Une couverture Combinant Tests et Preuves pour la Vérification Formelle

Viet Hoang Le

ONERA
THE FRENCH AEROSPACE LAB

Université Fédérale
Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

Présentée et soutenue le *Date de défense (11/07/2019)* **par :**

VIET HOANG LE

Une couverture Combinant Tests et Preuves pour la Vérification Formelle

---

École doctorale :
MITT : Domaine STIC : Sureté de logiciel et calcul de haute performance
Unité de Recherche :
MOIS-DTIS
Directeur de Thèse :
Virginie WIELS
Encadrant :
Loïc CORRENSON et *Julien SIGNOLES*
Rapporteurs :
Catherine DUBOIS et *Ioannis PARISSIS*
Examinateurs :
Hélène WAESELYNCK et *Delphine LONGUET*

Remerciements

Je tiens à remercier, tout d'abord, Madame Virginie Wiels, Monsieur Julien Signoles et Monsieur Loïc Correnson, qui m'ont encadré tout au long de ma thèse, qui ont parfaitement répondu à mes questions scientifiques et qui m'ont bien conseillé pour m'aider à réussir cette thèse. Je les remercie aussi pour leur gentillesse et leur disponibilité pour me guider dans la bonne direction. Grâce à leurs conseils professionnels et leurs encouragements, j'ai pu éviter le stress et garder la force pour terminer ma thèse.

Ensuite, Je souhaiterais à remercier les membres de mon jury de thèse.

Madame Catherine Dubois et Monsieur Ioannis Parissis m'ont fait l'honneur d'être rapporteurs de ma thèse, ils ont pris le temps de lire et évaluer mon manuscrit, ainsi que de me donner leurs opinions pendant le jury de ma thèse. Leurs remarques m'ont permis de me perfectionner mes contributions. Je les remercie pour tout cela.

Je tiens à remercier madame Hélène Waeselynck et madame Delphine Longuet pour leur participation au jury. Elles m'ont apporté une bonne vision du positionnement de ma thèse dans les recherches académiques et dans la pratique industrielle.

Puis, je voudrais remercier séparément les membres de trois équipes du CEA et de l'ONERA.

Premièrement, je remercie l'équipe CEATech Toulouse pour leur accueil et leur aide. Pendant ma thèse, les membres de CEATech Toulouse m'ont fourni un bureau, tous les appareils nécessaires et aussi des bons conseils. Leur aide a été très importante pour ma thèse.

Deuxièmement, je remercie les membres de l'équipe DTIS de ONERA. Les discussions m'ont donné une vision sur l'état de l'art de l'aéronautique qui m'a permis de réaliser l'importance de ma thèse.

Enfin, je remercie les membres du laboratoire LSL de CEA Nano-Inov pour leur accueil chaque fois que je venais à Paris. De plus, avec l'aide de leur outil, la plateforme Frama-C, j'ai pu implémenter les théories de ma thèse dans la pratique.

Mes derniers remerciements vont à la direction de la région Occitanie qui a financé ma thèse.

Table des matières

Liste des tableaux

Introduction

Problématique

La vérification d'un logiciel est une étape de son processus de développement. Elle sert à garantir l'absence de comportements inattendus et indésirables durant l'exécution du logiciel. Elle est particulièrement importante quand le logiciel fait partie d'un système critique qui peut causer la perte grave de ressources ou de vies humaines. Les ingénieurs vérification assurent notamment deux objectifs [Bro+10] :

1. *toutes* les spécifications définies sont satisfaites par le logiciel ;
2. ce logiciel satisfait *seulement* ces spécifications.

Le premier objectif est obligatoire mais le deuxième est seulement exigé dans certain cas. En pratique, les exigences de haut niveau peuvent être exprimées par plusieurs exigences de bas niveau. En vérifiant et en validant ces dernières, nous pouvons garantir les premières. Dans cette thèse, nous nous intéressons uniquement aux exigences de bas niveau [Bra+18].

Pour assurer qu'un logiciel satisfait sa spécification, plusieurs méthodes de vérification peuvent être utilisées, notamment le test ou la preuve. La vérification par test est la technique la plus souvent utilisée. Elle consiste à exécuter le logiciel sur des données d'entrée et à évaluer chaque couple entrée-sortie produit par rapport aux spécifications du logiciel, en utilisant un oracle dédié [Sof08]. La vérification par preuve est une autre technique de vérification qui consiste à raisonner rigoureusement sur le programme et ses spécifications [Flo93] et qui garantit la satisfaction des spécifications pour toutes les exécutions possibles du programme. Les méthodes de preuve peuvent être classées en trois familles : vérification déductive [Hoa78 ; Fil11], vérification de modèle [Mer08] et interprétation abstraite [CC77].

Pour s'acquitter complètement des objectifs n^0 1 et n^0 2 définis au premier paragraphe, un processus de vérification peut utiliser des analyses de couverture. C'est notamment le cas dans le cadre de la norme DO-178C/ED-12C [RTC11a ; Bro+10 ; Moy+13] qui est utilisée pour la certification des logiciels avioniques et dans lequel un processus de vérification doit garantir deux types de couverture : la couverture fonctionnelle et la couverture structurelle. La couverture fonctionnelle cible l'objectif n^0 1 : elle assure que toutes les spécifications définies pour le logiciel sont vérifiées par une technique de vérification (test ou preuve) et sont satisfaites par le programme selon les résultats des vérifications. La couverture structurelle cible l'objectif n^0 2. Cette couverture assure que tous les comportements des exécutions du logiciel sont définis pour les spécifications existantes. Pour la vérification par test, nous attei-

gnons cet objectif en *choisissant* un critère de couverture [AO08] : par exemple la couverture des instructions, la couverture des branches, ou la couverture MC/DC. En vérification par preuve, ces critères n'ont pas de signification et la couverture structurelle est assurée par quatre objectifs alternatifs : validité des méthodes de preuve, complétude, flot de données et absence du code étranger [Bro+10 ; Moy+13]. Les critères de couverture pour la vérification par test et celle par preuve sont donc fondamentalement différents.

Par ailleurs, les logiciels critiques étant de plus en plus complexes, l'utilisation d'une seule technique de vérification peut se révéler insuffisante. Pour la vérification par test, les différents cas nécessaires pour tester un logiciel sont de plus en plus nombreux et difficiles à trouver. Pour la vérification par preuve, les tentatives de preuve peuvent donner des résultats non-définitifs, typiquement quand une spécification n'est pas prouvée mais qu'aucun contre-exemple n'est trouvé pour celle-ci. L'utilisation combinée du test et de la preuve est ainsi une approche intéressante pour améliorer l'efficacité de la vérification en palliant les inconvénients de chacune des techniques. Cependant, comme les notions de couverture structurelle existantes sont très différentes selon que le test ou la preuve est utilisé, il est impossible de mener une analyse de couverture structurelle lorsque ces deux techniques sont combinées. L'objectif de cette thèse est de proposer une nouvelle notion de couverture adaptée aussi bien au test qu'à la preuve et permettant ainsi de combiner efficacement ces deux techniques.

Témoin

Pour définir cette nouvelle notion de couverture, nous avons besoin d'introduire la notion de témoin, qui est notre première contribution. Cette notion est définie pour formaliser et identifier rapidement les résultats obtenus pendant une campagne de vérification. Un témoin regroupe les quatre informations suivantes : la spécification vérifiée, la portion de logiciel sollicitée, la technique employée et le résultat de la vérification.

Selon la méthode de vérification utilisée, le témoin est différent. Par exemple, un témoin de preuve signifie qu'une spécification est vérifiée pour toutes les exécutions du programme, tandis qu'un témoin de test démontre qu'un certain chemin d'exécution la satisfait. Nous avons donc différents types de témoins. Dans cette thèse, nous serons amenés à proposer une dizaine de témoins différents, chacun joue un rôle particulier pour évaluer le degré de couverture d'une campagne.

Labels et Mutants

Deux autres notions, déjà existantes dans la littérature et que nous expliquons ci-après, sont aussi indispensables à notre analyse de couverture : celles de label et de mutant. Notre deuxième contribution est de les combiner d'une manière originale.

La notion de label a été définie et développée par Bardin et al [BKC14 ; Bar+14 ; Bar+15].

Chaque label est une formule logique attachée à un point de programme. Cette formule permet de partitionner l'ensemble des exécutions possibles et, donc, de partitionner la structure du code.

À partir de cette notion, Bardin et al. définissent la couverture de labels utilisée par un test. La couverture de labels permet de formaliser le critère de couverture structurelle utilisé par une campagne de test :

- un critère de couverture structurelle peut être représenté par un ensemble de labels ;
- si chaque label d'un tel ensemble est activé par au moins un cas de test, alors le critère de couverture structurelle représenté par cet ensemble est satisfait.

Dans notre approche, nous utilisons les deux points ci-dessus pour définir notre notion de couverture commune au test et à la preuve. Dans notre vision, les critères de couverture structurelle sont en effet représentés par des ensembles de labels.

Pour établir une relation entre le résultat de vérification et la notion de label, nous nous reposons sur la notion de mutants [Bud+80 ; DLS78] et de couverture d'un mutant [AO08]. Le principe quand nous utilisons la vérification par test, le suivant : une modification est faite en un point du programme source pour créer une version incorrecte de ce programme, nommée « mutant ». Nous refaisons alors les vérifications de la campagne sur le mutant. Si la vérification échoue, alors le mutant est « tué », ce qui signifie que les cas de test choisis sont capable de trouver l'erreur et qu'une ou plusieurs spécifications qui sont vérifiées par ces cas de test sont associées à la portion mutée du code. La couverture du mutant dépend du nombre de mutants tués par rapports au nombre de mutants créés.

Une méthode similaire a déjà été utilisée pour la vérification par preuve [CKV03]. Dans [CKV03], Hana et al proposent à revérifier les spécifications sur un mutant en utilisant les méthodes de preuves comme la vérification par modèle. Cela signifie que la couverture d'un mutant nous sert de point commun pour relier preuve et test et, en utilisant cette notion de mutant, nous pouvons définir une nouvelle notion de couverture commune au test et à la preuve.

Les travaux antécédents utilisent séparément les deux notions ci-dessus de labels et de mutants, voir l'état de l'art après d'introduction pour plus de détails. Dans cette thèse, nous mentions comment on peut les combiner pour obtenir un nouveau critère de couverture. En effet, la notion de label nous permet de partitionner la structure du code selon un critère de couverture structurelle choisi (1), tandis que, en associant le label à un mutant, nous pouvons identifier la spécification qu'il impacte (2). En combinant (1) et (2), nous pouvons qualifier le lien entre chaque spécification et chaque label, ce qui nous permet de définir une nouvelle notion de couverture indépendante d'une quelconque technique de vérification.

Couverture label-mutant

La nouvelle notion de couverture est donc définie en utilisant notre notion de témoin et une combinaison des labels et des mutants. Elle est appelée « couverture label-mutant ». Cette couverture permet d'apprécier la qualité de la vérification en établissant des relations

entre les spécifications, représentant les exigences du code, et les labels, représentant la structure du code. Plus précisément, chaque label est renforcé par un mutant correspondant à la notion de combinaison label-mutant. Nous vérifions chaque spécification sur le code original et sur chacun des mutants. Ces vérifications produisent des témoins, et en les analysant, nous déduisons pour chaque mutant, les spécifications dont la vérification par le code original et la vérification par ce mutant sont différentes (le mutant est tué et donc ces spécifications sont associées à ce mutant). La couverture label-mutant est satisfaite si et seulement si chaque spécification est associée à au moins un label et que chaque label est associé à au moins une spécification.

La vérification du logiciel, selon la norme DO-178 [Bro+10 ; Moy+13] est terminée si, et seulement si, elle satisfait à la fois la couverture fonctionnelle et la couverture structurelle. La couverture fonctionnelle est satisfaite si et seulement si toutes les spécifications définies sont vérifiées. Cela correspond au fait que chaque spécification est associée à au moins un label. La couverture structurelle est satisfaite si et seulement si tous les labels sont couverts par la vérification, c'est-à-dire au fait que chaque label est associé à au moins une spécification dans notre notion de couverture label-mutant. Notre couverture peut donc représenter à la fois la couverture structurelle et la couverture fonctionnelle. Autrement dit, nous pouvons établir que la vérification du logiciel est terminée dès que la couverture label-mutant est satisfaite.

Associée à notre notion de couverture label-mutant, nous proposons aussi la définition d'une matrice de couverture : c'est une présentation des résultats de vérification qui donne une vision synthétique de la satisfaction totale ou partielle de la couverture label-mutant. En utilisant cette matrice, nous pouvons trouver rapidement les spécifications non-vérifiées et les labels non-couverts. Ces informations permettent à l'ingénieur de vérification de compléter efficacement sa campagne de vérification. Nous pouvons même en déduire une *méthode d'interprétation des résultats* et une *plateforme logicielle* pour conduire le processus de vérification.

Plateforme de vérification et méthode d'interprétation

La couverture label-mutant exige d'établir des relations entre chaque spécification et chaque label. Un label est renforcé par un mutant. La relation entre une spécification et un label est établie dès que la vérification de la spécification peut tuer le mutant associé au label. La vérification peut être réalisée soit par test, soit par preuve, soit par une combinaison des deux. L'effort de vérification dépend du choix de la méthode de vérification utilisée. Si nous choisissons la bonne méthode, l'effort de vérification est réduit. Notre dernière contribution est une méthode d'interprétation qui cherche à *optimiser* le processus de vérification et une plateforme qui réalise *automatiquement* les vérifications.

La plateforme et la méthode d'interprétation forme un *cycle* entre les éléments fournis par l'ingénieur vérification et la matrice de couverture. D'abord, l'ingénieur définit le code, ses spécifications, le critère de couverture et les méthodes de vérification. Ensuite, la plateforme utilise ces éléments et construit automatiquement la matrice de couverture. Enfin, l'ingénieur

analyse la matrice de couverture en appliquant la méthode d'interprétation et conclut : soit le processus de vérification est terminé soit il définit des éléments additionnels de vérification pour cibler en priorité l'établissement des relations manquantes entre spécifications et labels. Puis, le cycle recommence jusqu'à la couverture complète. Nous avons développé un prototype de cette plateforme et l'avons utilisé pour expérimenter ce processus de vérification.

Nous présentons les différentes contributions que nous avons évoquées en quatre chapitres distincts. Le premier chapitre présente les notions nécessaires pour la présentation de notre notion de couverture. Le deuxième chapitre introduit la couverture label-mutant, c'est-à-dire notre nouvelle notion couverture. Le troisième chapitre présente la plateforme, son implémentation au sein de Frama-C [Kir+15] et une méthode d'interprétation pour la matrice de couverture. Le dernier chapitre détaille des problèmes encore ouverts et propose des extensions à nos travaux permettant de les résoudre.

État de l'art

Dans ce chapitre, nous allons présenter un état de l'art de notre domaine de recherche. Les travaux existants sont classés selon quatre aspects : label, mutant, combinaison de test et preuve et critères de couverture. Chaque aspect est présenté dans une section dédiée.

Labels

La notion de label, introduite et développée par Bardin et al. [BKC14 ; Bar+14], a été initialement utilisée pour représenter des critères de couverture structurelle de test. Le taux de couverture dépend du nombre de labels par lesquels est passé au moins un cas de test par rapport au nombre total de labels. Deux travaux liés aux labels sont particulièrement intéressants : la détection de chemins inatteignables et la notion d'hyperlabel.

Détection de chemins inatteignables D'après [Bar+15], les labels peuvent être utilisés pour détecter les chemins d'exécution inatteignables grâce à une analyse statique (preuve). Plus précisément, une fois générés, les labels correspondant au critère de couverture structurelle choisi, des techniques fondées sur l'interprétation abstraite et la preuve de programme permettent de trouver les labels par lesquels aucune exécution de la fonction ne passe. Comme ces labels déterminent des chemins d'exécution du code, ces chemins sont les chemins inatteignables. Cette technique de détection est utile parce qu'elle nous aide à trouver automatiquement les chemins et les instructions inatteignables du code.

Hyperlabel La notion d'hyperlabel [Mar+17a ; Mar+17b] est une extension de la notion de label en y ajoutant des relations inter-labels. Par exemple, il est possible d'ajouter une condition entre deux labels différents l_1 et l_2, comme « une exécution passant par l_1 doit aussi passer par l_2 », ou bien entre différentes exécutions. Avec la notion d'hyperlabel, nous pouvons représenter la plupart des critères de couverture structurelle existants [Mar+17a]. Nos travaux actuels ne considèrent pas encore les hyperlabels car les labels suffissent à représenter les critères de couverture structurelle utilisés à notre connaissance : couverture des instructions, couverture des branches et couverture MC/DC simplifiée [Hay+01]. Cette notion d'hyperlabel peut néanmoins a priori être introduite dans nos travaux sans problème particulier.

Mutants

Un mutant [DLS78 ; Bud+80] d'un programme est une version modifiée de ce programme. Il vise à quantifier la vérification. Par exemple, une vérification par test est un ensemble de cas de test. Un mutant est tué si et seulement si au moins un cas de test donne un résultat différent pour le mutant. La vérification par test peut ainsi être quantifiée en calculant le nombre de mutants tués par rapport au nombre total de mutants. Plusieurs travaux autour des mutants sont intéressants dans notre contexte. Nous les présentons ci-après

Opérateurs de mutation Plusieurs opérateurs de mutation différents [Agr+99 ; OVP96 ; DLS78] visant à muter le code du programme peuvent être définies. Ces opérateurs dépendent aussi du langage de programmation considéré. Par exemple, nous avons des opérateurs de mutation pour le langage C [Agr+99] et des opérateurs pour le langage Java [MOK06]. Le langage C possède des instructions « goto » que Java ne contient pas. Par conséquent, les opérateurs de mutation de « goto » n'existent que pour le langage C mais pas en Java.

À côté des opérateurs de mutation du code, il existe aussi des opérateurs de mutation de spécification [BG85 ; MB11 ; BOY00]. Dans nos travaux, et contrairement aux travaux existants, nous utilisons conjointement la mutation de code et la mutation de spécification. En utilisant ces deux notions de mutation, nous parvenons à obtenir plus d'informations que les travaux existants sur la couverture de la vérification.

Mutants équivalents et duplication de mutant Le problème des mutants équivalents [Pap+15] survient lorsque le mutant créé ressemble trop au code original tandis que celui de la duplication [Pap+15] se produit quand plusieurs mutants générés sont trop semblables. Ces deux problèmes interviennent fréquemment lorsque des outils automatiques de génération de mutants sont utilisés. Plusieurs travaux [Dev+17 ; Car+18 ; Pap+15] cherchent à résoudre ces problèmes. Même si notre outil n'est pas encore capable de générer automatiquement des mutants, il est probable que ces problèmes apparaîtront lorsque ce sera le cas. Dès lors, ces travaux nous seront utiles pour les résoudre.

Choix de l'opérateur de mutation Comme présenté ci-dessus, nous pouvons utiliser plusieurs opérateurs de mutation. Leur utilisation génère souvent trop de mutants et augmente ainsi le coût de vérification. Une façon de limiter ce problème est de choisir judicieusement les opérateurs de mutation nécessaires [MB99 ; Off+96 ; ORZ93]. Dans ces travaux, les opérateurs de mutation sont prédéfinis et sont appliqués l'un après l'autre au logiciel. Dans notre thèse, c'est un peu différent. Nous définissons au premier lieu les labels. En second lieu, selon le label et l'instruction choisis, nous sélectionnons l'opérateur de mutation nécessaire. Donc, nous n'avons besoin ici que d'un unique opérateur de mutation qui soit efficace, qui crée rarement ou ne crée jamais des mutants équivalents. Dans un tel cas, Delamaro et al. [Del+14] proposent d'utiliser la mutation qui supprime des instructions parce qu'il est très fort et efficace. Par contre, ils considèrent aussi d'autres opérateurs, également efficaces, comme

l'opérateur de modification de constantes. Par conséquent, dans notre travail, même si nous proposons seulement opérateurs de mutation différents, nous pouvons l'étendre naturellement à l'utilisation d'autres opérateurs.

Témoins

Nous ne sommes pas les premiers à introduire une notion de témoin. Beyer et al. [Bey+15 ; Bey+16 ; BD16] ont précédemment défini une telle notion pour décrire les résultats d'une vérification des spécifications d'un programme, avec des propriétés similaires.

Néanmoins, tandis que nos témoins considèrent la différence entre le test et la preuve, les témoins de Beyer et al. sont fondés uniquement sur la satisfaction de la spécification. Plus précisément, d'après leurs travaux, si l'outil de vérification montre la satisfaction de la spécification par le programme, alors cet outil produit un témoin de correction et, au contraire, s'il trouve un contre-exemple, alors nous avons un témoin de violation. Notre notion de témoin est plus riche car elle prend en compte deux familles des techniques de vérification, test et preuve, ainsi que les notions de mutant et de label. Nous exploitons les résultats possibles de chaque famille et, pour chaque possibilité, nous définissons un témoin particulier, ce qui engendre plus de témoins que Beyer et al.

Une autre différence entre ces deux notions de témoins est liée à leur incertitude. Les travaux de Beyer et al. supposent en effet que les résultats des outils de vérification sont incertains, ce qui est pris en compte par les témoins qui doivent être validés par un autre outil de vérification. En revanche, nous considérons que les résultats des outils de vérification sont fiables. Nous n'avons donc pas besoin de valider les témoins.

La dernière différence est liée à la représentation des témoins. Nos témoins sont représentés par une formule logique, tandis que ceux de Beyer et al. sont introduit par un automate. Cette représentation permet à Beyer et al. de valider un témoin en suivant son chemin dans l'automate.

De plus, les travaux de Beyer et al. nous suggèrent une optimisation possible dans la conduite d'une campagne de vérification. En effet, quand nous faisons un nouveau cycle dans le processus, nous pouvons réutiliser les témoins obtenus au cycle précédant. Nous y reviendrons dans le chapitre 4.

Combinaison test et preuve

Nous présentons ici quelques méthodes connues permettant de combiner test et preuve. D'après Bishop et al. [BBC13], il est possible de classer les différents types de combinaison entre tests et preuves en quatre niveaux :

1. (Séparément) : le test et la preuve sont appliqués séparément pour vérifier des parties

différentes du système ;

2. (Assistance) : la preuve soutient le test et vice versa (par exemple, la preuve aide la génération automatique de cas de test) ;
3. (Amitié) : la preuve contribue à la fois à générer automatiquement des cas de test et à analyser ses résultats ;
4. (Unification :) test et preuve sont complètement unifiés.

Dans cette section, nous présentons des méthodes de combinaison existantes en suivant cette classification.

JNuke Initialement, JNuke [AB05] est un outil qui propose un environnement pour tester un programme de Java. Pour réduire la vérification, la preuve (interprétation abstraite) est implémentée dans cet outil. Dans ce cas, JNuke choisit l'interprétation abstraite parce qu'elle est proche du test réalisé par JNuke. Néanmoins, le programme demeure à tester et à prouver indépendamment. Autrement dit, à partir du code source, l'outil produit un environnement pour tester le programme et un autre pour prouver le programme. Puis, les résultats issus des deux vérifications sont analysés. En raison de cette indépendance, la méthode de cet outil est classée au niveau « Séparément ».

Flinder-SCA Flinder-SCA [Kis+15] est un outil fondée sur la combinaison de test et preuve. Son implémentation repose sur deux plateformes distinctes : Frama-C [Kir+15] et Flinder [Lab]. Cet outil sert à détecter des vulnérabilités du logiciel, tout particulièrement des logiciels communiquant des informations (par exemple, un serveur d'internet). Il procède en trois étapes :

1. utiliser les méthodes de preuves (de la famille de l'interprétation abstraite) pour trouver des possibilités d'erreur ;
2. simplifier le code en respectant l'ensemble des erreurs possibles trouvées dans la première étape ;
3. tester pour confirmer ou informer les erreurs potentielles.

Plus précisément, cet outil commence d'abord par utiliser le greffon EVA [Büh+] de Frama-C. Ce greffon informe des erreurs possibles à l'exécution du code. Il est également possible d'utiliser une analyse de teinte [Den76] pour trouver les instructions du code qui peuvent propager des informations sensibles vers des objets qui le sont moins. Ensuite, le code est simplifié en préservant l'ensemble des erreurs potentielles. Cette étape permet de minimiser l'effort de test de l'étape suivante. Enfin, le code est testé en utilisant du fuzzing [Mic07], est implémenté dans Flinder, pour confirmer les erreurs potentielles trouvées lors de la première étape.

Dans cet outil, le test est utilisé pour confirmer le résultat de la preuve. Par conséquent, nous le classons dans la catégorie « Assistance ». Avec cet outil, la faille Heartbleed [Kis+15] est trouvée plus facilement que l'utilisation seule de test ou preuve. Cela montre que l'utilisation des méthodes combinées est plus efficace qu'utiliser le test ou la preuve seul.

Sante Sante (Static ANalysis and TEsting) [Che+11 ; Che+12 ; CDK12] est un autre greffon de Frama-C. Tout comme Flinder-SCA, cet outil introduit une vérification en trois étapes :

1. utiliser une méthode de preuve pour trouver les possibilités d'erreur ;
2. appliquer la technique du « slicing » pour simplifier le code ;
3. tester pour valider ces possibilités.

Ces étapes sont réalisées par des greffons implémentés dans Frama-C.

Pourtant, Sante et Flinder-SCA visent différents objectifs : la sûreté pour Sante et la sécurité pour Flinder-SCA. Les méthodes utilisées dans chaque étape sont donc différentes. Dans la première étape, les deux outils commencent par utiliser le greffon EVA de Frama-C. Mais, parce que Flinder-SCA vise plutôt la sécurité de l'information, Flinder-SCA permet aussi d'effectuer une analyse de teinte pour trouver les instructions concernées par des erreurs potentielles. Dans la troisième étape, tandis que Flinder-SCA utilise le fuzzing pour confirmer les erreurs, Sante réalise un test normal dont les cas passent par tous les chemins d'exécutions possibles du code avec l'aide de PathCrawler [WMM04 ; Wil+05], un générateur de cas de test. Donc, même si leurs procédures sont identiques, Sante et Flinder-SCA les réalisent différemment. Sante est aussi classé dans la catégorie « Assistance », comme Flinder-SCA.

staRVOOrS staRVOOrS [Chi+15 ; Ahr+17] (Static and Runtime Verification of Object-Oriented Software) est un outil de vérification pour Java. Il combine la vérification déductive de l'outil KeY [Ahr+16] avec la vérification à l'exécution de l'outil Larva [CAS09]. staRVOOrS repose sur un enchaînement de quatre étapes consécutive :

1. vérification déductive ;
2. raffinement de la spécification ;
3. traduction et instrumentation ;
4. génération de moniteur.

À la première étape, les spécifications du logiciel, écrites dans le langage de spécification ppDate [Chi+15], sont prouvées en utilisant KeY. Les preuves sont soit complètes soit partielles. La deuxième étape sert à raffiner les preuves partielles. Dans la troisième étape, chaque spécification est remplacée par une spécification équivalente au format DATE qui peut être utilisée par Larva. La quatrième étape consiste à utiliser Larva pour trouver les erreurs et les traces d'exécution qui mènent à ces erreurs. Nous classons ce cas au niveau « Assistance » parce que les tests sont construits à partir des preuves partielles.

Méthode combinée pour hypothèses implicites Les outils de vérification actuels, spécifiquement les outils de preuve, garantissent les spécifications du programme en s'appuyant sur des hypothèses implicites potentiellement incorrectes. Christakis et al [CMW12 ; CMW16] proposent de résoudre ce problème en combinant exécution symbolique (une méthode de preuve) et test. Plus précisément, dans leurs travaux, le code est vérifié par des outils de preuve qui exigent d'expliciter les hypothèses implicites sous forme de lemmes. Ces lemmes doivent également être vérifiés. Lorsque ce n'est pas le cas, les spécifications dont les preuves

reposent sur leur validité sont testées. Pour cela, une méthode appelée exécution symbolique dynamique est appliquée pour générer des cas de test permettant aussi de valider les hypothèses implicites. Cette méthode est aussi une méthode de la catégorie « Assistance ».

Combiner preuve et test de conformité Constant et al [Con+07] proposent une autre combinaison de test et preuve pour la vérification en boîte noire. Cette situation introduit trois niveaux d'abstractions : les exigences de haut niveau, une représentation par automate des opérations et l'implémentation. La preuve est utilisée pour vérifier chaque exigences avec l'automate défini, tandis que le test de conformité, une méthode de test de boîte noire, vérifie l'implémentation. Les cas de test dans ce cas sont générés à partir de l'automate. Nous considérons que le test dans ce cas vérifie la conformité de l'implémentation avec le modèle formel (l'automate validé par preuve). Ce cas est donc classé dans la catégorie « Assistance ».

Combinaison d'analyse dans SPARK Comar et al [CKM12] présentent également trois méthodes combinées possibles. La première consiste à diviser le code source en sous-programmes et à utiliser une méthode de vérification appropriée pour chacun d'eux. La deuxième méthode consiste à utiliser une méthode de vérification différente pour chaque spécification d'un logiciel. La troisième cherche à remplacer certains tests par des preuves. Donc, nous classons la première et deuxième suggestion dans la catégorie « Séparément » et la troisième dans la catégorie « Assistance ».

Combinaison unifiée du test et de la preuve Bishop et al [BBC13] introduisent un cas d'étude dans lequel le test et la preuve sont entièrement combinés. La technique présentée dans ce cas d'étude fonctionne uniquement pour les fonctions monotones. Elle consiste d'abord à prouver la monotonicité de l'implémentation de la fonction. Ensuite, à partir des spécifications formelles, elle définit des cas de test. Enfin, les résultats des cas de test sont combinés à la monotonicité de l'implémentation pour prouver que les spécifications vérifiées sont satisfaites par n'importe quelle valeur d'entrée. Cette combinaison est classée dans la catégorie « Unification ».

Autres notions de couverture

Par ailleurs, il existe une autre notion de couverture que la notre pour combiner test et preuve, appelée couverture de l'espace d'état [Bec+18]. C'est une couverture des données. Elles sert à trouver les données couvertes par les preuves partielles et compléter en testant les données non-couvertes. Donc, il n'existe pas de structure dans la notion de cette couverture. Néanmoins, ça peut être une extension de l'utilisation des labels. Nous y reviendrons dans le chapitre 4.

If I had an hour to solve a problem, I'd spend 55 minutes thinking about the problem and 5 minutes thinking about solutions.

Albert Einstein

1

Campagnes de vérification

Avant de résoudre un problème, nous devons le comprendre. Dans ce premier chapitre, nous allons donc présenter le problème étudié dans ma thèse.

Comme nous l'avons expliqué dans l'introduction, la vérification est soumise à deux analyses de couverture : fonctionnelle et structurelle. Les critères de couverture structurelle sont très différents selon que nous utilisons le test ou la preuve. Cela empêche une vérification combinée qui utiliserait à la fois une méthode de test et une méthode de preuve. Un critère de couverture partagé pour le test et la preuve permettrait cette vérification combinée. La définition de ce critère est notre objectif principal.

La section 1 introduit un exemple qui nous aide à éclaircir certaines notions. La section 2 présente le contexte informel d'une campagne de vérification, une notion que nous introduisons pour organiser le processus de vérification. La section 3 présente les notions de code et de spécification. Ces définitions sont utilisées pour représenter la couverture d'une vérification. La section 4 définit formellement les notions de test et de preuve, conjointement à une présentation détaillée du calcul de couverture. La section 5 introduit enfin une notion au cœur de notre solution : la mutation.

1.1 Exemple

Avant de présenter les notions formelles et informelles dont nous avons besoin, nous introduisons d'abord un exemple simple de fonction C spécifiée avec le langage ACSL [Bau+a] (cf. Figure 1.3) qui permettra d'illustrer ces notions.

```
/*@
 requires v  <  u;

 behavior S0:
 assumes a  <  v;
 ensures \result  ≡  v;

 behavior S1:
 assumes v  ≤  a  ≤  u;
 ensures \result  ≡  a;

 behavior S2:
 assumes a  >  u;
 ensures \result  ≡  u;
*/
```

FIGURE 1.1 – Les spécifications

```
int range(int a, int u, int v)
{
 int res = a;
 if(a  <  v){ res = v;}
 if(a  >  u){ res = u;}
 return res;
}
```

FIGURE 1.2 – Le code

FIGURE 1.3 – Un code avec spécifications

Cette fonction s'appelle `range`. Elle a trois variables en entrée : `a`, `u` et `v`. Les variables `u` et `v` sont respectivement la borne supérieure et la borne inférieure avec la contrainte : $\mathtt{v} < \mathtt{u}$. La variable `a` est l'entrée comparative permettant de déterminer la valeur de sortie.

Plus précisément, cette fonction a trois résultats possibles exprimés par les comportements (ACSL behavior) S0, S1 et S2 :

1. soit la valeur de la variable `a` est inférieure à la borne inférieure `v` : la fonction retourne alors la valeur de `v` (comportement S0) ;

2. soit la valeur de la variable `a` est supérieure à la borne inférieure `v` mais est inférieure à la borne supérieure `u` : la fonction retourne alors la valeur de `a` (comportement S1) ;

3. soit la valeur de la variable `a` est supérieure à la borne supérieure `u` : la fonction retourne alors la valeur de `u` (comportement S2).

Cette fonction C correspond à la fonction mathématique suivante :

$$(\mathtt{a},\mathtt{v},\mathtt{u}) \mapsto \begin{cases} \mathtt{v} & \text{si } \mathtt{v} \geq \mathtt{a} \\ \mathtt{a} & \text{si } \mathtt{v} \leq \mathtt{a} \leq \mathtt{u} \\ \mathtt{u} & \text{si } \mathtt{a} \geq \mathtt{u} \end{cases}$$

1.2 Contexte

Cette section présente informellement les notions qui nous aident à définir une campagne de vérification, une notion que nous définissons pour organiser le processus de vérification. Nous résumons aussi l'état de l'art de la pratique de la vérification et le problème qui mène au sujet de ma thèse.

Le modèle du cycle en V est un des modèles conceptuels très utilisé pour le développement logiciel. Le processus de « Vérification et Validation » (« V&V ») se trouve dans la partie droite de ce modèle pour garantir l'exactitude du logiciel avant de l'utiliser. Ce processus est effectué par l'ingénieur de validation.

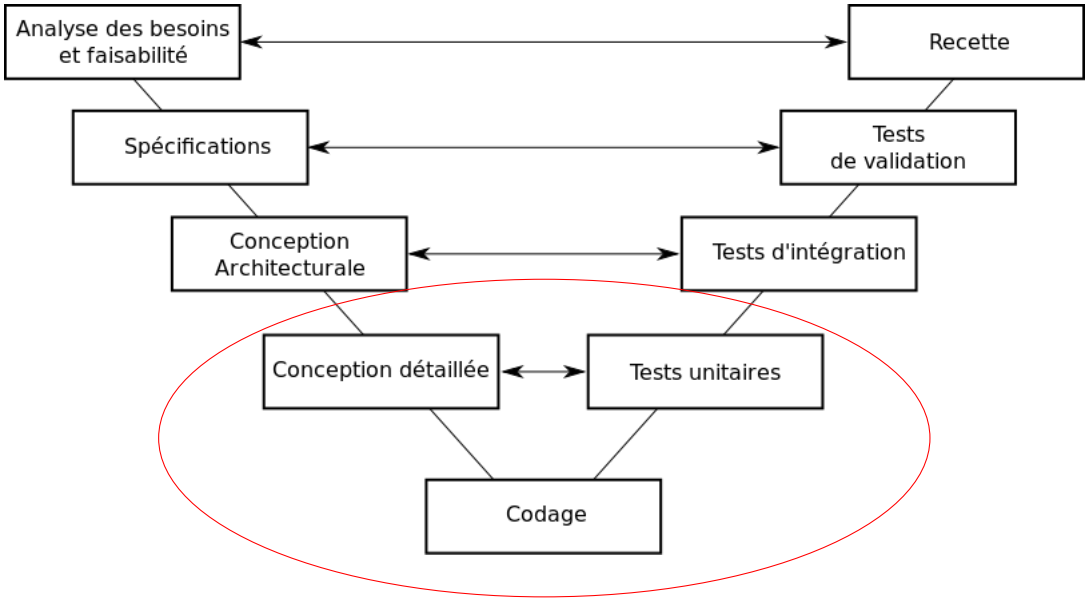


FIGURE 1.4 – Cycle en V [1]

La figure 1.4 présente les étapes d'un cycle en V. Le processus de « V&V » se trouve à partir de l'étape « Tests unitaires » jusqu'à l'étape « Tests de validation » dans cette figure.

