviales sur les tableaux. Ils sont corrects et prouvés par Wp. Nous appliquons StaDy sur des versions modifiées (ou *mutants*) des programmes C, qui sont générées de manière systématique. Chaque programme mutant est obtenu par l'application d'une *mutation* unique sur le programme d'origine. Les mutations sont les suivantes : modification d'un opérateur binaire dans le code ou dans la spécification, négation d'une condition dans le code, modification d'une relation dans la spéficiation, négation d'un prédicat dans la spécification, suppression d'une partie d'un invariant de boucle ou d'une postcondition dans la spécification. Nous n'appliquons pas de mutation à la précondition de la fonction sous vérification, et nous limitons les mutations possibles pour chaque opérateur binaire afin d'éviter de créer des expressions absurdes, notamment dans le cas des opérations arithmétiques sur les pointeurs.

La première étape consiste à essayer de prouver chaque mutant à l'aide de Wp. Les mutants prouvés respectent leur spécification et sont classifiés comme étant corrects. Nous appliquons ensuite la méthode NCD aux mutants restants. Cette étape classifie l'échec de preuve pour certains mutants comme étant due à une non-conformité, indiquant l'annotation violée ainsi qu'un NCCE. Une troisième étape est l'application de la méthode SWD aux mutants non encore classifiés, qui permet de classer certains d'entre eux comme ayant une faiblesse de sous-contrats, en indiquant l'annotation non prouvée, les contrats trop faibles et un SWCE. Si aucun contre-exemple n'a été trouvé par SWD, le mutant reste non classifié.

Résultats expérimentaux. La figure 10.5 présente les résultats de ces expérimentations. La colonne #mut contient le nombre de mutants générés pour chaque exemple. Les trois colonnes de l'étape *Preuve* contiennent respectivement le nombre de mutants prouvés (#✓), le temps d'exécution maximal et moyen (affiché sur deux lignes) pour les mutants prouvés ($t^{✓}$) et pour les mutants non prouvés ($t^{?}$). Notons que les mutants prouvés sont les mutants équivalents. Les quatre colonnes de l'étape NCD contiennent respectivement le nombre de mutants classifiés par NCD (#✗), la proportion de mutants classifiés par NCD parmi les mutants non prouvés (%), le temps d'exécution maximal et moyen pour les mutants classifiés par NCD ($t^{✗}$) et pour les mutants non classifiés ($t^{?}$). Les quatre colonnes de l'étape SWD contiennent respectivement le nombre de mutants classifiés par SWD (#✗), la proportion de mutants classifiés par SWD parmi les mutants non classifiés par NCD (%), le temps d'exécution maximal et moyen pour les mutants classifiés par SWD

($t^{\text{✗}}$) et pour les mutants non classifiés ($t^{?}$). Les deux colonnes NCD + SWD synthétisent les résultats des étapes NCD et SWD. La colonne % contient la proportion de mutants non équivalents classifiés par NCD ou SWD. La colonne t contient le temps maximal et le temps moyen pour tous les mutants sans distinction du résultat de la classification. La mesure du temps d'exécution s'arrête à la fin de la preuve (en cas de succès) ou au premier NCCE ou SWCE généré. La colonne # ? contient le nombre de mutants restant non classifiés à la fin de l'expérimentation.

Pour les 20 programmes considérés, 928 mutants ont été générés. 80 d'entre eux ont été prouvés à l'aide de WP. Parmi les 848 mutants non prouvés, NCD a détecté une non-conformité introduite par une mutation dans 776 mutants (91.5%), laissant 72 mutants non classifiés. Parmi ces 72 mutants, SWD a pu générer un contre-exemple (soit un NCCE soit un SWCE) pour 48 d'entre eux (66.7%), laissant finalement 24 programmes non classifiés. Ces 24 mutants peuvent être des mutants équivalents qui n'ont pas été prouvés par WP pour cause d'une incapacité de prouveur, ou des mutants dont la mutation s'est produite dans une annotation non traitée par STADY (la mutation étant actuellement indétectable), ou des mutants non équivalents pour lesquels la génération de tests n'a généré aucun contre-exemple mais est incomplète (c'est-à-dire qu'elle a été interrompue car elle ne respectait pas la limite de temps imposée).

Concernant **H1**, STADY a trouvé une raison précise pour les échecs de preuve et a produit un contre-exemple pour 824 des 848 mutants non prouvés, et a donc classifié 97.2% d'entre eux. Décider pour chaque mutant non classifié si un échec de preuve est causé par une incapacité de prouveur nécessite souvent de réduire le domaine des variables d'entrée, d'ajouter des lemmes supplémentaires ou d'essayer une preuve interactive (avec COQ par exemple). Cette dernière étape n'a pas été suffisamment approfondie dans nos travaux.

Concernant **H2**, NCD a diagnostiqué 776 des 848 mutants non prouvés (91.5%). SWD a diagnostiqué 48 des 72 mutants restants (66.7%), ce qui a contribué de manière significative (et complémentaire à NCD) au diagnostic et à l'explication de certains échecs de preuve.

Dans nos expérimentations, nous laissons à chaque prouveur 40 secondes pour prouver chaque propriété. Nous laissons 5 secondes à chaque session de génération de tests. Rappelons qu'il y a au plus une session pour l'étape NCD par mutant, mais qu'il y a autant de sessions que nécessaire pour l'étape SWD, le nombre dépendant du nombre de sous-contrats. Nous limitons également la profondeur d'exploration des chemins du programme : nous considérons uniquement les chemins qui contiennent au plus quatre itérations consécutives de chaque boucle. Ceci est un paramètre de la génération de tests de PATHCRAWLER, nous dirons que nous utilisons le critère *k-path* du générateur, avec $k = 4$. Le *timeout* de chaque session de test et le *k-path* limitent fortement la couverture de test mais STADY détecte néanmoins 97.2% des mutations introduites dans les programmes mutants. Ceci confirme l'hypothèse **H3** et démontre que la méthode proposée peut classifier les échecs de preuve et générer des contre-exemples de manière efficace même lorsque la couverture de test est partielle. De plus, la méthode peut également être utilisée sur des programmes dont le nombre de chemins à explorer ne peut pas être limité.

Concernant **H4**, sur les programmes considérés, WP nécessite en moyenne 2.6 secondes par mutant (au plus 4.4 secondes) lorsqu'il réussit à prouver la correction du programme. Il passe en moyenne 13.0 secondes (au plus 61.3 secondes) par mutant lorsque la preuve

du programme échoue. Le temps d'exécution total de StaDy est comparable : StaDy nécessite en moyenne 2.7 secondes par mutant non prouvé (au plus 19.9 secondes). Plus précisément, l'étape NCD a besoin de 2.4 secondes en moyenne (au plus 9.4 secondes) pour détecter une non-conformité, et en moyenne 2.5 secondes (au plus 8.3 secondes) lorsqu'elle n'aboutit pas. L'étape SWD a besoin de 2.4 secondes en moyenne (au plus 6.4 secondes) pour détecter une faiblesse de sous-contrat, et en moyenne 6.3 secondes (au plus 11.6 secondes) lorsqu'elle n'aboutit pas.

Bilan des expérimentations. Nos expérimentations ont montré que la méthode que nous proposons permet de classifier automatiquement une proportion importante des échecs de preuve, dans un temps comparable au temps d'une preuve automatique, et pour des programmes pour lesquels uniquement une couverture de test partielle est possible. SWD complète efficacement NCD afin d'obtenir un diagnostic plus complet et plus précis des échecs de preuve.