Le code est le produit d'un projet de développement logiciel. Il existe plusieurs niveaux de code : soit une seule fonction, soit un groupe de fonctions, soit une classe, soit un système complexe regroupant de nombreuses fonctions et classes. Le code que nous considérons dans cette thèse est une fonction. Un tel exemple de code a été introduit dans la section 1.1.

Une fonction est définie par une séquence d'instructions. Chaque instruction est une unité syntaxique qui effectue une action donnée. Ainsi, l'exemple de la section 1.1 contient deux types d'instruction. L'une permet de modifier la mémoire (`res = a;`) tandis que l'autre est une conditionnelle qui décide de la prochaine action à exécuter via l'évaluation d'une formule booléenne (`if (a < v)`).

1. `https://fr.wikipedia.org/wiki/Cycle_en_V`

Une spécification est la description formalisée des propriétés attendues du logiciel. Il existe plusieurs niveaux de spécification comme la spécification d'architecture ou la spécification fonctionnelle. Dans la figure 1.4, les spécifications sont déterminées à partir de l'étape « Spécifications » jusqu'à l'étape « Conception détaillée ». Chaque étape détermine un type de spécification différent. Les spécifications considérées dans ma thèse sont des spécifications fonctionnelles. Elles sont produites pendant l'étape de « Conception détaillée ».

Dans ma thèse, nous nous concentrons sur les vérifications unitaires, au niveau de la fonction. Notre travail se limite donc aux trois étapes entourées par le cercle de la figure 1.4.

1.2.1 Vérification et couverture par tests

Le test repose sur les notions de cas de test [Sof08], d'oracles [PP98 ; Wey80] et de verdict. Pour organiser le processus de vérification par test, nous définissons la notion d'activité de test. Chaque activité de test est composée de trois étapes :

1. définition des cas de test,
2. évaluation des cas de test en utilisant l'oracle,
3. émission d'un verdict.

La première étape consiste à définir les cas de test. Un cas de test est l'observation d'une exécution du code. Il est défini par une paire entrée/sortie de cette exécution. Nous pouvons écrire manuellement les cas de test ou utiliser un outil qui les crée automatiquement.

La deuxième étape exige un oracle pour évaluer chaque cas de test. Il prend en compte le couple entrée-sortie de chaque cas de test et confronte ce couple aux spécifications. Pour chaque spécification, l'oracle détermine si le couple entrée-sortie du cas de test satisfait ou ne satisfait pas cette spécification. L'oracle est soit l'observation de l'ingénieur, soit un outil qui évalue automatiquement les cas de test.

Dans la troisième étape, nous collectons et analysons les résultats de la deuxième étape pour définir si le logiciel contient une erreur ou si tous les objectifs sont satisfaits. Les objectifs dont nous parlons ici sont la couverture fonctionnelle et la couverture structurelle, présentés précédemment. À partir du verdict de test, nous pouvons mesurer le degré de couverture et donc, décider si le processus de vérification est complet ou non.

Pour tester la fonction `range` de la figure 1.3, nous définissons plusieurs fonctions `main` (cf figure 1.5) dans lesquelles `range` est exécutée avec des combinaisons d'entrées différentes. Chaque exécution de la fonction `range` est l'exécution d'un cas de test de cette fonction. Les résultats des cas de test envisagés par l'oracle sont déterminés à partir des trois spécifications de `range`. Dans cet exemple, toutes les spécifications sont satisfaites par le couple entrée-sortie de leur cas de test.

Nous pouvons réaliser plusieurs activités de test (chaque activité de test représente un ensemble de cas de test différent). En pratique, nous commençons par des activités dont les cas de test sont créés automatiquement par des outils (par exemple, le greffon PathCraw-

```
// trois cas de tests pour la fonction "range(a,u,v)"

int main(){
 range (6,15,10);
 // range(6,15,10) retourne l'entier 10
 // la spécification S0 est satisfaite
}

int main(){
 range (8,10,3);
 // range(8,10,3) retourne l'entier 8
 // la spécification S1 est satisfaite
}

int main(){
 range (3,2,1);
 // range(3,2,1) retourne l'entier 2
 // la spécification S2 est satisfaite
}
```

FIGURE 1.5 – Test de la fonction `range`

ler [WMM04] de Frama-C [Kir+15]). Si ces activités ne sont pas suffisantes, nous continuons avec des activités dont les cas de test sont construits manuellement.

La décision d'arrêter ou de continuer la vérification repose sur la notion de couverture. Selon la norme avionique DO-178C [Bro+10 ; Moy+13], le test doit garantir deux types de couverture : la couverture fonctionnelle et la couverture structurelle.

La couverture fonctionnelle a pour objectif d'assurer que des vérifications ont bien été menées pour chaque spécification. Elle est satisfaite quand toutes les spécifications définies ont été testées.

La couverture structurelle a pour objectif d'assurer que toute la structure du code a été exercée lors des vérifications. Cette notion de couverture est liée à l'exécution du code. Le calcul de cette couverture est soit manuel en traçant chaque exécution du code dans le test, soit automatique en utilisant un outil [SN14] comme le greffon LTest [Bar+14] de Frama-C. Il existe plusieurs critères de couverture structurelle [AO08 ; RTC11a] comme la couverture des instructions, la couverture des branches et la couverture MC/DC.

La couverture des instructions consiste à exercer toutes les instructions du code. La couverture des branches consiste à exercer toutes les branches (test de toutes les valeurs des décisions). La couverture MC/DC est une extension de la couverture des branches. Une expression booléenne appelée condition ne contient aucun connecteur logique. Une décision est une expression construite avec une ou plusieurs conditions liées par des connecteurs logiques. La couverture MC/DC est fondée sur les trois règles suivantes :

- toutes les évaluations de chaque décision doivent être vérifiées ;
- toutes les évaluations de chaque condition doivent être vérifiées ;
- la vérification démontre que le changement d'évaluation de chaque condition d'une décision modifie indépendamment l'évaluation de cette décision.

Nous pouvons arrêter le processus de vérification lorsque les couvertures fonctionnelles et

structurelles sont satisfaites. Ces deux notions de couvertures sont bien définies pour le test. Considérons le test de la fonction `range` ci-dessus. Il garantit la couverture fonctionnelle et la couverture des instructions, un critère de couverture structurelle. Donc, nous pouvons arrêter le processus de vérification.

1.2.2 Vérification et couverture par preuve

Dans notre travail, nous ne considérons que des méthodes de preuve, telles que celles dénotés par Floyd [Flo93] qui vérifient des propriétés sur l'ensemble des traces d'exécution. Elles sont classées en trois familles principales :

- vérification déductive [Hoa78 ; Fil11] ;
- vérification de modèle [Mer08] ;
- interprétation abstraite [CC77].

La vérification déductive repose sur la logique de Hoare [Hoa78] et le calcul de plus faible précondition. La preuve dont nous parlons en général est la vérification déductive.

La vérification de modèle exige un modèle des états possibles du programme. Elle garantit que les spécifications du programme sont valides dans tous les états du modèle.

L'interprétation abstraite est une théorie d'approximation conservative de la sémantique du programme, permettant une représentation finie de toutes les exécutions du programme. La spécification est vérifiée par des exécutions « abstraites » de cette approximation du programme.

Comme pour le test, nous utilisons la notion d'activité de preuve pour organiser le processus de vérification. Une activité de preuve consiste ici en une unique étape : tenter la preuve d'une spécification. Cette étape est réalisée soit manuellement soit automatiquement par un outil de preuve comme Frama-C/WP [Bau+b] associé au prouveur automatique Alt-Ergo [Bob+]. Suivant la technique de preuve utilisée, il existe des tâches manuelles différentes. Par exemple, avec la vérification déductive, nous pouvons prouver manuellement des spécifications en utilisant Frama-C/WP avec l'assistant de preuve Coq [BC04].

Une tentative de preuve d'une spécification donne lieu à l'un des trois résultats suivants :

- la spécification est bien prouvée ;
- la spécification est prouvée fausse auquel cas un contre-exemple est parfois fourni, ce qui ressemble à un test qui échoue ;
- la preuve échoue par manque de temps ou par manque de capacité des prouveurs utilisés.

Le troisième résultat pose un problème. Pour les deux premiers résultats, nous savons exactement que la spécification est satisfaite (premier résultat) ou non (deuxième résultat), alors que, dans le dernier cas, la spécification n'est pas encore vérifiée. Il faut alors trouver un autre moyen pour vérifier cette spécification, par exemple en utilisant une autre méthode de vérification.

Supposons que dans une activité de preuve, nous tentons de prouver les spécifications de la

fonction `range`. Si toutes les spécifications sont prouvées alors elles sont satisfaites par toutes les exécutions possibles de cette fonction. Si nous trouvons une exécution contradictoire pour la spécification « si $v > a$ alors v » alors nous savons qu'il existe une erreur et nous devons la corriger. Si l'essai de preuve pour « si $v \leq a \leq u$ alors a » échoue (n'est prouvée ni valide ni invalide) alors nous devons reprouver cette spécification avec une autre méthode de preuve.

Comme le test, nous pouvons réaliser plusieurs activités de preuve. En pratique, nous commençons par des outils de preuve automatique. Nous continuons par des outils de preuve partiellement manuels (par exemple, greffon WP de Frama-C associé avec l'assistante de preuve Coq) si la couverture de vérification n'est pas encore satisfaite. Selon la DO-178, nous avons en effet deux analyses de couverture à mener pour la preuve.

La couverture fonctionnelle est satisfaite si et seulement si toutes les spécifications définies sont prouvées.

La vérification par preuve est un domaine récent, et dans la DO-178B, il n'y a pas de critère structurel s'appliquant à la vérification déductive. Pour résoudre ce problème, la DO-333 [RTC11b] (un complément de la DO-178C/ED-12C [RTC11a ; Bro+10 ; Moy+13]) propose une approche alternative à la couverture structurelle. Elle vise à remplacer l'objectif de couverture structurelle par les quatre objectifs suivants :

- validité des méthodes de preuve : l'ingénieur doit analyser les hypothèses nécessaires à la validité de la preuve, par exemple des contraintes de l'environnement et ces hypothèses doivent être validées par ailleurs ;
- complétude : les spécifications sont complètes ;
- flot de données : les spécifications contrôlent toutes les dépendances entre entrées et sorties ;
- code étranger : chaque fragment de code dépend au moins d'une spécification (absence de code mort et de code non-atteignable).

La couverture structurelle de preuve est satisfaite si et seulement si les quatre objectifs ci-dessus sont satisfaits.

Le processus de vérification utilisant les méthodes de preuve est terminé si et seulement si les deux couvertures (fonctionnelle et structurelle) sont satisfaites.

1.2.3 Campagne de vérification

Dans cette thèse, nous introduisons la notion de campagne de vérification pour organiser le processus de vérification. Elle consiste en trois étapes successives, qui peuvent être répétées jusqu'à l'obtention d'une couverture complète :

1. Déterminer l'ensemble des objets de vérification (le code) et des objectifs de vérification (les spécifications et le critère de couverture structurelle choisi) pour chaque objet.
2. Réaliser les activités de vérification (soit une activité de test, soit une activité de preuve).
3. Analyser les résultats des activités de vérification.

Deux cas peuvent ici survenir selon qu'une erreur est trouvée ou non. Si une erreur est trouvée, nous terminons la campagne et nous informons l'utilisateur de cette erreur. Si aucune erreur n'est trouvée, nous calculons la couverture de toutes les activités de vérification et nous concluons de la manière suivante :

- si les deux couvertures sont atteintes, alors la campagne est terminée ;
- si l'une de deux couvertures n'est pas satisfaite, alors nous continuons la campagne en ajoutant d'autres activités de vérification. Dans ce cas, nous retournons à la deuxième étape pour réaliser les activités supplémentaires, ou à la première si les spécifications doivent être raffinées pour couvrir tous les comportements possibles du code.

Théoriquement, les étapes d'une campagne sont appliquées successivement l'une après l'autre : nous exécutons une étape si et seulement si l'étape précédente est finie. Toutes les campagnes commencent par la détermination des objets et de leurs objectifs et terminent quand une erreur est trouvée ou quand les deux couvertures de vérification sont satisfaites.

1.2.4 Problématique de la thèse

La situation n'est néanmoins pas si simple. En réalité, comme présenté précédemment, il n'existe pas de couverture structurelle pour la preuve. Néanmoins, nous pouvons substituer la couverture structurelle pour la preuve par quatre objectifs supplémentaires. En conséquence, nous pouvons calculer la couverture d'une campagne de vérification si et seulement si les activités de cette campagne choisissent soit seulement le test, soit seulement les méthodes de preuve pour effectuer la vérification.

La contrainte ci-dessus pose cependant un problème. Nous savons que l'utilisation des méthodes de preuve pour démontrer la satisfaction d'une spécification a un avantage important : elles sont exhaustives si la preuve aboutit. Néanmoins, s'il existe au moins une spécification que nous ne pouvons ni prouver ni trouver une exécution contradictoire, la campagne ne peut pas être terminée. Alors, nous devons continuer la vérification en réalisant une des deux actions suivantes :

- Si l'ingénieur veut garder les résultats de la campagne, il doit continuer cette dernière en ajoutant les activités (automatiques et/ou manuelles) qui permettent de prouver les spécifications restantes. Ce peut être un travail de longue haleine qui si nous ne parvenons pas au bout, bloque cette campagne.
- Si l'ingénieur ne souhaite pas conserver ses résultats, il suffit alors de remplacer la campagne par une nouvelle qui utilise les méthodes de test. L'ingénieur doit néanmoins redéfinir des objets et des objectifs afin qu'ils soient appropriés au calcul de la couverture des méthodes de test. Cette solution est parfois plus simple que la précédente mais l'utilisateur doit créer une nouvelle campagne de vérification et refaire toutes les parties du travail déjà effectuées.

En pratique, nous observons que les méthodes automatiques de preuve sont plus efficaces que les méthodes automatiques de test, et surtout, plus exhaustives (les risques de bugs résiduels malgré un succès de preuve sont plus faibles). En revanche, nous observons aussi que les méthodes manuelles de test sont plus faciles à maîtriser en temps et en efforts (nous pouvons estimer le temps nécessaire pour finir une campagne de test) alors que cela n'est pas

le cas pour la méthode de preuve (une preuve peut être arbitrairement longue).

Chacune des deux solutions est longue et peut demander beaucoup de travail manuel. Il serait plus efficace d'utiliser le test en complément des preuves effectuées.

Un autre inconvénient de cette contrainte est la réticence à utiliser les résultats des méthodes de preuve dans une campagne de vérification utilisant des méthodes de test. En pratique, une telle campagne repose sur des outils qui créent automatiquement les cas de test. Les cas de test ainsi créés ne satisfont pas toujours les critères de couverture. Pour les compléter, il faut créer manuellement des cas de test supplémentaires. Ce peut être fastidieux et, parfois, générer de mauvais cas de test (e.g. les cas de test qui sont déjà créés par les outils automatiques). Une solution possible à ce problème est l'utilisation des résultats des méthodes de preuve pour trouver le cas de test approprié et éliminer la partie inatteignable du code. Par exemple, dans [Bar+15], Bardin et al. proposent des méthodes de preuve pour trouver des chemins inatteignables pour générer les cas de test.

En conclusion, il serait intéressant de pouvoir utiliser aussi bien le test que la preuve pour la vérification d'une même fonction. Pour pouvoir mettre en œuvre cette approche combinée, il est nécessaire de définir une notion de couverture structurelle adaptée aussi bien au test qu'à la preuve. Définir une telle notion de couverture est le but de notre travail.

1.3 Objets et Objectifs de vérification

Cette section introduit les définitions liées au code et à la spécification. Selon la DO-178 [Bro+10 ; Moy+13], une campagne de vérification doit garantir deux couvertures : fonctionnelle et structurelle. La couverture fonctionnelle vise à assurer la validité de toutes les spécifications définies à l'égard du code. La couverture structurelle, en revanche, assure que la structure du code est entièrement exercée par la vérification des spécifications. La notion de structure du code dépend du critère de couverture structurelle choisi.

Pour mieux expliquer les relations entre code, spécification, couverture structurelle et couverture fonctionnelle, cette section est divisée en trois sous-sections. La première contient toutes les définitions préliminaires. La deuxième développe la notion de fragment de code qui est essentielle pour définir la couverture structurelle. La troisième introduit la notion de spécification, liée à la couverture fonctionnelle.

1.3.1 Définitions préliminaires

Cette partie présente des notions standards pour lesquelles nous introduisons nos propres notations nécessaire à la formalisation de nos travaux.

Point de programme : Un point du programme (ou emplacement) est un identificateur qui est apposé à une instruction du code. Cet emplacement est utilisé pour ordonner et identifier la position de l'instruction dans le code. Chaque fonction f est associée à un ensemble de points de programme $\mathcal{L}$ différents.

La figure 1.6 est dérivée de la figure 1.3 en y ajoutant les points de programme de la fonction `range`. L'ensemble des points de programme ajoutés est $\mathcal{L} \stackrel{\text{def}}{=} \{\texttt{li} \mid \texttt{i} \in [1,6]\} \cup \{\texttt{Pre}, \texttt{Post}\}$. `Pre` et `Post` sont deux points de programme importants de la fonction : le premier indique son point d'entrée tandis que le second désigne son point de sortie.

```
int range(int a, int u, int v)
{
Pre:
 l1: int res = a;
 l2: if(a  <  v){ l3: res = v;}
 l4: if(a  >  u){ l5: res = u;}
 l6: return res;
Post:
}
```

FIGURE 1.6 – Fonction `range` avec emplacements.

État mémoire : Un état mémoire m de l'ensemble des états mémoire $\mathcal{M}$ représente une capture du contenu de la mémoire à un instant donné. m contient les informations de la mémoire à cet instant comme les variables et leurs valeurs et les pointeurs.

La figure 1.7 introduit un appel à la fonction `range`. L'état mémoire m est capturé lorsque cette fonction commence à être exécutée. m contient les trois variables (`a`, `u`, `v`) et leurs valeurs [`a` $= 8$, `u` $= 10$, `v` $= 3$].

```
int main(){
 range (8,10,3);
}
```

FIGURE 1.7 – Appel à la fonction `range`

État du programme : Un état (l, m) d'un programme est un couple composé d'un emplacement et d'un état mémoire. Il représente une capture de l'état m de la mémoire à un endroit précis du code dénoté par l'emplacement l.

Par exemple, l'état mémoire $m_1 \equiv$ [`a` $= 8$, `u` $= 10$, `v` $= 3$] est un état mémoire capturé lorsque la fonction `range` commence à être exécutée. L'emplacement `Pre` dénote le point d'entrée de cette fonction. Donc, la capture de l'état mémoire m_1 à l'emplacement `Pre` forme l'état du programme $(\texttt{Pre}, m_1)$.

Instruction : Chaque instruction du programme est identifiée par un point de programme. Elle peut changer l'état mémoire correspondant (seules *quelques* instructions le peuvent)

et détermine le prochain emplacement (*toutes* les instructions le font). Autrement dit, elle consomme l'état du programme courant et crée un nouvel état du programme. Alors, nous classons les instructions en deux types : les instructions opérationnelles et les instructions conditionnelles. Les instructions opérationnelles contiennent seulement un emplacement de sortie. Ces instructions peuvent modifier l'état mémoire. Les instructions conditionnelles possèdent deux emplacements de sortie possibles dont un seul est choisi en fonction du résultat de l'évaluation de la condition de l'instruction. Dans notre contexte, ces instructions ne modifient pas l'état mémoire.

Soient l, l' et l'' trois emplacements différents et m, m' deux états mémoire différents, les deux types d'instruction sont déterminées comme ci-dessous :

$$\text{Instruction opérationnelle :} \quad (l, m) \xrightarrow{\texttt{Opp}} (l', m')$$

$$\text{Instruction conditionnelle :} \quad (l, m) \xrightarrow{\texttt{Cond}+} (l', m)$$
$$(l, m) \xrightarrow{\texttt{Cond}-} (l'', m)$$

La forme de ces types d'instruction suit la représentation classique d'un programme sous forme de graphe de flot de contrôle [All70].

Dans la fonction `range`, nous avons par exemple des instructions opérationnelles (figure 1.8) et des instructions conditionnelles (figure 1.9).

```
...
l1: int res = a;
l2:
...
l6: return res;
```

FIGURE 1.8 – Instructions opérationnelles

```
...
l2: if(a  <  v){ ... }
l4: if(a  <  v){ ... }
...
```

FIGURE 1.9 – Instructions conditionnelles

L'instruction opérationnelle dans ce cas introduit la variable `res` et sa valeur initiale dans la mémoire. C'est pourquoi l'état mémoire m est remplacé par un autre état mémoire m' identique à m sauf que la variable `res` y est associée à sa valeur `a'` résultant de l'évaluation de `a`. Elle passe aussi à l'emplacement suivant (`l2`). Cette situation est représentée formellement comme suit :

$$(\texttt{l1}, m) \xrightarrow{\texttt{res} \leftarrow \texttt{a}';} (\texttt{l2}, m' = m \cup \{\texttt{res} = \texttt{a}'\})$$

L'instruction conditionnelle n'introduit rien mais consomme l'état mémoire : elle remplace les variables (`a` et `v`) par leur valeur (`a'`, `v'`) qui est dénotée dans l'état mémoire actuel. Si $\texttt{a}' < \texttt{v}'$ est vrai après évaluation, elle saute à l'emplacement `l3`. Sinon, elle saute à l'emplacement `l4`. Cette situation est représentée formellement comme suit :

$$(\texttt{l2}, m) \xrightarrow{(\texttt{a}'<\texttt{v}')\, est\, vrai\, (+)} (\texttt{l3}, m)$$

$$(\texttt{l2}, m) \xrightarrow{(\texttt{a}'<\texttt{v}')\, est\, faux\, (-)} (\texttt{l4}, m)$$

Vecteur d'entrée : Un vecteur d'entrée $\overrightarrow{x} \stackrel{\text{def}}{=} x_1, ..., x_n$ est une liste des valeurs d'entrée d'une fonction. Pour la fonction `range` de l'exemple, on a un vecteur d'entrée $\overrightarrow{x} = 8, 10, 3$ (figure 1.7) dont chaque valeur correspond à une variable d'entrée (resp. `a`, `u`, `v`) de cette fonction.

Exécution : Une exécution $f(\overrightarrow{x})$ d'une fonction f est initiée à partir d'un vecteur d'entrée $\overrightarrow{x} = x_1, ..., x_n$. Elle est représentée par une suite (finie ou infinie) $(l_i, m_i)_{0 \leq i}$ d'états du programme. Le prochain état (l_{i+1}, m_{i+1}) de l'exécution du programme est déterminé à partir de l'application de l'instruction à l'emplacement l_i à l'état du programme courant (l_i, m_i).

Dans notre exemple, en considérant la fonction `range` et un vecteur d'entrée $\overrightarrow{x} = 8, 10, 3$, l'exécution `range`$(\overrightarrow{x})$ est définie par la suite (s) des états de programme suivante :

$$s = (\texttt{Pre}, m_1), (\texttt{l1}, m_1), (\texttt{l2}, m_2), (\texttt{l4}, m_2), (\texttt{l6}, m_2), (Post, m_3)$$

avec `Pre`, `l1`, $\cdots$, `Post` les emplacements de la fonction `range` et m_1, m_2 et m_3 les états mémoire suivants :

m_1 :		m_2 :		m_3 :	
	$\texttt{a} = 8$		$\texttt{a} = 8$		$\texttt{a} = 8$
	$\texttt{u} = 10$		$\texttt{u} = 10$		$\texttt{u} = 10$
	$\texttt{v} = 3$		$\texttt{v} = 3$		$\texttt{v} = 3$
			$\texttt{res} = 8$		$\texttt{res} = 8$
					$result = 8$

Dans l'exemple ci-dessus, *result* représente le résultat de l'appel `range`$(\overrightarrow{x})$.

Séquence : Soit une exécution $f(\overrightarrow{x}) = (l_n, m_n)_{0 \leq n}$. Une séquence d'états entre deux emplacements l_i et l_j $(i < j)$ est dénotée par $s : (l_i, m_i) \hookrightarrow_{f(\overrightarrow{x})} (l_j, m_j)$.

Par exemple, la séquence $s : (l1, m_1) \hookrightarrow_{\texttt{range}(8,10,3)} (l4, m_2)$ de l'exécution ci-dessus est définie par la suite des états de programme suivants :

$$s = (\texttt{l1}, m_1), (\texttt{l2}, m_2), (\texttt{l3}, m_2), (\texttt{l4}, m_2)$$

1.3.2 Objet de vérification

L'objet de vérification est le code. Il peut être étudié à différents niveaux : le programme en entier, une classe ou une fonction comme dans notre exemple. Dans notre travail, nous nous concentrons sur les vérifications unitaires. Cela signifie que l'objet à vérifier dans notre contexte est une fonction.

Fragmenter le code : Pour montrer que l'ensemble du code est vérifié, nous le divisons en plusieurs morceaux appelés fragments. Ces fragments sont créés selon un critère donné de couverture structurelle. Démontrer qu'une campagne de vérification couvre tous les fragments permet de garantir la couverture totale du code.

Par exemple, la couverture des instructions exige que chaque instruction soit exécutée et vérifiée. Selon ce critère, chaque instruction est un fragment. Si la campagne de vérification couvre tous les fragments (toutes les instructions) alors la couverture structurelle est atteinte.

Pourtant, la notion d'emplacement fonctionne seulement avec la couverture des instructions. Pour représenter les autres couvertures possibles, la notion d'emplacement n'est pas suffisante. Par exemple, pour la couverture MC/DC, nous ne pouvons pas utiliser seulement l'emplacement pour représenter toutes les combinaisons possibles de chaque instruction conditionnelle.

Label : Pour permettre l'identification de chaque fragment pendant la vérification, nous étiquetons chacun par un label. La notion de label, introduite par Bardin et al. [BKC14 ; Bar+14 ; Bar+15], est construite en groupant un emplacement l et une propriété $h \in \mathcal{P}(\mathcal{M})$ sur les états de la mémoire. Elle peut interpréter un état mémoire comme une valeur proportionnelle en remplaçant chacune de ces variables par la valeur qui lui est associée dans l'état mémoire et en évaluant la formule résultant. h est donc la propriété de l'ensemble des états mémoire $\mathcal{M}$ dont l'évaluation $h(m)$ de chaque état mémoire m retourne « vrai ».

La figure 1.10 montre un exemple de label. À l'emplacement `l2`, nous trouvons une instruction conditionnelle qui contient deux variables (`a` et `v`). Nous souhaitons connaître ce qui se passe si $\texttt{a} > \texttt{v}$, si $\texttt{a} \equiv \texttt{v}$ et si $\texttt{a} < \texttt{v}$. Pour cela, nous définissons trois labels à l'emplacement `l2`, chacun représente une de ces trois situations. Chaque label représente un ensemble des états mémoire identifié par la relation entre `a` et `v`. Par exemple, l'état mémoire $m \equiv [\texttt{a} = 8, \texttt{u} = 10, \texttt{v} = 3]$ appartient au label $(\texttt{l2}, \texttt{a} > \texttt{v})$ car 8 est supérieur à 3.

```
int range(int a, int u, int v)
{
Pre:
 ...

 // label (l2, a  >  v)
 // label (l2, a  ≡  v)
 // label (l2, a  <  v)
 l2: if(a  <  v){ l3: res = v;}

 ...
Post:
}
```

FIGURE 1.10 – Fonction `range` avec label

Nous disons que l'exécution $f(\vec{x})$ *passe par* un label $L = (l, h)$ (i.e. le label l est *couvert par* l'exécution $f(\vec{x})$), ce qui est noté $f(\vec{x}) \rightsquigarrow (l, h)$, si et seulement si :

$$f(\vec{x}) \rightsquigarrow (l,h) \stackrel{\text{def}}{=} \text{il existe } m \in h \text{ tel que } (l,m) \in f(\vec{x}).$$

Par exemple, considérons l'exécution $\texttt{range}(8,10,3)$ et les labels de la figure 1.10. L'état du programme $(\texttt{l2}, m_2)$ est un état de cette exécution et est le seul état qui contient l'emplacement `l2`. Comme présenté précédemment, l'état mémoire m_2 contient les variables et leurs valeurs : $[\texttt{a} = 8; \texttt{u} = 10; \texttt{v} = 3; \texttt{res} = 3]$. En évaluant la condition du label avec cette mémoire, nous obtenons $8 > 3$ qui est une formule vraie. Autrement dit, m_2 est un membre de l'ensemble caractérisé par le label considéré.

Si nous pouvons montrer que chaque label est couvert par au moins une exécution *via* au moins une activité de vérification (même sans identifier l'exécution), nous pouvons déduire que l'ensemble du code est couvert par l'ensemble de ces activités. Autrement dit, la campagne de vérification constituée de ces dernières permet d'obtenir une couverture structurelle totale.

Critères de couverture structurelle La couverture structurelle [Bro+10 ; Moy+13] est la couverture de la structure d'une fonction dans la campagne de vérification. Il existe différents critères de couverture structurelle [RTC11a] (e.g. la couverture des instructions, la couverture des branches, la couverture MC/DC simplifiée, etc). Pour chaque critère de couverture, Bardin et al. [BKC14 ; Bar+14] ont défini les labels nécessaires pour le représenter. Ils définissent aussi LTest [Bar+14] dont le module LAnnotate peut générer automatiquement les labels selon un critère de couverture choisi.

Les méthodes de test utilisent un ensemble de labels pour représenter la structure d'une fonction. Comme nous l'avons montré, un label représente un fragment du code. Si nous pouvons montrer que tous les fragments du code sont vérifiés par une campagne de couverture (via leurs labels), alors le critère de couverture structurelle représenté par les labels est garanti pour cette campagne.

Par exemple, considérons la couverture des instructions et la couverture des branches. Comme exprimé ci-dessus, la couverture des instructions assure que la vérification exerce toutes les instructions du code tandis que la couverture des branches concerne les évaluations des formules booléennes des instructions conditionnelles. Pour chaque critère, nous fragmentons différemment le code (figure 1.11 et figure 1.12). Un critère de couverture structurelle est satisfait si et seulement si chaque label créé pour ce critère est couvert par au moins une exécution dans la campagne de vérification.

La notion de couverture structurelle a été définie pour le test mais n'est pas adaptée pour les méthodes de preuve. Néanmoins, elle peut être remplacée par d'autres critères comme la complétude des spécifications ou l'absence de code mort. Un autre problème concerne les labels inatteignables. Le test seul n'est pas capable de détecter ces labels.

```
int range(int a, int u, int v)
{
Pre:

 // label (l1, vrai)
 l1: int res = a;

 // label (l2, vrai)
 l2: if(a  <  v){

  // label (l3, vrai)
  l3: res = v;
 }

 // label (l4, vrai)
 l4: if(a  >  u){

  // label (l5, vrai)
  l5: res = u;
 }

 // label (l6, vrai)
 l6: return res;
Post:
}
```

FIGURE 1.11 – Labels pour la couverture des instructions

```
int range(int a, int u, int v)
{
Pre:
 l1: int res = a;

 // label (l2, a < v)
 // label (l2, a ≥ v)
 l2: if(a  <  v){
  l3: res = v;
 }

 // label (l4, a > u)
 // label (l4, a ≤ u)
 l4: if(a  >  u){
  l5: res = u;
 }

 l6: return res;
Post:
}
```

FIGURE 1.12 – Labels pour la couverture des branches

1.3.3 Objectifs de vérification

Nous représentons les objectifs de vérification sous la forme d'une collection de spécifications. Chaque spécification S est constituée de deux parties : une hypothèse et une conclusion. Une telle spécification est notée par $S \stackrel{\text{def}}{=} H \Rightarrow C$.

Partie hypothèse : Une hypothèse H correspond à $H \stackrel{\text{def}}{=} (l, h)$ avec l un emplacement dans la fonction (en général, l'état Pre de la fonction) et h une propriété sur les états de la mémoire.

Nous pouvons remarquer que les notions d'hypothèses et de labels sont formalisées par le même type d'objet mathématique. Néanmoins, dans la section 2.2, nous étendons la notion de label (l, h) avec une notion de mutant M, tandis que la notion d'hypothèse restera inchangée. Elles deviendront donc différentes.

Partie conclusion : Soient l et l' deux emplacements dans la fonction tels que $l \leq l'$ et $r \in \mathcal{P}(\mathcal{M} \times \mathcal{M})$ une relation entre deux états mémoire. Une conclusion C correspond au triplet l, l' et r et est notée $C \stackrel{\text{def}}{=} (l, l', r)$.

Spécification : Comme nous l'avons indiqué au début de cette section, les objectifs de vérification sont des spécifications $S \stackrel{\text{def}}{=} H \Rightarrow C$ constituées d'une hypothèse H et d'une conclusion C telles que l'emplacement de l'hypothèse H est aussi le premier emplacement de la conclusion C. Comme pour un label, nous montrons la satisfaction de chaque spécification en utilisant la notion de « passer par ». Ainsi, $f(\vec{x}) \rightsquigarrow (l,h)$ (resp. $f(\vec{x}) \rightsquigarrow (l,l',r)$) dénote que $f(\vec{x})$ passe par l'hypothèse $H = (l,h)$ (resp. la conclusion $C = (l,l',r)$). Par contre, *passer par* l'hypothèse H et *passer par* la conclusion C ne sont pas deux notions identiques. En effet, pour l'hypothèse H, $f(\vec{x}) \rightsquigarrow (l,h)$ indique que l'exécution $f(\vec{x})$ atteint l et que l'état de mémoire correspondant satisfait h, alors que pour la conclusion C, $f(\vec{x}) \rightsquigarrow (l,l',r)$ signifie que l'exécution atteint toujours l' après l et que pour chaque séquence $s : (l,m) \hookrightarrow_{f(\vec{x})} (l',m')$ de l'exécution $f(\vec{x})$, les deux états mémoire correspondants (m et m') satisfont la relation r (i.e. $(m,m') \in r$). Plus formellement, les notions de *passer par* une hypothèse et une conclusion sont définies ci-dessous :

$$f(\vec{x}) \rightsquigarrow (l,h) \stackrel{\text{def}}{=} \exists (l_i,m_i) \in f(\vec{x}), l_i \equiv l \wedge m_i \in h;$$
$$f(\vec{x}) \rightsquigarrow (l,l',r) \stackrel{\text{def}}{=} \forall (m,m') \in \mathcal{M}^2, \exists\, s : (l,m) \hookrightarrow_{f(\vec{x})} (l',m'), (m,m') \in r.$$

Pour la fonction `range` de notre exemple, nous pouvons définir les objectifs de vérification par les trois spécifications S_1, S_2 et S_3 suivantes :

$$\begin{aligned} S_0 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{a} < \texttt{v} < \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{v}) \\ S_1 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{v} \leq \texttt{a} \leq \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{a}) \\ S_2 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{v} < \texttt{u} < \texttt{a}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{u}) \end{aligned}$$

Chaque spécification représente une situation de sortie de la fonction `range`.

1.4 Méthode de vérification et Couverture

Dans notre contexte, une activité de vérification est soit un test unitaire (i.e. la vérification des fonctions par une méthode de test [Sof08]) soit une preuve unitaire (i.e. la vérification des fonctions par une méthode de preuve [Fil11 ; Flo93]). N'importe laquelle de ces deux activités peut être utilisée pour vérifier une spécification. Chacune fournit des garanties sur la satisfaction de la spécification vérifiée.

Dans cette section, nous utilisons la notion de campagne de vérification précisée précédemment pour organiser le processus de vérification. Une campagne doit contenir trois étapes :

- déterminer le code, les spécifications et les critères de couverture souhaités ;
- définir et réaliser les activités de vérification pour vérifier toutes les spécifications avec le code donné ;

– analyser les résultats, identifier les erreurs le cas échéant, déterminer la satisfaction des critères de couverture et décider de terminer ou de retourner à la deuxième étape.

Une campagne de vérification peut avoir plusieurs activités de vérification. Elle termine si et seulement si aucune d'erreur n'est trouvée et si les critères de couverture sont tous satisfaits.

Comme présenté précédemment, selon la DO-178, il existe deux types de couverture : la couverture fonctionnelle et la couverture structurelle. La couverture fonctionnelle consiste à garantir la vérification de chaque spécification. La couverture structurelle consiste à montrer qu'aucune instruction ni chemin d'exécution n'est oublié pendant une campagne de vérification. Peu importe la méthode de vérification utilisée : les deux couvertures doivent être satisfaites.

1.4.1 Test et Couverture

Comme définie précédemment, une activité de test contient deux étapes :

1. Générer les cas de test
   – manuellement : l'ingénieur définit les cas de test selon ses besoins propres. Par exemple, pour garantir la couverture des branches, l'ingénieur analyse le code et détermine exactement un cas de test pour chaque branche.
   – automatiquement : plusieurs outils [Bar+14 ; KHS11] sont capables de créer automatiquement des cas de test appropriés aux spécifications ou qui satisfont un critère de couverture donné. Par exemple, le greffon LTest [Bar+14] de Frama-C peut générer les cas de test selon un critère de couverture choisi.
2. Évaluer chaque cas de test en utilisant un oracle
   – manuellement : l'ingénieur évalue chaque cas de test avec chaque spécification. Par exemple, l'ingénieur peut exécuter chaque cas de test et vérifier la relation entrée sortie de chaque cas de test.
   – automatiquement : il existe des outils qui évaluent automatiquement chaque cas de test avec chaque spécification comme le greffon E-ACSL [SKV17] de Frama-C. Il existe aussi une proposition à développer un outil [PP98] qui permet de créer l'oracle à partir d'une documentation formelle.

Automatiser en grande partie les activités de test en utilisant un outil de génération de cas de test est tout à fait classique. Néanmoins, ici, nous ne nous intéressons pas à ces outils en tant que tels, mais uniquement aux bénéfices de l'automatisation des activités de test qu'ils permettent.

Soit une spécification $S \stackrel{\text{def}}{=} H \Rightarrow C$ et une exécution $f(\overrightarrow{x})$ de la fonction f. Le test de la spécification S pour l'exécution $f(\overrightarrow{x})$, peu importe qu'il soit manuel ou automatique, doit produire un des résultat suivants :

– **Satisfaction par test :** La spécification S est satisfaite par rapport à un cas de test contenant l'exécution $f(\overrightarrow{x})$ si et seulement si $f(\overrightarrow{x})$ passe par l'hypothèse et la

conclusion de S :

$$f(\vec{x}) \rightsquigarrow S \stackrel{\text{def}}{=} (f(\vec{x}) \rightsquigarrow H) \wedge (f(\vec{x}) \rightsquigarrow C)$$

– **Non-satisfaction par test :** La spécification S n'est pas satisfaite par rapport à un cas de test contenant l'exécution $f(\vec{x})$ si et seulement si $f(\vec{x})$ passe par l'hypothèse mais ne passe pas par la conclusion de S :

$$f(\vec{x}) \not\rightsquigarrow S \stackrel{\text{def}}{=} (f(\vec{x}) \rightsquigarrow H) \wedge (f(\vec{x}) \not\rightsquigarrow C)$$

– **Hors cas de test** Si l'exécution $f(\vec{x})$ d'un cas de test ne passe pas par l'hypothèse H de la spécification S, nous concluons que ce cas de test n'est pas un véritable cas de test pour cette spécification :

$$f(\vec{x}) \not\sim S \stackrel{\text{def}}{=} (f(\vec{x}) \not\rightsquigarrow H)$$

Une campagne de test est une campagne de vérification qui utilise seulement des méthodes de test. Comme les autres campagnes de vérification, elle est terminée si et seulement si les critères de couvertures sont atteints. Chaque campagne doit satisfaire deux couvertures : couverture fonctionnelle et couverture structurelle. Une campagne de test garantit la couverture fonctionnelle si et seulement si, pour chaque spécification définie dans la campagne, il existe au moins un cas de test qui satisfait cette spécification. Une campagne de test garantit la couverture structurelle si et seulement si, pour chaque label créé dans la campagne selon le critère choisi (couverture des instructions, couverture des branches, etc), il existe au moins un cas de test dont l'exécution passe par ce label.

Dans notre travail, pour synthétiser les résultats des tests par rapport à la couverture, nous introduisons une représentation par matrice appelée matrice de couverture. Cette matrice nous aide à visualiser la couverture fonctionnelle et la couverture structurelle d'une campagne. La figure 1.13 montre la structure de cette matrice. Chaque colonne de la matrice représente une spécification et chaque ligne représente un label. La cellule à l'intersection entre une spécification $(S_i)_{i\geq 0}$ et un label $(L_j)_{j\geq 0}$ est marquée par l'information de couverture $Cover(S_i, L_j)$ de S_i et L_j. Cette information est définie comme suit :

$$Cover(S_i, L_j) \stackrel{\text{def}}{=} \begin{cases} \text{symbole « T » si l'exécution du test qui passe par } S_i \text{ passe aussi par } L_j\,; \\ \text{cellule vide sinon.} \end{cases}$$

	...	Spécification S_i	...
...			
Label L_j		$Cover(S_i, L_j)$	
...			

FIGURE 1.13 – Modèle de table pour synthétiser la couverture de test

La cellule (S_i, L_j) qui est marquée par T porte deux information complémentaires :

1. La spécification S_i de cette cellule est satisfaite par une exécution d'une activité de

	S_0	S_1	S_2
$\mathtt{L}_1 = (\mathtt{l1}, \mathtt{true})$	T	T	T
$\mathtt{L}_2 = (\mathtt{l3}, \mathtt{true})$	T		
$\mathtt{L}_3 = (\mathtt{l5}, \mathtt{true})$			T

FIGURE 1.14 – Exemple de table dans une campagne de test pour la fonction `range`

test de la campagne,

2. Le label L_j est couvert par cette exécution.

La couverture fonctionnelle assure que toutes les spécifications sont vérifiées dans la campagne. Elle est satisfaite si et seulement si chaque colonne contient une marque `T`. La couverture structurelle demande que chaque label doit être couvert par une exécution d'une activité de test de la campagne. Elle est satisfaite si et seulement si chaque ligne contient une marque `T`.

Reprenons notre exemple de la figure 1.3 en considérant une campagne de test. Cette campagne contient une activité de test qui a trois cas de test. L'exécution de chaque cas de test se trouve dans la figure 1.5. Le résultat synthétisé par la matrice de couverture est montré dans la figure 1.14. À partir de cette table, nous voyons que chaque spécification contient au moins une marque `T`. Autrement dit, toutes les spécifications sont testées dans la campagne. La couverture fonctionnelle est donc satisfaite. Nous voyons aussi que chaque label contient au moins une marque `T` donc que chaque label est couvert par au moins une exécution de la campagne de test. La couverture structurelle est ainsi également satisfaite.

En pratique, une campagne de vérification par test commence par des activités de test qui sont totalement automatisées à l'aide d'outils pour générer et évaluer les cas de test pour chaque spécification définie. Il y a néanmoins un problème : puisque nous utilisons les outils pour définir les cas de test, nous ne pouvons pas contrôler les résultats. Ainsi, dans la majorité des cas, il existe des spécifications qui ne sont pas vérifiées et des labels qui ne sont pas couverts par les exécutions des activités de la campagne. Pour ces cas, nous devons définir manuellement des cas de test complémentaires pour terminer la campagne de vérification. C'est un travail potentiellement fastidieux parce que nous ne savons pas quels cas de test sont nécessaires pour compléter la campagne. Pourtant, nous devons la terminer et les méthodes de preuve peuvent aider. Donc, ensuite, nous présentons la vérification par preuve et la couverture.

1.4.2 Preuve et Couverture

Une activité de preuve contient seulement une étape : choisir la méthode de preuve et l'utiliser pour prouver les spécifications. Cette étape est soit complètement automatique (par exemple, en utilisant Frama-C/WP [Bau+b] et le prouveur automatique Alt-Ergo [Bob+]) soit partiellement manuelle (par exemple, le Frama-C/WP avec l'assistant de preuve Coq [BC04]). Dans notre thèse, nous ne nous intéressons pas à la fonctionnalité de l'activité de preuve, mais plutôt aux résultats de ces activités. Considérons la spécification $S \stackrel{\text{def}}{=} H \Rightarrow C$ d'une fonction

f. Une activité de preuve qui essaye de prouver S retourne un des résultats suivants :

- **Satisfaction par preuve :** la tentative de preuve de l'activité de preuve réussit à prouver la spécification S. Dans ce cas, pour toutes les exécutions $f(\overrightarrow{x})$ de la fonction f, soit S est satisfaite soit S n'est pas pertinente car son hypothèse H n'est pas satisfaite :

$$f \models S \stackrel{\text{def}}{=} \forall f(\overrightarrow{x}), \begin{cases} (f(\overrightarrow{x}) \rightsquigarrow H) \wedge (f(\overrightarrow{x}) \rightsquigarrow C), \\ \text{ou } f(\overrightarrow{x}) \not\rightsquigarrow H \end{cases}$$

- **Non-satisfaction par preuve :** la tentative de preuve trouve un contre-exemple, c'est-à-dire une exécution $f(\overrightarrow{x})$ quelconque, qui contredit la spécification S. Cela signifie que la spécification est non-satisfaite par rapport à cette exécution :

$$f \not\rightsquigarrow S \stackrel{\text{def}}{=} \exists f(\overrightarrow{x}), (f(\overrightarrow{x}) \rightsquigarrow H) \wedge (f(\overrightarrow{x}) \not\rightsquigarrow C)$$

- **Résultat indéterminé :** la tentative de preuve ne peut ni prouver la spécification S, ni trouver une exécution qui contredit S. Cette situation qui peut avoir plusieurs origines [Pet+16] est notée par $f \not\models S$. Dans ce cas, nous devons vérifier S avec une autre activité de preuve.

Une campagne de preuve est une campagne qui utilise seulement les activités de preuve. Comme les autres campagnes de vérification, elle doit satisfaire deux couvertures : la couverture fonctionnelle et la couverture structurelle. Dans cette campagne, la couverture fonctionnelle exige que des activités de preuve prouvent toutes les spécifications définies dans la campagne. En revanche, la couverture structurelle ne peut pas être directement satisfaite par une campagne de preuve car il n'existe aucun critère de couverture adapté à la preuve. Selon la DO-333 [RTC11b], le supplément Méthodes Formelles du DO-178C/ED12C [RTC11a], l'objectif de couverture structurelle peut être remplacé par les quatre objectifs suivants :

1. **Sûreté des méthodes de preuve :** normalement, l'essai de preuve prouve les spécifications utilisant des invariances, des lemmes, des contraintes (e.g. les contraintes de l'environnement). Ces éléments doivent être vérifiés aussi.
2. **Complétude :** les spécifications de la campagne de vérification sont complètes. Cela signifie que pour toutes les exécutions $f(\overrightarrow{x})$ possible d'une fonction f, il existe une spécification $S = H \Rightarrow C$ qui est prouvée dans la campagne et telle que $f(\overrightarrow{x}) \rightsquigarrow H$.
3. **Flot de données :** toutes les dépendances entre entrées et sorties doivent être définies dans les spécifications.
4. **Code étranger :** il n'existe aucun code mort (les instructions qui sont exécutées mais redondantes) ou code non-atteignable (les instructions qui ne sont passées par aucune exécution du code).

Nous pouvons atteindre manuellement ou automatiquement les objectifs ci-dessus. Par exemple, certains outils [Cuo+12] peuvent garantir l'absence des dépendances non-définies entre entrées et sorties (Flot de données). Il existe aussi des moyens [Neu+16 ; CGK98] qui nous aident à trouver du code mort et du code non-atteignable (code étranger).

Pour synthétiser la couverture, nous introduisons aussi une représentation par table (figure 1.15). Néanmoins, puisque nous n'avons pas de notion de couverture structurelle pour la preuve, nous pouvons expliciter seulement la couverture fonctionnelle. Cette table ne contient

···	Spécification S_i	···
	$Proof(S_i)$	

FIGURE 1.15 – Modèle de table pour synthétiser la couverture de preuve

S_0	S_1	S_2
P	P	P

FIGURE 1.16 – Exemple de table dans une campagne de preuve pour la fonction `range`

donc que les colonnes représentant les spécifications. $Proof(S_i)$ est l'information de couverture utilisée pour la spécification S_i. Elle est définie comme suit :

$$Proof(S_i) \stackrel{\text{def}}{=} \begin{cases} \text{symbole « P » si } S_i \text{ est prouvée dans la campagne} \\ \text{cellule vide sinon} \end{cases}$$

Créons à présent une campagne de preuve pour vérifier les trois spécifications de la fonction `range`. Cette campagne contient une activité de preuve qui essaye de prouver ces spécifications. La figure 1.16 montre le résultat de cette activité. Nous voyons que toutes les spécifications de l'exemple sont prouvées. Cela signifie que la couverture fonctionnelle est satisfaite. Cette table ne garantit rien concernant la couverture structurelle. Pour cela, nous devons montrer les objectifs supplémentaires précédemment présentés via l'utilisation d'autres techniques et outils. Cela demande donc plus d'efforts de la part de l'ingénieur.

Comme pour une campagne de test, nous pouvons aussi automatiser une campagne de preuve en utilisant des outils. Si toutes les spécifications sont prouvées, alors tout va bien. Néanmoins, dans la majorité des cas, il y a des spécifications que l'outil ne peut pas prouver et pour lesquelles il ne peut pas non plus trouver un contre-exemple. Dans ce cas, deux choix s'offrent à nous. Premièrement, nous ajoutons des annotations supplémentaires comme des lemmes et prouvons interactivement les spécifications non prouvées. Si ce n'est pas possible (ou demande trop d'effort), nous devons nous tourner vers la seconde solution qui consiste à jeter cette campagne et tous ses résultats associés et à la remplacer par une campagne de test. Ce choix gaspille donc tous les efforts de preuve déjà dépensés.

1.4.3 Combinaison Test et Preuve et Couverture

À cause de la différence dans le calcul de couverture des activités de test et des activités de preuve, nous ne pouvons pas déterminer pour l'instant la satisfaction de la couverture si la campagne utilise à la fois des activités de test et des activités de preuve.

Néanmoins, certaines spécifications sont plus faciles à tester qu'à prouver, alors que d'autres, au contraire, sont plus faciles à prouver qu'à tester. Une utilisation combinée des activités de test et de preuve peut donc augmenter la probabilité de terminer une campagne de

vérification. De plus, nous avons présenté ci-dessus, aux sections 1.4.1 et 1.4.2, des problèmes des campagnes de vérification automatique qui sont résolus par une combinaison de test et de preuve. Une campagne de test automatique repose sur des outils qui génèrent automatiquement les cas de test. Pourtant, il existe le plus souvent des spécifications qui ne sont pas vérifiées ou des labels qui ne sont couverts par aucune exécution. Dans ce cas, nous devons créer manuellement des cas de test additionnels. Une méthode de preuve permet de prouver des spécifications et de trouver des labels inatteignables (cf [Bar+15]). Les résultats de cette méthode peuvent nous aider à réduire l'effort nécessaire pour définir ces cas de test dans la campagne de test. Réciproquement, pour une campagne de preuve, il existe parfois des spécifications qui ne sont pas vérifiées. Il serait très utile de pouvoir garder les résultats de cette campagne et de la compléter par des activités de test qui ne vérifient que les spécifications restantes. La clé pour réaliser ces propositions est d'avoir un critère de couverture commun au test et à la preuve.

C'est ce que nous réalisons dans cette thèse en proposant une définition de couverture utilisable lorsque certaines spécifications sont prouvées et les autres testées. En analysant les informations données précédemment, le problème vient de la notion de couverture structurelle : tandis que les méthodes de test peuvent utiliser des labels pour déterminer la couverture structurelle totale, les méthodes de preuve utilisent une approche alternative pour la garantir. Pour résoudre ce problème, nous proposons une méthode qui relie la notion de label aux méthodes de preuve. Nous montrerons aussi que cette méthode est utile pour le test et, qu'en conséquence, nous pouvons l'utiliser quand nous combinons plusieurs outils de vérification (e.g. un outil de test et un outil de preuve).

1.5 Mutation

Dans les sections précédentes, nous avons introduit les formalisations des définitions de base. À partir de ces définitions, nous pouvons formaliser l'objet principal de cette thèse : déterminer une méthode qui peut relier la notion de label à la notion de preuve, afin de calculer la couverture d'une combinaison test et preuve.

Notre solution repose fortement sur le concept de mutation, introduit dans cette section. En effet, la couverture par mutation peut être vue comme un cas particulier de couverture structurelle pour le test, et nous verrons que la mutation peut aussi être utilisée pour définir la couverture structurelle d'une preuve.

Ici, les mutations [Bud+80 ; DLS78] auxquelles nous nous intéressons sont les mutations au sein des fonctions. Le résultat d'une mutation est une autre fonction, appelée « mutant ». Il existe dans la littérature plusieurs opérateurs de mutation [Agr+99 ; OVP96 ; MOK06] qui modifient les fonctions de manières différentes (effacer une instruction, changer une valeur, $\cdots$). Nous ne déterminons ici aucun nouvel opérateur de mutation, mais nous utilisons plutôt d'une manière spécifique les opérateurs existants afin de les adapter à notre méthode.

La figure 1.17 représente un mutant de la fonction `range` introduite figure 1.6. Pour obtenir

```
int range(int a, int u, int v){
 ...
 l2: if(a  <  v){ l3: res = v;}
 ...
}
```

Code original

```
int range_mut(int a, int u, int v){
 ...
 // supprimer l'instruction res = v
 l2: if(a  <  v){ l3: ;}
 ...
}
```

Code après mutation

FIGURE 1.17 – Mutation par suppression d'instruction au label `l3`

ce mutant, nous utilisons un opérateur de mutation qui supprime l'instruction se trouvant à l'emplacement `l3`.

Le choix de l'opérateur de mutation dépend du critère de couverture structurelle que nous voulons réaliser [AO08]. L'exemple ci-dessus choisit l'effacement d'une instruction opérationnelle parce que nous choisissons la couverture des instructions. Si nous choisissons la couverture des branches, nous pouvons choisir une autre opération de mutation (cf. figure 1.18). Cette opération remplace la formule booléenne des instructions conditionnelles par une constante booléenne (`vrai` ou `faux`) pour rediriger les exécutions vers la branche choisie.

```
int range(int a, int u, int v){
 ...
 l2: if(a  <  v){ l3: res = v;}
 ...
}
```

Code original

```
int range_mut(int a, int u, int v){
 ...
 // remplacer la condition par 1
 l2: if(1){ l3: res = v;}
 ...
}
```

Code après mutation

FIGURE 1.18 – Mutation par modification de condition au label `l2`

1.5.1 Mutation et Test

Dans la littérature [AO08], la mutation est utilisée comme un critère de couverture structurelle pour le test. Elle sert à évaluer la qualité des cas de test. Chaque mutant modifie le comportement du code source. Chaque cas de test est exécuté sur la fonction d'origine et sur le mutant, et nous nous intéressons aux différences de comportement des deux fonctions. Deux situations peuvent se produire :

- Si le cas de test se comporte de manière identique sur la fonction et son mutant, il est considéré comme indépendant de l'emplacement. Il existe cependant deux possibilités pour cette situation :
  - l'oracle du cas de test est trop faible pour détecter une différence de comportement des deux fonctions et donc, nous devons utiliser un autre oracle ;
  - tous les comportements existants sont effectivement indépendants de la partie mu-

tée du code. Dans cette situation, nous devons redéfinir l'ensemble des comportements.
- Si le cas de test donne un résultat différent dans le mutant, alors c'est un cas de test fortement corrélé à l'emplacement. Cette situation confirme la qualité de l'oracle. Le mutant, dans ce cas, est dit « tué ».

Dans notre formalisme, la notion du test par mutation est modélisée en considérant un test de spécification S avec une fonction source f et un mutant f', créé en mutant l'instruction à l'emplacement l de f. Soit un vecteur d'entrée quelconque $\overrightarrow{x}$. Jusqu'à présent, nous considérons que $f(\overrightarrow{x})$ est un vrai cas de test pour la spécification $S = H \Rightarrow C$ si et seulement si $f(\overrightarrow{x}) \rightsquigarrow H$. Le verdict du cas de test est observé via le passage de $f(\overrightarrow{x})$ par la conclusion C. Ici, nous supposons que $f(\overrightarrow{x}) \rightsquigarrow C$.

Maintenant, nous vérifions aussi la spécification S avec le mutant f' avec le même vecteur d'entrée $\overrightarrow{x}$. Il existe trois résultats possibles :
- $f'(\overrightarrow{x}) \not\rightsquigarrow H$: le mutant est tué mais la position mutée précède celle de l'hypothèse H. Cette mutation modifie la fonction de la façon que $f(\overrightarrow{x}) \rightsquigarrow H$ devienne faux. Dans ce cas, le mutant créé n'a donc aucune relation avec cette spécification. Il n'y a pas de couverture entre S et l.
- $f'(\overrightarrow{x}) \rightsquigarrow S$: le mutant n'est pas tué mais nous pouvons continuer à tester cette spécification en utilisant d'autres cas de test. Si le mutant n'est toujours pas tué après l'exécution de nombreux cas de test, nous pouvons penser que S est indépendante de la partie mutée.
- $f'(\overrightarrow{x}) \not\rightsquigarrow S$: le mutant est tué. Dans ce cas, le cas de test est un cas de test couvrant cette spécification.

En conclusion, nous trouvons que en testant les spécifications dans les mutants, nous pouvons relier chaque spécification à chaque portion du code, et plus général, à un label. Cela nous aide à établir une couverture structurelle pour le test en ne concernant que les résultats du test. Donc, nous pensons que la notion du test par mutation est très utile pour définir notre critère de couverture.

1.5.2 Mutation et Preuve

Certains travaux de la littérature comme [CKV03] introduisent déjà l'utilisation de mutants avec des méthodes de preuve. Par contre, dans le travail dans [CKV03], la mutation est appliquée à la vérification de modèle pour vérifier chaque partie du modèle avec les exigences du logiciel. Ici, nous introduisons ce concept dans notre formalisme en mutant directement chaque instruction du code.

Supposons qu'une certaine spécification S soit prouvée pour la fonction f. Nous considérons maintenant un mutant f' créé par modification de l'instruction à l'emplacement l. Lorsque nous tentons de prouver S avec f', nous avons trois résultats possibles :
- $f' \models S$: nous pouvons en conclure que la spécification et la partie mutée sont indépendantes l'une de l'autre ;

- $f' \not\rightsquigarrow S$: l'essai de preuve trouve un contre-exemple de f' pour S. Cela signifie que le mutant est tué et donc qu'il existe une relation entre S et l ;
- $f' \not\models S$: la spécification S qui est prouvée pour f n'est pas encore prouvée pour f'. Alors, il est possible mais pas certain que l'instruction modifiée à l'emplacement l a une relation avec la spécification S. Dans ce cas, nous pouvons créer un test qui vérifie S et qui exerce l'instruction à l'emplacement l pour confirmer cette possibilité.

Nous voyons donc que nous pouvons exploiter les résultats de preuve sur le mutant en l pour établir un lien entre une spécification et un label de programme. Il ressemble à l'utilisation des résultats de test sur le mutant en l. Nous pouvons donc utiliser la notion de mutant pour définir le nouveau critère de couverture. Ce critère est détaillé dans le chapitre suivant.

Quis custodiet ipsos custodes ?
Juvénal

Mais qui gardera ces gardiens ?
traduction de Olivier Sers

2

La couverture label-mutant

Dans le chapitre précédent, nous avons identifié le problème principal que nous souhaitons résoudre. Maintenant, nous allons introduire formellement notre solution : une nouvelle notion de couverture. Informellement, celle-ci consiste d'abord à considérer un fragment d'une fonction et une spécification déjà vérifiée, puis à créer un ou plusieurs mutants de ce fragment, ensuite à les vérifier et enfin à comparer les résultats pour déterminer l'impact du fragment muté sur les spécifications précédemment vérifiées.

Dans une campagne de vérification, les spécifications sont définies à partir des exigences du programme et les fragments sont identifiés selon le critère de couverture choisi. Une spécification est dite *vérifiée* si cette spécification est impactée par au moins un fragment. La vérification atteint une couverture fonctionnelle si et seulement si toutes les spécifications sont vérifiées. De même, un fragment a été *couvert* si ce fragment a un impact sur au moins une spécification. La couverture structurelle est satisfaite si et seulement si tous les fragments définis sont couverts.

La présentation de la nouvelle notion de couverture comprend la définition de cinq éléments principaux qui sont présentés dans cinq sections dédiées de ce chapitre. Ce sont :

1. une notion de témoin pour formaliser les activités de vérification ;
2. les notions de mutants de code et de mutants de spécification ;
3. des témoins supplémentaires dénotant les vérifications des mutants ;
4. un ordre permettant de trier une collection de mutants ;
5. une notion de campagne de vérification synthétisant les résultats d'une campagne de vérification et permettant d'en mesurer la couverture.

2.1 Témoin de vérification

Le chapitre précédent a introduit la notion de campagne de vérification. Chaque campagne est composée de différentes activités de vérification. Chaque activité utilise une méthode de vérification dédiée pour effectuer ses actions de vérification. Chaque action est ainsi constituée des éléments suivants :

- la spécification vérifiée ;
- le résultat produit par la vérification ;
- **(optionnel)** les labels exercés par la vérification.

Chaque enregistrement de ce type est appelé un « témoin ».

Nous considérons ici deux techniques de vérification : les méthodes de test et les méthodes de preuve. Pour chaque technique, nous analysons leurs résultats possibles (voir les sections 1.4.1 et 1.4.2 pour savoir la formalisation des résultats de vérification) et argumentons pour déterminer le type de témoins que nous voulons utiliser.

- Témoins de **Test :**
  - **(`T`) :** L'exécution $f(\vec{x})$ de l'activité de test satisfait la spécification $S = H \Rightarrow C$ et passe par un label L. Nous utilisons le témoin de test nommé `T` pour enregistrer ce résultat. Alors, `T` est défini comme suit :

$$\texttt{T}(S, L, \vec{x}) \quad \stackrel{\text{def}}{=} \quad (f(\vec{x}) \rightsquigarrow S) \wedge (f(\vec{x}) \rightsquigarrow L).$$

  - **(`FT`) :** L'exécution $f(\vec{x})$ de l'activité de test ne satisfait pas la spécification $S = H \Rightarrow C$ et passe par un label L. Nous utilisons le témoin d'échec de test nommé `FT` pour enregistrer ce résultat. `FT` est défini comme suit :

$$\texttt{FT}(S, L, \vec{x}) \quad \stackrel{\text{def}}{=} \quad (f(\vec{x}) \not\rightsquigarrow S) \wedge (f(\vec{x}) \rightsquigarrow L).$$

  - **Sans témoin :** L'exécution $f(\vec{x})$ de l'activité de test ne passe pas l'hypothèse H de la spécification $S = H \Rightarrow C$. Comme nous ne considérons pas ce cas comme un test de S, nous n'utilisons aucun témoin pour S dans ce cas.
- **Témoins de Preuve :**
  - **(`P`) :** La spécification S de la fonction f est prouvée. Cela assure ainsi qu'aucune contradiction est trouvée pour cette spécification. Nous utilisons le témoin de preuve nommé `P` pour enregistrer ce résultat. `P` est défini comme suit :

$$\texttt{P}(S) \quad \stackrel{\text{def}}{=} \quad f \models S$$

  - **(`FT`) :** Un contre-exemple, l'exécution $f(\vec{x})$, est trouvée pour la spécification S. Ce résultat est formellement identique au résultat enregistré par le témoin d'échec de test. Afin de définir le moins de témoin possible, nous réutilisons le témoin d'échec de test `FT`.
  - **Sans témoin :** La tentative de preuve ne donne aucun résultat définitif pour la spécification S.

– **Témoin d'inatteignabilité :**
  – **(U) :** Le label L est démontré inatteignable par n'importe quelle exécution de la fonction f. Nous utilisons le témoin d'inatteignabilité nommé `U` pour enregistrer ce résultat. `U` est défini comme suit :

$$\texttt{U}(L) \quad \stackrel{\text{def}}{=} \quad \forall \vec{x}, (f(\vec{x}) \not\rightsquigarrow L).$$

Nous avons donc maintenant défini 5 témoins différents, nommé `T`, `FT`, `P`, `FP` et `U`.

Retournons à l'exemple de la fonction `range` (cf 1.1) avec trois spécifications :

$$\begin{aligned} S_0 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{a} < \texttt{v} < \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{v}) \\ S_1 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{v} \leq \texttt{a} \leq \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{a}) \\ S_2 &\stackrel{\text{def}}{=} (\texttt{Pre}, \texttt{v} < \texttt{u} < \texttt{a}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{u}) \end{aligned}$$

Imaginons qu'il existe `range'` une implémentation alternative et fausse (i.e : un mutant) de la fonction `range`, tel que dans `range'`, S_0 et S_1 sont satisfaites tandis que S_2 n'est pas satisfaite. Nous créons une activité de test et une activité de preuve pour `range'`.

L'activité de test contient trois cas de test. Chaque cas de test vérifie une des spécifications de la fonction `range'`. Le premier cas de test utilise le vecteur d'entrée $\vec{x_1}$ qui satisfait S_0 et passe par un label L_1. Nous avons donc un témoin $\texttt{T}(S_0, L_1, \vec{x_1})$. Le deuxième cas de test utilise le vecteur d'entrée $\vec{x_2}$ qui satisfait S_1 et passe par deux labels L_2 et L_2'. Nous avons donc deux témoins $\texttt{T}(S_1, L_2, \vec{x_2})$ et $\texttt{T}(S_1, L_2', \vec{x_2})$. Le dernier cas de test utilise le vecteur d'entrée $\vec{x_3}$ qui ne satisfait pas S_2 et passe par le label L_1. Nous avons donc un témoin $\texttt{FT}(S_2, L_1, \vec{x_3})$.

Notre activité de preuve réussit à prouver S_0. Nous obtenons donc le témoin $\texttt{P}(S_0)$. Néanmoins, cette activité de preuve est trop faible et ne donne aucun résultat définitif pour S_1. Autrement dit, nous n'obtenons aucun témoin pour S_1. En outre, notre activité trouve un contre-exemple, une exécution $\texttt{range}'(\vec{x})$, pour S_2. Cette exécution passe par le label L. Nous avons donc un témoin $\texttt{FT}(S_2, L, \vec{x})$.

2.2 Mutant de code et de spécification

Jusqu'ici, nous n'avons considéré que des mutations du code mais il existe deux éléments du programme qui peuvent être modifiés : la fonction (objet de la vérification) et la spécification (objectif de la vérification). Nous appelons la nouvelle fonction (resp. spécification), un mutant de la fonction source (resp. de la spécification source).

Les travaux existants n'utilisent qu'un seul type de mutant : soit des mutants de fonctions source [DLS78 ; Bud+80] soit des mutants de spécifications [BG85 ; MB11]. Ces travaux aident à évaluer la qualité d'une vérification soit par rapport au code soit par rapport à la fonction, en fonction du mutant choisi. Néanmoins, aucune étude ne combine l'utilisation des deux types de mutant. Dans notre cas, au contraire, nous proposons d'utiliser les deux en même

temps pour définir notre nouvelle notion de couverture de vérification.

Cette section présente deux concepts de mutants : mutant de code (appelé mutant étiqueté) et mutant de spécification (appelé opposition). Un mutant étiqueté est un mutant de la fonction source qu'on adjoint à un label. L'opposition est obtenue en appliquant un opérateur de mutation spécifique à une spécification. Combiner les deux nous permettra ensuite de définir une couverture claire pour les activités de vérification qui peut être appliquée de manière uniforme pour chaque méthode de vérification utilisée.

2.2.1 Mutant étiqueté : mutant du code renforcé par un label

Le chapitre 1 a introduit la notion de label pour étiqueter chaque fragment du code en fonction du critère de couverture choisi. Nous renforçons à présent un label (l, h) quelconque en lui ajoutant un mutant M. Ce mutant appelé *mutant étiqueté* est obtenu en modifiant le fragment étiqueté par le label (l, h). Les définitions introduites jusqu'alors sont étendues au label renforcé (l, h, M). Par exemple, la définition de « passer par » s'appliquant à (l, h) devient :

$$f(\overrightarrow{x}) \rightsquigarrow (l, h, M) \quad \stackrel{\text{def}}{=} \quad \exists m \in h \text{ tel que } (l, m) \in f(\overrightarrow{x})$$

Un mutant étiqueté *ne modifie que les exécutions qui passent par son label*. Plus précisément, considérons f une fonction source, (l, h, M) un label de f et $\overrightarrow{x}$ un vecteur d'entrée de f. Seuls les états mémoire des états de programme de $f(\overrightarrow{x})$ qui se trouvent après l'emplacement du label sont modifiés par rapport à l'exécution d'origine. Nous présentons à présent deux opérateurs de mutation intéressants qui respectent cette contrainte.

Supprimer une instruction : cet opérateur de mutation efface totalement une instruction. Il fonctionne seulement avec les instructions opérationnelles. Considérons une instruction `St` à l'emplacement l et un label renforcé (l, h, f'). Pour n'importe quel couple (l, m) à l'emplacement l, on a ainsi :

$$\text{Dans la fonction } f : \quad (l, m) \xrightarrow{\texttt{St}} (l', m')$$

$$\begin{aligned} \text{Dans le mutant } f' : \quad & (l, m) \xrightarrow{h+} (l', m) \\ & (l, m) \xrightarrow{h-} (l'', m) \xrightarrow{\texttt{St}} (l', m') \end{aligned}$$

Informellement cette transformation implique que les exécutions du mutant qui passent par le label (l, h, f') suivent la branche $h+$ où `St` ne s'exécute jamais. Inversement, les exécutions qui ne passent pas par le label suivent la branche $h-$ où `St` est exécutée comme dans la fonction d'origine.