Validité des expérimentations. Comme c'est souvent le cas dans le domaine de la vérification logicielle, une des principales menaces à la validité de nos expérimentations est liée à la représentativité des résultats. Dans notre cas, la nature du problème nous restreint aux programmes annotés et réalistes qui ne peuvent pas être générés automatiquement ou être récupérés dans des bases de données de code non spécifié existantes. Afin de réduire cette atteinte à la validité de nos expérimentations, nous avons utilisé des programmes provenant d'un *benchmark* indépendant [Burghardt et al., 2015], créé pour illustrer les différents usages du langage de spécification ACSL pour la vérification déductive de programmes avec Frama-C.

Le passage à l'échelle des résultats est également une autre menace potentielle puisque nous n'avons pas démontré leur validité pour des programmes de plus grande taille. Néanmoins, du fait du raisonnement modulaire de la vérification déductive, nous sommes persuadés que la technique proposée devrait uniquement être appliquée au niveau unitaire, chaque fonction prise séparément, puisqu'un ingénieur validation essaiera de prouver un programme de cette manière.

La préoccupation majeure du passage à l'échelle est liée à l'utilisation du test structurel qui peut souvent atteindre le *timeout* fixé sans avoir couvert tous les chemins d'exécution faisables du programme. Afin de résoudre ce problème, nous avons étudié l'impact d'une couverture de test partielle sur l'efficacité de la méthode (voir **H3**) et proposé une manière simple de réduire le domaine des variables d'entrée (en utilisant la clause `typically`).

Un autre obstacle à la validité peut être dû aux mesures utilisées. Afin de réduire cette menace, nous avons mesuré de manière précise nos résultats, ce qui inclut le temps d'analyse pour chaque étape et chaque mutant, les valeurs moyennes et maximales, en séparant les échecs de preuve classifiés et non classifiés. Une de nos préoccupations est de produire des situations réalistes dans lesquelles un ingénieur validation peut avoir besoin d'aide afin de diagnostiquer des échecs de preuve. Bien que les premiers utilisateurs de StaDy aient apprécié ses résultats, nous n'avons pas encore eu l'occasion d'évaluer proprement la pertinence des résultats avec un groupe représentatif d'utilisateurs. En attendant de conduire une telle expérimentation, nous avons simulé les erreurs des utilisateurs par des mutations, ce qui permet de recréer des situations fréquentes et problématiques (code incorrect, annotation incorrecte, spécification incomplète) aboutissant à des échecs de preuve. Cette approche semble appropriée pour les non-conformités et les faiblesses de sous-contrats, mais l'est certainement moins pour les cas plus subtils d'incapacités de prouveur. Ces résultats devraient être confirmés à l'avenir par une

évaluation de StaDy par un échantillon représentatif d'utilisateurs réels.

Conclusion du chapitre

Dans ce chapitre, nous avons présenté l'architecture du greffon StaDy qui implémente la méthode présentée aux chapitres 6, 7 et 8. Nous avons également présenté les résultats de nos expérimentations évaluant la capacité de diagnostic des échecs de preuve et les performances de StaDy. Ces expérimentations ont été menées sur un échantillon de programmes provenant d'un *benchmark* indépendant pour ne pas fausser les résultats. Nous avons simulé les erreurs de programmation et de spécification par une technique automatique de mutation. Ces expérimentations ont montré que la méthode proposée permet de classifier automatiquement une proportion importante des échecs de preuve, dans un temps comparable au temps d'une preuve automatique, et pour des programmes pour lesquels uniquement une couverture de test partielle est possible. Nos expérimentations montrent également que SWD complète efficacement NCD afin d'obtenir un diagnostic plus complet et plus précis des échecs de preuve.

Il faudrait maintenant évaluer StaDy, d'une part sur des exemples de plus grande taille, d'autre part avec un échantillon représentatif d'utilisateurs réels.

		Preuve			NCD				SWD				NCD + SWD		
Exemple	#mut	#✓	$t^{✓}$	$t^{?}$	#✗	%	$t^{✗}$	$t^{?}$	#✗	%	$t^{✗}$	$t^{?}$	%	t	#?
binary_search	99	5	$\leqslant 3.4$ ≈ 3.3	$\leqslant 50.1$ ≈ 19.9	86	91.5	$\leqslant 9.4$ ≈ 2.6	$\leqslant 2.7$ ≈ 2.6	8	100.0	$\leqslant 5.9$ ≈ 3.7	— —	100.0	$\leqslant 9.4$ ≈ 2.9	0
binary_search2	99	5	$\leqslant 3.3$ ≈ 3.3	$\leqslant 50.4$ ≈ 19.9	86	91.5	$\leqslant 5.4$ ≈ 2.5	$\leqslant 2.6$ ≈ 2.5	8	100.0	$\leqslant 6.4$ ≈ 3.9	— —	100.0	$\leqslant 9.0$ ≈ 2.8	0
lower_bound	52	1	$\leqslant 2.5$ ≈ 2.4	$\leqslant 49.4$ ≈ 26.2	46	90.2	$\leqslant 5.1$ ≈ 2.3	$\leqslant 2.4$ ≈ 2.3	5	100.0	$\leqslant 2.3$ ≈ 2.2	— —	100.0	$\leqslant 5.1$ ≈ 2.5	0
upper_bound	52	1	$\leqslant 2.6$ ≈ 2.5	$\leqslant 49.8$ ≈ 26.7	46	90.2	$\leqslant 5.2$ ≈ 2.2	$\leqslant 2.4$ ≈ 2.3	5	100.0	$\leqslant 2.3$ ≈ 2.2	— —	100.0	$\leqslant 5.2$ ≈ 2.5	0
max_element	52	5	$\leqslant 3.6$ ≈ 2.9	$\leqslant 20.5$ ≈ 9.1	41	87.2	$\leqslant 5.3$ ≈ 2.8	$\leqslant 2.5$ ≈ 2.4	3	50.0	$\leqslant 2.6$ ≈ 2.4	$\leqslant 4.8$ ≈ 4.7	93.6	$\leqslant 7.3$ ≈ 3.2	3
max_element2	52	3	$\leqslant 3.4$ ≈ 2.9	$\leqslant 20.1$ ≈ 8.9	43	87.8	$\leqslant 5.7$ ≈ 2.5	$\leqslant 2.6$ ≈ 2.5	3	50.0	$\leqslant 2.5$ ≈ 2.4	$\leqslant 5.3$ ≈ 5.0	93.9	$\leqslant 7.8$ ≈ 3.0	3
max_seq	125	26	$\leqslant 4.4$ ≈ 3.9	$\leqslant 22.1$ ≈ 9.4	87	87.9	$\leqslant 6.3$ ≈ 3.0	$\leqslant 3.7$ ≈ 3.1	0	0.0	— —	$\leqslant 6.4$ ≈ 6.1	87.9	$\leqslant 9.5$ ≈ 3.8	12
min_element	52	3	$\leqslant 3.4$ ≈ 3.0	$\leqslant 16.3$ ≈ 8.5	43	87.8	$\leqslant 5.3$ ≈ 2.6	$\leqslant 2.9$ ≈ 2.6	3	50.0	$\leqslant 2.8$ ≈ 2.5	$\leqslant 5.8$ ≈ 5.3	93.9	$\leqslant 8.4$ ≈ 3.1	3
copy	23	2	$\leqslant 2.5$ ≈ 2.3	$\leqslant 40.9$ ≈ 12.6	20	95.2	$\leqslant 2.5$ ≈ 2.1	$\leqslant 2.0$ ≈ 2.0	1	100.0	$\leqslant 2.0$ ≈ 2.0	— —	100.0	$\leqslant 4.0$ ≈ 2.2	0
fill	23	2	$\leqslant 1.9$ ≈ 1.9	$\leqslant 8.5$ ≈ 4.3	20	95.2	$\leqslant 1.9$ ≈ 1.9	$\leqslant 2.5$ ≈ 2.5	1	100.0	$\leqslant 1.9$ ≈ 1.9	— —	100.0	$\leqslant 4.4$ ≈ 2.0	0
iota	29	2	$\leqslant 2.2$ ≈ 2.1	$\leqslant 12.9$ ≈ 7.8	25	92.6	$\leqslant 5.0$ ≈ 2.4	$\leqslant 2.1$ ≈ 2.0	2	100.0	$\leqslant 2.1$ ≈ 2.0	— —	100.0	$\leqslant 5.0$ ≈ 2.5	0
replace_copy	25	2	$\leqslant 3.2$ ≈ 3.2	$\leqslant 61.3$ ≈ 16.4	21	91.3	$\leqslant 5.5$ ≈ 3.2	$\leqslant 2.8$ ≈ 2.8	2	100.0	$\leqslant 2.5$ ≈ 2.4	— —	100.0	$\leqslant 5.5$ ≈ 3.4	0
reverse_copy	27	2	$\leqslant 2.0$ ≈ 1.9	$\leqslant 41.2$ ≈ 14.5	24	96.0	$\leqslant 2.1$ ≈ 2.1	$\leqslant 2.0$ ≈ 2.0	1	100.0	$\leqslant 1.9$ ≈ 1.9	— —	100.0	$\leqslant 4.0$ ≈ 2.1	0
adjacent_find	32	4	$\leqslant 2.8$ ≈ 2.5	$\leqslant 57.4$ ≈ 21.8	26	92.9	$\leqslant 5.3$ ≈ 2.4	$\leqslant 8.3$ ≈ 5.2	1	50.0	$\leqslant 2.4$ ≈ 2.4	$\leqslant 11.6$ ≈ 11.6	96.4	$\leqslant 19.9$ ≈ 3.1	1
equal	25	2	$\leqslant 2.7$ ≈ 2.3	$\leqslant 15.6$ ≈ 9.9	22	95.7	$\leqslant 5.1$ ≈ 2.4	$\leqslant 2.1$ ≈ 2.1	1	100.0	$\leqslant 2.1$ ≈ 2.1	— —	100.0	$\leqslant 5.1$ ≈ 2.5	0
equal2	25	2	$\leqslant 1.5$ ≈ 1.4	$\leqslant 15.3$ ≈ 9.3	22	95.7	$\leqslant 5.0$ ≈ 2.2	$\leqslant 2.4$ ≈ 2.4	1	100.0	$\leqslant 1.9$ ≈ 1.9	— —	100.0	$\leqslant 5.0$ ≈ 2.3	0
equal3	60	7	$\leqslant 2.7$ ≈ 2.6	$\leqslant 15.3$ ≈ 9.3	51	96.2	$\leqslant 3.4$ ≈ 2.5	$\leqslant 2.8$ ≈ 2.6	0	0.0	— —	$\leqslant 5.5$ ≈ 5.3	96.2	$\leqslant 8.3$ ≈ 2.7	2
find	25	2	$\leqslant 2.4$ ≈ 2.4	$\leqslant 20.6$ ≈ 8.8	22	95.7	$\leqslant 5.0$ ≈ 2.3	$\leqslant 2.0$ ≈ 2.0	1	100.0	$\leqslant 2.1$ ≈ 2.1	— —	100.0	$\leqslant 5.0$ ≈ 2.3	0
find2	26	2	$\leqslant 2.9$ ≈ 2.4	$\leqslant 15.6$ ≈ 8.2	23	95.8	$\leqslant 5.0$ ≈ 2.2	$\leqslant 2.4$ ≈ 2.4	1	100.0	$\leqslant 2.1$ ≈ 2.1	— —	100.0	$\leqslant 5.0$ ≈ 2.3	0
mismatch	25	2	$\leqslant 2.2$ ≈ 2.1	$\leqslant 15.0$ ≈ 9.3	22	95.7	$\leqslant 5.2$ ≈ 2.4	$\leqslant 2.3$ ≈ 2.3	1	100.0	$\leqslant 2.2$ ≈ 2.2	— —	100.0	$\leqslant 5.2$ ≈ 2.5	0
Total	928	80 /928			776 /848	91.5			48 /72	66.7			97.2		24
Max			$\leqslant 4.4$	$\leqslant 61.3$		96.2	$\leqslant 9.4$	$\leqslant 8.3$		100.0	$\leqslant 6.4$	$\leqslant 11.6$	100.0	$\leqslant 19.9$	
Mean			≈ 2.6	≈ 13.0		92.6	≈ 2.4	≈ 2.5		80.0	≈ 2.4	≈ 6.3	98.1	≈ 2.7	

FIGURE 10.5 – Diagnostic des échecs de preuve sur des mutants

11

BILAN ET PERSPECTIVES

Dans ce chapitre, nous rappelons les objectifs de la thèse en partie 11.1 puis nous présentons le bilan des travaux réalisés en partie 11.2 ainsi que les perspectives envisagées en partie 11.3.

11.1 RAPPEL DES OBJECTIFS

Notre objectif principal était de fournir une méthode de diagnostic automatique des échecs de preuve afin de décider si la cause d'un échec de preuve est une non-conformité entre le code et la spécification, ou la faiblesse d'un ou plusieurs sous-contrats de la spécification, ou enfin une incapacité de l'outil de vérification déductive. Nos travaux ont donc pour but d'améliorer et de faciliter le processus de spécification et de preuve des programmes qui est long et difficile et nécessite une connaissance des outils de preuve de programmes. En effet, il est souvent difficile pour l'utilisateur de décider laquelle de ces trois raisons est la cause de l'échec de la preuve car cette information n'est pas (ou rarement) donnée par le prouveur et requiert donc une revue approfondie du code et de la spécification.

Notre objectif était également de fournir une implémentation efficace de cette méthode de diagnostic des échecs de preuve, et d'évaluer les capacités de cet outil sur différents programmes.

11.2 BILAN

Nous avons d'abord développé une méthode de validation à l'exécution des annotations E-ACSL liées au modèlé mémoire et implémenté une bibliothèque C permettant de valider ces annotations à l'exécution.

Nous avons ensuite proposé une méthode originale d'aide à la vérification déductive, par génération de tests structurels sur une version instrumentée du programme d'origine. Cette méthode comporte une première étape de détection de non-conformités, puis une seconde étape de détection de faiblesses de sous-contrats (contrats de boucle ou de fonction appelée).

Nous avons dans un premier temps conçu une méthode de détection des non-conformités du code par rapport à la spécification dans les programmes C. Cette méthode

nécessite une traduction en C des annotations E-ACSL afin d'expliciter les violations potentielles d'annotations par la création de nouvelles branches dans le programme. La préservation de la sémantique par cette traduction nous assure de trouver les erreurs dans le code et dans la spécification par génération de tests structurels à condition de couvrir tous les chemins d'exécution faisables du programme. Nous avons formalisé cette traduction afin de justifier sa correction. Puis nous avons implémenté cette méthode dans STADY et expérimenté son efficacité par des mutations. Il est apparu que sur des programmes mutants, nous étions capables de détecter les erreurs introduites. En revanche lorsqu'aucune erreur n'a été détectée, le programme mutant peut être équivalent au programme d'origine, ou bien on peut avoir introduit une faiblesse de sous-contrat. Une telle faiblesse ne rend pas le programme invalide, mais empêche la vérification déductive du programme.

Nous avons alors décidé dans un second temps d'adapter notre méthode de détection des non-conformités afin de pouvoir détecter les faiblesses de sous-contrats. Cette nouvelle méthode nécessite une traduction en C des annotations E-ACSL différente de la première, car nous voulons pouvoir "exécuter" certains contrats au lieu du code qu'ils spécifient. Nous avons intégré cette méthode à STADY et expérimenté son efficacité par des mutations dans le cadre du diagnostic des échecs de preuve. Nos expérimentations ont permis de mettre en évidence le gain apporté par cette deuxième méthode par rapport à une détection des non-conformités seule.