Ainsi, la figure 2.1 montre l'utilisation de l'opérateur de mutation « supprimer une instruction » sur un exemple. Dans cette figure, la partie gauche montre un extrait du code issu de l'exemple de la figure 1.3. Cet extrait contient trois labels. À droite, nous pouvons trouver le mutant résultant correspondant au label $(\texttt{l3}, \texttt{v} > 0)$.

```
...
 //@ label (l3,v > 0);
 //@ label (l3,v ≡ 0);
 //@ label (l3,v < 0);
 l3: res = v;
 l4:
...
```

Code original

```
...
 // mutant de (l3,v > 0);
 l3: if (!(v > 0)) {ln: res = v;}
 l4:
...
```

Code après mutation

FIGURE 2.1 – Mutation : supprimer l'instruction à l'emplacement `l3`

Comparons le code source et son mutant, nous pouvons facilement voir que les exécutions qui passent par ce label sautent directement à la position `l4` tandis que les exécutions qui ne passent pas par ce label doivent passer par un emplacement `ln` ajoutant l'instruction `res = a;`.

Modifier une expression : cet opérateur de mutation modifie l'expression d'une instruction donnée. L'expression que cet opérateur modifie diffère suivant le type de l'instruction. Ici, notre opérateur considère seulement trois instructions : les instructions opérationnelles qui modifient les valeurs des variables « $\texttt{x} = \texttt{e};$ », les instructions opérationnelles qui retournent la valeur « $\texttt{return}\,\texttt{e};$ » et les instructions conditionnelles « $\texttt{if}(\texttt{e})\,\{\cdots\}\,\texttt{else}\,\{\cdots\}$ ». Dans chaque cas, l'expression « `e` » représente celle qui est modifiée par notre opérateur.

Considérons une instruction `St` à l'emplacement l et un état de programme (l, m) associé. L'instruction $\texttt{St}'$ obtenue en modifiant l'expression de `St` à un label (l, h, M), est définie comme suit :

$$\text{Dans le code source } P : \quad (l,m) \xrightarrow{\texttt{St}} (l',m')$$

$$\text{Dans le mutant } M : \quad (l,m) \xrightarrow{\texttt{h+}} (l'',m) \xrightarrow{\texttt{St}'} (l',m'')$$
$$(l,m) \xrightarrow{\texttt{h-}} (l''',m) \xrightarrow{\texttt{St}} (l',m')$$

Informellement, cette transformation implique que, dans ce mutant, si l'exécution passe par le label (l, h, M), l'instruction suivante à exécuter est $\texttt{St}'$ et non `St`. Si l'exécution ne passe pas par le label, alors l'instruction suivante à exécuter est toujours `St`.

La figure 2.2 présente un exemple d'utilisation de l'opérateur "modifier une expression". Comme appliquer la définition formelle sur cet exemple produirait un code occupant trop de place, nous avons choisi de présenter seulement une version simplifiée mais équivalente.

```
...
 //@ label (l3,v > 0);
 //@ label (l3,v ≡ 0);
 //@ label (l3,v < 0);
 l3: res = v;
...
```

Code original

```
...
 // mutant de label (l3,v > 0);
 l3: res = (v > 0)?(v+1):(v);
...
```

Code après mutation

FIGURE 2.2 – Mutation : ajoutant +1 à l'expression `v` à `l3`

La mutation est bien appliquée dans cette situation. Considérons une exécution quelconque. Si elle passe par ce label, la variable `res` prend pour valeur $\mathtt{v}+1$. Sinon, la nouvelle valeur de la variable `res` est `v` (comme dans la fonction d'origine).

En le combinant avec un label, nous pouvons expliciter différents critères de couverture structurelle. La figure 2.2 montre ainsi une couverture des instructions spécifiques en exigeant que l'instruction au label `l3` soit vérifiée avec trois intervalles de valeurs pour `v` : $(\mathtt{v} > 0)$, $(\mathtt{v} \equiv 0)$ et $(\mathtt{v} < 0)$.

Maintenant, considérons la couverture de branche. Elle impose que l'instruction conditionnelle « `if`$(\mathtt{a} < \mathtt{v})$ » soit vérifiée avec trois labels : $(\mathtt{a} < \mathtt{v})$, $(\mathtt{a} \equiv \mathtt{v})$ et $(\mathtt{a} > \mathtt{v})$. Dans ce cas, la meilleure modification est le remplacement de l'expression booléenne $(\mathtt{a} < \mathtt{v})$ par son opposition $(!(\mathtt{a} < \mathtt{v}))$. La figure 2.3 montre un exemple d'un tel mutant.

```
...
 //@ label (l2,a < v);
 //@ label (l2,a ≡ v);
 //@ label (l2,a > v);
 l2: if(a < v)
 { ... };
...
```

Code original

```
...
 // mutant au label (l2,a ≡ v);
 l2: if((a ≡ v)?(!(a < v)):(a < v))
 { ... };
...
```

Code après mutation

FIGURE 2.3 – Mutation : remplacer l'expression $(\mathtt{a} < \mathtt{v})$ par sa négation $!(\mathtt{a} < \mathtt{v})$

2.2.2 Opposition : mutant de spécification

En plus de muter le code, nous pouvons aussi muter la spécification via différents opérateurs [BOY00]. Néanmoins ici, nous n'utilisons que la mutation opérant sur la conclusion de la spécification. Le mutant ainsi créé est appelé l'*opposition* de la spécification d'origine.

Considérons une spécification S d'hypothèse H et de conclusion C (i.e. $S \stackrel{\text{def}}{=} H \Rightarrow C$). L'opposition $S' = \mathtt{Opp}(S)$ de S est obtenue en ajoutant le symbole de négation $\neg$ à la conclusion C :

$$\mathtt{Opp}(S) \quad \stackrel{\text{def}}{=} \quad H \Rightarrow \neg C$$

Quand nous mutons une spécification en utilisant l'opérateur de mutation ci-dessus, une autre spécification (opposition) est créée. Il est possible de vérifier cette nouvelle spécification par d'autres actions de vérification. Grâce à l'utilisation d'outils de vérification automatique, l'ajout des actions de vérification pour les oppositions à la campagne de vérification ne pose aucun problème de passage à l'échelle. Pour les activités de test, initialement, nous utilisons les oracles qui évaluent automatiquement les cas de test. L'ajout des oppositions exige seulement l'application de ces oracles pour évaluer les cas de test existants pour les nouvelles spécifications. Par exemple, si nous utilisons le greffon E-ACSL [SKV17] de Frama-C [Kir+15] pour évaluer les cas de test pour les spécifications originales, nous pouvons facilement et automatiquement réutiliser E-ACSL pour évaluer ces cas de test pour les oppositions de ces spécifications. Pour les activités de preuve, il suffit d'appliquer les outils de preuve sur les nouvelles spécifications. Par exemple, après avoir défini l'opposition des spécifications, nous utilisons le greffon WP [Bau+b] de Frama-C pour vérifier les spécifications originales et leur opposition.

De plus, la vérification de l'opposition donne plusieurs informations sur les spécifications et les outils de vérification. Ainsi, l'opposition d'une spécification est la complétude de cette spécification. En effet, si l'une est satisfaite alors l'autre doit nécessairement être insatisfaite. Par contre, que se passe-t-il si une spécification et son opposition sont toutes les deux satisfaites par une activité de vérification ?

Considérons une spécification $S = H \Rightarrow C$ d'une fonction f et son opposition $\texttt{Opp}(S) = H \Rightarrow \neg C$. Nous définissons une activité de preuve et une activité de test qui les vérifient.

Imaginons que l'activité de preuve réussit à prouver cette spécification et son opposition, c'est-à-dire que ces activités produisent deux témoins :

$$\begin{aligned} \texttt{P}(S) &\stackrel{\text{def}}{=} \forall \overrightarrow{x},\ \text{soit } (f(\overrightarrow{x}) \rightsquigarrow H \wedge C) \text{ soit } (f(\overrightarrow{x}) \not\rightsquigarrow H). \\ \texttt{P}(\texttt{Opp}(S)) &\stackrel{\text{def}}{=} \forall \overrightarrow{x},\ \text{soit } (f(\overrightarrow{x}) \rightsquigarrow H \wedge \neg C) \text{ soit } (f(\overrightarrow{x}) \not\rightsquigarrow H). \end{aligned}$$

Selon les formules ci-dessus, l'essai de preuve réussit à prouver la spécification et son opposition si et seulement si l'hypothèse H est toujours fausse. Si l'hypothèse H n'est pas fausse mais que l'essai de preuve réussit toujours, alors l'outil de preuve est nécessairement incorrect.

La vérification de la spécification et de son opposition par une activité de test nous permet aussi d'évaluer la validité de l'outil de test. Selon notre formalisme, une spécification $S = H \Rightarrow C$ est satisfaite par $f(\overrightarrow{x})$ si et seulement si $(f(\overrightarrow{x}) \rightsquigarrow H \wedge C)$. D'après cette formulation, si $f(\overrightarrow{x})$ satisfait à la fois S et $\texttt{Opp}(S)$ alors la formule suivante serait satisfaite :

$$(f(\overrightarrow{x}) \rightsquigarrow H \wedge C) \wedge (f(\overrightarrow{x}) \rightsquigarrow H \wedge \neg C)$$

Cette formule est incohérente et donc, l'outil qui conduit à cette situation est défectueux.

Le mutant du code et de la spécification nous aide à valoriser la vérification et nous donne l'information de couverture. Grâce aux outils automatiques de test et de preuve, la vérification de mutant ne représente que très peu d'effort supplémentaire. En utilisant ces mutants, nous approchons d'une couverture qui fonctionne avec test et preuve.

2.3 Extension de témoin aux mutants

Utiliser les mutants de code et de spécifications nécessite d'ajouter des actions supplémentaires à une campagne de vérification. Ainsi, au total, quatre éléments peuvent étendre la nature des témoins :

- la méthode de vérification (test ou preuve) ;
- le résultat de la vérification (succès ou échec) ;
- le mutant de code utilisé ;
- le mutant de spécification (opposition) utilisé.

Chaque élément multiplie le nombre de type de témoin par deux. Il y a donc au total $2^4 = 16$ types de témoin. Heureusement, il est possible de réduire ce nombre. En effet, nous pouvons nous rendre compte par exemple que la vérification de la spécification et celle de son opposition est un couple dont le résultat "succès" de l'une est identique au résultat "échec" de l'autre.

Par exemple, considérons une spécification $S = H \Rightarrow C$ de la fonction f :

- Si l'opposition de S ($H \Rightarrow \neg C$) est prouvée alors aucune exécution de f ne viole cette opposition. Dans ce cas, il est normal que S ne soit pas prouvée (`FP`) ou qu'il existe un contre-exemple pour S (`FT`).
- Si nous trouvons un test qui peut valider l'opposition de S, alors on est sûr que S est invalide lorsque nous la testons par la même exécution. Ainsi, le témoin `FT`, ce qui représente l'invalidité de S, est aussi inutile.

Donc, les témoins qui montrent l'échec d'une vérification ne sont pas intéressants et sont enlevés de notre thèse. Par conséquent, seuls 9 types de témoin sont conservés.

Parmi ceux-ci, ceux montrant la validation des spécifications par test et par preuve (`P` et `T`) et l'inatteignabilité d'un label (`U`) ont déjà été introduits dans la section 2.1. Ils sont les enregistrements des résultats provenant de la vérification d'une spécification par le code original. Les 6 autres témoins sont nommés à partir de deux témoins `P` et `T` ci-dessus en ajoutant :

- (`M`) si le témoin enregistre le résultat de la vérification du mutant,
- (`O`) si le témoin enregistre le résultat de la vérification de l'opposition.

Convention : si nous avons les deux caractères `O` et `M` dans le nom du témoin, `O` est placé avant `M`

Ces 6 autres types de témoins sont présentés ci-après :

- *Témoin* `OP` *:* témoin de preuve de l'opposition dans le code initial, lorsque l'opposition de la spécification S ($\texttt{Opp}(S) = H \Rightarrow \neg C$) est *prouvée* dans le code initial :

$$\texttt{OP}(S) \quad \stackrel{\text{def}}{=} \quad f \models \texttt{Opp}(S)$$

– *Témoin* `OT` *:* témoin de test de l'opposition dans le code initial, lorsque l'opposition de la spécification S est *testée* par l'exécution $f(\overrightarrow{x})$ du code source et que cette exécution passe aussi par le label L :

$$\texttt{OT}(S, L, \overrightarrow{x}) \quad \stackrel{\text{def}}{=} \quad (f(\overrightarrow{x}) \rightsquigarrow \texttt{Opp}(S)) \wedge (f(\overrightarrow{x}) \rightsquigarrow L)$$

– *Témoin* `MP` *:* témoin de preuve de la spécification dans le mutant étiqueté, lorsque la spécification S est *prouvée* dans le mutant étiqueté f' du label $L = (l, h, f')$:

$$\texttt{MP}(S, L) \quad \stackrel{\text{def}}{=} \quad f' \models S$$

– *Témoin* `MT` *:* témoin de test de la spécification dans le mutant étiqueté, lorsque la spécification S est *testée* dans le mutant étiqueté f' du label $L = (l, h, f')$ pendant l'exécution $M(\overrightarrow{x})$ passant par L :

$$\texttt{MT}(S, L, \overrightarrow{x}) \quad \stackrel{\text{def}}{=} \quad (f'(\overrightarrow{x}) \rightsquigarrow S) \wedge (f'(\overrightarrow{x}) \rightsquigarrow L)$$

– *Témoin* `OMP` *:* témoin de preuve de l'opposition dans le mutant étiqueté, lorsque l'opposition de la spécification S est *prouvée* dans le mutant étiqueté f' du label $L = (l, h, f')$:

$$\texttt{OMP}(S, L) \quad \stackrel{\text{def}}{=} \quad f' \models \texttt{Opp}(S)$$

– *Témoin* `OMT` *:* témoin de test de l'opposition dans le mutant étiqueté, lorsque l'opposition de la spécification S est *testée* par une exécution $f'(\overrightarrow{x})$ du mutant f' qui passe par le label $L = (l, h, f')$:

$$\texttt{OMT}(S, L, \overrightarrow{x}) \quad \stackrel{\text{def}}{=} \quad (f'(\overrightarrow{x}) \rightsquigarrow \texttt{Opp}(S)) \wedge (f'(\overrightarrow{x}) \rightsquigarrow L)$$

Les témoins permettent de créer des enregistrements formels pour toutes les actions de la campagne de vérification.

2.4 Priorité des témoins

Les sections précédentes ont introduit 9 types de témoin différents répartis entre témoins de preuve (`P`, `OP`, `MP`, `OMP`) et témoins de test (`T`, `OT`, `MT`, `OMT`). Les témoins `FP` et `FT` sont écartés car il y a équivalence avec `OP` et `OT`. Dans une même campagne de vérification, pour une même spécification, il peut y avoir plusieurs témoins. Néanmoins, afin de faciliter la détermination de la couverture, nous allons ordonner les témoins, ce qui permettra de n'en retenir qu'au plus deux pour chaque relation spécification-fragment. L'un portera l'information concernant la vérification tandis que l'autre portera celle de la couverture. Par conséquent, nous devons définir un ordre pour décider quels témoins garder et quels témoins supprimer.

Cette priorisation est déterminée en comparant et en valorisant les témoins par rapport aux trois éléments suivants :

– le mutant étiqueté ;
– la méthode de vérification (test ou preuve) ;
– le mutant d'opposition.

Ce sont ces éléments qui affectent le choix des témoins. Chacun divise l'ensemble des témoins en des couples différents, ce qui est détaillé dans les trois sous-sections suivantes.

2.4.1 Priorité par rapport à la mutation

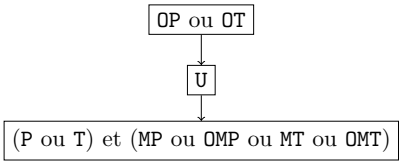
L'objectif principal de toutes les campagnes de vérification est de trouver des erreurs. Ensuite, quand aucune erreur n'est trouvée, ces campagnes visent à garantir que tout le code et toutes les spécifications sont bien couverts via les critères de couverture.

Pendant une campagne de vérification, obtenir des témoins qui annoncent l'existence d'une erreur est suffisant pour terminer la campagne de vérification. Comme présenté précédemment, la vérification de la spécification et celle de son opposition est un couple dont le résultat « succès » de l'une est identique au résultat « échec » de l'autre. L'existence d'une erreur est équivalente au résultat « échec » d'une spécification et donc, est équivalente au résultat « succès » de son opposition. Ceci est caractérisé par le témoin `OP` et le témoin `OT` qui sont donc les plus prioritaires.

Ensuite, un autre objectif d'une campagne de vérification est de trouver les labels inatteignables. Ainsi, dans la liste de priorité, le prochain témoin à considérer est `U`. En effet, si nous trouvons au plus tôt les labels concernés par ce témoin, nous pouvons réduire le nombre de labels considérés dans les prochaines vérifications.

Enfin, si aucun témoin `OP` ou témoin `OT` n'est produit pendant une campagne de vérification, nous nous intéressons au calcul de couverture. Dans ce cas, la campagne contient seulement les témoins `P` et `T` qui dénotent les spécifications qui sont prouvées (`P`) ou testées (`T`) dans le code initial. Nous devons également vérifier les mutants étiquetés pour déterminer la couverture des activités de la campagne. La vérification des mutants produit un des témoins suivants : `MP`, `MT`, `OMP` et `OMT`.

L'ordre de priorité partiel est ainsi déterminé par ce schéma :



Ce schéma indique par exemple que, pour une campagne de vérification donnée, un témoin `OP` ou `OT` est plus important que la combinaison d'un témoin de satisfaction de spécification (`P` ou `T`) et d'un témoin venu de la vérification du mutant du code (`MP`, `OMP`, `MT` ou `OMT`).

```
int range(int a, int u, int v){
 int res = v;
 if(a  <  v){ res = v;}
 if(a  >  u){ res = u;}
 return res;
}
```

FIGURE 2.4 – Une version alternative et incorrecte de la fonction `range`

Par exemple, considérons une version alternative et incorrecte de la fonction `range` (figure 2.4) et la spécification $S_2 = (\texttt{Pre}, \texttt{v} \leq \texttt{a} \leq \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, result \equiv \texttt{a})$. Nous testons S_2 avec cette version incorrecte et deux exécutions $\texttt{range}(4, 10, 3)$ et $\texttt{range}(3, 10, 3)$. Le cas de test utilisant l'exécution $\texttt{range}(4, 10, 3)$ produit un témoin `OT` tandis que celui utilisant l'exécution $\texttt{range}(3, 10, 3)$ crée un témoin `T`. Dans ce cas, puisqu'une erreur est trouvée (témoin `OT`), nous pouvons éliminer le témoin `T`.

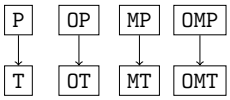
2.4.2 Priorité par rapport à la méthode de vérification

Considérons deux témoins $\texttt{P}(S)$ et $\texttt{T}(S, L, \overrightarrow{x})$. Leur formalisation est la suivante (cf. section 1.4) :

$$\texttt{P}(S) \quad \stackrel{\text{def}}{=} \quad \forall \overrightarrow{x},\ \text{soit}\ (f(\overrightarrow{x}) \rightsquigarrow S)\ \text{soit}\ (f(\overrightarrow{x}) \not\rightsquigarrow H).$$

$$\texttt{T}(S, L, \overrightarrow{x}) \quad \stackrel{\text{def}}{=} \quad (f(\overrightarrow{x}) \rightsquigarrow S) \wedge (f(\overrightarrow{x}) \rightsquigarrow L).$$

D'après ces formules, le témoin de preuve garantit la satisfaction de la spécification par rapport à toutes les exécutions qui passent par son hypothèse, alors que le témoin de test ne garantit cette satisfaction que pour l'exécution de certains cas de test. En conséquence, un témoin de preuve `P` est plus fort qu'un témoin de test `T`. Autrement dit, un témoin de preuve donne plus de confiance qu'un témoin de test. Nous pouvons ainsi en déduire l'ordre de priorité suivant :



Il indique que le témoin de preuve est prioritaire par rapport au témoin de test, pour chaque type de témoin.

Par exemple, considérons la fonction `range` (figure 1.3) et la spécification $S_2 = (\texttt{Pre}, \texttt{v} \leq \texttt{a} \leq \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, result \equiv \texttt{a})$. S_2 peut à la fois être prouvée (témoin `P`) et testée en

utilisant l'exécution $\texttt{range}(8, 10, 3)$ (témoin `T`). Dans ce cas, `T` n'apporte aucune information particulière parce que `P` couvre ce témoin. Nous ne garderons donc pas `T`.

2.4.3 Priorité par rapport à la relation spécification-opposition

Il est parfois possible d'obtenir deux témoins contradictoires en même temps : un témoin montrant que la spécification S est également satisfaite pour une fonction f et un autre témoin montrant que l'opposition $\texttt{Opp}(S)$ de S est également satisfaite pour f. Cette situation peut survenir dans deux cas différents : les témoins de preuve et les témoins de test.

Les témoins contradictoires sont des témoins de preuves (soit `P` et `OP`, soit `MP` et `OMP`). Nous isolons et considérons chaque couple individuellement.

`P` et `OP` : Cette situation correspond formellement au cas suivant :

$$\begin{aligned}
&\texttt{P}(S) \wedge \texttt{OP}(S) \stackrel{\text{def}}{=} \\
&\quad \forall \overrightarrow{x},\ \text{soit } (f(\overrightarrow{x}) \rightsquigarrow H \wedge C) \text{ soit } (f(\overrightarrow{x}) \not\rightsquigarrow H) \\
&\quad \wedge \forall \overrightarrow{x},\ \text{soit } (f(\overrightarrow{x}) \rightsquigarrow H \wedge \neg C) \text{ soit } (f(\overrightarrow{x}) \not\rightsquigarrow H) \\
&\quad \equiv \forall \overrightarrow{x}, (f(\overrightarrow{x}) \not\rightsquigarrow H)
\end{aligned}$$

Ainsi, la formule $\texttt{P}(S) \wedge \texttt{OP}(S, L)$ est équivalente à $\forall \overrightarrow{x}, (f(\overrightarrow{x}) \rightsquigarrow \neg H)$. Cette situation ne peut donc survenir que lorsque aucune exécution ne satisfait l'hypothèse H de S , c'est-à-dire que cette spécification est inutile. Dans ce cas, nous choisissons de garder le témoin `OP` de manière à signaler ce problème le plus tôt possible. Le témoin `P` est, quant à lui, supprimé.

Par exemple, considérons la spécification $S = (l, false) \Rightarrow C$. Cette spécification est toujours prouvée parce que son hypothèse est toujours fausse. L'opposition de S est toujours prouvée aussi. En conséquence, la preuve de S n'apporte rien. Nous gardons dans cette situation le témoin `OP` pour signaler l'inutilité de cette spécification le plus tôt possible.

`MP` et `OMP` : Nous pouvons avoir ces deux témoins pour la même raison que ci-dessus : mais dans l'un mutant et non dans le code initial. Par contre, le problème dans ce cas vient du mutant. Dans cette situation, nous pouvons déduire deux faits :

- l'emplacement de la mutation se trouve avant l'emplacement de l'hypothèse.
- la mutation est la cause du changement dans la satisfaction de cette hypothèse.

Nous savons donc que la mutation change la décision de satisfaction de la spécification mais que la partie mutée et la spécification sont indépendantes. Dans ce cas là, $\texttt{OMP}(S, L)$ (qui est nécessaire pour montrer que S et L sont fortement connectés) ne rend pas bien compte de la situation. Par conséquent, nous gardons seulement le témoins `MP`.

Considérons la fonction `main` (figure 2.5) qui exécute trois fois la fonction `range`. Certains

```
int main(){
 //label (l,true,M)
 range (8,10,3);
 range (6,15,10);
 range (3,2,1);
 return 0;
Post:}
```

FIGURE 2.5 – Exécution de fonction `range`

outils de preuve (par exemple, ceux fondés sur des techniques d'interprétation abstraite) prouvent les spécifications en utilisant les exécutions de cette fonction. Les spécifications de la fonction `range` sont prouvées dans le code source. Maintenant, la mutation transforme « `range`$(8, 10, 3)$; » en « `range`$(2, 10, 3)$; » dans la fonction `main` (mutant `M`). $S_1 = (\texttt{Pre}, \texttt{v} < \texttt{a} < \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{a})$ est prouvée. Son opposition est aussi prouvée. Autrement dit, dans cette situation, nous avons `MP` et `OMP` en même temps. Nous ne voulons pas accepter cette situation comme une couverture entre un label et une spécification. Alors, nous gardons uniquement le témoin `MP`.

Les témoins sont des témoins de tests (soit `T` et `OT`, soit `MT` et `OMT`). Nous isolons et considérons individuellement chaque couple.

`T` et `OT` : cette situation correspond formellement au cas suivant :

$$\begin{aligned}\texttt{T}(S, L, \overrightarrow{x}) \wedge \texttt{OT}(S, L, \overrightarrow{y}) \stackrel{\text{def}}{=} \\ (f(\overrightarrow{x}) \rightsquigarrow S) \wedge (f(\overrightarrow{x}) \rightsquigarrow L) \\ \wedge (f(\overrightarrow{y}) \rightsquigarrow \texttt{Opp}(S)) \wedge (f(\overrightarrow{y}) \rightsquigarrow L)\end{aligned}$$

Cette formule est valide si et seulement si $\overrightarrow{x}$ et $\overrightarrow{y}$ sont différents, c'est-à-dire que `T` et `OT` sont obtenus par deux cas de test distincts. Le test vise toujours à trouver une contradiction de la spécification par rapport au code. Par conséquent, le témoin qui trouve l'erreur (`OT`) est prioritaire sur le témoin qui ne trouve rien (`T`). Donc, nous gardons le témoin `OT`.

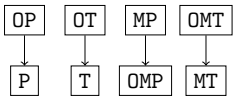
Par exemple, considérons de nouveau la version incorrecte de la fonction `range` (figure 2.4) et la spécification $S_2 = (\texttt{Pre}, \texttt{v} \leq \texttt{a} \leq \texttt{u}) \Rightarrow (\texttt{Pre}, \texttt{Post}, \mathit{result} \equiv \texttt{a})$. Comme présenté précédemment, nous pouvons tester cette fonction par 2 exécutions `range`$(3, 8, 3)$ et `range`$(4, 8, 3)$. L'une produit le témoin `T` tandis que l'autre produit le témoin `OT`. Puisque l'erreur est essentielle dans la vérification, nous gardons `OT` (même conclusion avec la première priorité).

`MT` et `OMT` : Comme pour le cas `T` et `OT`, nous pouvons avoir deux témoins `MT` et `OMT` pour une même spécification avec deux cas de test différents. Selon la vérification du mutant, le cas de test qui peut tuer le mutant est plus intéressant que celui qui ne peut pas. Dans ce cas

là, $\texttt{OMT}(S, L, \vec{x})$ est capable de tuer le mutant. Autrement dit, $\texttt{OMT}(S, L, \vec{x})$ est plus fort que $\texttt{MT}(S, L, \vec{y})$. Par conséquent, nous gardons seulement le témoin `OMT`.

Nous n'introduisons pas d'exemple spécifique dans ce cas. Lorsque la version incorrecte de la fonction `range` (figure 2.4) est considérée comme un mutant étiqueté de la fonction `range`, le test de cette fonction produit les témoins `MT` et `OMT` (au lieu de `T` et `OT`). Nous pouvons en déduire la même conclusion que dans le cas précédent.

Nous résumons toutes les discussions ci-dessus par le schéma suivant :



2.4.4 Cas particulier

Cette sous-section discute un cas particulier qui se trouve en dehors de notre formulation. Ce cas a déjà été évoqué précédemment à la fin de la section 2.2.

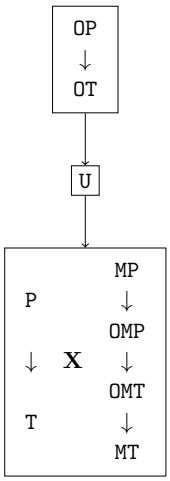
Plusieurs outils de test ne suivent pas notre formalisme. En effet, dans notre contexte, une spécification $S = H \Rightarrow C$ est satisfaite par l'exécution $f(\vec{x})$ si et seulement si $f(\vec{x}) \rightsquigarrow H \wedge C$. Par contre, pour certains outils de test, S est satisfaite par l'exécution $f(\vec{x})$ si $f(\vec{x}) \rightsquigarrow H \wedge C$ ou $f(\vec{x}) \not\rightsquigarrow H$. Leur critère de satisfaction est donc plus faible que le nôtre.

À cause de ce problème, nous pouvons obtenir des témoins `T` et `OT` avec le même vecteur d'entrée $\vec{x}$ et la même spécification S. La même situation est possible pour `MT` et `OMT`. Dans notre contexte, les exécutions $f(\vec{x})$ qui ne passent pas par l'hypothèse de la spécification S ne sont pas capables de tester S. C'est pourquoi, si un outil de test crée `T` et `OT`, resp. `MT` et `OMT`, avec le même vecteur d'entrée $\vec{x}$ et la même spécification S, nous abandonnons ces témoins. Néanmoins, nous acceptons encore les autres témoins produits par un tel outil.

2.4.5 Synthèse

Nous avons trois règles de priorité sous forme de schémas. Cette partie vise à fusionner ces règles.

Avant d'appliquer les trois règles, nous devons prendre en compte attentivement le cas particulier évoqué en section 2.4.4 pour éviter les cas en dehors de notre formalisme. Ensuite, nous appliquons les trois ordres de priorité. Pour éviter l'utilisation non-organisée des règles de priorité, nous combinons les trois schémas en un seul. La version complète des règles de priorité est :



D'après cette formulation, les témoins qui exhibent une erreur sont plus prioritaires que les autres. Parmi les témoins d'erreur, `OP` est plus prioritaire que `OT`. Si la campagne de vérification ne produit aucun témoin d'erreur, nous identifions les témoins `U` pour trouver les témoins inatteignables et nous nous intéressons ensuite à la combinaison d'un témoin de vérification (`P` ou `T`) et d'un témoin de couverture (`MP`, `OMP`, `OMT` ou `MT`). Selon le schéma, un témoin de preuve `P` est meilleur qu'un témoin de test `T`. Dans le cas des témoins de couverture, nous devons suivre attentivement le schéma $\texttt{MP} \succ \texttt{OMP} \succ \texttt{OMT} \succ \texttt{MT}$ pour garder les témoins les plus utiles. $\succ$ signifie qu'un témoin précède un autre témoin.

2.5 Matrice de couverture

Pour synthétiser les résultats de la campagne de vérification, nous utilisons une structure de tableau appelée matrice de couverture. Nous proposons aussi une version consolidée de cette matrice permettant de mieux identifier les résultats obtenus.

2.5.1 Structure d'une matrice de couverture

Une matrice de couverture permet à un ingénieur de consulter l'avancement d'une campagne de vérification. Chaque colonne de cette matrice représente une spécification, tandis que chaque ligne représente un label. Une matrice enregistre les résultats d'une campagne de vérification (éventuellement toujours en cours). La table 2.1 présente les marques qui peuvent être stockées dans les cellules de la matrice. Ces marques correspondent aux différentes combinaisons des témoins obtenus de la vérification d'une spécification S d'un programme source f associé au mutant f' du label L. Ces combinaisons de témoins possibles sont présentées dans la table 2.2.

La marque notée dans une cellule de la matrice encode une synthèse des informations de vérification *et* de couverture obtenus pour une spécification S et un label L. La table 2.1 donne

	Specification	Qualification
Label	(empty)	Aucune vérification n'est réalisée
	KO	Une erreur est trouvée
	U	Le label est inatteignable
	?	Aucune information de vérification n'est disponible
	$P?$ $T?$	Aucune information de couverture est disponible
	P✗ T✗	La spécification est vérifié mais n'est pas corrélé au label
	P✓ T✓	La spécification est vérifié et couverte par le label

TABLE 2.1 – Marques possibles dans les cellules d'une matrice de couverture.

Spé. à f	Spé. à f'	Témoin	Vérification	Couverture
aucun témoin	aucun témoin		(empty)	
opp. prouvée	n'importe quel	$\texttt{OP}(S)$	KO	
opp. testée	témoin	$\texttt{OT}(S, L, \overrightarrow{x})$		
label inattei.	aucun témoin	$\texttt{U}(L)$	U	
aucun témoin	prouvée	$\texttt{MP}(S, L)$	?	
	opp. prouvé	$\texttt{OMP}(S, L)$		
prouvée	aucun témoin	$\texttt{P}(S)$	P	?
	testée	$\texttt{P}(S) \wedge (\exists \overrightarrow{x}.\texttt{MT}(S, L, \overrightarrow{x}))$		
testée	testée	$\exists \overrightarrow{x}.\ [\ \texttt{T}(S, L, \overrightarrow{x}) \wedge \texttt{MT}(S, L, \overrightarrow{x})\ ]$	T	
prouvée	prouvée	$\texttt{P}(S) \wedge \texttt{MP}(S, L)$	P	✗
testée	prouvée	$(\exists \overrightarrow{x}.\texttt{T}(S, L, \overrightarrow{x})) \wedge \texttt{MP}(S, L)$	T	
prouvée	opp. prouvée	$\texttt{P}(S) \wedge \texttt{OMP}(S, L)$	P	✓
	opp. testée	$\texttt{P}(S) \wedge (\exists \overrightarrow{x}.\texttt{OMT}(S, L, \overrightarrow{x}))$		
testée	opp. prouvée	$(\exists \overrightarrow{x}.\texttt{T}(S, L, \overrightarrow{x})) \wedge \texttt{OMP}(S, L)$	T	
	opp. testée	$\exists \overrightarrow{x}.[\ \texttt{T}(S, L, \overrightarrow{x}) \wedge \texttt{OMT}(S, L, \overrightarrow{x})\ ]$		

TABLE 2.2 – Marques des informations de vérification et de couverture.

une vue synthétique des connections possibles entre labels et spécifications. Une cellule vide mentionne qu'aucune activité de vérification n'a encore été réalisée. La marque KO montre que la spécification S est invalidée (témoin `OP` et `OT` selon la table 2.2) dans le programme original, et aucune information de couverture n'est ici nécessaire. La marque ? indique que seule une information de couverture (`SP` et `OP`) est disponible. La marque U est une marque spécifique. Elle vient du témoin `U` qui montre que le label est inatteignable. Les autres marques sont composées de deux indications. La première indication (T ou P) encode l'information de vérification obtenue par un témoin `T` ou un témoin `P`. La marque T indique que la spécification S est testée tandis que P est obtenue quand S est prouvée. La seconde indication dénote l'information de couverture structurelle obtenue. Ici, la marque ? montre que S est valide mais qu'aucune relation structurelle entre S et L n'a été trouvée. La marque ✗ indique que, même si S est valide, S et L ne sont pas corrélés. Autrement dit, le mutant f' en L n'est pas distinguable de f pour cette spécification. La marque ✓ indique que S est valide dans le code source et invalide dans le mutant étiqueté de label L, ce qui démontre en outre que S et L sont structurellement corrélés.

2.5.2 Aide aux tests manuels

Dans le cadre d'une combinaisons test & preuve dans laquelle toutes les activités automatiques ont été menées, la matrice de couverture nous aide à définir les cas de test qu'il faut ajouter pour obtenir une couverture complète. Plus précisément, pour définir les cas de test d'une spécification S non-vérifiée ou non-couverte, nous nous concentrons sur chaque label L pour lequel la cellule de la matrice à l'intersection entre la ligne de L et la colonne de S est soit vide soit marquée par ?, P? ou T?. En effet, ces labels définissent implicitement des chemins d'exécution qu'il faut exercer par des nouvelles activités de vérification. En revanche, il est inutile de considérer des cas de test qui passeraient par les labels marqués par P✗ ou T✗ parce qu'ils ne sont pas corrélés à la spécification concernée.