Nous avons ensuite conçu une méthode combinant la détection des non-conformités et la détection des faiblesses de sous-contrats afin d'aider au diagnostic des échecs de preuve. Cette méthode combinée essaie d'abord d'exhiber des non-conformités. Si aucune non-conformité n'est détectée, elle recherche des faiblesses de sous-contrats. Si aucune de ces deux phases n'a fourni de raison quant à l'échec de la preuve et si la génération de tests a couvert tous les chemins d'exécution faisables du programme, alors l'échec de la preuve est dû à une incapacité du prouveur. Sinon le problème reste non résolu.

Nous avons implémenté les différentes composantes de la méthode dans le greffon STADY. La méthode de diagnostic elle-même n'est pas encore implémentée. D'autre part, l'implémentation souffre de quelques limitations dues au support incomplet du langage de spécification E-ACSL. Ces deux points sont laissés en perspectives.

Dans le cadre de travaux effectués avec R. Genestier et A. Giorgetti, STADY a été appliqué sur des programmes générant des tableaux structurés et a permis de faciliter la vérification déductive de ces programmes.

11.2.1 PUBLICATIONS ASSOCIÉES À LA THÈSE

Dans le cadre de nos travaux, nous avons présenté quatre communications de recherche lors de conférences internationales :

L'article [Kosmatov et al., 2013], publié à RV 2013, présente notre méthode de validation à l'exécution des annotations E-ACSL liées au modèle mémoire et l'implémentation d'une bibliothèque permettant de valider ces annotations à l'exécution.

L'article [Petiot et al., 2014b], publié à TAP 2014, détaille l'aspect méthodologique de notre méthode de détection des non-conformités entre le code et sa spécification.

L'article [Petiot et al., 2014a], publié à SCAM 2014, présente la traduction des annotations ACSL en C pour la détection de non-conformités entre le code et la spécification.

L'article [Genestier et al., 2015], publié à TAP 2015, recense plusieurs programmes C dont la vérification déductive a été facilitée par STADY.

Le rapport de recherche [Petiot et al., 2015] présente la méthode globale de diagnostic des échecs de preuve combinant la détection des non-conformités entre le code et la spécification et la détection des faiblesses de sous-contrats.

11.3 PERSPECTIVES

Nous suggérons maintenant quelques perspectives faisant suite à nos travaux dans le cadre des combinaisons d'analyses statiques et dynamiques et de l'aide à la vérification déductive.

11.3.1 FORMALISATION DE LA MÉTHODE

Compte tenu de l'importance de la traduction des annotations ACSL en C dans notre approche, aussi bien pour la détection des non-conformités que pour la détection des faiblesses de sous-contrats, la correction de cette traduction doit être assurée. Nous avons donné une première justification de cette correction dans le chapitre 4, mais il est possible d'aller plus loin dans cette direction et de formaliser cette traduction à l'aide d'un outil comme COQ [Castéran et al., 2004] ou WHY3 [Filliâtre et al., 2013], d'une manière similaire à [Herms et al., 2012].

Il serait également utile d'appliquer ce principe au générateur de tests. Ceci peut se traduire par la formalisation en COQ de certaines parties de PATHCRAWLER et/ou par l'utilisation d'un solveur de contraintes certifié comme celui formalisé dans [Carlier et al., 2012].

Ces deux pistes permettraient de s'assurer formellement de la correction des résultats produits par nos outils.

11.3.2 EXTENSION DE LA PRISE EN CHARGE D'ACSL

Dans nos travaux nous avons pris en charge la majorité des constructions du langage E-ACSL (sous-ensemble "exécutable" d'ACSL). Il serait peut-être possible de traiter des constructions du langage ACSL qui sont en dehors du langage E-ACSL, comme les prédicats inductifs. [Tollitte et al., 2012] propose une méthode d'extraction de code fonctionnel (pour COQ) à partir de spécifications inductives. Peut-être que ces travaux peuvent servir de base à une méthode de traduction des spécifications inductives en code impératif (en C notamment) ce qui permettrait de prendre en charge les prédicats inductifs dans STADY.

Notre implémentation ne traite pas non plus les annotations faisant intervenir les nombres réels mathématiques et les nombres flottants du C. De telles annotations permettent notamment d'exprimer les erreurs d'arrondi se produisant lors des calculs sur les nombres flottants [Goubault et al., 2011]. Leur intégration au sein de STADY nécessiterait de formaliser leur traduction en C et de pouvoir les exécuter symboliquement et concrètement

lors de la génération de tests, d'une manière similaire à ce que nous avons réalisé pour les entiers mathématiques d'E-ACSL.

11.3.3 AUTOMATISATION DE LA MÉTHODE

La méthode de détection des non-conformités et la méthode de détection des faiblesses de sous-contrats sont implémentées dans STADY, mais la méthode les combinant ne l'est pas. L'utilisation actuelle de STADY est la suivante. STADY doit être lancé une première fois en mode "détection de non-conformités"' (son mode par défaut). Puis, si aucune non-conformité n'a été détectée, STADY peut être lancé un nombre quelconque de fois en mode "détection de faiblesses de sous-contrats". Dans ce second mode, l'utilisateur indique à STADY pour quels contrats une faiblesse doit être recherchée. Il revient donc à l'utilisateur de réduire ou d'augmenter progressivement l'ensemble des contrats au fur et à mesure des exécutions de STADY.

Ce processus pourrait être automatisé au cours d'une seule exécution de STADY qui lancerait d'abord une détection de non-conformités puis une ou plusieurs détections de faiblesses de sous-contrats. Les contrats pour lesquels une faiblesse peut être recherchée pourraient être déterminés en analysant les dépendances entre les annotations du programme.

11.3.4 EXPÉRIMENTATIONS AVEC DES UTILISATEURS

Au cours de cette thèse nous avons expérimenté l'efficacité de notre outil en terme de capacité de diagnostic et de temps d'exécution. Néanmoins, nous n'avons pas pu mesurer l'impact réel de l'outil du point de vue d'un utilisateur qui a pour tâche de spécifier et vérifier un programme.

Nous pourrions par exemple former deux groupes d'utilisateurs débutants dans le domaine de la vérification déductive, avec le niveau le plus homogène possible. Les utilisateurs du premier groupe auraient uniquement à leur disposition l'outil de preuve, tandis que ceux du second groupe pourraient également utiliser STADY. Les deux groupes auraient à spécifier et prouver formellement des programmes et nous pourrions ainsi comparer les résultats produits par chacun des groupes.

Une telle expérimentation permettrait de quantifier et qualifier l'aide au diagnostic des échecs de preuve fourni par STADY et donc l'apport d'un tel outil pour la spécification et la vérification déductive des programmes.

BIBLIOGRAPHIE

[Pat, 2011] (2011). **PathCrawler Online**. http://pathcrawler-online.com.

[Ahn et al., 2010] Ahn, K., et Denney, E. (2010). **Testing first-order logic axioms in program verification**. Dans Fraser, G., et Gargantini, A., éditeurs, *Tests and Proofs*, volume 6143 de *Lecture Notes in Computer Science*, pages 22–37. Springer Berlin Heidelberg.

[Ahrendt et al., 2015] Ahrendt, W., Chimento, J., Pace, G., et Schneider, G. (2015). **A specification language for static and runtime verification of data and control properties**. Dans Bjørner, N., et de Boer, F., éditeurs, *FM 2015 : Formal Methods*, volume 9109 de *Lecture Notes in Computer Science*, pages 108–125. Springer International Publishing.

[Arndt, 2010] Arndt, J. (2010). **Matters Computational - Ideas, Algorithms, Source Code [The fxtbook]**. Published electronically at http ://www.jjj.de.