	S0	**S1**	**S2**
l1	P✗	P?	P✗
l2	**P✓**	P✗	P✗
l3	P✗	P✗	**P✓**

FIGURE 2.6 – Exemple d'aide à la vérification

Par exemple, reprenons l'exemple de la matrice de la figure 2.6. Cette matrice contient les résultats issus d'une preuve automatique. Dans cette matrice, nous constatons que la spécification `S1` est vérifiée mais nous ne savons pas quel label est associé à cette spécification. Donc, nous devons ajouter un test pour compléter l'information de couverture manquante. Les cellules entre les labels `l2` et `l3` et de la spécification concernée `S1` sont marquées par P✗. Donc, nous ne considérons jamais ces labels pour créer les cas de test. Néanmoins, la cellule

entre le label `l1` et la spécification concernée `S1` est marqué par « $P?$ ». Nous devons donc chercher une exécution qui passe le label `l1` pour définir un nouveau cas de test pour `S1`.

2.5.3 Détection des erreurs de spécification

Notre manière de synthétiser des informations de couverture structurelle peut également aider à détecter des spécifications non-pertinentes. Il s'agit des spécifications dont les cellules ne sont marquées que par une marque $P✗$ ou $T✗$. Par exemple, dans la figure 2.7, les cellules de la spécification `S0` ne sont marquées que par $P✗$. Cela veut dire que cette spécification n'est corrélée à aucune portion de code du programme. Cette spécification contient donc une incohérence, ou bien une hypothèse inatteignable, ou bien une conclusion toujours vraie.

	S0	S1	S2
LQ1	P✗	P✓	P✗
LQ2	P✗	P✗	P✓

FIGURE 2.7 – S0 non-pertinente

2.5.4 Consolidation

Une matrice de couverture est souvent de grande taille, ce qui peut compliquer son exploitation. Par exemple, pour notre exemple (la fonction `transform` introduite à la section 3.3), nous avons 13 spécifications et 13 labels, donc 169 cellules au total pour la matrice de couverture, ce qui la rend difficile à analyser. Pour résoudre ce problème, nous introduisons une méthode qui consolide les informations par ligne et par colonne. Cette méthode ajoute une colonne appelée *consolidation de spécification* et une ligne appelée *consolidation de label*. Pour finalement mesurer le degré de couverture obtenue après une campagne, il suffit alors de ne considérer que la consolidation de spécification et que la consolidation de label. Autrement dit, au lieu d'analyser toute la matrice, nous n'avons qu'à analyser une ligne et une colonne. Le format de ces lignes et colonnes de consolidation est donnée dans la figure 2.8.

Même si les marques utilisées dans les consolidations de spécification et de label sont syntaxiquement similaires (table 2.3 et table 2.4), leur sémantique diffère légèrement de celle des cellules de la matrice.

Pour la consolidation d'une spécification (table 2.3), nous pouvons utiliser une des 5 marques suivantes : P, T, ?, ✗ et KO pour encoder son degré de couverture. La marque P (*resp.* T) dénote que la spécification S est prouvée (*resp.* testée) et que cette spécification est liée à *au moins* un label. La marque ✗ indique qu'aucun label n'est en corrélation avec la spécification S (cf. section 2.5.3). La marque ? est utilisée quand aucun témoin de vérification n'a été trouvé pour S. Il est alors nécessaire de compléter la campagne de vérification par des activités de vérification complémentaires ou une relecture attentive de la spécification. La marque KO montre que la spécification S est invalide. Autrement dit, le code contient une

	Specification
Label	(empty) KO U ? P? T? P✗ T✗ P✓ T✓

consolidation de label	Qualification
✓	Le label est associé à au moins une spécification
U	Le label est inatteignable
✗	Le label n'est associé à aucune spécification
?	Manque d'information
KO	Une erreur est trouvé dans la portion du code

consolidation de spécification		Qualification
	P	La spécification est prouvée et est associée à au moins un label
	T	La spécification est testée et est associée à au moins un label
	?	La spécification n'est pas vérifiée
	✗	La spécification est non-pertinente
	KO	La spécification est invalide

FIGURE 2.8 – Matrice de couverture consolidée.

erreur.

La consolidation de label utilise également des marques pour encoder l'information sur la couverture (table 2.4). La marque ✓ indique que le label L est lié à *au moins* une spécification. La marque ✗ dénote l'indépendance du label L avec *toute* spécification existante. Cette situation est intéressante car elle pointe une partie du code dont la fonctionnalité n'est pas spécifiée. Il faudra alors compléter la spécification de programme (cela ressemble au critère « code étrangé » de la DO-333). La marque U dénote que le label L est inatteignable et donc qu'il est indépendant de toute spécification (existante ou non). Cette situation est aussi intéressante car elle pointe une partie du code qui n'est pas utilisée (cela ressemble aussi au critère « code étrangé » de la DO-333). La marque ? est utilisée dans tous les autres cas pour indiquer l'absence d'information de couverture pertinente. La marque KO montre qu'il existe une erreur dans le code et qu'elle est peut-être liée au label marqué par cette marque.

La couverture label-mutant est dite *complète* lorsque, après consolidation de la matrice de couverture, nous n'obtenons que des marques P ou T comme consolidation de spécification et des marques ✓ comme consolidation de label. La première exigence (uniquement P ou T comme consolidation de spécification) permet de garantir que toutes les spécifications sont suffisamment vérifiées. Cela correspond à la satisfaction de la couverture fonctionnelle. La deuxième exigence (uniquement ✓ comme consolidation de label), qui démontre que tous les labels sont couverts, est équivalente à la satisfaction de la couverture structurelle. Donc, si la matrice d'une campagne de vérification satisfait ces deux exigences, cette campagne satisfait

Si la spécification S possède ...	Alors le symbole pour la consolidation des cellules de S est ...	Ce qui signifie que ...
Au moins une cellule P✓	P	S est prouvée
Au moins une cellule T✓	T	S est testée
Toutes les cellules avec soit P✗ soit T✗	✗	S n'est associé à aucun label
Au moins une cellule KO	KO	S est invalide
Autres cas	?	la vérification de S n'a aucune conclusion décisive

TABLE 2.3 – Règles pour consolider les résultats par spécification.

Si le label L possède ...	Alors le symbole pour la consolidation des cellule de L est ...	Ce qui signifie que ...
Au moins une cellule P✓	✓	L est fortement connecté à au moins une spécification
Au moins une cellule T✓		
Toutes les cellules avec soit P✗ soit T✗	✗	*L n'a aucune corrélation avec n'importe quelle spécification*
Toutes les cellules avec U	U	*L est inatteignable*
Au moins une cellule KO	KO	une erreur a été trouvée
Autres cas	?	le label L n'est pas couvert par la campagne

TABLE 2.4 – Règles pour consolider les résultats par label.

à la fois l'exigence de couverture fonctionnelle et l'exigence de couverture structurelle, ce qui permet d'en déduire que la campagne est terminée.

	S0	S1	S2	Consolidation Label
l1	P✗	P✓	P✗	✓
l2	P✓	P✗	P✗	✓
l3	P✗	P✗	P✓	✓
Consolidation Specification	P	P	P	

FIGURE 2.9 – Matrice de couverture complète de la fonction `range`

La figure 2.9 montre une matrice de couverture complète de la fonction `range`. Cette matrice résulte d'une combinaison test et preuve. Dans cette matrice, chaque spécification est prouvée (le symbole P dans la consolidation de spécification) et chaque label est associé à une spécification (la symbole ✓ dans la consolidation de label). Dans ce cas, grâce à la preuve, le test n'est utilisé que pour définir le label associé à la spécification `S1`.

2.6 Conclusion

En utilisant les notions de témoin, de label et de mutant, nous avons défini un critère de couverture commun pour le test et la preuve. En reliant chaque spécification et chaque label, nous avons une technique qui sert à calculer le degré de couverture sans concerner trop la méthode de vérification utilisée. Nous proposons aussi la matrice de couverture pour synthétiser les résultats de vérification et pour identifier rapidement le degré de couverture de la vérification.

La matrice de couverture est un outil très utile pour le pilotage d'une campagne de vérification. Elle nous aide à détecter les spécifications non-pertinentes. Elle permet d'identifier aussi les informations manquantes dans la campagne de vérification. Ces informations nous aident à ajouter les activités de vérification nécessaires, en aidant par exemple à définir des cas de test manuels. Cette utilisation de notre méthode de couverture et de la matrice de couverture associée pendant une campagne de couverture est présentée plus en détails dans le chapitre suivant.

Assume nothing.
Believe nobody.
Check everything.

John Cockram

3

Implémentation & Évaluation

Ce chapitre est dédié à l'intégration de la notion de couverture du chapitre précédent au sein d'une campagne de vérification. En effet, nous introduisons ici deux éléments : une plateforme automatique et une méthode d'intégration. La plateforme automatique ajoute les labels et les mutants à partir du critère de couverture, réalise les activités de vérification et remplit la matrice de couverture. La méthode d'intégration aide à faire usage de cette matrice.

La plateforme est implémentée comme un enchaînement des greffons de Frama-C, une plateforme extensible et collaborative dédiée à l'analyse de programmes écrits en C [Kir+15]. Cette plateforme contient des générateurs principaux (générateur de mutant, générateur de témoin et générateur de matrice de couverture) et un contrôleur. Son utilisation est présentée sur un cas d'étude.

Ce cas d'étude réalise la vérification d'une fonction, appelée `transform`, avec notre plateforme. En suivant une campagne de vérification effectuée avec notre plateforme, nous voulons en montrer l'utilisation et comment y intégrer la matrice de couverture pour effectuer une campagne de vérification complète. De plus, à la fin de ce chapitre, nous comparons aussi un exemple de la campagne combinée qui utilise à la fois test et preuve pour satisfaire notre couverture.

3.1 Utilisation de la couverture label-mutant

Dans le chapitre précédent, nous avons défini une notion de couverture qui nous aide à mêler les activités de test et de preuve dans une campagne de vérification. Dans cette section, nous proposons un moyen d'utilisation de cette couverture, introduit dans la figure 3.1. Il repose sur une *plateforme* dans laquelle les activités de vérification sont réalisées automatiquement et produisent une matrice de couverture, ainsi qu'une *méthode* qui montre l'interprétation de la matrice de couverture pour utiliser la plateforme. Plus précisément, d'abord, l'ingénieur ajoute à notre plateforme de vérification tous les éléments nécessaires : le code, les spécifications, le critère de couverture, les configurations de l'activité de preuve et de test automatique, les cas de test manuels et/ou les preuves manuelles. Puis, notre plateforme est automatiquement exécutée et construit une matrice de couverture. Enfin, l'ingénieur exploite la matrice de couverture et décide de l'action suivante d'après la méthode d'interprétation : soit terminer la vérification, soit ajouter des éléments et relancer le cycle de vérification.

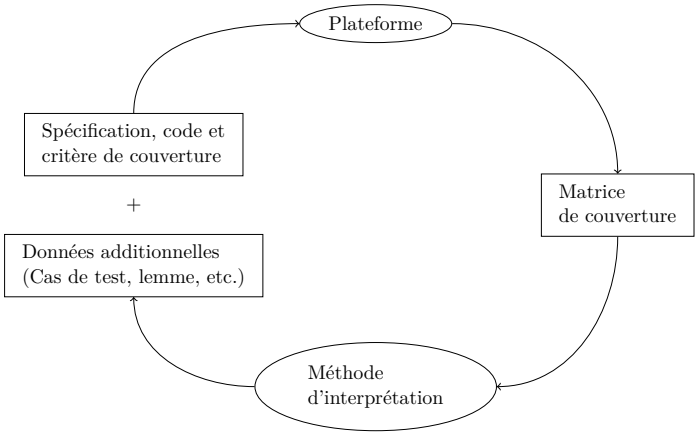


FIGURE 3.1 – Procédure de la vérification.

La présentation de cette section est séparée en trois parties. La première introduit la plateforme, la deuxième présente la méthode d'interprétation de la matrice de couverture, et la dernière présente une proposition pour générer les labels et les mutants pour différents critères de couverture.

3.1.1 Plateforme de vérification

La plateforme de vérification, détaillée dans la figure 3.2, est un outil permettant de réaliser la vérification automatique. Elle contient un générateur de labels et de mutants, un générateur de témoins pour chaque méthode de vérification, un générateur de matrice de couverture et un contrôleur. À partir des éléments fournis par l'utilisateur (code, spécification, critère de couverture, ...), la plateforme est automatiquement exécutée et génère une matrice de couverture.

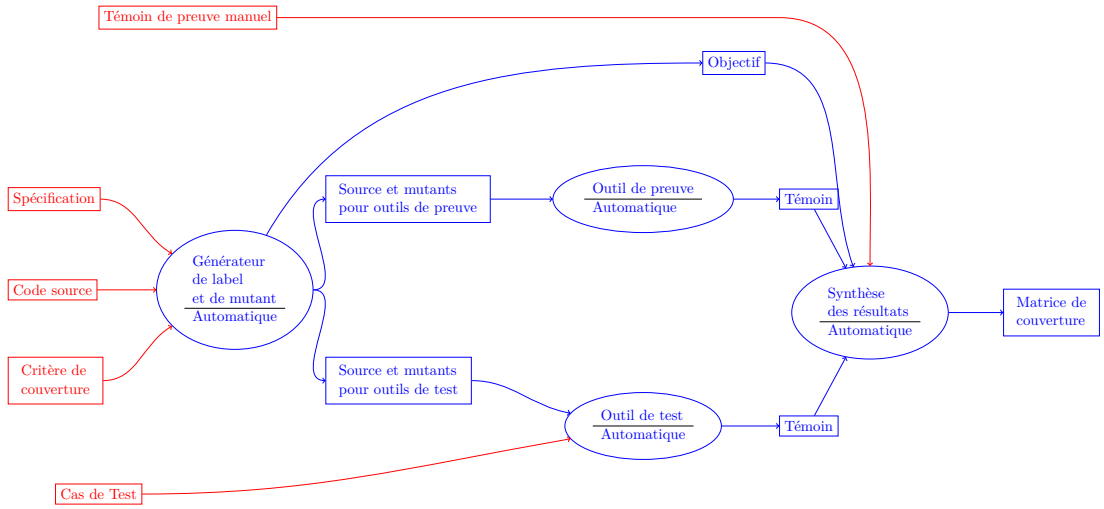


FIGURE 3.2 – Plateforme pour automatiser une campagne de vérification et pour construire la matrice de vérification correspondante.

Générateur de labels et de mutants Il est capable de générer les labels, les mutants et les oppositions à partir du code, des spécifications et du critère de couverture choisis. Dans notre travail, le choix du mutant et du label est soit automatique, soit manuel, soit les deux. Nous utilisons la couverture label-mutant pour représenter les critères de couverture (la couverture des instructions, la couverture des branches et la couverture MC/DC). Pour chaque critère de couverture, nous proposons des labels et des mutants convenables, ce qui est détaillé à la sous-section 3.1.3. Ce générateur produit automatiquement les labels et les mutants à partir du critère de couverture choisi. Néanmoins, l'ingénieur peut aussi définir ses propres labels et mutants et les ajouter manuellement au générateur.

Générateur de témoins Il sert à générer des témoins au format JSON à partir des résultats des activités de vérification. Néanmoins, puisque chaque méthode de vérification fonctionne différemment, il est difficile d'avoir un générateur de mutants commun à toutes les méthodes de vérification. Des générateurs dédiés à chacune doivent donc être construits. Leur format de sortie en JSON permet d'être néanmoins uniforme dans le traitement des témoins a posteriori.

Par exemple, considérons deux activités de vérification d'une même campagne : une activité de preuve reposant sur la vérification déductive et une activité de test dont les cas de test sont automatiquement générés. Pour chaque méthode (la vérification déductive et le test), un générateur de témoin est créé. Le générateur pour la vérification déductive génère des témoins de preuve `P`, `OP`, `MP` et `OMP`, alors que le générateur pour le test génère des témoins de test `T`, `OT`, `MT` et `OMT`. Tous les témoins sont écrits au format JSON.

Générateur de la matrice de couverture Il s'agit du dernier générateur de notre plateforme. Ce générateur collecte les témoins des générateurs de témoin et construit la matrice de couverture qui constitue la sortie de notre plateforme. L'ingénieur peut ensuite appliquer

la méthode d'interprétation de la section 3.1.2 pour définir sa prochaine action.

Contrôleur Comme son nom l'indique, cet outil contrôle l'exécution de la plateforme. L'ingénieur doit détailler toutes les activités de vérification (les outils de vérification, la configuration de chaque outil, etc.) dans un fichier JSON décrivant la campagne de vérification. Puis, il configure et exécute le contrôleur. Ce dernier lance automatiquement en séquence les outils de vérification et les générateurs de la plateforme. L'exécution totale ou partielle de la campagne de vérification dépend de la configuration du contrôleur. Par exemple, si l'ingénieur ne veut réaliser que quelques activités de la campagne mais pas toutes, il peut configurer le contrôleur et notre plateforme exécute donc seulement ces activités.

3.1.2 Méthode d'interprétation

La méthode d'interprétation sert à aider l'ingénieur à décider de la prochaine action à réaliser. Chaque situation possible d'une matrice est présentée ici. Comme nous l'avons déjà expliqué en section 2.5, il est plus facile d'interpréter les résultats d'une matrice à partir de sa version consolidée par ligne (consolidation de spécification) et par colonne (consolidation de label). C'est ce que nous allons effectuer ici.

Vérification complète La campagne de vérification est terminée lorsque les couvertures fonctionnelle et structurelle sont complètes. D'après la section 2.5, cette situation est équivalente à une matrice de couverture dont la consolidation de spécification est remplie uniquement de symboles « P » ou « T » et la consolidation de label n'est remplie que de symboles « ✓ », comme sur l'exemple de la figure 3.3.

	S0	S1	S2	Consolidation Label
l1	P✗	P✓	P✗	✓
l2	P✓	P✗	P✗	✓
l3	P✗	P✗	P✓	✓
Consolidation Specification	P	P	P	

FIGURE 3.3 – Exemple d'une vérification complète.

Erreur trouvée Dans une matrice de couverture, le symbole « KO » implique l'existence d'une erreur, soit dans le code soit dans une spécification. Autrement dit, si nous trouvons « KO » dans la consolidation de spécification (comme sur l'exemple de la figure 3.4), cela signifie que la spécification correspondante est invalide par rapport au code source. L'ingénieur doit donc rapporter cette erreur. Nous pouvons utiliser l'information de label et de spécification pour localiser l'origine du problème rencontré. Après la correction de cette erreur, il peut recommencer la campagne de vérification.

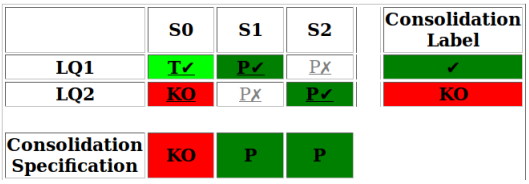


FIGURE 3.4 – Exemple : la spécification `S0` est erronée.

Spécification non-pertinente Une spécification est non-pertinente si cette spécification est prouvée à la fois pour le code original et pour tous ses mutants. La figure 3.5 montre un exemple d'une telle situation dans laquelle la spécification `S0` est marquée par le symbole « ✗ » dans la consolidation de spécification. Ce symbole signifie que la spécification associée est prouvée à la fois pour le code original et pour tous ses mutants (soit elle est toujours vraie soit elle ne s'applique à aucune partie du code). Elle est donc non-pertinente et doit être corrigée.

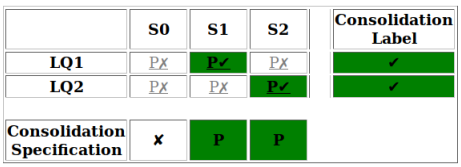


FIGURE 3.5 – Exemple : la spécification `S0` est non-pertinente.

Label inatteignable ou mort Un label inatteignable ou mort est marqué par le symbole « ✗ » dans la consolidation de label. La figure 3.6 montre un exemple d'une telle situation dans laquelle toutes les spécifications sont vérifiées mais le label `LQ2` n'est pas encore couvert. Par conséquent, soit `LQ2` est un label qui représente un code mort ou inatteignable, soit il manque une spécification qui le couvre.

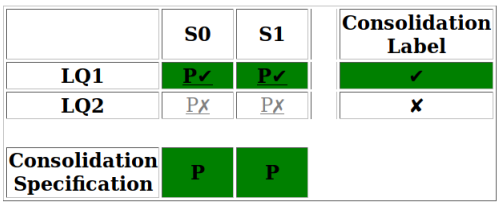


FIGURE 3.6 – Exemple : le label `LQ2` est soit inatteignable soit mort

Vérification incomplète Il s'agit de la situation la plus fréquente d'une campagne de vérification. Nous la rencontrons quand il reste encore une cellule vide ou une marque « ? » dans la consolidation de spécification ou la consolidation de label. L'ingénieur doit alors continuer sa campagne de vérification en ajoutant des activités de vérification supplémentaires.

Pour cela, il peut également s'appuyer sur les résultats de la matrice, ce que nous présentons sur un exemple ci-dessous :

	S0	S1	S2	Consolidation Label
l1	P✗	P?	P✗	?
l2	P✓	P✗	P✗	✓
l3	P✗	P✗	P✓	✓
Consolidation Specification	P	?	P	

FIGURE 3.7 – Exemple d'une campagne incomplète.

Considérons ainsi la figure 3.7 qui présente une matrice correspondant à cette situation. En analysant cette matrice, nous constatons que la spécification `S1` est vérifiée sans avoir de label corrélé. Pour déterminer un tel label, nous devons continuer la campagne de vérification en y ajoutant une autre activité de vérification, par exemple un cas de test pour vérifier `S1`. En examinant la colonne `S1` de la matrice, nous observons que la cellule du label `l1` et de la spécification `S1` est marquée du symbole « `P?` » ; les autres sont marquées par la symbole `P✗` qui signifie que le label n'a aucune corrélation avec `S1`. Par conséquent, le cas de test à créer doit cibler le label `l1`.

Nous voyons donc, comme pour le cas d'erreur, que la matrice permet de guider le choix des activités de vérification à mener pour compléter la campagne.

3.1.3 Proposition des labels et des mutants pour les critères de couverture

Cette section présente une façon de générer les labels et les mutants pour différents critères de couverture. Notre génération des labels est fortement influencée par les travaux de Bardin et al. [BKC14 ; Bar+14]. Le choix de mutant dépend du critère de couverture choisi et de l'instruction à muter. Le mutant créé doit satisfaire deux conditions : l'opérateur de mutation est le plus fort possible, tandis que le mutant créé doit pouvoir être compilé et exécuté. D'après [Del+14], deux opérateurs donnent en général de très bon résultat sur ce type d'approche : « supprimer une instruction » et « modifier une expression ».

Couverture des instructions Elle garantit que chaque instruction est couverte par une activité de vérification. Pour cela, notre générateur ajoute des labels « `//@ label L : \true` » au niveau de toutes les instructions opérationnelles. Chaque label représente une instruction à couvrir.

Le choix d'un opérateur de mutation pour cette instruction est difficile. En effet, pour montrer qu'une instruction est couverte par une activité de vérification, une solution simple est de revérifier le code après avoir effacé cette instruction. Autrement dit, l'opérateur de mutation choisi est « supprimer une instruction ». Néanmoins, certaines instructions spécifiques comme « `return e;` » ne doivent pas être effacées parce que l'effacement de ces instructions violerait

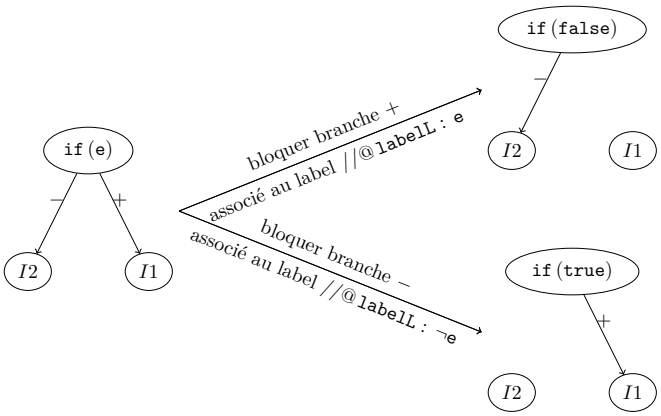


FIGURE 3.8 – Mutant créé par effacer de branche de la condition « `if (e)` »

la sémantique du code. Après avoir effacé ces instructions, le code ne pourrait plus être compilé et alors, ne pourrait plus être vérifié. C'est pourquoi, pour ces instructions, nous proposons d'utiliser un autre opérateur de mutation. D'après [Del+14], l'opérateur « modifier une expression » est aussi fort que « supprimer une instruction ». Il remplace l'expression d'une telle instruction (par exemple : `e` dans l'instruction « `return e;` ») par une constante (par exemple : `0`).

Couverture des branches Cette couverture sert à garantir que chaque branche est vérifiée. Elle est plus forte et subsume la couverture des instructions. Pour ce critère de couverture, nous commençons donc par ajouter les mêmes labels et les mêmes mutants que pour la couverture des instructions. Ensuite, nous ajoutons en complément des labels et des mutants pour chaque instruction conditionnelle « `if (e)` » du code de la manière suivante.

En nous inspirant des travaux de Bardin et al. [BKC14 ; Bar+14], nous ajoutons deux labels « `//@label L : e` » et « `//@labelL : ¬e` » pour chaque instruction conditionnelle « `if (e)` ». Pour le choix d'opérateur de mutation, nous visons la création d'un mutant par branche de la conditionnelle « `if (e)` », chacun associé à un label de cette branche. Le premier est créé en effaçant la branche positive de la conditionnelle (celle qui est exécutée lorsque la condition `e` est vraie, voir la section 1.3). Le deuxième mutant est créé en effaçant la branche négative de cette instruction conditionnelle (celle qui est exécutée lorsque la condition `e` est fausse). En vérifiant chacun de ces deux mutants, nous pouvons savoir si la branche effacée dans ce mutant est vérifiée ou pas.

L'effacement de branche est expliqué graphiquement dans la figure 3.8. Dans cette figure, le premier mutant, associé au label « `//@label L : e` », est créé en remplaçant `e` par `false`. Le label « `//@label L : e` » représente la branche positive de l'instruction. En remplaçant la condition par `false`, nous effaçons la branche positive. De la même manière, le second mutant, associé au label « `//@label L : ¬e` », est créé en remplaçant `e` par `true`. Cela correspond à l'effacement de la branche négative.

Couverture MC/DC simplifiée [Hay+01] Cette couverture est une extension de la couverture des branches. Elle demande de garantir d'abord la couverture des instructions. Elle demande aussi de garantir que l'évaluation de chaque condition affecte indépendamment la décision. Pour comprendre l'exigence de cette couverture, ainsi que les labels et les mutants qui en découlent, nous devons introduire trois notions : la condition, la décision et l'inversion de branche.

La *condition* est une expression logique atomique, c'est-à-dire une condition qui ne contient aucun connecteur logique (par exemple, $\mathtt{a} < \mathtt{b}$, $\mathtt{a} > \mathtt{b}$, $\mathtt{a} = \mathtt{b}$). Dans l'instruction conditionnelle « $\mathtt{if}\,(\mathtt{e})\,\{\cdots\}\,\mathtt{else}\,\{\cdots\}$ », $\mathtt{e}$ est une expression logique appelée *décision*. Une décision est construite par une ou plusieurs conditions reliées entre elles par des connecteurs logiques $\wedge$, $\vee$, $\neg$, $\Rightarrow$. Par exemple, la décision $\mathtt{e} \stackrel{\text{def}}{=} \mathtt{e_1} \wedge (\mathtt{e_2} \vee \mathtt{e_3})$ contient trois conditions e_1, e_2 et e_3. Chaque expression logique a deux valeurs de vérité possibles `true` et `false` (vrai et faux en français).

La couverture MC/DC simplifiée consiste à vérifier l'effet produit par un changement de la valeur d'une seule condition sur la valeur de la décision à laquelle elle appartient. Par conséquent, les labels ajoutés représentent une combinaison des valeurs des différentes conditions. Par exemple, $\mathtt{L} = (\neg\mathtt{e_1} \wedge \mathtt{e_2} \wedge \mathtt{e_3})$ signifie que le label représente une combinaison dans laquelle $\mathtt{e_2}$ et $\mathtt{e_3}$ sont vrais et $\mathtt{e_1}$ est faux. En remplaçant les conditions de la décision par les valeurs de combinaison, nous obtenons une valeur de vérité pour cette décision. Pour respecter la couverture MC/DC simplifiée, nous ajoutons une collection de labels représentant des combinaisons telles que pour chaque condition, nous ayons toujours deux labels ne différant que par cette condition et tels que en appliquant ces combinaisons dans la décision, les valeurs de vérité de décision soient différentes. Par exemple, pour la décision $e = e_1 \wedge (e_2 \vee e_3)$, nous avons les deux labels suivants :

$$L = (\neg e_1 \wedge e_2 \wedge e_3)$$
$$L' = (e_1 \wedge e_2 \wedge e_3)$$

En appliquant la combinaison du label L à la décision e, la valeur de vérité de e est `false`, tandis que la valeur de vérité de e est `true` avec la combinaison du label L'. La différente entre L et L' est la valeur de vérité de e_1. Elle est `false` dans L, et `true` dans L'. Le critère MC/DC est satisfait si et seulement si nous pouvons garantir que la combinaison dans chaque label est couverte par la vérification.

La notion d'inversion de branche est nécessaire pour définir les mutants pour MC/DC. L'inversion de branche est le changement de l'expression $\mathtt{e}$ de l'instruction « $\mathtt{if}\,(\mathtt{e})\,\{\cdots\}\,\mathtt{else}\,\{\cdots\}$ » par sa négation $\neg\mathtt{e}$. Pour montrer si une combinaison est couverte par une activité de vérification, nous comparons les résultats de vérification du code source et du mutant créé par l'inversion de la branche liée à cette combinaison. Seules les exécutions passant par le label associé à un mutant passent la branche inversée. En comparant la vérification du code source et du mutant, nous pouvons déterminer si la vérification couvre ce label ou pas. Si deux vérifications donnent des résultats identiques, nous savons que la vérification ne couvre pas ce label et, inversement, si deux vérifications donnent un ou plusieurs résultats différents, la

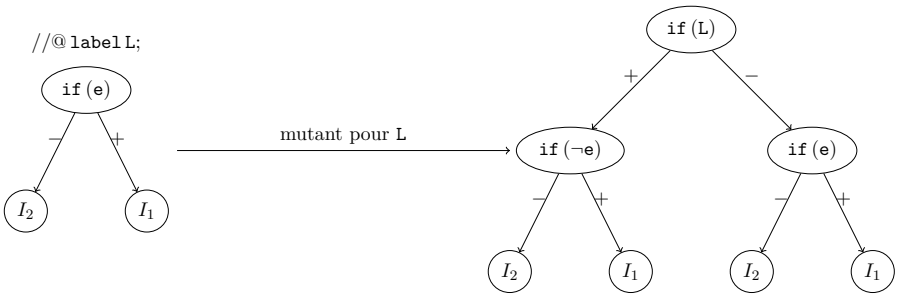


FIGURE 3.9 – Mutant créé en inversant la branche de la condition « `if (e)` » quand le label L est vrai.

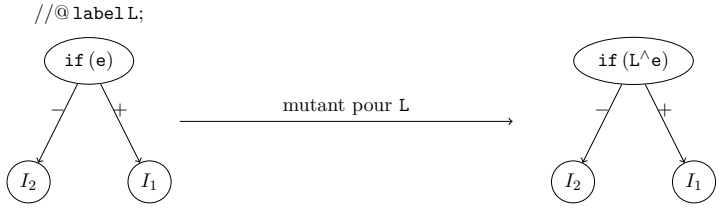


FIGURE 3.10 – Mutant créé en utilisant « ou exclusif » pour inverser la branche.

vérification couvre ce label.

Le mutant créé est représenté graphiquement dans la figure 3.9. Dans cette figure, nous trouvons que les exécutions qui ne passent pas par le label L suivent l'instruction conditionnelle source « `if (e) I1 else I2` » tandis que celles qui passent par L passent l'instruction conditionnelle inversée « `if ¬(e) I1 else I2` ». Nous pouvons utiliser l'opérateur « ou exclusif » pour réduire cela, remplacer « `if (L){...}else{...}` » par « `if (L^e)` » (figure 3.10). Après avoir vérifié le code source et le mutant créé, nous en déduisons la couverture du label L. Si tous les labels sont couverts, la couverture MC/DC simplifiée est satisfaite.

3.2 Prototype de la plateforme

Dans la section précédente, nous avons présenté une méthode d'interprétation de la matrice de couverture et une plateforme automatique de vérification. La méthode d'interprétation guide l'ingénieur pour décider de ce qu'il doit ensuite faire. La plateforme est un outil qui réalise automatiquement les activités de vérification et construit la matrice de couverture. Nous avons développé un prototype de cette plateforme au sein de Frama-C [Kir+15]. Ce prototype est introduit dans cette section.

La figure 3.11 montre l'état actuel de notre prototype. Il manque encore quelques fonctionnalités comparativement à notre plateforme théorique présentée dans la figure 3.2. Par exemple, nous envisageons d'ajouter automatiquement les labels et les mutants à partir d'un

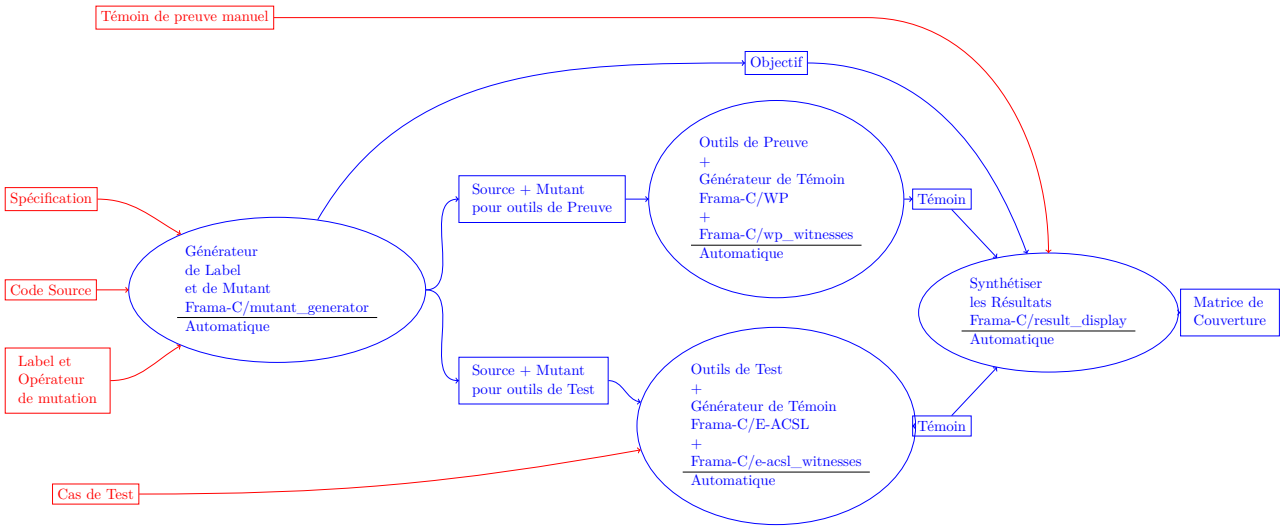


FIGURE 3.11 – Prototype de la plateforme.

critère de couverture choisi (figure 3.2). Pour l'instant, notre prototype exige de les ajouter manuellement (figure 3.11). Notre prototype génère ensuite les fichiers des mutants et ajoute les oppositions au code original et aux mutants. Comme les outils de vérification fonctionnent chacun d'une manière différente, notre générateur doit créer des codes appropriés pour chacun d'eux. Par exemple, il doit garder les annotations d'aide à la preuve dans les fichiers analysés par l'outil de preuve, tandis qu'il les efface des fichiers réservés à l'outil de test.

Notre plateforme est implémentée au sein de Frama-C. Chaque outil développé est donc un greffon de Frama-C. Il s'agit des greffons Frama-C/mutant_generator, Frama-C/mp_witnesses, Frama-C/e-acsl_witnesses, Frama-C/result_display de Frama-C indiques sur la figure 3.11. Nous présentons ici les parties de Frama-C sur lesquelles nous reposons, avant d'introduire nos propres développements.

3.2.1 Frama-C

Pour alléger les notations, dans cette section, nous réduisons le nom des greffons pour y enlever le préfixe Frama-C/. Par exemple, Frama-C/WP devient WP et Frama-C/E-ACSL devient E-ACSL.

Frama-C [Kir+15] est un logiciel qui sert à analyser le code source de programmes écrits en C. Il possède plusieurs greffons qui permettent notamment d'utiliser la vérification déductive (WP), la vérification dynamique (E-ACSL), ou l'interprétation abstraite (EVA). Ces greffons peuvent collaborer entre eux. Par exemple, comme nous allons le détailler par la suite, E-ACSL et WP peuvent collaborer pour vérifier conjointement un ensemble de propriétés donné. D'après [Sig+], il est possible de créer des greffons de Frama-C et de les ajouter au sein de Frama-C. C'est l'approche que nous avons choisie pour notre plateforme.

```
/*@
 requires v  <  u;

 behavior S1:
 assumes a  <  v;
 ensures \result == v;

 behavior S2:
 assumes v  ≤  a  ≤  u;
 ensures \result == a;

 behavior S3:
 assumes a  >  u;
 ensures \result == u;
*/
int range(int a, int u, int v)
{
 ...
}
```

FIGURE 3.12 – Exemple de spécifications ACSL.

Les codes C analysés par Frama-C peuvent être spécifiés avec des annotations ACSL [Bau+a]. Un exemple de spécifications écrites en ACSL se trouve dans la figure 3.12. Dans cette figure, l'annotation ACSL est comprise entre $/*@$ et $*/$. Cette annotation contient les mots clés « `requires` », « `behavior` », « `assumes` », « `ensures` » et « `result` ». Chaque mot clé « `requires` », « `assumes` » ou « `ensures` » est associé à une condition appelée clause. La clause « `requires` » indique l'exigence de l'entrée que tous les appels à la fonction `range` doivent satisfaire. Un « `behavior` » représente un comportement. Chaque « `behavior` » contient une clause « `assumes` » et une clause « `ensures` ». Toutes les exécutions du code doivent, soit ne pas satisfaire la clause « `assumes` » soit satisfaire la clause « `assumes` » et la clause « `ensures` ». Le mot clé « `\result` » représente la valeur sortie du code.

Dans notre travail, une spécification $S \stackrel{\text{def}}{=} H \Rightarrow C$ est constituée de deux parties : une hypothèse H et une conclusion C. Nous allons exprimer une telle spécification sous forme d'un comportement ACSL (`behavior`). Dans celui-ci, une clause « `assumes` » exprime l'hypothèse H tandis qu'une clause « `ensures` » exprime la conclusion C. La clause « `requires` » représente une hypothèse globale. Autrement dit, pour définir une spécification associée à un comportement `B`, nous devons combiner les « `requires` » de la fonction et les « `assumes` » de `B` pour créer l'hypothèse générale tandis que nous utilisons la clause « `ensures` » pour identifier la conclusion. À partir de l'exemple précédent, nous pouvons déduire la spécification `S1` correspondant au comportement de même nom comme suit :

$$\texttt{S1} = \underbrace{(\texttt{v} < \texttt{u} \wedge \texttt{a} < \texttt{v})}_{\text{hypothèse}} \Rightarrow \underbrace{(\backslash\texttt{result} == \texttt{v})}_{\text{conclusion}}$$

Par ailleurs, les greffons de Frama-C sont capables de générer, modifier ou utiliser des spécifications écrites en ACSL. Par exemple, le greffon WP présenté ci-après est capable de *prouver* la spécification `S1`.

WP [Bau+b] est le greffon de vérification déductive de Frama-C. Il permet de prouver des spécifications ACSL en utilisant des démonstrateurs automatisés de théorèmes comme Alt-Ergo [Bob+] ou des assistants de preuve comme Coq [BC04]. Dans cette situation, WP peut être vu comme un outil intermédiaire qui manipule les spécifications ACSL et le code de la fonction `range` et les transforme en formules mathématiques. En prouvant automatiquement ces formules avec Alt-Ergo ou de manière assistée avec Coq, nous pouvons prouver que la spécification `S1` est valide.

Les modèles mémoires de WP permettent de choisir comment les pointeurs et les accès mémoires sont modélisés par des formules mathématiques. Par exemple, le modèle mémoire Caveat simule le comportement de l'analyseur Caveat [SFF04]. Ce modèle fait usage d'une détection de variables spécifique qui augmente la probabilité de prouver les annotations contenant des pointeurs.

E-ACSL [SKV17] est un greffon d'analyse dynamique de Frama-C permettant d'évaluer les annotations formelles pendant les exécutions d'un programme C. Plus précisément, E-ACSL transforme le code C annoté d'origine en un code équivalent dans lequel une vérification *dynamique* des annotations est effectuée. Pour que le code généré soit exécutable, ce greffon exige une fonction « main » à partir de laquelle débuter l'exécution. Ensuite, le code généré est compilé et exécuté. L'exécution reporte si le code viole une spécification annotée dans le code source. Dans notre contexte, nous utilisons E-ACSL pour tester des fonctions formellement spécifiées similaires à celle de la fonction `range` précédemment introduite. Pour cela, nous pouvons par exemple ajouter la fonction `main` suivante pour représenter les cas de test sur les vecteurs d'entrée (6,15,10), (8,10,3) et (3,2,1).

```
// test de la fonction "range(a,u,v)"
int main(){
 range (6,15,10);
 // range(6,15,10) retourne l'entier 10
 // S1, "si a < v < u alors v", est satisfaite

 range (8,10,3);
 // range(8,10,3) retourne l'entier 8
 // S2, "si v ≤ a ≤ u alors a", est satisfaite

 range (3,2,1);
 // range(3,2,1) retourne l'entier 2
 // S3, "si v < u < a alors u", est satisfait

 return 0;
Post:}
```

Notre prototype ne pourrait pas créer automatiquement les cas de test pour E-ACSL. Nous devons donc créer ces cas de test. Ils sont définis soit manuellement soit automatiquement par un générateur de cas de test (par exemple : Frama-C/PathCrawler [Wil+05]).

Le greffon E-ACSL n'analyse qu'un sous-ensemble du langage ACSL, également appelé E-ACSL [Sig ; DKS13]. Puisque ce langage est une version réduite d'ACSL, les autres outils de Frama-C (e.g. WP) sont également capables de vérifier les spécifications E-ACSL. Comme nous utilisons E-ACSL dans notre étude, nous nous limitons donc désormais au langage E-ACSL pour

exprimer nos spécifications. L'exemple de fonction `range` est compatible avec E-ACSL.

3.2.2 Prototype

Le prototype décrit dans la figure 3.11 contient les mêmes générateurs que ceux de notre plateforme théorique : un générateur de labels et de mutants, des générateurs de témoin et un générateur de matrice de couverture. Ces générateurs sont contrôlés par un contrôleur. Nous fournissons ci-après quelques détails de chaque générateur et du contrôleur.

Générateur de labels et de mutants Du point de vue théorique, comme nous l'avons vu, ce générateur peut déduire automatiquement les labels et les mutants à partir du critère de couverture choisi. Néanmoins, celui de notre prototype, appelé « `mutant_generator` », exige encore un ajout manuel du code, des spécifications et surtout des annotations représentant les labels et les opérateurs de mutation. Si ces éléments sont ajoutés, ce générateur crée ensuite automatiquement les mutants et les oppositions. En particulier, il génère différemment les mutants et les oppositions pour la preuve et le test. Par exemple, pour le test, nous ajoutons une assertion E-ACSL. Les exécutions qui passent cette assertion passent par le label. Pour la preuve, nous n'avons pas besoin de cette assertion.

Plus précisément, l'utilisateur débute par les annotations de labels //@ `label L : A`. `L` est le nom du label et `A` est la condition du label. Ensuite, pour chaque annotation de label, un opérateur de mutation //@ `mutant L : Op` est ajouté. `L` est le nom du mutant et `Op` est l'opérateur de mutant choisi. À partir de ces annotations, notre prototype produit automatiquement les mutants.

Le générateur crée un fichier par mutant. Puisque les outils de vérification peuvent vérifier la spécification et son opposition en même temps, nous n'avons pas besoin d'un fichier particulier pour ces oppositions. Ce générateur liste aussi les spécifications et les labels du code. Ces spécifications et ces labels sont sauvegardés dans des fichiers JSON. Ils sont plus tard utilisés pour définir le nombre de lignes et de colonnes de la matrice de couverture.

Générateurs de témoins Ces générateurs sont associés aux outils de vérification utilisés, à savoir les greffons WP et E-ACSL. Le greffon « `wp_witnesses` », associé au greffon WP, peut extraire directement des résultats de preuve d'un témoin. Le greffon « `e-acsl_witnesses` », associé au greffon E-ACSL, construit un témoin à partir des informations produites par E-ACSL lors des vérifications dynamiques des annotations. Ces témoins sont sauvegardés dans des fichiers JSON pour être utilisés plus tard lors de la construction de la matrice de couverture. Les greffons associés n'interviennent pas pendant l'exécution des greffons de vérification WP et E-ACSL. Ils prennent juste les résultats de ces greffons pour créer les témoins. La partie difficile de ces générateurs est la classification des témoins. En effet, l'outil de vérification vérifie les spécifications et leurs oppositions en même temps, notre générateur de témoin doit distinguer les résultats obtenus pour les spécifications de ceux obtenus pour les oppositions.

Générateur de matrice de couverture Ce générateur, appelé « `result_display` », collecte tous les fichiers JSON qui contiennent les spécifications, les labels et les témoins pour construire une matrice de couverture. Elle est présentée dans un fichier HTML. Quand nous cliquons sur une marque dans une cellule d'une telle matrice, les témoins liés à cette marque sont affichés.

Contrôleur Le contrôleur porte le nom de la plateforme : « `witness` ». Il sert à lire le fichier qui contient les informations d'une campagne de vérification et à définir ensuite les générateurs utilisés, leur ordre d'exécution et à les exécuter selon cet ordre. Nous pouvons configurer le contrôleur pour exécuter la plateforme comme nous voulons. Par exemple, une fois collectés les témoins des différentes activités d'une campagne de vérification, nous pouvons construire la matrice de couverture en utilisant seulement ceux-ci.

3.2.3 Conclusion

Nous avons développé un prototype de notre plateforme au sein de Frama-C. Il sert à réaliser automatiquement une campagne de vérification et à produire la matrice de couverture résultante. À partir de cette matrice, nous pouvons appliquer la méthode d'interprétation présentée à la section 3.1.2 pour déterminer si la vérification est complète et terminée.

Greffon	nombre de lignes de code
mutant_generator	4 187
wp_witnesses	834
e-acsl_witnesses	1 318
result_display	3 888
witness	2 794

FIGURE 3.13 – Nombre de lignes de code pour chaque partie du prototype.

Notre prototype contient environ 13000 lignes de code écrites en Ocaml. Leur répartition par générateur est présentée dans la figure 3.11. Le générateur de labels et de mutants fut le plus difficile à créer. Ensuite, ce furent ceux pour générer la matrice de couverture « `result_display` » et pour contrôler le prototype « `witness` ». En revanche, l'extraction de témoins à partir des greffons de vérification existants, comme WP ou E-ACSL, fut relativement facile à réaliser.

3.3 Cas d'étude

Nous avons maintenant une plateforme de vérification basée sur Frama-C qui met en œuvre notre notion de couverture label-mutant. Dans cette section, nous présentons l'utilisation de notre greffon pour construire et réaliser une campagne de vérification sur un cas d'étude.

Ce cas d'étude est une fonction réaliste appelée `transform`. Son code `C` complet (incluant spécifications, labels et mutants) est introduit en annexe A. Nous ne présentons ici que les éléments nécessaires à sa compréhension.

3.3.1 Présentation de la fonction `transform`

La fonction `transform` calcule un signal de sortie à partir d'un signal d'entrée en utilisant une régression linéaire. Cette régression linéaire, présentée dans la figure 3.14, dépend des paramètres x_1, x_2, y_1 et y_2. Ces valeurs sont toutes bornées par une borne supérieure s_{max} et une borne inférieure s_{min}. En conséquence, les paramètres doivent satisfaire les contraintes suivantes :

$$(1) \quad s_{min} \leq x_1 < x_2 \leq s_{max}$$
$$(2) \quad s_{min} \leq y_1 \leq s_{max}$$
$$(3) \quad s_{min} \leq y_2 \leq s_{max}$$

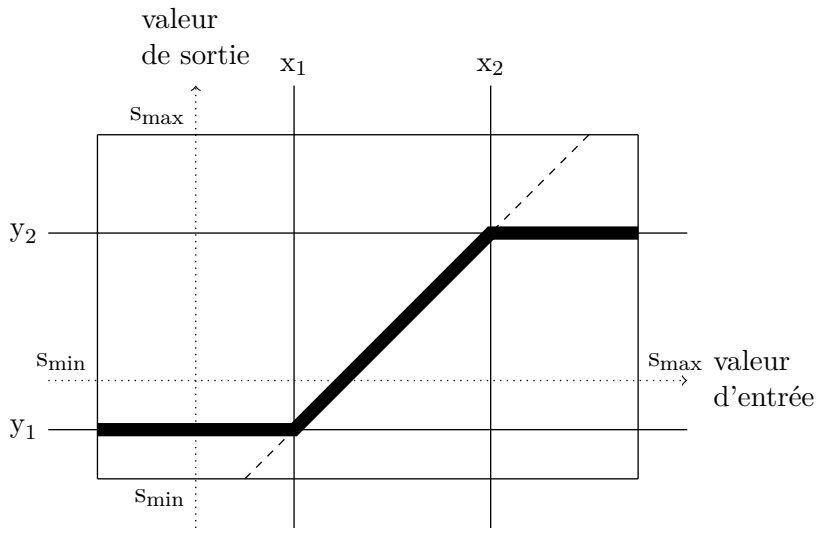


FIGURE 3.14 – Régression linéaire pour la transformation.

Dans le corps de la fonction `transform`, la borne supérieure s_{max} et la borne inférieure s_{min} sont représentées par des variables globales. Les paramètres x_i et y_i avec ($i \in \{1, 2\}$) sont fournis en paramètre de la fonction `transform` au travers d'une structure appelée `Block`. Les signaux (entrée et sortie) sont fournis *via* une structure appelée `Signal`. Cette structure contient deux champs : l'un contenant la valeur du signal et l'autre déterminant sa validité. La figure 3.15 présente un squelette de cette fonction qui retourne 0 lorsque les contraintes (1), (2) et (3) ci-avant sont satisfaites et un code d'erreur sinon.

Les objectifs de vérification de cette fonction sont des exigences informelles déterminées lors du processus de développement du logiciel. Ces objectifs sont formalisés et classés en

```
// Groupe de spécifications ERR: S0 à S6
// Groupe de spécifications OK: S7
// Groupe de spécifications VALUE: S8 à S10
// Groupe de spécifications VALID: S11 à S12
int transform (Block *p, Signal *input, Signal *output)
{
 // 1. vérifier la validité du block
 // 2. modifier la valeur du signal de sortie en fonction du signal d'entrée
 // 3. configurer le drapeau d'erreur du signal de sortie
}
```

FIGURE 3.15 – Squelette de la fonction `transform`.

quatre groupes de spécifications (voir la figure 3.15). Le groupe **ERR** contient 7 spécifications étiquetées de `S0` à `S6`. Les spécifications de ce groupe caractérisent les codes d'erreur retournés par la fonction (chaque spécification caractérise un code d'erreur différent) lorsque les paramètres sont invalides. Le groupe **OK** contient la spécification `S7` qui formalise le résultat lorsque toutes les contraintes sont satisfaites. Les trois spécifications du groupe **VALUE**, étiquetées de `S8` à `S10`, définissent les valeurs attendues du signal de sortie, tandis que les deux spécifications du groupe **VALID**, étiquetées `S11` et `S12`, contrôlent le drapeau d'erreur du signal de sortie.

Nous vérifions la fonction `transform` avec Frama-C. Les spécifications ci-dessus sont ainsi écrites dans le langage de spécification E-ACSL.

Un ingénieur de vérification qui ne bénéficie pas de notre outil, mais seulement de la plateforme Frama-C peut choisir d'utiliser soit WP pour prouver les spécifications, soit E-ACSL pour les vérifier dynamiquement. Dans le cas de WP, une spécification du groupe **VALUE** n'est pas prouvée. En conséquence, il ne peut pas calculer la couverture de la preuve en utilisant la méthode de couverture de preuve de la DO-178C [Bro+10 ; Moy+13] qui demande que toutes les spécifications soient prouvées pour définir la satisfaction de la couverture. Avec E-ACSL, l'ingénieur peut utiliser les labels (voir la section 1.3.2) pour représenter un critère de couverture selon la DO-178C [RTC11a]. Après avoir appliqué E-ACSL aux cas de test, la couverture est calculée en comptant le nombre de labels passés par les exécutions de ces cas de test. Néanmoins, pour utiliser E-ACSL, il doit déterminer manuellement au moins 10 cas de test. Sept d'entre eux sont utilisés pour tester l'ensemble des situations où les contraintes sont invalides. Les trois autres correspondent à des valeurs attendues du signal de sortie lorsque les contraintes sont satisfaites. L'ingénieur de vérification doit donc dépenser beaucoup d'efforts pour couvrir toutes les spécifications.

Pour conclure, sans une notion de couverture commune au test et à la preuve, il est difficile et coûteux de compléter une campagne de vérification. Nous allons réaliser maintenant une campagne de vérification sur le cas d'étude en suivant notre approche qui permet de combiner les méthodes de test et preuve.

3.3.2 Ajout des labels et des mutants

L'utilisateur commence par déterminer les labels et les opérateurs de mutation correspondant au critère de couverture structurelle que nous souhaitons couvrir (couverture des instructions, couverture des branches, etc.). Ici, nous choisissons la couverture des instructions en y ajoutant quelques labels supplémentaires permettant de montrer un avantage de notre outil : pouvoir choisir des labels et des opérateurs de mutation. Cependant, comme précisé précédemment, notre outil n'est pas encore capable de générer automatiquement des labels et des opérateurs de mutation. Nous les définissons donc manuellement. Après l'avoir fait, notre outil est en mesure de créer les mutants automatiquement.

Maintenant, nous expliquons comment ajouter manuellement des labels et des opérateurs de mutation. Nous utilisons deux commentaires séparés : un pour expliciter le label et un autre pour noter l'opérateur de mutation utilisé pour ce label. Cette séparation est nécessaire pour permettre d'ajouter un label à un point de programme donné, tout en associant l'opérateur de mutation attaché à ce label à un autre point. Le label et son opérateur de mutation sont reliés par leur nom. Par exemple, considérons le code suivant :

```
...
 //@ label L1: a  <  v;
 if (a  <  v) {
 //@ mutant L1: replace_assign (0);
 return -1;
 }
...
```

Dans le code, l'opérateur de mutation « //@`mutant L1 : replace_assign(0);` » est associé au label « //@`label L1 : a < v;` ». `L1` est le nom commun au label et à l'opérateur de mutation. La fonction `replace_assign` représente l'opérateur de mutation « modifier une expression ». La séparation de la définition du label et du mutant est utile pour certains des critères de couverture structurelle, notamment la couverture des branches. Cette couverture exige que toutes les branches soient exécutées et vérifiées. Le label, dans ce cas, représente les exécutions qui passent par la branche `a < v` et donc passent par l'instruction « `return − 1;` ». Ici, au lieu de muter l'instruction « `if (a < v)` », nous préférons muter « `return − 1;` » et associer le mutant créé au label `L1`.

Le code source de la fonction `transform` qui contient tous les labels et les opérateurs de mutation de notre exemple est donné dans l'annexe A. En utilisant le greffon « `witness` », nous exécutons le greffon « `mutant_generator` » pour créer automatiquement des mutants de la fonction `transform` à partir des annotations de label et de mutant que l'utilisateur a ajoutées.

3.3.3 Vérification et couverture de la fonction `transform`

La suite de la campagne de vérification consiste à exécuter les activités de vérification. Nous illustrons l'usage de notre plateforme pour déduire les activités de vérification, activité

par activité. D'abord, notre campagne de vérification commence avec une seule activité de preuve automatique : l'utilisation du greffon WP pour effectuer la vérification déductive. La campagne est explicitée dans un fichier JSON présenté figure 3.16.

```
{
 "functions" : [
   "transform"
 ],
 "activities" : [
   {
    "name" : "proof_without_model",
    "method" : "wp",
    "option" :
     {
      "wp_option" : ""
     }
   },
 ]
}
```

FIGURE 3.16 – Exemple d'une campagne de vérification.

Initialement, elle contient une activité appelée `proof_without_model`. Cette activité utilise le greffon WP de Frama-C pour prouver les spécifications. Le greffon « `witness` » identifie `proof_without_model` et exécute les greffons correspondant « `wp_witnesses` » (pour effectuer la vérification déductive de WP et produire les témoins) et « `result_display` » (pour afficher le résultat à partir du témoin). L'exécution de notre outil produit une matrice de couverture présentée figure 3.17. Dans celle-ci, nous pouvons observer grâce à la ligne « Consolidation Specification », que cinq spécifications ne sont pas vérifiées (absence de marque) ou sont vérifiées mais qu'aucun label associé n'a été trouvé (marque « `?` »). La colonne « Consolidation Label » montre en outre que cinq labels ne sont pas couverts (marque « `?` »).

Les trois spécifications `S10`, `S11` et `S12` ne sont pas vérifiées lors de l'activité `proof_without_model`. En lisant ces spécifications, nous observons qu'elles contiennent des pointeurs compliquant la preuve automatique. Pour surmonter cette difficulté, nous continuons notre campagne en y ajoutant une activité de preuve automatique, appelée `proof_with_model`. Elle utilise encore WP pour prouver les spécifications, mais avec le modèle mémoire **caveat** (voir la section 3.2.1) pour pouvoir prouver plus de spécifications quand il n'y a pas d'alias entre les pointeurs en argument de la fonction, ce que nous pouvons supposer ici.

Une fois cette activité terminée, les résultats courants de la campagne sont synthétisés dans une nouvelle matrice de couverture présentée en figure 3.18. Cette matrice montre que deux spécifications supplémentaires sont désormais prouvées, à savoir `S11` et `S12`. En observant la matrice, de plus près, nous remarquons que, parmi les cinq spécifications non-couvertes, nous en avons quatre (`S8`, `S9`, `S11` et `S12`) qui sont prouvées mais pour lesquelles aucun label n'est associé (marque « `?` » dans la consolidation de spécification).

La preuve n'est pas suffisante pour continuer la vérification à cause des flottants de la

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Consolidation Label
Lerror1	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗				✓
Lerror2	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗				✓
Lerror3	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗				✓
Lerror4	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗				✓
Lerror5	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗				✓
Lerror6	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗				✓
Lerror7	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗				✓
Llow1	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?	P✗				?
Llow2	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?	P✗				?
Lhigh	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?				?
Lmedium	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		?		?
Lvalidity	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗				?
Lok	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗				✓
Consolidation Specification	P	P	P	P	P	P	P	P	?	?				

Couple Spécification-Label :
- P✗ = absence de relation démontrée
- P✓ ou T✓ = relation démontrée
- P? ou T? = relation possible
- ? = seule information de couverture
- vide = pas d'information

Consolidation Spécification :
- P = spécification prouvée et associée à au moins un label
- T = spécification testée et associée à au moins un label
- ? = spécification vérifiée mais sans information sur le label associé
- vide = spécification non vérifiée

Consolidation Label :
- ✓ = label associé à au moins une spécification vérifiée
- ? = sans information sur la spécification associée

FIGURE 3.17 – Matrice de couverture après l'exécution de l'activité `proof_without_model`.

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Consolidation Label
Lerror1	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror2	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror3	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror4	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror5	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror6	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror7	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗		P✗	P✗	✓
Llow1	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?	P✗		P✗	P✗	?
Llow2	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?	P✗		P✗	P✗	?
Lhigh	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P?		P✗	P✗	?
Lmedium	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	?
Lvalidity	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P?	P?	?
Lok	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗		P✗	P✗	✓
Consolidation Specification	P	P	P	P	P	P	P	P	?	?		?	?	

FIGURE 3.18 – Matrice de couverture après exécution de `proof_with_model`.

spécification S10. Nous pensons ensuite à ajouter des cas de test. Notre matrice de couverture peut nous aider à définir facilement les cas de test. Considérons la version réduite de la matrice

de la figure 3.18 donnée dans la figure 3.19. Nous utilisons cette version pour montrer les labels considérés en déterminant les cas de test des spécifications `S8` et `S9`. Plus précisément, dans la matrice, « `P✗` » signifie que la spécification n'est pas associée au label correspondant. Donc, nous ne considérons pas un tel label pour définir le cas de test pour une telle spécification. Il reste seulement les cellules marquées « `P?` ». Ce symbole signifie que le label correspondant est possiblement reliée à la spécification indiquée une relation avec spécification. Ainsi, nous nous fondons sur les cellules marquées `P?` pour créer les cas de test. Par exemple, pour créer un cas de test qui vérifie la spécification `S8`, nous considérons seulement les exécutions qui passent par le label `Llow1` ou `Llow2`.

	S8	**S9**
Llow1	P?	P✗
Llow2	P?	P✗
Lhigh	P✗	P?
Consolidation Specification	?	?

Consolidation Label
?
?
?

FIGURE 3.19 – Matrice de couverture réduite de l'activité `proof_with_model` visant à créer les cas de test pour `S8` et `S9`.

```
#define DUMMY_V 583278.0
#define DUMMY_S 53264

int testcase (double v, int s){
 Block my_block ;
 Block *p = &my_block;
 p→x1 = 2.;
 p→x2 = 3.;
 p→y1 = 2.;
 p→y2 = 3.;

 Signal my_input;
 Signal *x = &my_input;

 Signal my_output;
 Signal *y = &my_output;

 x→v = v;
 x→s = s;

 y→v = DUMMY_V;
 y→s = DUMMY_S;

 transform(p,x,y);
}
```

FIGURE 3.20 – Forme générale des cas de test.

La troisième activité, l'activité de test manuel, appelée `test_value`, repose sur E-ACSL pour tester les spécifications `S8` à `S10` (les spécifications du groupe **VALUE**). Parce que les spécifications restantes ne s'intéressent qu'au signal d'entrée avec des paramètres de « `block` »

bien formés, notre cas de test suit la forme de la fonction « `testcase` » (figure 3.20). Comme nous avons trois spécifications différentes, nous avons besoin de trois cas de test qui utilisent les exécutions présentées dans la figure 3.21. Dans cette situation, le greffon « `witness` » exécute « `e-acsl_witnesses` » pour produire des témoins.

```
// Cas de test pour S8
int main(){
 testcase (1.5, 1);
}

// Cas de test pour S9
int main(){
 testcase (3.456, 1);
}

// Cas de test pour S10
int main(){
 testcase (2.769, 1);
}
```

FIGURE 3.21 – Cas de test pour `S8`, `S9` et `S10`.

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Consolidation Label
Lerror1	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror2	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror3	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror4	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror5	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror6	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror7	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗		P✗	P✗	✓
Llow1	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗		P✗	P✗	✓
Llow2	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗		P✗	P✗	✓
Lhigh	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓		P✗	P✗	✓
Lmedium	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	T✓	P✗	P✗	✓
Lvalidity	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	T?	P✓	P?	✓
Lok	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	T?	P✗	P✗	✓
Consolidation Specification	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**P**	**T**	**P**	**?**	

FIGURE 3.22 – Matrice de couverture après l'exécution de l'activité `test_value`.

Après avoir mis à jour la matrice de couverture avec les résultats de l'activité `test_value`, la matrice obtenue est présentée dans la figure 3.22. En les observant, nous pouvons déduire que chaque label est associé à au moins une spécification et que toutes les spécifications sauf `S12` sont associées à au moins un label. Par conséquent, seule une vérification pour `S12` est manquante. En réutilisant la même technique que ci-dessus, nous ajoutons le cas de test de la figure 3.23 en ne considérant que le label `Lvalidity`, le seul à pouvoir avoir une relation avec `S12`. La dernière activité, appelée `test_coverage`, réalise ce cas de test. Les résultats de cette activité sont combinés à ceux des activités précédentes pour construire la matrice de couverture finale de ce cas d'étude. Elle est présentée dans la figure 3.24.

Selon cette matrice, chaque spécification est impactée par au moins un label. D'après

```
int main(){
 testcase(20000000,1);
}
```

FIGURE 3.23 – Cas de test pour S12.

	S0	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12	Consolidation Label
Lerror1	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror2	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror3	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror4	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror5	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror6	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗	P✗		P✗	P✗	✓
Lerror7	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	P✗		P✗	P✗	✓
Llow1	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗		P✗	P✗	✓
Llow2	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗		P✗	P✗	✓
Lhigh	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓		P✗	P✗	✓
Lmedium	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	T✓	P✗	P✗	✓
Lvalidity	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✗	T?	P✓	P✓	✓
Lok	P✗	P✗	P✗	P✗	P✗	P✗	P✗	P✓	P✗	P✗	T?	P✗	P✗	✓
Consolidation Specification	P	P	P	P	P	P	P	P	P	P	T	P	P	

FIGURE 3.24 – Matrice de couverture après l'exécution de l'activité `test_coverage`.

notre définition de couverture (voir section 2.5), cela correspond à la satisfaction du critère de couverture fonctionnelle. Nous avons aussi une corrélation de chaque label avec au moins une spécification, ce qui satisfait le critère de couverture structurelle. Ainsi, cette campagne de vérification satisfait à la fois la couverture fonctionnelle et la couverture structurelle, ce qui nous permet de la conclure à une couverture complète.

Dans cet exemple, pour atteindre à une couverture complète, nous n'avons besoin que quatre cas de test, un pour vérifier la spécification S10, les trois autres pour compléter l'information de couverture aux résultats de preuve. Si nous testons ces spécifications, nous avons besoin de plus de cas de test que dans cet exemple. Cet exemple montre donc l'avantage de notre plateforme et notre critère de couverture. Dans la section suivante, nous voulons montrer l'équivalence de notre couverture et la couverture actuel pour test et pour preuve.

3.4 Norme DO-178

La norme DO-178C [RTC11a ; Bro+10 ; Moy+13] est la dernière version de la norme DO-178 définie pour le processus de vérification d'un logiciel avionique. Pour utiliser la couverture label-mutant dans le développement d'un tel logiciel, nous devons démontrer que notre notion est compatible avec la DO-178. Cette norme impose que la vérification d'un logiciel avionique satisfait la couverture structurelle et la couverture fonctionnelle. La notion de couverture

fonctionnelle d'une vérification par preuve est équivalente à celle d'une vérification par test. Néanmoins, la manière d'appliquer la notion de couverture structurelle à une vérification par preuve est différente de celle d'une vérification par test. Le test repose sur plusieurs critères de couverture structurelle (couverture des instructions, couverture de branche, couverture MC/DC, etc.) tandis que la preuve, selon la norme DO-333 qui est le complément « méthodes formelles » de la norme [RTC11b], approche la couverture structurelle par quatre objectifs alternatifs (sûreté des méthodes de preuve, complétude, flot de données et code étranger).

Notre couverture, la couverture label-mutant, propose une autre approche pour utiliser la preuve dans la vérification, différente de celle de la DO-333 [RTC11b]. Elle peut être vue comme une reformulation de la DO-178B [RTC92], l'ancienne version de la DO-178 qui ne propose aucune couverture pour la preuve. Nous proposons d'utiliser les différentes notions de critère de couverture structurelle correspondant aux différents niveaux A, B et C de la DO-178 pour la preuve. Ici, la manière de calculer ces couvertures est modifiée pour qu'elle soit adaptée à la fois au test et à la preuve.

Dans cette section, nous illustrons une campagne de vérification qui combine les activités de test et de preuve en nous reposant sur la couverture label-mutant. Nous comparons ensuite cette campagne avec une campagne par test utilisant les critères de couverture traditionnels.

3.4.1 Code inatteignable et code mort

Une instruction inatteignable est une instruction par laquelle aucune exécution ne passe. Une instruction morte est couverte par au moins une exécution mais ne correspond à aucune exigence définie. Une vérification utilisant notre couverture peut détecter ces instructions, auquel cas la vérification finale est incomplète. Une telle détection est exigée par la DO-178. Par exemple, un des quatre objectifs alternatifs proposés pour la vérification par preuve consiste à trouver ces instructions. Notre couverture ressemble à la DO-178 dans ce cas.

Certaines méthodes de preuve peuvent prouver l'inatteignabilité d'une instruction. Par exemple, [Bar+15] propose de combiner deux greffons de Frama-C, appelés EVA et WP, pour détecter les labels inatteignables. Ainsi, utiliser une telle méthode détecte non seulement les témoins de preuve mais aussi les labels inatteignables. Dans ce cas, le label correspondant est marqué par le témoin U comme sur la figure 3.25.

	`S1`	`S2`
L1	P✓	P✓
L2	U	U
Consolidation Specification	P	P

Consolidation Label
✓
U

FIGURE 3.25 – Exemple de label inatteignable.

Il existe une autre manière pour détecter la possibilité du code inatteignable ou du code mort dans la matrice de couverture. La figure 3.26 montre un exemple d'une telle situation. Dans la matrice de cette figure, les cellules du label `LQ2` sont marquées par `P✗` et la consolidation correspondante est marquée par `✗`. `LQ2` est ainsi un label qui n'est associé à aucune spécification définie. Soit il manque une spécification reliée à ce label, soit ce label représente un code mort ou un code inatteignable.

	S0	S1	Consolidation Label
LQ1	P✓	P✓	✓
LQ2	P✗	P✗	✘
Consolidation Specification	P	P	

FIGURE 3.26 – Autre exemple de label inatteignable ou mort.

3.4.2 Simulation d'une campagne de couverture compatible avec la DO-178B

Dans cette partie, nous comparons les résultats possibles obtenus lors d'une campagne combinée complète (utilisant à la fois une méthode de test et une méthode de preuve) avec une campagne de test complète. Une campagne complète est une campagne qui satisfait à la fois la couverture fonctionnelle et la couverture structurelle. La campagne de test satisfait ces couvertures via les critères présentés dans la DO-178B. La campagne combinée satisfait ces couvertures à l'aide de notre couverture.

D'après notre notion de couverture, une spécification S et un label L sont reliés si et seulement si les vérifications du code original et du mutant produisent un des couples de témoins suivants : `T` et `OMT`, `T` et `OMP`, `P` et `OMT` et `P` et `OMP`. Nous comparons à présent chaque couple de témoins ci-dessus aux résultats possibles d'une campagne de test.

`T` et `OMT` correspond à un cas de test qui vérifie la spécification S tout en couvrant le label L. Ce cas de test valide S dans le code original et invalide S dans le mutant. Il s'agit d'un cas de test pertinent pour vérifier cette spécification. En combinant ces témoins, nous pouvons conclure que toutes les spécifications sont vérifiées, et donc, que la couverture fonctionnelle est satisfaite. Le couple de témoin `T` et `OMT` correspond aussi au cas où le label L est couvert par un cas de test. En combinant ces témoins, nous pouvons conclure que tous les labels sont vérifiés par les cas de test, et donc, que la couverture structurelle est satisfaite. En conclusion, ce couple de témoin n'est rien d'autre qu'un cas de test dans une campagne de test dont les couverture sont calculées d'après la norme DO-178B.