[Ball, 2005] Ball, T. (2005). **A theory of predicate-complete test coverage and generation**. Dans de Boer, F., Bonsangue, M., Graf, S., et de Roever, W.-P., éditeurs, *Formal Methods for Components and Objects*, volume 3657 de *Lecture Notes in Computer Science*, pages 1–22. Springer Berlin Heidelberg.

[Ball et al., 2011] Ball, T., Levin, V., et Rajamani, S. K. (2011). **A decade of software model checking with slam**. *Commun. ACM*, 54(7) :68–76.

[Barnett et al., 2006] Barnett, M., Chang, B.-Y., DeLine, R., Jacobs, B., et Leino, K. (2006). **Boogie : A modular reusable verifier for object-oriented programs**. Dans de Boer, F., Bonsangue, M., Graf, S., et de Roever, W.-P., éditeurs, *Formal Methods for Components and Objects*, volume 4111 de *Lecture Notes in Computer Science*, pages 364–387. Springer Berlin Heidelberg.

[Barrett et al., 2011] Barrett, C., Conway, C., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., et Tinelli, C. (2011). **CVC4**. Dans Gopalakrishnan, G., et Qadeer, S., éditeurs, *Computer Aided Verification*, volume 6806 de *Lecture Notes in Computer Science*, pages 171–177. Springer Berlin Heidelberg.

[Barrett et al., 2007] Barrett, C., et Tinelli, C. (2007). **CVC3**. Dans Damm, W., et Hermanns, H., éditeurs, *Computer Aided Verification*, volume 4590 de *Lecture Notes in Computer Science*, pages 298–302. Springer Berlin Heidelberg.

[Baudin et al., 2015] Baudin, P., Cuoq, P., Filliâtre, J. C., Marché, C., Monate, B., Moy, Y., et Prevosto, V. (2015). **ACSL : ANSI/ISO C Specification Language Version 1.9**. Published electronically at http://frama-c.com/acsl.html.

[Beckman et al., 2008] Beckman, N., Nori, A. V., Rajamani, S. K., et Simmons, R. J. (2008). **Proofs from tests**. Dans *ISSTA*, pages 3–14. ACM.

[Beyer et al., 2007] Beyer, D., Henzinger, T., Jhala, R., et Majumdar, R. (2007). **The software model checker blast**. *International Journal on Software Tools for Technology Transfer*, 9(5-6) :505–525.

[Botella et al., 2009] Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., et Williams, N. (2009). **Automating structural testing of c programs : Experience with pathcrawler**. Dans *Automation of Software Test, 2009. AST '09. ICSE Workshop on*, pages 70–78. IEEE.

[Boyapati et al., 2002] Boyapati, C., Khurshid, S., et Marinov, D. (2002). **Korat : Automated testing based on java predicates**. Dans *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA. ACM.

[Burghardt et al., 2015] Burghardt, J., et Gerlach, J. (2015). **ACSL by Example**. Published electronically at http://www.fokus.fraunhofer.de/download/acsl_by_example.

[Cadar et al., 2008a] Cadar, C., Dunbar, D., et Engler, D. R. (2008a). **KLEE : unassisted and automatic generation of high-coverage tests for complex systems programs**. Dans *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association.

[Cadar et al., 2008b] Cadar, C., Ganesh, V., Pawlowski, P. M., Dill, D. L., et Engler, D. R. (2008b). **Exe : Automatically generating inputs of death**. *ACM Trans. Inf. Syst. Secur.*, 12(2) :10 :1–10 :38.

[Canet et al., 2009] Canet, G., Cuoq, P., et Monate, B. (2009). **A value analysis for C programs**. Dans *Proceedings of the 2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, SCAM '09, pages 123–124, Washington, DC, USA. IEEE Computer Society.

[Carlier et al., 2012] Carlier, M., Dubois, C., et Gotlieb, A. (2012). **A certified constraint solver over finite domains**. Dans Giannakopoulou, D., et Méry, D., éditeurs, *FM 2012 : Formal Methods*, volume 7436 de *Lecture Notes in Computer Science*, pages 116–131. Springer Berlin Heidelberg.

[Castéran et al., 2004] Castéran, P., et Bertot, Y. (2004). **Interactive theorem proving and program development. Coq'Art : The Calculus of inductive constructions.** Texts in Theoretical Computer Science. Springer Berlin Heidelberg.

[Chaki et al., 2003] Chaki, S., Clarke, E., Groce, A., Jha, S., et Veith, H. (2003). **Modular verification of software components in C**. Dans *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 385–395, Washington, DC, USA. IEEE Computer Society.

[Chebaro et al., 2012a] Chebaro, O., Delahaye, M., et Kosmatov, N. (2012a). **Testing Inexecutable Conditions on Input Pointers in C Programs with SANTE**. Dans *ICSSEA 2012 - 24th International Conference on Software & Systems Engineering and their Applications*, pages 1–7, Paris, France. 7 pages - Session 11 : Testing - http ://icssea.enst.fr/icssea12/.

[Chebaro et al., 2012b] Chebaro, O., Kosmatov, N., Giorgetti, A., et Julliand, J. (2012b). **Program slicing enhances a verification technique combining static and dynamic analysis**. Dans *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1284–1291, New York, NY, USA. ACM.

[Chebaro, 2011] Chebaro, O. C. (2011). **Classification of errors threats by static analysis, program sclicing and structural testing of programs**. Theses, Université de Franche-Comté.

[Cheon et al., 2005] Cheon, Y., et Leavens, G. T. (2005). **A contextual interpretation of undefinedness for runtime assertion checking**. Dans *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, AADEBUG'05, pages 149–158, New York, NY, USA. ACM.

[Christakis et al., 2012] Christakis, M., Müller, P., et Wüstholz, V. (2012). **Collaborative verification and testing with explicit assumptions**. Dans Giannakopoulou, D., et Méry, D., éditeurs, *FM 2012 : Formal Methods*, volume 7436 de *Lecture Notes in Computer Science*, pages 132–146. Springer Berlin Heidelberg.

[Claessen et al., 2008] Claessen, K., et Svensson, H. (2008). **Finding counter examples in induction proofs**. Dans Beckert, B., et Hähnle, R., éditeurs, *Tests and Proofs*, volume 4966 de *Lecture Notes in Computer Science*, pages 48–65. Springer Berlin Heidelberg.

[Clarke et al., 2003] Clarke, E., Grumberg, O., Jha, S., Lu, Y., et Veith, H. (2003). **Counterexample-guided abstraction refinement for symbolic model checking**. *J. ACM*, 50 :752–794.

[Clarke et al., 1982] Clarke, E. M., et Emerson, E. A. (1982). **Design and synthesis of synchronization skeletons using branching-time temporal logic**. Dans *Logic of Programs, Workshop*, pages 52–71, London, UK. Springer.

[Clarke et al., 2009] Clarke, E. M., Emerson, E. A., et Sifakis, J. (2009). **Model checking : algorithmic verification and debugging**. *Commun. ACM*, 52(11) :74–84.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., et Sistla, A. P. (1986). **Automatic verification of finite-state concurrent systems using temporal logic specifications**. *ACM Trans. Program. Lang. Syst.*, 8(2) :244–263.

[Clarke, 1976] Clarke, L. A. (1976). **A system to generate test data and symbolically execute programs**. *IEEE Trans. Softw. Eng.*, 2 :215–222.

[Clarke et al., 2006] Clarke, L. A., et Rosenblum, D. S. (2006). **A historical perspective on runtime assertion checking in software development**. *SIGSOFT Softw. Eng. Notes*, 31(3) :25–37.

[Cok et al., 2005] Cok, D., et Kiniry, J. (2005). **Esc/java2 : Uniting esc/java and jml**. Dans Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., et Muntean, T., éditeurs, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 de *Lecture Notes in Computer Science*, pages 108–128. Springer Berlin Heidelberg.