`T` et `OMP` correspond à une situation dans laquelle on a à la fois un cas de test qui vérifie la spécification S et une preuve qui montre que cette spécification est invalide pour toutes

les exécutions du mutant associé au label L. Nous pouvons en déduire que le cas de test qui vérifie S dans le code original invalide S dans le mutant. Nous avons donc deux informations : le cas de test vérifie S et couvre le label L. Il est équivalent au cas `T` et `OMT` ci-dessus et est donc équivalent à un cas de test dans une campagne de test classique.

`P` et `OMT` est un couple de témoins représentant une preuve de la spécification S dans le code original et un test qui invalide S dans le mutant. Nous pouvons en déduire deux choses : d'une part, aucune exécution du code original n'invalide S et d'autre part, un cas de test invalide S dans le mutant. Dans cette situation, nous pouvons déduire que ce cas de test valide aussi S dans le code original. Il est donc semblable à un cas de test d'une campagne de test classique.

`P` et `OMP` montre que la spécification S est valide pour toutes les exécutions du code source et est invalide pour toutes les exécutions du mutant associé au label L. Donc, chaque cas de test qui peut tester S peut produire un couple de témoins `T` et `OMT`. Nous savons qu'il existe au moins un cas de test parce que, si aucune exécution n'est passée par S, nous pouvons avoir aussi un témoin `OP` et cette situation est considérée comme une erreur (voir la section 2.4). La campagne pour cette situation n'est pas complète parce qu'elle contient une erreur. C'est pourquoi la combinaison de `P` et `OMP` peut se comparer à une campagne de test qui teste S pour toutes les exécutions possibles. Cela correspond à la campagne de test la plus complète imaginable.

Nous avons démontré que chaque couple de témoins correspond aux résultats d'un ou de plusieurs cas de test. Une campagne combinée utilisant notre couverture peut donc simuler une campagne de test utilisant les critères de couverture présentés dans la norme DO-178B.

3.4.3 Correspondance avec les normes DO-178C et DO-333

Dans cette section, nous montrons comment notre couverture est compatible avec les quatre objectifs alternatifs indiqués dans la norme DO-333 [Bro+10 ; Moy+13] : sureté des méthodes de preuve, complétude, flot de données et code étranger.

Sûreté des méthodes de preuve consiste à garantir que les méthodes de preuve utilisées sont correctes. Dans notre travail, nous supposons que c'est effectivement le cas. Ce principe est donc indépendant de notre étude et reste à établir par ailleurs.

Complétude et flot de données sert à démontrer que toutes les spécifications sont complètes et que les spécifications contrôlent toutes les dépendances entre entrées et sorties. Notre couverture exige de démontrer une relation entre chaque spécification et chaque label correspondant à une portion du code. Si chaque spécification est associée à au moins un label et si chaque label est associé à au moins une spécification, nous pouvons donc garantir ces deux

objectifs. Plus précisément, notre couverture garantit les objectifs « complétudes » et « flot de données » de la norme DO-333.

Code étranger vise à garantir l'absence de code mort et de code inatteignable. Cet objectif est inclus dans notre notion de couverture (voir section 3.4.1).

Les objectifs de couverture annoncées dans la norme DO-333 sont implicites dans notre notion de couverture. Autrement dit, une campagne combinée fondée sur notre notion de couverture peut remplacer une campagne de preuve satisfaisant la norme DO-333.

3.5 Conclusion

Cette section a montré qu'une campagne combinée utilisant notre couverture peut simuler une campagne de test compatible avec la norme DO-178B et une campagne de preuve compatible avec la norme DO-333. Cela signifie qu'une campagne combinée utilisant notre couverture peut déduire les mêmes résultats qu'une campagne de test utilisant la norme DO-178B. Pourtant, il existe encore des développements à effectuer pour perfectionner notre méthode. Ces développements sont introduits dans le chapitre suivant.

Never perfect.
Perfection goal that changes.
Never stops moving.
Can chase, cannot catch.

Abathur - Starcraft II

4

Extensions Possibles

Notre nouvelle notion de couverture (couverture label-mutant) est maintenant définie. Nous avons aussi une plateforme qui réalise automatiquement la vérification, construit la matrice de vérification associée et une méthode d'interprétation de cette matrice. Néanmoins, cette plateforme est encore un prototype ayant un certain nombre de limitations. Nous présentons dans ce chapitre les extensions possibles pour la rendre plus opérationnelle.

Parmi ces extensions figurent notamment la prise en compte des boucles, des appels de fonctions, une phase de normalisation, une optimisation de la gestion des témoins OMP, ainsi que l'intégration de l'interprétation abstraite. Pour les boucles, nous proposons deux approches différentes. L'une propose d'ajouter un compteur qui compte le nombre d'itérations de la boucle et une condition permettant d'en sortir quand le nombre d'itérations est atteint. L'autre propose de considérer une boucle comme un appel de fonction. Pour l'appel de fonctions, nous proposons de stratifier les fonctions en niveaux et de les traiter tour à tour. La normalisation résout un problème spécifique à l'écriture des contrats, notamment lorsque nous utilisons des comportements ACSL. L'optimisation des témoins `OMP` permet de réduire encore plus le coût de la vérification. Enfin, nous proposons l'ajout à notre méthode d'une autre famille de méthode de preuve, l'interprétation abstraite. Par ailleurs, ce chapitre introduit aussi d'autres perspectives possibles pour notre travail.

4.1 Boucle

La première extension que nous envisageons est la prise en compte des boucles. Cette construction, inévitable en pratique, est néanmoins souvent un problème pour la vérification. Pour le test, la couverture de boucle la plus fréquente [AO08] consiste à tester chaque boucle avec 0,1 et au moins 2 itérations de la boucle. Pour la preuve, nous devons ajouter les invariants de boucle [FMV11] pour prouver les spécifications. Jusqu'à présent, tous les exemples que nous avons traités n'avaient pas de boucle.

Ici, nous introduisons un exemple d'une fonction C qui utilise la structure de boucle (voir la figure 4.1). Cette fonction est une vérification des valeurs d'un tableau d'entiers fourni en entrée de la fonction par la variable `t`. La variable `n` est le nombre d'éléments du tableau. Cette fonction retourne 0 si tous les entiers dans le tableau valent 0. Sinon, la fonction retourne 1.

```
/*@
 ...
*/
int all_zeros(int t[], int n)
{
 int k;
 for (k = 0; k  <  n; k++){
  if (t[k] != 0){
   return 0;
  }
 }
 return 1;
}
```

FIGURE 4.1 – Exemple de fonctions contenant d'une boucle.

4.1.1 Compteur d'itérations

Notre couverture repose sur des labels et des mutants. Les mutants sont utilisés pour représenter le critère de couverture de test pour que la preuve puisse être utilisée. C'est pourquoi ici, nous proposons d'utiliser les mutants pour représenter la couverture de boucle ci-dessus. Plus précisément, nous proposons d'utiliser deux opérateurs de mutation, « supprimer une instruction » et « modifier une expression » (voir la section 2.2). Pourtant, ces deux opérateurs ne suffisent pas pour expliciter la couverture de boucle (par exemple, « supprimer une instruction » ne permet pas d'expliciter la partie « au moins 2 itérations » de la couverture de boucle). Donc, nous introduisons un autre opérateur de mutation.

L'opérateur proposé dans cette section ajoute un compteur propre à chaque boucle et une instruction permettant d'en sortir. La figure 4.2 montre une boucle après l'application d'un tel opérateur de mutation. L'entier « `compteur_de_boucle` » est utilisé pour compter le nombre d'itérations de cette boucle. L'instruction « `break` » permet ensuite de sortir de la boucle lorsque le nombre d'itérations souhaité est atteint (dans l'exemple, ce nombre est 10 itérations).

```
/*@
 ...
*/
int all_zeros(int t[], int n)
{
 ...
 int compteur_de_boucle = 0;
 for (k = 0; k  <  n; k++){
  if(compteur_de_boucle  ≡  10){
   break;
  }
  compteur_de_boucle ++;
  if (t[k] != 0){
   return 0;
  }
 }
 return 1;
}
```

FIGURE 4.2 – `all_zeros` après ajout du compteur d'itérations.

Dans l'exemple, seules les exécutions dont les valeurs d'entrée permettent à la boucle d'être exécutée au moins 10 fois sont modifiées dans le mutant. Nous ne pouvons plus prouver les spécifications correspondant à de telles exécutions (si cela était possible, nous pourrions prouver leur opposition). Les autres spécifications peuvent être prouvées dans ce mutant. Cette situation ressemble à un test qui satisfait la partie « au moins 2 itérations » de la couverture de boucle. En conclusion, cet opérateur de mutation nous aide à représenter les parties « 1 itération » et « au moins 2 itérations » de la couverture de boucle. Pour la partie « 0 itération », nous pouvons utiliser un autre critère. Dans l'exemple de la figure 4.2, nous pouvons remplacer « 0 itération » par la couverture de l'instruction « `return 1;` ».

La proposition dans cette section utilise le mutant pour représenter la couverture de boucle. Cela ressemble à notre couverture. Néanmoins, il demande plus de travaux pour avoir une couverture de boucle commune pour la vérification par test et la vérification de preuve.

4.1.2 Invariant de boucle

Un invariant de boucle permet d'identifier une propriété qui est vraie avant d'entrer dans la boucle et qui est préservée par chaque itération. Il ressemble à une spécification entre les entrées et les sorties de chaque itération de boucle. Donc, nous pouvons voir une boucle comme un appel d'une fonction dont chaque spécification est un invariant de cette boucle. La vérification devient alors similaire au cas de l'appel de fonction dont une solution est présentée dans la prochaine section.

La figure 4.3 montre un exemple d'invariants de boucle. Deux invariants `S1` et `S2` sont considérés comme les spécifications d'une fonction dont la contenu est le corps de la boucle. Nous relions ensuite chaque invariant à chaque instruction dans le corps de boucle.

```
/*@
 ...
*/
int all_zeros(int *t, int n)
{
 int k;
 /*@
 @ loop invariant S1: 0 ≤ k ≤ n;
 @ loop invariant S2: \forall integer j; 0 ≤ j < k  ⇒  t[j] ≡ 0;
 @ loop variant n-k;
 */
 for (k = 0; k  <  n; k++){
  if (t[k] != 0){
   return 0;
  }
 }
 return 1;
}
```

FIGURE 4.3 – Invariant de boucle.

4.2 Appel de fonctions

Cette section discute de notre notion de couverture label-mutant dans le cas où une fonction, nommée `P`, appelle une autre fonction, nommée `Q`. Nous avons, dans ce cas, deux groupes de spécifications et deux groupes de labels qui ont des liens entre eux, ce qui pose la question suivante : notre notion de couverture exige-t-elle la corrélation entre les spécifications et les labels de la fonction `P` et les spécifications et les labels de la fonction `Q` ? Ce problème est appelé le problème de l'interprocéduralité.

4.2.1 Présentation du problème

Tout d'abord, nous allons présenter le problème en détail. Comme présenté précédemment, notre couverture label-mutant utilise des labels pour représenter un critère de couverture structurelle (couverture des instructions, couverture des branches, etc.). Chaque label doit être associé à un mutant qui modifie seulement les exécutions passant par ce label. En vérifiant le code original et celui du mutant, nous pouvons détecter quelle spécification a un effet sur le label associé au mutant.

Maintenant, considérons la figure 4.4. Cette figure contient deux fonctions `P` et `Q`. Pour la vérification de la fonction `Q`, nous pouvons appliquer normalement la couverture label-mutant. `P` appelle la fonction `Q`. Si nous considérons que les labels de la fonction `Q` sont aussi des labels de la fonction `P`, alors nous devons relier aussi les spécifications de la fonction `P` avec les labels de fonction `Q` par la vérification de `P`. Cette action exige plusieurs activités de vérification. Ensuite si nous avons une fonction `R` quelconque qui appelle la fonction `P`, la vérification des spécifications de `R` a aussi besoin des labels de `P` et de `Q`. En conséquence, la vérification risque rapidement de ne pas passer à l'échelle.

```
/*@
 behavior S:
 ensures \result  ≡  \abs(x+1);
*/
int P(int x)
{
 int n = Q(x+1);
 //@ label LP : (1 ≡ 1);
 //@ mutant LP : replace_assign (-1);
 return n;
}

/*@
 behavior S0:
  assumes x  >  0;
  ensures \result  ≡  x;

 behavior S1:
  assumes x = 0;
  ensures \result  ≡  0;

 behavior S2:
  assumes x  <  0;
  ensures \result  ≡  -x;
*/
int Q(int x)
{
 if(x  ≥  0){
  //@ label LQ1 : (1 ≡ 1);
  //@ mutant LQ1 : replace_assign (-1);
  return x;
 }else{
  //@ label LQ2 : (1 ≡ 1);
  //@ mutant LQ2 : replace_assign (-1);
  return -x;
 }
}
```

FIGURE 4.4 – Exemple d'appel de fonction.

Un autre problème, mais qui va nous guider vers une solution, vient des méthodes de preuve. Certaines méthodes de preuve (par exemple, la vérification déductive effectuée par le greffon WP [Bau+b] de la plateforme Frama-C) prouve les spécifications de `P` en utilisant seulement les propriétés (spécifications) de la fonction `Q`. Ainsi, même sur un mutant de `Q`, l'essai de preuve de `P`, qui repose sur les propriétés inchangées de `Q`, ne modifie pas ses résultats. Par conséquent, aucune information de couverture n'est générée dans cette situation.

En revanche, la situation ci-dessus suggère aussi une solution : la vérification ne relie pas directement les spécifications de `P` aux labels de `Q` mais elle relie plutôt les spécifications de `P` aux spécifications de `Q`. Cette solution est détaillée dans la section suivante.

4.2.2 Couverture par niveau

La couverture par niveau est une solution au problème de l'interprocéduralité. C'est une extension de la couverture label-mutant. Elle commence par analyser la profondeur des fonctions dans le graphe d'appels des fonctions. Nous distinguons alors deux cas :

Fonctions feuilles (Niveau 0) Les fonctions feuilles n'appellent aucune fonction. Nous pouvons appliquer normalement la couverture label-mutant pour les vérifier.

Fonctions nœuds (Niveau n)$_{n>0}$ Les fonctions nœuds appellent au moins une fonction. Dans ce cas, nous devons ajouter une relation spécification-spécification (couverture par niveau), décrite ci-après, en combinant des relations label-spécification (couverture label-mutant).

Une relation spécification-spécification lie chaque spécification de la fonction vérifiée avec une de celles de la fonction appelée. La spécification de la fonction appelée est alors considérée comme un label de la fonction vérifiée.

Retournons à l'exemple présenté en figure 4.4. La fonction `P` contient une spécification `S` qui appelle une fonction `Q` contenant trois spécifications `S0`, `S1` et `S2`. Pour déterminer l'impact de `S0`, `S1` et `S2` sur `S`, nous créons des mutants de la fonction `Q` et les utilisons pour vérifier `S`.

Nous proposons ici un opérateur qui applique deux modifications sur la fonction `Q`. La première modification est utilisée pour la preuve et la seconde modification est nécessaire pour le test. Elles sont présentées ci-après.

Première modification : nous remplaçons la spécification de `Q` par son opposition (cf. figure 4.5). Comme plusieurs méthodes de preuve ne dépendent que des spécifications de `Q` pour prouver les spécifications de `P`, la modification de spécification est suffisante. Néanmoins, cette modification de spécification n'est pas suffisante pour les méthodes de test. En effet, le test exige le code de `Q` mais pas ses spécifications pour tester la spécification `S` de la fonction

P. Cela conduit à la seconde modification.

```
/*@
 behavior S0_opp:
  assumes x  >  0;
  ensures !(\result  ≡  x);
*/
```

FIGURE 4.5 – Modification de la spécification S0.

Seconde modification : nous mutons aussi le code de Q. Supposons que la fonction appelée Q est déjà vérifiée et a une matrice de couverture (cf. figure 4.6). Grâce à cette matrice, nous identifions un label et son mutant qui sont associés à chaque spécification de Q. Nous utilisons alors ces mutants pour la vérification de P. Selon la matrice de la figure 4.6, la spécification S0 est associée au label LQ1. Nous considérons la spécification S0 comme le label. Ainsi, le mutant du label LQ1 devient le mutant associé au nouveau label (label S0) pour tester la fonction P.

	S0	S1	S2	Consolidation Label
LQ1	P✓	P✓	P✗	✓
LQ2	P✗	P✗	P✓	✓
Consolidation Specification	P	P	P	

FIGURE 4.6 – Matrice de couverture pour la fonction Q.

Nous pouvons distinguer deux cas particuliers en fonction du mutant choisi. D'une part, la vérification de la fonction appelée (la fonction Q dans l'exemple) peut montrer qu'une spécification impacte un ou plusieurs labels. Dans ce cas, pour vérifier la fonction courante (la fonction P dans l'exemple), nous choisissons seulement un label pour combiner le mutant associé à ce label avec le mutant de la spécification. D'autre part, lorsqu'un seul label est corrélé à au moins deux spécifications, nous pouvons combiner le mutant associé à ce label avec le mutant de chaque mutation pour vérifier la fonction courante.

Dans la couverture par niveau, l'important est la relation spécification-spécification. Par exemple, après la vérification de la fonction P, les résultats doivent montrer la relation entre la spécification S de la fonction P et les spécifications S0, S1 et S2 de la fonction Q.

Autre avantage de la mutation à deux phases : elle nous incite à adapter les différentes exigences des méthodes de vérification. Comme présenté au début de la section, certaines méthodes de vérification prouvent la spécification de P en utilisant seulement les spécifications de Q. Ici, les spécifications S0, S1 et S2 de la fonction Q sont utilisées comme lemmes pour prouver la spécification S de la fonction P. Ainsi, en changeant S0 par son mutant S0', nous pouvons en déduire qu'il existe une relation entre S et S0. Si S est encore prouvée, alors S0 n'a aucune corrélation avec S. En revanche, si la vérification trouve un contre-exemple pour S alors S impacte S0. Avec les autres méthodes qui ont besoin du code de la fonction, la

mutation à deux étapes produit aussi un mutant de code. L'impact des deux spécifications `S` et `S0` est établi indirectement via le code muté (par exemple, le mutant associé au label `LQ1` a une corrélation avec `S0` d'après la matrice de mutation présentée en figure 4.6).

Il n'y a pas besoin d'une nouvelle matrice de couverture pour synthétiser les résultats par niveau. En effet, puisque nous considérons les spécifications de la fonction appelée `Q` comme les labels de la fonction `P`, il nous suffit d'ajouter des lignes dans la matrice de couverture de `P`, chaque ligne correspondant à une spécification de `Q`. La figure 4.7 présente un exemple schématique de la matrice pour la vérification de la fonction `P`.

	Spécification `S`	consolidation de label
Spécification `S0` de `Q`		
Spécification `S1` de `Q`		
Spécification `S2` de `Q`		
Label `LP` de `P`		

consolidation de spécification	

FIGURE 4.7 – Exemple d'une matrice de couverture pour la vérification de la fonction `P` appelant la fonction `Q`.

4.2.3 Conclusion

La couverture par niveau est nécessaire pour la couverture label-mutant. Néanmoins, nous ne considérons pas les fonctions récursives ici puisque nous ne pouvons pas déterminer le niveau d'une telle fonction. Il demande donc plus de travaux pour mettre en œuvre la couverture de cette section.

4.3 Normalisation des spécifications

Dans notre contexte, une spécification doit être écrite sous la forme $S \stackrel{\text{def}}{=} H \Rightarrow C$, appelée forme normale, dans laquelle H est une hypothèse et C est une conclusion (cf. section 1.3.3). H ne contient que les éléments portant sur l'état initial (les conditions des entrées) et C ne contient que les éléments portant sur l'état final (les relations entre la sortie et les entrées). Néanmoins, en pratique, un ingénieur de vérification n'exprime pas toujours les spécifications sous cette forme. Une phase de normalisation est donc nécessaire pour isoler les éléments de l'état initial et les éléments de l'état final et construire les spécifications sous la forme exigée.

4.3.1 Langage E-ACSL et description du problème

Nous utilisons le langage E-ACSL [Sig ; DKS13] pour exprimer nos spécifications. E-ACSL propose une notion appelée « `behavior` » utilisée pour exprimer une spécification. Un « `behavior` » comporte notamment deux éléments : les clauses « `assumes` » et « `ensures` ». D'après la formulation de E-ACSL, un « `behavior` » décrit par « `assumes` A `ensures` B » est équivalent à la formule $A \Rightarrow B$. Cependant, elle n'est pas nécessairement en forme normale. Voici un exemple :

```
/*@
 ...
 behavior S1:
 assumes u > v;
 ensures \result ≡ v ⇒ a < v ⇒ \result ≡ u;
 ...
*/
int unknow(int a, int u, int v)
{
 ...
}
```

La spécification `S1` de la fonction `unknow` est exprimée comme suit :

$$\texttt{u} > \texttt{v} \Rightarrow \backslash\texttt{result} \equiv \texttt{v} \Rightarrow \texttt{a} < \texttt{v} \Rightarrow \backslash\texttt{result} \equiv \texttt{u}.$$

Dans cette spécification, l'élément dédié à l'hypothèse (par exemple, $\texttt{a} < \texttt{v}$) est mélangé aux éléments dédiés à la conclusion (par exemple, $\backslash\texttt{result} \equiv \texttt{v}$). Néanmoins, il existe une autre forme de la spécification `S1` définie comme suit :

```
/*@
 ...
 behavior S1:
 assumes u > v && a < v;
 ensures \result ≡ v ⇒ \result ≡ u;
 ...
*/
int unknow(int a, int u, int v)
{
 ...
}
```

Dans cette forme, l'hypothèse contient seulement les éléments portant sur l'état initial ($\texttt{u} > \texttt{v}$ et $\texttt{a} < \texttt{v}$) ainsi que la conclusion contient seulement les éléments portant sur l'état final ($\backslash\texttt{result} \equiv \texttt{v}$ et $\backslash\texttt{result} \equiv \texttt{u}$). Cela est une forme normale de la spécification `S1` que nous voulons utiliser dans notre travail. Il nous faut donc d'une méthode pour normaliser les spécifications.

4.3.2 Mise en forme normale

Dans cette section, nous définissons une méthode pour transformer une formule quelconque en une formule équivalente en forme normale. La transformation met tout d'abord la spécification en forme normale conjonctive. Chaque clause disjonctive est ensuite transformée en une forme $H \Rightarrow C$ dans laquelle H ne contient que des éléments sur l'état initial et C ne contient que des éléments sur l'état final. Le résultat de cette transformation est une formule en forme normale.

Dans la première étape, nous définissons la forme normale conjonctive de notre formule originale. Une forme normale conjonctive doit avoir une distinction entre les termes dédiés à l'hypothèse et ceux dédiés à la conclusion. Autrement dit, elle a la forme suivante :

$$\bigwedge_{i \in \mathbb{N}} (H_i \vee C_i).$$

avec H_i les termes dédiés à l'hypothèse liés par les connecteurs logiques $\vee$ et C_i les termes dédiés à la conclusion également liés par des connecteurs logiques $\vee$.

Retournons à l'exemple ci-dessus. Nous y transformons la spécification `S1` en une forme normale conjonctive. En constatant que les formules contenant `\result` portent nécessairement sur l'état final, voici la transformation :

$$\begin{array}{lcl} \texttt{S1} & = & \texttt{u} > \texttt{v} \Rightarrow \backslash\texttt{result} \equiv \texttt{v} \Rightarrow \texttt{a} < \texttt{v} \Rightarrow \backslash\texttt{result} \equiv \texttt{u} \\ & = & \neg(\texttt{u} > \texttt{v}) \vee \neg(\texttt{a} < \texttt{v}) \vee \neg(\backslash\texttt{result} \equiv \texttt{v}) \vee \backslash\texttt{result} \equiv \texttt{u} \end{array}$$

Après modification, nous obtenons donc la formule en forme normale conjonctive suivante :

$$\neg(\texttt{u} > \texttt{v}) \vee \neg(\texttt{a} < \texttt{v}) \vee \neg(\backslash\texttt{result} \equiv \texttt{v}) \vee \backslash\texttt{result} \equiv \texttt{u}$$

Nous remarquons que chaque clause conjonctive $(H_i \vee C_i)$ de forme normale peut être transformée sous la forme que nous souhaitons, c'est-à-dire $(H \Rightarrow C)$. Donc, nous modifions chaque clause conjonctive de la formule ci-dessus pour obtenir la formule suivante :

$$(\texttt{u} > \texttt{v} \wedge \texttt{a} < \texttt{v}) \Rightarrow (\neg(\backslash\texttt{result} \equiv \texttt{v}) \vee \backslash\texttt{result} \equiv \texttt{u})$$

Chaque clause $(\neg H_i) \Rightarrow C_i$ est une spécification conforme à notre écriture. En utilisant le langage ACSL, nous pouvons transformer chaque clause en un « `behavior` » représentant une spécification de la fonction.

La formule en forme normale de l'exemple contient une clause conjonctive. Autrement dit, la spécification `S1` est normalisée en une spécification :

$$\texttt{S1\_norm} \quad = \quad (u > v \wedge a < v) \Rightarrow (\neg(\backslash result \equiv v) \vee \backslash result \equiv u);$$

La clause peut être écrite en ACSL. Le résultat est présenté dans la figure 4.8.

```
/*@
 ...
 behavior S1_norm:
 assumes u  >  v && a  <  v ;
 ensures !(\result  ≡  v) || \result  ≡  u;

 ...
*/
int unknow(int a, int u, int v)
{
 ...
}
```

FIGURE 4.8 – Spécifications ACSL après normalisation.

En conclusion, nous pouvons normaliser une formule logique quelconque. C'est essentiel pour notre notion de couverture parce que la matrice de couverture repose fortement sur les spécifications définies. Utiliser une spécification en forme normale $H \Rightarrow C$ nous aide à identifier plus vite la propriété représentée par cette spécification et, à l'aide de la matrice de couverture, à identifier les instructions ou les chemins d'exécution exercés par cette propriété.

4.4 Optimisation des témoins OMP

Comme présenté précédemment, il est possible d'obtenir une situation dans laquelle l'opposition d'une certaine spécification S est prouvée par le mutant associé à un certain label L (ce qui est représenté par un témoin d'opposition de preuve `OMP`). Dans ce cas, si cette spécification est prouvée ou testée par le code original, nous pouvons conclure immédiatement que cette spécification S impacte ce label L. Cependant, il est souvent difficile d'établir une preuve d'opposition.

4.4.1 Description du problème

Nous illustrons le problème sur l'exemple ci-dessous.

Considérons la spécification ACSL et le code d'une fonction appelée `one_code_error` présentés dans la figure 4.9. Cette fonction est une version alternative de la fonction `transform` qui renvoie seulement un signal d'erreur (-1) pour n'importe quelle erreur. Comme présenté précédemment pour la fonction `transform`, nous avons défini un groupe, appelé **ERR**, contenant 7 spécifications, chacune caractérisée par un code d'erreur différent. Pour la version `one_code_error`, nous définissons et remplaçons ces 7 spécifications par une seule spécification, `error_mode`, caractérisée seulement par un signal d'erreur (-1) pour toutes les erreurs du code.

La spécification `error_mode` peut être prouvée en utilisant le greffon WP de Frama-C.

```
#define BOUND_SIGNAL 100000000.

#define BCK(p) \
 ( \
  (-BOUND_SIGNAL  ≤  (p)→x1) \
  && ((p)→x2  ≤  BOUND_SIGNAL) \
  && ((p)→x1  <  (p)→x2) \
  && (-BOUND_SIGNAL  ≤  (p)→y1  ≤  BOUND_SIGNAL) \
  && (-BOUND_SIGNAL  ≤  (p)→y2  ≤  BOUND_SIGNAL) \
  )


/*@
...

 behavior error_mode:
 assumes !BCK(p);
 assigns \nothing;
 ensures \result  ≡  -1;

...
*/

int one_code_error (Block *p, Signal *x, Signal *y)
{
Pre:
 l0: if(p→x1  <  -BOUND_SIGNAL){
  //@ label Lerror1: 1  ≡  1;
  //@ mutant Lerror1: replace_assign (0);
  l1: return -1;
 }


 l2: if(p→x2  >  -BOUND_SIGNAL){
  //@ label Lerror2: 1  ≡  1;
  //@ mutant Lerror2: replace_assign (0);
 l3: return -1;
 }

...
Post:}
```

FIGURE 4.9 – Extrait de la fonction `one_code_error`.

Autrement dit, nous avons un témoin de preuve pour `error_mode`. D'après notre notion de couverture (voir le chapitre 2), nous devons vérifier le mutant associé à chaque label et, si nous réussissons à établir les témoins d'opposition d'un mutant associé à un label L, nous pouvons conclure que la spécification impacte L. Ici, nous vérifions le mutant associé au label `Lerror1` pour déterminer la corrélation entre ce label et `error_mode`. Nous pouvons obtenir deux témoins d'opposition : le témoin d'opposition de test et le témoin d'opposition de preuve. Le témoin d'opposition de test est obtenu en testant le mutant par une exécution passant par le label `Lerror1`. Dans ce cas, l'ingénieur doit déterminer un cas de test. En revanche, le témoin d'opposition de preuve ne peut pas être créé pour la vérification du mutant associé au label `Lerror1`. En effet, l'essai de preuve de l'opposition de la spécification `error_mode` est toujours en échec parce que cette fonction contient des contre-exemples pour l'opposition de la spécification `error_mode` (par exemple, une exécution qui passe par le label `Lerror2`). Néanmoins, ces contre-exemples n'apportent aucune valeur ajoutée car ils ne passent pas par le label considéré `Lerror1`. Il nous faut donc un moyen pour éviter ces contre-exemples dans la preuve.

4.4.2 Blocage des chemins non-pertinents

Dans cette partie, nous envisageons un moyen pour permettre aux outils de preuve d'éviter les exécutions qui ne passent pas par le label considéré dans le code du mutant. Pour cela, nous utilisons « `assert false;` » pour bloquer les chemins inatteignables depuis le label considéré. Nous expliquons cette solution sur des graphes de flot de contrôle. La figure 4.10 montre le flot de contrôle de la fonction `one_code_error`.

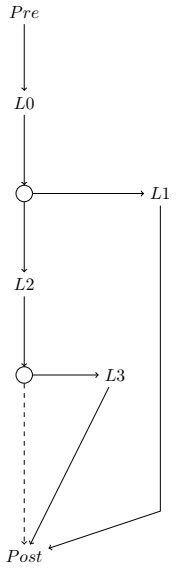


FIGURE 4.10 – Flot de contrôle de la fonction `one_code_error`.

Considérons deux labels $L1$ et $L3$. En bloquant les chemins d'exécution inatteignables depuis ces labels, nous obtenons le graphe de flot de contrôle de la figure 4.11 pour le label $L1$

et 4.12 pour le label $L3$. Dans ces figures, les chemins qui ne conduisent pas au label considéré sont marqués par **X**. La preuve ne prend donc pas ces chemins en considération pour prouver les spécifications de `one_code_error`, ce qui nous permet d'obtenir un témoin OMP.

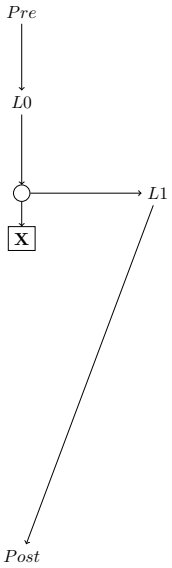


FIGURE 4.11 – Flot de contrôle après bloquer L1.

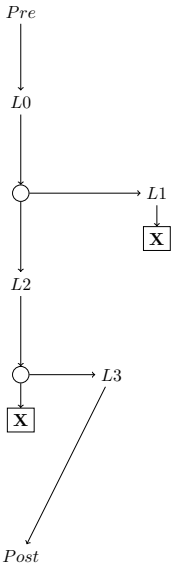


FIGURE 4.12 – Flot de contrôle après bloquer L3.

4.5 Interprétation abstraite

Notre nouvelle notion de couverture, présentée dans le chapitre 2, fonctionne avec toutes les méthodes de vérification. Néanmoins, l'implémentation actuelle, présentée dans le chapitre 3, ne prend en compte que le greffon WP de Frama-C qui réalise la vérification déductive et E-ACSL, une analyse dynamique qui représente le test. Nous proposons d'intégrer à notre plateforme le greffon EVA [BBY17 ; Büh+] implémentant une méthode de vérification par interprétation abstraite [CC77].

4.5.1 EVA

Le greffon EVA de Frama-C est une implémentation d'une analyse de valeurs par interprétation abstraite. Elle sert à calculer automatiquement l'ensemble des valeurs possibles pour chaque variable à chaque point de programme et, à partir de ces ensembles, à déduire la validité de chaque spécification $H \Rightarrow C$. EVA est souvent utilisé pour prévenir les erreurs à l'exécution comme les divisions par zéro ou pour détecter des chemins ou des instructions inatteignables.

Pour vérifier une fonction avec EVA, comme pour le greffon E-ACSL de Frama-C, nous

devons définir une fonction « `main` » à partir de laquelle la fonction vérifiée est exécutée. Pourtant, cette exécution représente en réalité un ensemble d'exécutions possibles en affectant par exemple un intervalle de valeurs au lieu d'une valeur précise à chaque variable. EVA évalue les chemins d'exécution dont l'exécution est établie lorsque nous remplaçons la variable par une valeur de l'intervalle. C'est un calcul par sur-approximation. À partir des chemins d'exécution trouvés, nous pouvons estimer les valeurs des variables à tout point du programme et, à partir de ces valeurs, définir la validité de chaque spécification ACSL.

Voici un exemple d'une fonction « `main` » de EVA :

```
// exécution de la fonction "range(a,u,v)"
int main(){
 range (Frama_C_interval(0,10), 5, 3);
 // initialisation de a avec l'intervalle [0,10]

 range (8, Frama_C_interval(0,10), 3);
 // initialisation de u avec l'intervalle [0,10]

 range (8, 5, Frama_C_interval(0,10));
 // initialisation de v avec l'intervalle [0,10]

 return 0;
}
```

Dans l'exemple ci-dessus, nous effectuons trois appels à une fonction `range` ayant trois paramètres. Dans chaque appel, une des trois variables est spécifiée par un intervalle [0,10]. Pour chaque exemple, EVA calcule et définit la validité de chaque spécification de la fonction `range`.

4.5.2 Avantage de EVA pour notre plateforme

EVA offre plusieurs atouts pour être intégré à notre plateforme.

Considérons la forme que nous utilisons pour une spécification $H \Rightarrow C$. L'hypothèse H caractérise l'ensemble des états mémoire en un point de programme donné P. Un état mémoire contient les informations de la mémoire comme les variables et leurs valeurs. Comme EVA sur-approxime l'ensemble des valeurs possibles pour chaque variable en chaque point de programme, nous pouvons évaluer l'ensemble des états mémoire possibles au point de programme P et comparer cet ensemble à celui caractérisé par H pour en déduire si H est valide ou non.

Considérons par exemple, une fonctions `abs` retournant la valeur absolue d'un entier de l'architecture 16-bit donné en paramètre. Une de ses spécifications est la suivante : `x` $> 0 \Rightarrow$ `\result` $>$ `x`. Nous pouvons prouver cette spécification en utilisant EVA à partir du point d'entrée suivant :

```
int main(){
 abs (Frama_C_interval(1,32767));
}
```

32767 est la valeur maximale du type entier (`int`) du programme C dans l'architecture

16-bit. Ici, EVA prouve cette spécification pour toute valeur d'entrée `x` de cette fonction telle que $\texttt{x} \in [\texttt{1}, \texttt{32767}]$.