[Comar et al., 2012] Comar, C., Kanig, J., et Moy, Y. (2012). **Integration von formaler verifikation und test**. Dans *Automotive - Safety & Security 2012, Sicherheit und Zuverlässigkeit für automobile Informationstechnik, 14.-15. November 2012, Karlsruhe, Proceedings*, LNI, pages 133–148. GI.

[Conchon, 2012] Conchon, S. (2012). **SMT Techniques and their Applications : from Alt-Ergo to Cubicle**. Thèse d'habilitation, Université Paris-Sud. In English, http://www.lri.fr/~conchon/publis/conchonHDR.pdf.

[Conchon et al., 2007] Conchon, S., Contejean, E., Kanig, J., et Lescuyer, S. (2007). **Lightweight integration of the Ergo theorem prover inside a proof assistant**. Dans *Proceedings of the second workshop on Automated formal methods*, AFM '07, pages 55–59, New York, NY, USA. ACM.

[Correnson, 2014] Correnson, L. (2014). **Qed. computing what remains to be proved**. Dans Badger, J., et Rozier, K., éditeurs, *NASA Formal Methods*, volume 8430 de *Lecture Notes in Computer Science*, pages 215–229. Springer International Publishing.

[Cousot et al., 1992] Cousot, P., et Cousot, R. (1992). **Abstract interpretation frameworks**. *Journal of Logic and Computation*, 2(4) :511–547.

[Cousot et al., 2007] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., et Rival, X. (2007). **Combination of abstractions in the ASTRÉE static analyzer**. Dans *Proceedings of the 11th Asian computing science conference on Advances in computer science : secure software and related issues*, ASIAN'06, pages 272–300, Berlin, Heidelberg. Springer.

[Csallner et al., 2004] Csallner, C., et Smaragdakis, Y. (2004). **JCrasher : an automatic robustness tester for Java**. *Software : Practice and Experience*, 34(11) :1025–1050.

[Csallner et al., 2005] Csallner, C., et Smaragdakis, Y. (2005). **Check 'n' Crash : combining static checking and testing**. Dans *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 422–431, New York, NY, USA. ACM.

[Csallner et al., 2008] Csallner, C., Smaragdakis, Y., et Xie, T. (2008). **DSD-Crasher : A hybrid analysis tool for bug finding**. *ACM Trans. Softw. Eng. Methodol.*, 17 :8 :1–8 :37.

[De Moura et al., 2008] De Moura, L., et Bjørner, N. (2008). **Z3 : an efficient SMT solver**. Dans *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer.

[Delahaye et al., 2013] Delahaye, M., Kosmatov, N., et Signoles, J. (2013). **Common specification language for static and dynamic analysis of c programs**. Dans *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1230–1235, New York, NY, USA. ACM.

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

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

[Dross et al., 2014] Dross, C., Efstathopoulos, P., Lesens, D., Mentré, D., et Moy, Y. (2014). **Rail, space, security : Three case studies for SPARK 2014**. Dans *ERTS*. Published electronically at http://www.spark-2014.org.

[Engel et al., 2007] Engel, C., et Hähnle, R. (2007). **Generating unit tests from formal proofs**. Dans Gurevich, Y., et Meyer, B., éditeurs, *Tests and Proofs*, volume 4454 de *Lecture Notes in Computer Science*, pages 169–188. Springer Berlin Heidelberg.

[Ernst, 2003] Ernst, M. D. (2003). **Static and dynamic analysis : Synergy and duality**. Dans *WODA 2003 : ICSE Workshop on Dynamic Analysis*, pages 24–27, Portland, OR. ACM.

[Ernst et al., 2001] Ernst, M. D., Cockrell, J., Griswold, W. G., et Notkin, D. (2001). **Dynamically discovering likely program invariants to support program evolution**. *IEEE Trans. Softw. Eng.*, 27 :99–123.

[Filliâtre et al., 2013] Filliâtre, J.-C., et Paskevich, A. (2013). **Why3 – where programs meet provers**. Dans Felleisen, M., et Gardner, P., éditeurs, *Programming Languages and Systems*, volume 7792 de *Lecture Notes in Computer Science*, pages 125–128. Springer Berlin Heidelberg.

[Flanagan et al., 2002] Flanagan, C., Leino, K. R. M., Lillibridge, M., Nelson, G., Saxe, J. B., et Stata, R. (2002). **Extended static checking for Java**. Dans *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245, New York, NY, USA. ACM.

[Floyd, 1963] Floyd, R. W. (1963). **Syntactic analysis and operator precedence**. *J. ACM*, 10 :316–333.

[Ge et al., 2011] Ge, X., Taneja, K., Xie, T., et Tillmann, N. (2011). **DyTa : dynamic symbolic execution guided with static verification results**. Dans *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 992–994, New York, NY, USA. ACM.

[Genestier et al., 2015] Genestier, R., Giorgetti, A., et Petiot, G. (2015). **Sequential generation of structured arrays and its deductive verification**. Dans Blanchette, J. C., et Kosmatov, N., éditeurs, *Tests and Proofs*, volume 9154 de *Lecture Notes in Computer Science*, pages 109–128. Springer International Publishing.

[Gladisch, 2009] Gladisch, C. (2009). **Could we have chosen a better loop invariant or method contract ?** Dans Dubois, C., éditeur, *Tests and Proofs*, volume 5668 de *Lecture Notes in Computer Science*, pages 74–89. Springer Berlin Heidelberg.

[Godefroid, 2007] Godefroid, P. (2007). **Compositional dynamic test generation**. *SIGPLAN Not.*, 42 :47–54.

[Godefroid et al., 2005] Godefroid, P., Klarlund, N., et Sen, K. (2005). **DART : directed automated random testing**. Dans *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 213–223, New York, NY, USA. ACM.

[Godefroid et al., 2008] Godefroid, P., Levin, M. Y., et Molnar, D. A. (2008). **Automated whitebox fuzz testing**. Dans *Proceedings of the Network and Distributed System Security Symposium, NDSS 2008, San Diego, California, USA, 10th February - 13th February 2008*. The Internet Society.

[Godefroid et al., 2010] Godefroid, P., Nori, A. V., Rajamani, S. K., et Tetali, S. D. (2010). **Compositional may-must program analysis : unleashing the power of alternation**. Dans *Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '10, pages 43–56, New York, NY, USA. ACM.

[Gordon et al., 1993] Gordon, M. J. C., et Melham, T. F., éditeurs (1993). **Introduction to HOL : a theorem proving environment for higher order logic**. Cambridge University Press, New York, NY, USA.

[Goubault et al., 2011] Goubault, E., et Putot, S. (2011). **Static analysis of finite precision computations**. Dans *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 232–247, Berlin, Heidelberg. Springer.

[Granlund et al., 2014] Granlund, T., et the GMP development team (2014). **GNU MP : The GNU Multiple Precision Arithmetic Library**, 6.0.0 édition. Published electronically at http://gmplib.org/.

[Gulavani et al., 2006] Gulavani, B. S., Henzinger, T. A., Kannan, Y., Nori, A. V., et Rajamani, S. K. (2006). **SYNERGY : a new algorithm for property checking**. Dans *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 117–127, New York, NY, USA. ACM.

[Gupta et al., 2009] Gupta, A., Majumdar, R., et Rybalchenko, A. (2009). **From tests to proofs**. Dans *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 262–276, Berlin, Heidelberg. Springer.

[Halbwachs et al., 1991] Halbwachs, N., Caspi, P., Raymond, P., et Pilaud, D. (1991). **The synchronous dataflow programming language Lustre**. *Proceedings of the IEEE*, 79(9) :1305–1320.

[Hatcliff et al., 2012] Hatcliff, J., Leavens, G. T., Leino, K. R. M., Müller, P., et Parkinson, M. (2012). **Behavioral interface specification languages**. *ACM Comput. Surv.*, 44(3) :16 :1–16 :58.

[Henzinger et al., 1994] Henzinger, T. A., Nicollin, X., Sifakis, J., et Yovine, S. (1994). **Symbolic model checking for real-time systems**. *Inf. Comput.*, 111(2) :193–244.

[Herms et al., 2012] Herms, P., Marché, C., et Monate, B. (2012). **A certified multi-prover verification condition generator**. Dans Joshi, R., Müller, P., et Podelski, A., éditeurs, *Verified Software : Theories, Tools, Experiments*, volume 7152 de *Lecture Notes in Computer Science*, pages 2–17. Springer Berlin Heidelberg.

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

[Hojjat et al., 2012] Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., et Rümmer, P. (2012). **Accelerating interpolants**. Dans Chakraborty, S., et Mukund, M., éditeurs, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 187–202. Springer Berlin Heidelberg.

[Holzmann, 1997] Holzmann, G. J. (1997). **The model checker SPIN**. *IEEE Trans. Softw. Eng.*, 23 :279–295.

[IEEE, 2011] IEEE (2011). **IEEE Guide–Adoption of the project management institute (PMI(r)) standard a guide to the project management body of knowledge (PMBOK(r) guide)**. Rapport technique.

[Jiang et al., 2012] Jiang, S., Li, W., Li, H., Zhang, Y., Zhang, H., et Liu, Y. (2012). **Fault localization for null pointer exception based on stack trace and program slicing**. Dans *Quality Software (QSIC), 2012 12th International Conference on*, pages 9 –12. IEEE.

[Kanig et al., 2014] Kanig, J., Chapman, R., Comar, C., Guitton, J., Moy, Y., et Rees, E. (2014). **Explicit assumptions - a prenup for marrying static and dynamic program verification**. Dans Seidl, M., et Tillmann, N., éditeurs, *Tests and Proofs*, volume 8570 de *Lecture Notes in Computer Science*, pages 142–157. Springer International Publishing.

[King, 1976] King, J. C. (1976). **Symbolic execution and program testing**. *Commun. ACM*, 19(7) :385–394.

[Kirchner et al., 2015] Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., et Yakobowski, B. (2015). **Frama-C : A software analysis perspective**. *Formal Aspects of Computing*, 27(3) :573–609.

[Kosmatov et al., 2013] Kosmatov, N., Petiot, G., et Signoles, J. (2013). **An optimized memory monitoring for runtime assertion checking of C programs**. Dans Legay, A., et Bensalem, S., éditeurs, *Runtime Verification*, volume 8174 de *Lecture Notes in Computer Science*, pages 167–182. Springer Berlin Heidelberg.

[Landi, 1992] Landi, W. (1992). **Undecidability of static analysis**. *ACM Lett. Program. Lang. Syst.*, 1 :323–337.

[Le Goues et al., 2011] Le Goues, C., Leino, K., et Moskal, M. (2011). **The boogie verification debugger (tool paper)**. Dans Barthe, G., Pardo, A., et Schneider, G., éditeurs, *Software Engineering and Formal Methods*, volume 7041 de *Lecture Notes in Computer Science*, pages 407–414. Springer Berlin Heidelberg.

[Leavens et al., 1999] Leavens, G. T., Baker, A. L., et Ruby, C. (1999). **Preliminary design of JML : A behavioral interface specification language for Java**. Rapport technique, Department of Computer Science, Iowa State University.

[Leavens et al., 2005] Leavens, G. T., Cheon, Y., Clifton, C., Ruby, C., et Cok, D. R. (2005). **How the design of JML accommodates both runtime assertion checking and formal verification**. *Science of Computer Programming*, 55(1–3) :185 – 208. Formal Methods for Components and Objects : Pragmatic aspects and applications.

[Leino et al., 2014] Leino, K. R. M., et Wüstholz, V. (2014). **The dafny integrated development environment**. Dans Dubois, C., Giannakopoulou, D., et Méry, D., éditeurs, Proceedings 1st Workshop on *Formal Integrated Development Environment,* Grenoble, France, April 6, 2014, volume 149 de *Electronic Proceedings in Theoretical Computer Science*, pages 3–15. Open Publishing Association.

[Logozzo, 2011] Logozzo, F. (2011). **Practical verification for the working programmer with CodeContracts and abstract interpretation**. Dans *Proceedings of the 12th international conference on Verification, model checking, and abstract interpretation*, VMCAI'11, pages 19–22, Berlin, Heidelberg. Springer.

[Marre et al., 2000] Marre, B., et Arnould, A. (2000). **Test sequences generation from LUSTRE descriptions : GATeL**. Dans *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 229–237, Washington, DC, USA. IEEE Computer Society.

[Meyer, 1988] Meyer, B. (1988). **Object-Oriented Software Construction**. Prentice-Hall.

[Müller et al., 2011] Müller, P., et Ruskiewicz, J. (2011). **Using debuggers to understand failed verification attempts**. Dans Butler, M., et Schulte, W., éditeurs, *FM 2011 : Formal Methods*, volume 6664 de *Lecture Notes in Computer Science*, pages 73–87. Springer Berlin Heidelberg.

[Necula et al., 2002] Necula, G. C., McPeak, S., Rahul, S. P., et Weimer, W. (2002). **CIL : Intermediate language and tools for analysis and transformation of C programs**. Dans *Proceedings of the 11th International Conference on Compiler Construction*, CC '02, pages 213–228, London, UK. Springer.

[Nethercote et al., 2007] Nethercote, N., et Seward, J. (2007). **Valgrind : a framework for heavyweight dynamic binary instrumentation**. Dans *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA. ACM.

[Nguyen et al., 2008] Nguyen, H., Kuncak, V., et Chin, W.-N. (2008). **Runtime checking for separation logic**. Dans Logozzo, F., Peled, D., et Zuck, L., éditeurs, *Verification, Model Checking, and Abstract Interpretation*, volume 4905 de *Lecture Notes in Computer Science*, pages 203–217. Springer Berlin Heidelberg.

[Nielson et al., 1999] Nielson, F., Nielson, H. R., et Hankin, C. (1999). **Principles of Program Analysis**. Springer New York, Inc., Secaucus, NJ, USA.

[Nori et al., 2009] Nori, A. V., Rajamani, S. K., Tetali, S., et Thakur, A. V. (2009). **The Yogi project : Software property checking via static analysis and testing**. Dans *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems : Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009,*, TACAS '09, pages 178–181, Berlin, Heidelberg. Springer.

[Pasareanu et al., 2007] Pasareanu, C. S., Pelánek, R., et Visser, W. (2007). **Predicate abstraction with under-approximation refinement**. *Logical Methods in Computer Science*, 3(1).

[Petiot et al., 2014a] Petiot, G., Botella, B., Julliand, J., Kosmatov, N., et Signoles, J. (2014a). **Instrumentation of annotated C programs for test generation**. Dans *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, pages 105–114. IEEE.

[Petiot et al., 2015] Petiot, G., Kosmatov, N., Botella, B., Giorgetti, A., et Julliand, J. (2015). **Your Proof Fails ? Testing Helps to Find the Reason**. Published electronically at http://arxiv.org/pdf/1508.01691.pdf.

[Petiot et al., 2014b] Petiot, G., Kosmatov, N., Giorgetti, A., et Julliand, J. (2014b). **How test generation helps software specification and deductive verification in Frama-C**. Dans Seidl, M., et Tillmann, N., éditeurs, *Tests and Proofs*, volume 8570 de *Lecture Notes in Computer Science*, pages 204–211. Springer International Publishing.