Un autre avantage de EVA pour notre notion de couverture est lié aux labels. Considérons une spécification $S = H \Rightarrow C$ d'une fonction f et un intervalle I pour son entrée tel que chaque état mémoire entrée de f est un état mémoire possible pour l'hypothèse H, ainsi que des labels $(L_i)_{0 \leq i \leq n}$. Nous exécutons EVA avec $f(I)$ pour vérifier la spécification S. En exécutant EVA, nous pouvons obtenir deux informations : la validité de S et les labels qui n'ont aucune corrélation avec S.

Retournons à la fonction `abs` ci-dessus. Elle a trois spécifications :

$$
\begin{aligned}
S1 &= \texttt{x} > \texttt{0} \Rightarrow \backslash\texttt{result} > \texttt{0} \\
S2 &= \texttt{x} \equiv \texttt{0} \Rightarrow \backslash\texttt{result} \equiv \texttt{0} \\
S3 &= \texttt{x} < \texttt{0} \Rightarrow \backslash\texttt{result} < \texttt{0}
\end{aligned}
$$

Nous prouvons ces spécifications avec EVA. Voici la matrice de couverture qu'on pourrait obtenir pour cette situation :

	S1	S2	S3
L1	P✓	P✓	✗
L2	✗	✗	P✓

La marque ✗ entre une spécification et un label signifie qu' EVA n'utilise pas la portion du code associée à ce label pour prouver cette spécification.

En combinant les deux avantages sus-mentionnés, nous pouvons obtenir la situation suivante : nous exécutons une fonction sur un intervalle qui correspond à l'hypothèse H d'une spécification. Puis en utilisant EVA, nous pouvons trouver facilement les labels qui n'ont aucune corrélation avec cette spécification. Cela nous aide à marquer tout de suite quelques cellules de la matrice de couverture avec cette information de couverture. Ainsi, nous n'avons plus à considérer ces cellules lors des activités de vérification suivantes.

4.6 Autres Perspectives

Nous proposons dans cette section d'autres perspectives possibles pour lesquelles nous ne développons pas de solutions particulières.

Vérification de modèle Il existe trois familles de méthodes de vérification statiques. La vérification déductive est l'une d'elle et est au cœur de notre travail. Nous proposons aussi une extension à l'interprétation abstraite. La vérification de modèle [Mer08] est la troisième

famille des méthodes de vérification statiques. Il serait intéressant d'étendre notre travail à cette méthode.

Hyperlabel Les hyperlabels [Mar+17a ; Mar+17b] sont une extension des labels qui permettent notamment de représenter plus de critères de couverture structurelle. Nous pourrions utiliser les hyperlabels pour maximiser le nombre de critères que nous traitons.

Développement de mutant Nous proposons ici une nouvelle utilisation des opérateurs de mutation. Notre approche pourrait être enrichie par l'utilisation d'un catalogue de mutants. Dans ce catalogue, pour chaque couple label-instruction, nous devons déterminer la modification appropriée. Il faudrait également s'intéresser à la problématique de l'équivalence des mutants [Dev+17 ; Car+18] afin d'améliorer encore l'efficacité de l'approche. En effet, puisque nous définissons manuellement les opérateurs de mutation, nous pouvons aujourd'hui éviter le problème d'équivalence des mutants. Cependant, dans l'avenir, ces opérateurs doivent être ajoutés automatiquement selon le critère de couverture choisi. Il deviendrait alors possible de créer un mutant équivalent au code original ou de créer des mutants équivalents. Autrement dit, ce problème deviendrait alors probablement réel dans nos travaux.

Couverture de l'espace d'état La couverture des ensembles des états [Bec+18] est intéressante pour nos travaux. Cette couverture est appliquée sur des preuves partielles. Autrement dit, en utilisant cette couverture, même si nous ne réussissons pas à prouver une spécification, nous pouvons en déduire des états qui sont déjà vérifiés. En combinant cette couverture avec notre propre notion de couverture, nous pourrions obtenir des résultats encore plus précis, résolvant ainsi plus de situations. Par exemple, considérons la spécification dont la conclusion est « $\texttt{\textbackslash result} \equiv \texttt{a} \vee \texttt{\textbackslash result} \equiv \texttt{b}$ ». Nous pourrions déterminer les labels liés uniquement à « $\texttt{\textbackslash result} \equiv \texttt{a}$ » ou à « $\texttt{\textbackslash result} \equiv \texttt{b}$ » selon la preuve partielle obtenue, permettant ainsi de raffiner notre notion de couverture.

Gestion des correctifs Les travaux sur les témoins de Beyer et al. [Bey+15 ; Bey+16 ; BD16] permettent d'envisager une autre extension pour notre travail : la réutilisation des témoins quand nous vérifions le code après correction d'une erreur. Elle permettrait de ne revérifier que les spécifications et les portions du code liées à cette correction.

En effet, un processus de vérification trouve souvent des erreurs. Dans notre travail, ces erreurs sont enregistrées par des témoins. Ensuite, ces erreurs doivent être corrigées. Après correction, nous devons revérifier le code. Idéalement, nous souhaiterions que les outils de vérification puissent lire les témoins et ne revérifier que les portions du code liées aux erreurs corrigées.

Problème du test unitaire Plus tôt, nous avons évoqué une solution au problème des appels de fonction. Elle ne fonctionne néanmoins que pour le test d'intégration. Pour le test

unitaire [Ran+99], nous n'avons pas encore de bonne solution. Pour une fonction `P` qui appelle une fonction `Q`, le test unitaire de `P` utilise le remplaçant de la fonction `Q` suivant :

```
int K;
Tr INJ[];
Tp COL[];
Tr Q (Tp p){
 COL[K] = p;
 Tr r = INJ[K];
 K++;
 return r;
}
```

Cette fonction utilise un compteur `K` et deux tables `INJ` et `COL`. `COL` enregistre les entrées de la fonction `Q`. `INJ` contient des sorties injectées avant test.

Le test unitaire fonctionne en boîte noire, c'est-à-dire sans avoir connaissance du code de la fonction `Q`. Nous ne pouvons alors pas appliquer la couverture label-mutant ici parce que nous ne connaissons pas `Q` et que nous ne pouvons donc pas muter son code. Nous avons donc besoin de définir une autre notion de couverture dans cette situation. Par exemple, nous pourrions définir une mutation des valeurs injectées, comme si c'était des labels.

Conclusion

L'utilisation combinée du test et de la preuve est rendue difficile par l'absence d'une notion de couverture commune. Cette couverture est nécessaire pour rendre compte de la qualité de la vérification effectuée. Elle est d'ailleurs un objectif incontournable des standards de certification du logiciel dans les domaines critiques tels que l'aéronautique. Ma thèse vise à résoudre ce problème en proposant une nouvelle notion de couverture, la couverture label-mutant. Elle fournit aussi une méthode d'interprétation qui permet d'optimiser le pilotage d'une campagne de vérification et une plateforme qui réalise automatiquement la vérification et construit la matrice de couverture. La méthode d'interprétation a été implémentée comme un prototype au sein de la plateforme de vérification Frama-C.

La couverture label-mutant La couverture label-mutant est une combinaison de la couverture de label et de la couverture par mutation. Le label sert à représenter le critère de couverture structurelle. Le mutant vise à mettre en relation chaque label avec chaque spécification. Plus précisément, chaque label est renforcé par un mutant. Ensuite, nous vérifions chaque spécification pour le code d'origine et pour chaque mutant. Si le résultat de la vérification d'une spécification diffère entre le code origine et le mutant, alors nous pouvons conclure que la spécification vérifiée est bien liée au label.

Un résultat d'une vérification est formulé et collecté grâce à une notion de témoin. Chaque résultat de vérification, soit par test soit par preuve, produit un témoin. En utilisant la notion de témoin, nous trouvons qu'un mutant «tué» dépend fortement du résultat de la vérification mais pas de la méthode de vérification utilisée. Autrement dit, nous pouvons mêler la vérification par test et la vérification par preuve pour tuer les mutants. En conclusion, grâce à la couverture label-mutant, nous pouvons utiliser n'importe quelle méthode de vérification (soit test, soit preuve, soit leur combinaison) dans le processus de vérification.

Cette couverture est analysée et comparée par rapport aux objectifs de la norme DO-178, standard de certification pour le logiciel aéronautique. La norme DO-178 exige que toutes les spécifications soient vérifiées (couverture fonctionnelle) et que la vérification couvre toute la structure du code (couverture structurelle). Notre couverture permet de répondre à ces objectifs grâce à la mise en lumière des liens entre spécifications et labels.

Nous proposons aussi la notion de matrice de couverture qui permet d'expliciter et de synthétiser les résultats. Tout d'abord, les témoins produits par les vérifications sont analysés pour déterminer les informations de vérification des spécifications et les informations de cou-

verture des labels. Ces informations sont classées et consolidées dans la matrice de couverture de manière automatique et synthétique. Nous pouvons ainsi déduire les spécifications qui impactent au moins un label et les labels qui ont une relation avec au moins une spécification. Cela donne une vision globale de la satisfaction des couvertures fonctionnelle et structurelle obtenues.

Méthode et plateforme expérimentale. Comme présenté précédemment, la couverture label-mutant nous autorise à choisir librement la méthode de vérification utilisée. Néanmoins, le choix de la méthode de vérification utilisée affecte l'effort de vérification. Nous avons également proposé un cycle qui cherche à guider le processus de vérification en privilégiant la vérification automatique. Ce cycle repose sur deux éléments : une plateforme qui réalise automatiquement la vérification et génère la matrice de vérification, ainsi qu'une méthode d'interprétation qui s'appuie sur la matrice de couverture pour guider la campagne.

Une prototype de la plateforme est implémenté au sein de Frama-C. Un ingénieur peut utiliser la plateforme pour contrôler la vérification et peut intervenir si nécessaire. Par exemple, il peut lire la matrice de couverture et appliquer la méthode d'interprétation. Au cas où la vérification n'est pas complète, l'ingénieur peut ensuite ajouter des activités de vérification additionnelles. La plateforme, après configuration, peut réaliser seulement les activités additionnelles.

Nous discutons également d'un certain nombre d'extensions à notre couverture et à notre plateforme, afin d'enlever certaines limites. Nous présentons des ébauches de solutions pour certaines d'entre elles, qui devraient notamment permettre de faciliter le passage à l'échelle de nos travaux.

A

Fonction `transform`

L'implémentation des structures

```
typedef struct {double v; int s;} Signal;

typedef struct { double x1,x2,y1,y2;} Block;
```

L'implémentation de la fonction

```
#define BOUND_SIGNAL 100000000.
#define EPS_SIGNAL 0.00001
#define EPS 0.0001

#define BCK(p) \
 ( \
  (-BOUND_SIGNAL  ≤  (p)→x1)  \
  && ((p)→x2  ≤  BOUND_SIGNAL)  \
  && ((p)→x1  <  (p)→x2) \
  && (-BOUND_SIGNAL  ≤  (p)→y1  ≤  BOUND_SIGNAL)  \
  && (-BOUND_SIGNAL  ≤  (p)→y2  ≤  BOUND_SIGNAL)  \
 )

#define ERROR (p,x,y) \
 ( \
  ((y→v - p→y1) / (p→y2 - p→y1)) - ((x→v - p→x1) / (p→x2 - p→x1))  \
 )

/*@

////////////////////////////////
// groupe ERR :
```

```
 behavior S0:
 assumes (p→x1  <  -BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -1;

 behavior S1:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes (p→x2  >  BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -2;

 behavior S2:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes !(p→x2  >  BOUND_SIGNAL);
 assumes (p→x1  ≥  p→x2);
 assigns \nothing;
 ensures \result == -3;

 behavior S3:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes !(p→x2  >  BOUND_SIGNAL);
 assumes !(p→x1  ≥  p→x2);
 assumes (p→y1  <  -BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -4;

 behavior S4:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes !(p→x2  >  BOUND_SIGNAL);
 assumes !(p→x1  ≥  p→x2);
 assumes !(p→y1  <  -BOUND_SIGNAL);
 assumes (p→y1  >  BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -5;

 behavior S5:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes !(p→x2  >  BOUND_SIGNAL);
 assumes !(p→x1  ≥  p→x2);
 assumes !(p→y1  <  -BOUND_SIGNAL);
 assumes !(p→y1  >  BOUND_SIGNAL);
 assumes (p→y2  <  -BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -6;

 behavior S6:
 assumes !(p→x1  <  -BOUND_SIGNAL);
 assumes !(p→x2  >  BOUND_SIGNAL);
 assumes !(p→x1  ≥  p→x2);
 assumes !(p→y1  <  -BOUND_SIGNAL);
 assumes !(p→y1  >  BOUND_SIGNAL);
 assumes !(p→y2  <  -BOUND_SIGNAL);
 assumes        (p→y2  >  BOUND_SIGNAL);
 assigns \nothing;
 ensures \result == -7;

////////////////////////////
// groupe OK :

 behavior S7:
 assumes BCK(p);
 assigns *y, x→s;
```

```
 ensures \result == 0;

////////////////////////////
// groupe VALUE :

 behavior S8:
 assumes BCK(p);
 assumes -BOUND_SIGNAL  ≤  x→v  ≤  p→x1;
 assigns *y, x→s;
 ensures ls: (y→v == p→y1);

 behavior S9:
 assumes BCK(p);
 assumes p→x2  ≤  x→v  ≤  BOUND_SIGNAL;
 assigns *y, x→s;
 ensures hs: (y→v == p→y2);

 behavior S10:
 assumes BCK(p);
 assumes p→x1  ≤  x→v  ≤  p→x2;
 assigns *y, x→s;
 ensures ms: -EPS  <  ERROR(p,x,y)  <  EPS;


////////////////////////////
// groupe VALID :

 behavior S11:
 assumes BCK(p);
 assumes -BOUND_SIGNAL  ≤  x→v  ≤  BOUND_SIGNAL;
 assigns *y, x→s;
 ensures y→s == 0;

 behavior S12:
 assumes BCK(p);
 assumes (-BOUND_SIGNAL  >  x→v) || (BOUND_SIGNAL  <  x→v);
 assigns *y, x→s;
 ensures y→s == 1;

 complete behaviors;
 disjoint behaviors;
*/

int transform (Block *p, Signal *x, Signal *y){
 if(p→x1  <  -BOUND_SIGNAL){
  //@ label Lerror1: 1 == 1;
  //@ mutant Lerror1: replace_assign (0);
  return -1;
 }

 if(p→x2  >  BOUND_SIGNAL){
  //@ label Lerror2: 1 == 1;
  //@ mutant Lerror2: replace_assign (0);
  return -2;
 }

 if(p→x1  ≥  p→x2){
  //@ label Lerror3: 1 == 1;
  //@ mutant Lerror3: replace_assign (0);
  return -3;
 }

 if(p→y1  <  -BOUND_SIGNAL){
```

```
 //@ label Lerror4: 1 == 1;
 //@ mutant Lerror4: replace_assign (0);
 return -4;
}

if(p→y1 > BOUND_SIGNAL){
 //@ label Lerror5: 1 == 1;
 //@ mutant Lerror5: replace_assign (0);
 return sum (x,y);
}

if(p→y2 < -BOUND_SIGNAL){
 //@ label Lerror6: 1 == 1;
 //@ mutant Lerror6: replace_assign (0);
 return -6;
}

if(p→y2 > BOUND_SIGNAL){
 //@ label Lerror7: 1 == 1;
 //@ mutant Lerror7: replace_assign (0);
 return -7;
}

//@ label Llow1: x→v ≤ p→x1;
//@ label Llow2: x→v ≤ p→x1;
//@ mutant Llow1 : replace_condition (0);
//@ label Lhigh: x→v ≥ p→x2;
//@ label Lmedium: p→x1 ≤ x→v ≤ p→x2;
if(x→v ≤ p→x1){
 //@ mutant Llow2: erase;
 y→v = p→y1;
}else{
 if(x→v ≥ p→x2){
  //@ mutant Lhigh: erase;
  y→v = p→y2 ;
 }else{
  double k = (x→v - p→x1) * (p→y1 - p→y2) / (p→x1 - p→x2) ;
  //@ mutant Lmedium: erase;
  y→v = p→y1 + k;
 }
}

//@ label Lvalidity : 1==1;
//@ mutant Lvalidity : erase;
y→s = ((x→v < -BOUND_SIGNAL)||(x→v > BOUND_SIGNAL))? 1 : 0;

//@ label Lok : 1 == 1;
//@ mutant Lok: replace_assign (-1);
return 0;
}
```

Bibliographie

[AB05] Cyrille Artho et Armin Biere. “Combined Static and Dynamic Analysis”. In : *Electronic Notes in Theoretical Computer Science (ENTCS)* (2005), p. 3-14.

[Agr+99] Hiralal Agrawal, Richard A. Demillo, Bob Hathaway, William Hsu, Wynne Hsu, Edward W. Krauser, R J. Martin, Aditya P. Mathur et Eugene Spafford. *Design Of Mutant Operators For The C Programming Language*. Rapp. tech. SERC-TR-41-P. Purdue University, 1999.

[Ahr+16] Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt et Mattias Ulbrich. *Deductive Software Verification - The KeY Book*. Springer, 2016.

[Ahr+17] Wolfgang Ahrendt, Jesús Mauricio Chimento, Gordon J. Pace et Gerardo Schneider. “Verifying data- and control-oriented properties combining static and runtime verification : theory and tools”. In : *Formal Methods in System Design* (2017), p. 200-265.

[All70] Frances Allen. “Control Flow Analysis”. In : *Symposium on Compiler Optimization*. 1970, p. 1-19.

[AO08] Paul Ammann et Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[Bar+14] Sébastien Bardin, Omar Chebaro, Mickaël Delahaye et Nikolai Kosmatov. “An All-in-One Toolkit for automated White-Box Testing”. In : *International Conference on Tests and Proofs (TAP)*. 2014, p. 53-60.

[Bar+15] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon et Jean-Yves Marion. “Sound and quasi-Complete Detection of Infeasible Test Requirements”. In : *International Conference on Software Testing, Verification and Validation (ICST)*. 2015, p. 1-10.

[Bau+a] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy et Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*. http ://frama-c.com/acsl.html.

[Bau+b] Patrick Baudin, François Bobot, Loïc Correnson et Zaynah Dargaye. *WP Plug-in Manual*. https ://frama-c.com/download/frama-c-wp-manual.pdf.

[BBC13] Peter Bishop, Robin Bloomfield et Lukasz Cyra. “Combining Testing and Proof to Gain High Assurance in Software : A case study”. In : *IEEE International Symposium on Software Reliability Engineering (ISSRE)*. 2013, p. 248-257.

[BBY17] Sandrine Blazy, David Bühler et Boris Yakobowski. “Structuring Abstract Interpreters Through State and Value Abstractions”. In : *International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*. 2017, p. 112-130.

[BC04] Yves Bertot et Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art : The Calculus of Inductive Constructions*. Springer, 2004.

[BD16] Dirk Beyer et Matthias Dangl. “Verification-Aided Debugging : An Interactive Web-Service for Exploring Error Witnesses”. In : *International Conference on Computer Aided Verification (CAV)*. 2016, p. 502-509.

[Bec+18] Bernhard Beckert, Mihai Herda, Stefan Kobischke et Mattias Ulbrich. “Towards a Notion of Coverage for Incomplete Program-Correctness Proofs”. In : *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*. 2018, p. 53-63.

[Bey+15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann et Andreas Stahlbauer. “Witness Validation and Stepwise Testification across Software Verifiers”. In : *Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. 2015, p. 721-733.

[Bey+16] Dirk Beyer, Matthias Dangl, Daniel Dietsch et Matthias Heizmann. “Correctness Witnesses : Exchanging Verification Results between Verifiers”. In : *ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 2016, p. 326-337.

[BG85] Timothy A. Budd et Ajei S. Gopal. “Program testing by specification mutation”. In : *Computer Languages* (1985), p. 63 -73.

[BKC14] Sébastien Bardin, Nikolai Kosmatov et François Cheynier. “Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria”. In : *International Conference on Software Testing, Verification and Validation (ICST)*. 2014, p. 173-182.

[Bob+] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer et Alain Mebsout. *The Alt-Ergo automated theorem prover*. http ://alt-ergo.lri.fr.

[BOY00] Paul E. Black, Vadim Okun et Yaacov Yesha. “Mutation Operators for Specifications”. In : *IEEE International conference on Automated Software Engineering (ASE)*. 2000, p. 81-88.

[Bra+18] Abderrahmane Brahmi, David Delmas, Mohamed Habib Essoussi, Famantanantsoa Randimbivololona, Abdellatif Atki et Thomas Marie. “Formalise to automate : deployment of a safe and cost-efficient process for avionics software”. In : *European Congress on Embedded Real Time Software and Systems (ERTS)*. 2018.

[Bro+10] Duncan Brown, Hervé Delseny, Kelly Hayhurst et Virginie Wiels. “Guidance for Using Formal Methods in a Certification Context”. In : *European Congress on Embedded Real Time Software and Systems (ERTS)*. 2010.

[Bud+80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton et Frederick G. Sayward. "Theoretical and empirical studies on using program mutation to test the functional correctness of programs". In : *ACM SIGPLAN SIGACT symposium on Principles of programming languages (POPL)* (1980), p. 220-233.

[Büh+] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perrelle et Virgile Prevosto. *Eva - The Evolved Value Analysis plug-in*. https ://frama-c.com/download/frama-c-value-analysis.pdf.

[Car+18] Luiz Carvalho, Marcio Augusto Guimaraes, Marcio Ribeiro, Leonardo Fernandes, Mustafa Al-Hajjaji, Rohit Gheyi et Thomas Thum. "Equivalent Mutants in Configurable Systems : An Empirical Study". In : *International Workshop on Variability Modelling on Software-Intensive Systems (VAMOS)*. 2018.

[CAS09] Jesus Mauricio Chimento, Wolfgang Ahrendt et Gerardo Schneider. "Larva - Safer Monitoring of Real-Time Java Programs (Tool Paper)". In : *IEEE International Conference on Software Engineering and Formal Methods (SEFM)*. 2009, p. 33-37.

[CC77] Patrick Cousot et Radhia Cousot. "Abstract Interpretation : A Unified Lattice Model for Static Analysis of Programs by Construction of Approximation of Fixpoints". In : *ACM Symposium on Principles of Programming Language (POPL)* (1977), p. 238-252.

[CDK12] Omar Chebaro, Mickaël Delahaye et Nikolai Kosmatov. "Testing Inexecutable Conditions on Input Pointers in C Programs with SANTE". In : *International Conference on Software & Systems Engineering and their Applications (ICSSEA)*. 2012, p. 1-7.

[CGK98] Yih-Fam Chen, Emden R. Gansner et Eleftherios Koutsofios. "A C++ Data Model Supporting Reachability Analysis and Dead Code Detection". In : *IEEE Transactions on Software Engineering* (1998), p. 682-694.

[Che+11] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti et Jacques Julliand. "The SANTE Tool : Value Analysis, Program Slicing and Test Generation for C Program Debugging". In : *International Conference on Tests and Proofs (TAP)*. 2011, p. 78-83.

[Che+12] Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti et Jacques Julliand. "Program Slicing Enhances a Verification Technique Combining Static and Dynamic Analysis". In : *ACM Symposium on Applied Computing (SAC)*. 2012, p. 1284-1291.

[Chi+15] Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace et Gerardo Schneider. "staRVOOrs : A Tool for Combined Static and Runtime Verification of Java". In : *Runtime Verification*. 2015, p. 297-305.

[CKM12] Cyrille Comar, Johannes Kanig et Yannick Moy. *Integrating Formal Program Verification with Testing*. Rapp. tech. Hi-Lite-ERTS-2012. AdaCore, 2012.

[CKV03] Hana Chockler, Orna Kupferman et Moshe Vardi. "Coverage Metrics for Formal Verification". In : *Correct Hardware Design and Verification Methods*. 2003, p. 111-125.

[CMW12] Maria Christakis, Peter Müller et Valentin Wüstholz. “Collaboration Verification and Testing With Explicit Assumptions”. In : 2012, p. 132-146.

[CMW16] Maria Christakis, Peter Müller et Valentin Wüstholz. “Guiding Dynamic Symbolic Execution toward Unverified Program Executions”. In : 2016, p. 144-155.

[Con+07] Camille Constant, Thierry Jeron, Hervé Marchand et Vlad Rusu. “Integrating Formal Verification and Conformance Testing for Reactive Systems”. In : 2007, p. 558-574.

[Cuo+12] Pascal Cuoq, David Delmas, Stéphane Duprat et Victoria Moya Lamiel. “Fan-C, a Frama-C plug-in for data flow verification”. In : *The embedded real-time software and systems congress (ERTS2)*. 2012.

[Del+14] Marcio E. Delamaro, Lin Deng, Vinicius H. S. Durelli, Nan Li et Jeff Offutt. “Experimental Evaluation of SDL and One-Op Mutation for C”. In : *International Conference on Software Testing, Verification and Validation (ICST 2014)*. 2014, p. 203-212.

[Den76] Dorothy E. Denning. “A Lattice Model of Secure Information Flow”. In : *Communications of the ACM* (1976), p. 236-243.

[Dev+17] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens et Patrick Heymans. “Automata Language Equivalence vs. Simulations for Model-based Mutant Equivalence : An Empirical Evaluation”. In : *International Conference on Software Testing, Verification and Validation (ICST)*. 2017, p. 424-429.

[DKS13] Mickaël Delahaye, Nikolaï Kosmatov et Julien Signoles. “Common Specification Language for Static and Dynamic Analysis of C Programs”. In : *Symposium on Applied Computing (SAC)*. 2013, p. 1230-1235.

[DLS78] Richard A. DeMillo, Richard J. Lipton et Frederick G. Sayward. “Hints on Test Data Selection : Help for the Practicing Programmer”. In : *IEEE Computer* (1978), p. 34-41.

[Fil11] Jean-Christophe Filliâtre. “Deductive software verification”. In : *International Journal on Software Tools for Technology Transfer* (2011), p. 397-403.

[Flo93] Robert W. Floyd. “Assigning Meanings to Programs”. In : *Program Verification : Fundamental Issues in Computer Science*. 1993, p. 65-81.

[FMV11] Carlo A. Furia, Bertrand Meyer et Sergey Velder. “Loop Invariants : Analysis, Classification and Examples”. In : *ACM Computing Surveys (CSUR)*. 2011.

[Hay+01] Kelly Hayhurst, Dan Veerhusen, John Chilenski et Leanna Rierson. *A Practical Tutorial on Modified Condition/Decision Coverage*. Rapp. tech. 20010057789. NASA Langley Research Center, 2001.

[Hoa78] Charles A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In : *Programming Methodology : A Collection of Articles by Members of IFIP WG2.3*. 1978, p. 89-100.

[KHS11] Mohammad Reza Keyvanpour, Hajar Homayouni et Hossein Shirazi. “Automatic Software Test Case Generation”. In : *Journal of Software Engineering* (2011), p. 91-101.

[Kir+15] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles et Boris Yakobowski. “Frama-C : A software analysis perspective”. In : *Formal Aspects of Computing*. 2015, p. 573-609.

[Kis+15] Balázs Kiss, Nikolai Kosmatov, Dillon Pariente et Armand Puccetti. “Combining Static and Dynamic Analyses for Vulnerability Detection : Illustration on Heartbleed”. In : *International Haifa Verification Conference (HVC 2015)*. 2015, p. 39-50.

[Lab] Search Lab. *Flinder security testing platform*. http ://www.flinder.hu.

[Mar+17a] Michaël Marcozzi, Mickaël Delahaye, Sébastien Bardin, Nikolai Kosmatov et Virgile Prevosto. “Generic and Effective Specification of Structural Test Objectives”. In : *International Conference on Software Testing, Verification and Validation (ICST)*. 2017, p. 436-441.

[Mar+17b] Michaël Marcozzi, Sébastien Bardin, Mickaël Delahaye, Nikolai Kosmatov et Virgile Prevosto. “Taming Coverage Criteria Heterogeneity with LTest”. In : *International Conference on Software Testing, Verification and Validation (ICST)*. 2017, p. 500-507.

[MB11] Lei Mi et Kerong Ben. “A Method of Software Specification Mutation Testing Based on UML State Diagram for Consistency Checking”. In : *Procedia Engineering* (2011), p. 110 -114.

[MB99] Elfurjani S. Mresa et Leonardo Bottaci. “Efficiency of Mutation Operators and Selective Mutation Strategies : An empirical Study”. In : *Software Testing Verification and Reliability*. 1999, p. 205-232.

[Mer08] Stephan Merz. “An introduction to model checking”. In : *Modeling and Verification of Real-Time Systems - Formalisms and Software Tools*. 2008, p. 81-116.

[MOK06] Yu-Seung Ma, Jeff Offutt et Yong-Rae Kwon. “MuJava : A Mutation System for Java”. In : *International Conference on Software Engineering (ICSE)*. 2006, p. 827-830.

[Moy+13] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels et Benjamin Monate. “Testing or Formal Verification : DO-178C Alternatives and Industrial Experience”. In : *IEEE Software*. 2013, p. 50-57.

[Neu+16] Felix Neubauer, Karsten Scheibler, Bernd Becker, Ahmed Mahdi, Martin Fränzle, Tino Teige, Tom Bienmüller et Detlef Fehrer. “Accurate Dead Code Detection in Embedded C Code by Arithmetic Constraint Solving”. In : *Workshop on Satisfiability Checking and Symbolic Computation co-located with International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SC2SYNASC)*. 2016, p. 32-38.

[Off+96] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch et Christian Zapf. “An Experimental Determination of Sufficient Mutant Operators”. In : *ACM Transactions on Software Engineeringg and Methodology (TOSEM)*. 1996, p. 99-118.

[ORZ93] A. Jefferson Offutt, Gregg Rothermel et Christian Zapf. “An Experimental Evaluation of Selective Mutation”. In : *International Conference on Software Engineering*. 1993, 100-107g.

[OVP96] A. Jefferson Offutt, Jeff Voas et Jeff Payne. *Mutation Operators for Ada*. Rapp. tech. ISSE-TR-96-09. George Mason University, 1996.

[Pap+15] Mike Papadakis, Yue Jia, Mark Harman et Yves Le Traon. “Trivial Compiler Equivalence : A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique”. In : *IEEE International Conference on Software Engineering (ICSE)*. 2015, p. 936-946.

[Pet+16] Guillaume Petiot, Nikolai Kosmatov, Bernard Botella, Alain Giorgetti et Jacques Julliand. “Your Proof Fails ? Testing Helps to Find the Reason”. In : *Tests and Proofs (TAP)*. 2016, p. 130-150.

[PP98] Dennis Peters et David Lorge Parnas. “Using Test Oracles Generated from Program Documentation”. In : *IEEE Transactions on software engineering* (1998), p. 161-173.

[Ran+99] Famantanantsoa Randimbivololona, Jean Souyris, Patrick Baudin, Anne Pacalet, Jacques Raguideau et Dominique Schoen. “Applying Formal Proof Techniques to Avionics Software : A Pragmatic Approach”. In : *International Symposium on Formal Methods (FM)*. 1999, p. 1798-1815.

[SFF04] Jean Souyris et Denis Favre-Félix. “Proof of Properties in Avionics”. In : *Building the Information Society*. 2004, p. 527-535.

[Sig] Julien Signoles. *E-ACSL : Executable ANSI/ISO C Specification Language*. http ://frama-c.com/download/e-acsl/e-acsl.pdf.

[Sig+] Julien Signoles, Thibaud Antignac, Loïc Correnson, Matthieu Lemerre et Virgile Prevosto. *Plug-in Development Guide*. https ://frama-c.com/download/frama-c-plugin-development-guide.pdf.

[SKV17] Julien Signoles, Nikolai Kosmatov et Kostyantyn Vorobyov. “E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. Tool Paper.” In : *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. 2017, p. 164-173.

[SN14] Sneha Shelke et Sangeeta Nagpure. “The Study of Various Code Coverage Tools”. In : *International Journal of Computer Trends and Technology (IJCTT)* (2014), p. 46-49.

[Wey80] Elaine J. Weyuker. *On Testing Nontestable Programs*. Rapp. tech. 025. New York University, 1980.

[Wil+05] Nicky Williams, Bruno Marre, Patricia Mouy et Muriel Roger. "PathCrawler : Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In : *European Dependable Computing Conference (EDCC)*. 2005, p. 281-292.

[WMM04] Nicky Williams, Bruno Marre et Patricia Mouy. "On-the-Fly Generation of K-Path Tests for C Functions". In : *IEEE International Conference on Automated Software Engineering (ASE)*. 2004, p. 290-293.

[Mic07] Michael Sutton and Adam Greene and Pedram Amini. *Fuzzing : Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[RTC11a] RTCA (Firm). and EUROCAE (Agency). *DO-178C, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Incorporated, 2011.

[RTC11b] RTCA (Firm). SC-205 and EUROCAE (Agency). Working Group 71. *Formal Methods Supplement to DO-178C and DO-278A*. RTCA, Incorporated, 2011.

[RTC92] RTCA (Firm). and EUROCAE (Agency). *DO-178B, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Incorporated, 1992.

[Sof08] Software and Engineering Standards Committee. "IEEE Standard for Software and System Test Documentation". In : *IEEE Std 829-2008* (2008), p. 1-150.

Résumé —

Actuellement, le développement d'un logiciel de taille industriel repose généralement sur des tests ou des preuves unitaires pour garantir rigoureusement ses exigences. En outre, il a déjà été montré que l'utilisation combinée du test et de la preuve unitaires est plus efficace que l'utilisation d'une seule de ces deux techniques. Néanmoins, un ingénieur en vérification hésite encore à utiliser ces deux techniques conjointement, faute d'une notion de couverture commune au test et à la preuve. Définir une telle notion est l'objet de cette thèse.

En effet, nous introduisons une nouvelle couverture, appelée « couverture label-mutant ». Elle permet de représenter les critères de couverture structurelle habituels du test, comme la couverture des instructions, la couverture des branches ou la couverture MC/DC et de décider si le critère choisi est satisfait en utilisant une technique de vérification formelle, qu'elle soit par test, par preuve ou par une combinaison des deux. Elle permet également de représenter les critères de couverture fonctionnelle. Nous introduisons aussi dans cette thèse une méthode reposant sur des outils automatiques de test et de preuve pour réduire l'effort de vérification tout en satisfaisant le critère de couverture choisi. Cette méthode est mise en œuvre au sein de la plateforme d'analyse de code C (Frama-C), fournissant ainsi à un ingénieur un moyen opérationnel pour contrôler et réaliser la vérification qu'il souhaite.

Mots clés : vérification de programmes, test unitaire, preuve de programmes, combinaison d'analyse, couverture de code

Abstract —

Currently, industrial-strength software development usually relies on unit testing or unit proof in order to ensure high-level requirements. Combining these techniques has already been demonstrated more effective than using one of them alone. The verification engineer is yet not been to combine these techniques because of the lack of a common notion of coverage for testing and proving. Defining such a notion is the main objective of this thesis.

We introduce here a new notion of coverage, named « label-mutant coverage ». It subsumes most existing structural coverage criteria for unit testing, including statement coverage, branch coverage or MC/DC coverage, while allowing to decide whether the chosen criterion is satisfied by relying on a formal verification technique, either testing or proving or both. It also subsumes functional coverage criteria. Furthermore, we also introduce a method that makes use of automatic tools for testing or proving in order to reduce the verification cost while satisfying the chosen coverage criterion. This method is implemented inside Frama-C, a framework for verification of C code (Frama-C). This way, it offers to the engineer a way to control and to perform the expected verifications.

Keywords : program verification, unit testing, program proving, analysis combination, code coverage