[Queille et al., 1982] Queille, J., et Sifakis, J. (1982). **Specification and verification of concurrent systems in CESAR**. Dans Dezani-Ciancaglini, M., et Montanari, U., éditeurs, *International Symposium on Programming*, volume 137 de *Lecture Notes in Computer Science*, pages 337–351. Springer Berlin Heidelberg.

[Romano et al., 2011] Romano, D., Di Penta, M., et Antoniol, G. (2011). **An approach for search based testing of null pointer exceptions**. Dans *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 160–169.

[Rosenblum, 1992] Rosenblum, D. S. (1992). **Towards a method of programming with assertions**. Dans *Proceedings of the 14th International Conference on Software Engineering*, ICSE '92, pages 92–104, New York, NY, USA. ACM.

[Schiller et al., 2012] Schiller, T. W., et Ernst, M. D. (2012). **Reducing the barriers to writing verified specifications**. Dans *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 95–112, New York, NY, USA. ACM.

[Schimpf et al., 2010] Schimpf, J., et Shen, K. (2010). **Eclipse - from LP to CLP**. *CoRR*, abs/1012.4240. Published electronically at http://arxiv.org/abs/1012.4240.

[Sen et al., 2006] Sen, K., et Agha, G. (2006). **CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools**. Dans *CAV*, pages 419–423. Springer.

[Signoles, 2015] Signoles, J. (2015). **E-ACSL : Executable ANSI/ISO C Specification Language**. Published electronically at http://frama-c.com/download/e-acsl/e-acsl.pdf.

[Signoles et al., 2015] Signoles, J., Correnson, L., Lemerre, M., et Prevosto, V. (2015). **Frama-C Developer Manual**. Published electronically at http://frama-c.com/download/frama-c-plugin-development-guide.pdf.

[Slabý et al., 2012] Slabý, J., Strejcek, J., et Trtík, M. (2012). **Checking properties described by state machines : On synergy of instrumentation, slicing, and symbolic**

execution. Dans Stoelinga, M., et Pinger, R., éditeurs, *Formal Methods for Industrial Critical Systems*, volume 7437 de *Lecture Notes in Computer Science*, pages 207–221. Springer Berlin Heidelberg.

[Sleator et al., 1985] Sleator, D. D., et Tarjan, R. E. (1985). **Self-adjusting binary search trees**. *J. ACM*, 32(3) :652–686.

[Szpankowski, 1990] Szpankowski, W. (1990). **Patricia tries again revisited**. *J. ACM*, 37(4) :691–711.

[Tillmann et al., 2008] Tillmann, N., et De Halleux, J. (2008). **Pex : white box test generation for .NET**. Dans *Proceedings of the 2nd international conference on Tests and proofs*, TAP'08, pages 134–153, Berlin, Heidelberg. Springer.

[Tollitte et al., 2012] Tollitte, P.-N., Delahaye, D., et Dubois, C. (2012). **Producing certified functional code from inductive specifications**. Dans Hawblitzel, C., et Miller, D., éditeurs, *Certified Programs and Proofs*, volume 7679 de *Lecture Notes in Computer Science*, pages 76–91. Springer Berlin Heidelberg.

[Tschannen et al., 2014] Tschannen, J., Furia, C., Nordio, M., et Meyer, B. (2014). **Program checking with less hassle**. Dans Cohen, E., et Rybalchenko, A., éditeurs, *Verified Software : Theories, Tools, Experiments*, volume 8164 de *Lecture Notes in Computer Science*, pages 149–169. Springer Berlin Heidelberg.

[Wenzel et al., 2008] Wenzel, M., Paulson, L. C., et Nipkow, T. (2008). **The isabelle framework**. Dans Mohamed, A., Munoz, et Tahar, éditeurs, *Theorem Proving in Higher Order Logics (TPHOLs 2008)*, volume 5170 de *LNCS*, pages 33–38. Springer.

[Whitner et al., 1989] Whitner, R. B., et Balci, O. (1989). **Guidelines for selecting and using simulation model verification techniques**. Dans *Proceedings of the 21st conference on Winter simulation*, WSC '89, pages 559–568, New York, NY, USA. ACM.

[Williams et al., 2004] Williams, N., Marre, B., et Mouy, P. (2004). **On-the-fly generation of k-path tests for C functions**. Dans *19th IEEE International Conference on Automated Software Engineering (ASE 2004), 20-25 September 2004, Linz, Austria*, pages 290–293.

[Zee et al., 2007] Zee, K., Kuncak, V., Taylor, M., et Rinard, M. (2007). **Runtime checking for program verification**. Dans Sokolsky, O., et Taşıran, S., éditeurs, *Runtime Verification*, volume 4839 de *Lecture Notes in Computer Science*, pages 202–213. Springer Berlin Heidelberg.

[Zitser et al., 2004] Zitser, M., Lippmann, R., et Leek, T. (2004). **Testing static analysis tools using exploitable buffer overflows from open source code**. *SIGSOFT Softw. Eng. Notes*, 29 :97–106.

TABLE DES FIGURES

Document réalisé avec LaTeX et :
le style LaTeX pour Thèse de Doctorat créé par S. Galland — http://www.multiagent.fr/ThesisStyle
la collection de paquets `tex-upmethodology` — http://www.arakhne.org/tex-upmethodology/

Résumé :

La vérification de logiciels repose le plus souvent sur une spécification formelle encodant les propriétés du programme à vérifier. La tâche de spécification et de vérification déductive des programmes est longue et difficile et nécessite une connaissance des outils de preuve de programmes. En effet, un échec de preuve de programme peut être dû à une non-conformité du code par rapport à sa spécification, à un contrat de boucle ou de fonction appelée trop faible pour prouver une autre propriété, ou à une incapacité du prouveur. Il est souvent difficile pour l'utilisateur de décider laquelle de ces trois raisons est la cause de l'échec de la preuve car cette information n'est pas (ou rarement) donnée par le prouveur et requiert donc une revue approfondie du code et de la spécification.
L'objectif de cette thèse est de fournir une méthode de diagnostic automatique des échecs de preuve afin d'améliorer le processus de spécification et de preuve des programmes C. Nous nous plaçons dans le cadre de la plate-forme d'analyse des programmes C FRAMA-C, qui fournit un langage de spécification unique ACSL, un greffon de vérification déductive WP et un générateur de tests structurels PATHCRAWLER. La méthode que nous proposons consiste à diagnostiquer les échecs de preuve en utilisant la génération de tests structurels sur une version instrumentée du programme d'origine.

Mots-clés : analyse statique, analyse dynamique, méthodes formelles, preuve, test, FRAMA-C

Abstract:

Software verification often relies on a formal specification encoding the program properties to check. Formally specifying and deductively verifying programs is difficult and time consuming and requires some knowledge about theorem provers. Indeed, a proof failure for a program can be due to a non-compliance between the code and its specification, a loop or callee contrat being insufficient to prove another property, or a prover incapacity. It is often difficult for the user to decide which one of these three reasons causes a given proof failure. Indeed, this feedback is not (or rarely) provided by the theorem prover thus requires a thorough review of the code and the specification.
This thesis develops a method to automatically diagnose proof failures and facilitate the specification and verification task. This work takes place within the analysis framework for C programs FRAMA-C, that provides the specification language ACSL, the deductive verification plugin WP, and the structural test generator PATHCRAWLER. The proposed method consists in diagnosing proof failures using structural test generation on an instrumented version of the program under verification.

Keywords: static analysis, dynamic analysis, formal methods, proof, testing, FRAMA-C

SPIM

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex
■ tél. +33 (0)3 81 66 66 02 ■ ed-spim@univ-fcomte.fr ■ www.ed-spim.univ-fcomte.fr

UFC
UNIVERSITÉ DE FRANCHE-COMTÉ