À la recherche de tous les vrais bugs

Arthur Correnson

À la recherche de tous les vrais bugs

Verification formelle d'un détecteur de bugs automatique

Arthur Correnson[1]

[1]CISPA Helmoltz Center for Information Security

Pour assurer le bon fonctionnement des logiciels, il est crucial de les tester pendant leur développement. Pour faciliter cette tâche, de nombreuses méthodes de test automatique existent et permettent de détecter rapidement des erreurs de programmation. Pour être efficace et sûr, un détecteur de bugs automatique se doit d'être aussi précis et exhaustif que possible : il ne doit détecter que des *vrais bugs* et, si possible, être capable de détecter *tous* les *bugs*. Dans cet article, nous présentons un détecteur automatique de *bugs* formellement vérifié en Coq. En particulier nous prouvons que, sous certaines hypothèses, notre détecteur est précis et exhaustif.

1 Introduction

Contexte. Pour vérifier ou valider les logiciels, il existe plusieurs méthodes, principalement la preuve et le test. Les outils de preuve permettent d'établir, a posteriori, qu'un logiciel est conforme à une certaine spécification (c'est à dire, prouver que le logiciel *ne se trompe pas*). D'autre part, les outils de test aident à identifier des défauts dans les logiciels pendant leur conception en détectant des scénarios d'utilisation dans lesquels un logiciel *se trompe*. Qu'il s'agisse de preuve ou de test, l'usage d'outils automatiques pour valider la qualité des logiciels est de plus en plus fréquent. Cela amène à se poser la question de la fiabilité de ces outils. En réponse à cette question, beaucoup de travaux se sont concentrés sur l'étude de la correction des outils de preuve automatique comme par exemple les vérificateurs de modèles, les démonstrateurs SMT, ou les analyseurs statiques par interprétation abstraite. Dans ce contexte, la question posée est claire : si un vérificateur prétend qu'un programme respecte sa spécification, est-ce vraiment le cas ?

Motivations. Contrairement aux vérificateurs automatiques, les approches de **test** automatiques visent à exposer des défauts plutôt que de prouver leur absence. En conséquence, il est souvent admis que ces outils n'ont pas besoin d'être très fiables. Cela est dommage car, en raison de sa simplicité de mise en œuvre, le test reste la méthode la plus largement utilisée pour vérifier la robustesse des logiciels. Utilisé à bon escient, le test peut donner des garanties formelles. Par ailleurs les méthodes de test automatique sont aussi sujettes à de nombreuses erreurs qui peuvent avoir des conséquences néfastes [God05]. Deux types d'erreurs peuvent se produire : le signalement de *faux bugs*, et le manquement de *bugs* que l'on espérait être détectés. Le premier type d'erreurs peut, à l'extrême, rendre un outil de test complètement inutile. En effet, inonder les utilisateurs de fausses alarmes ne fait que déplacer le problème de trouver des erreurs dans un logiciel vers celui de trouver des erreurs dans l'outil de recherche de *bugs* lui-même. Symétriquement, un détecteur de *bugs* qui manque trop d'erreurs donne de très faibles garanties et peut donner la fausse illusion qu'un

logiciel est robuste. Pour palier ces deux problèmes, il est désirable de pouvoir interpréter précisément les résultats d'un détecteur de *bugs* automatique. Si un *bug* est signalé, qu'est-ce que cela veut dire ? Si aucun *bug* n'est signalé, quelles garanties avons-nous obtenues ?

Contributions. Dans cet article, nous proposons de prouver la correction d'un détecteur de *bugs* à l'aide de l'assistant de preuve Coq. L'objectif est d'être capable d'interpréter formellement les verdicts de l'outil, qu'ils soient positifs ou négatifs. Plus précisément, nous prouvons un résultat de *précision* : tout *bug* détecté est un véritable *bug* dans le programme analysé ; ainsi qu'un résultat d'exhaustivité : tout *bug* finira éventuellement par être détecté. L'outil est fondé sur des méthodes d'exécution symbolique et permet d'analyser des programmes impératifs simples.

Tous les résultats présentés dans cet article sont issus d'un travail plus approfondi et déjà soumis en anglais à la conference *Foundations of Software Engineering* [CS23]. L'objectif principal de cet article est de faire une synthèse de ces travaux en Français. L'accent est mis sur la présentation de l'approche plutôt que sur les détails techniques des preuves.

2 Vérifier les vérificateurs

Comment faire confiance aux méthodes formelles ? Établir la correction d'un vérificateur automatique peut passer par plusieurs approches. Par exemple, on peut exiger que les vérificateurs produisent des preuves en plus d'un simple résultat binaire "oui/non". Ces preuves jouent le rôle de certificat de correction et peuvent être relues par un expert, ou par un validateur automatique. Par exemple, les démonstrateurs automatiques VeriT [BCBdODF09] et CVC4 [BCD+11] produisent des preuves qui peuvent être validée par un validateur de preuves fiable comme SMTCoq [AFG+11]. D'autres outils de preuve comme les vérificateurs de modèles peuvent suivre cette approche [Nam01]. Par exemple, le vérificateur SLAB [DKFW10] produit des certificats. Une autre approche et de vérifier une fois pour toute la correction des vérificateurs en utilisant un autre vérificateur. Cette approche peut sembler vaine à première vue. Pourquoi ferions-nous plus confiance au vérificateur du vérificateur qu'au vérificateur lui même ? Pour éviter cette spirale sans fin, il existe une solution *simple* qui fait consensus : utiliser un *assistant de preuve* comme second vérificateur.

Les assistants de preuves comme base de confiance. Les assistants de preuve sont des outils de vérification *minimalistes* et *robustes*. Ils permettent d'écrire des programmes et leur preuve de correction dans un langage formel. Contrairement à d'autres outils plus automatiques (par exemple, les démonstrateurs SMT), les preuves sont écrites manuellement par un expert et sont ensuite mécaniquement vérifiées par l'assistant. La tâche de valider la correction des preuves est suffisamment simple et élémentaire pour que l'on puisse faire l'hypothèse que l'assistant de preuve ne se trompe jamais. Par ailleurs, comme les preuves sont faites par l'utilisateur, on peut se permettre d'écrire des théorèmes et de définir des objets plus complexes. En particulier, on peut implémenter et prouver la correction de logiciels aussi complexes qu'un compilateur C [LBK+16] ou un analyseur statique [JLB+15] à l'aide d'un assistant à la démonstration. Cette méthode, comme toutes les autres, n'est pas parfaite (un assistant de preuve pourrait toujours contenir un *bug* !), mais elle donne un niveau de garanties difficilement égalable.

Vérifier les prouveurs. Les assistants de preuve ont été utilisés avec succès pour prouver la correction de divers vérificateurs automatiques. Par exemple, Verasco est un interprète abstrait pour le langage C dont la correction a été entièrement vérifiée en Coq [JLB+15]. D'autre méthodes de vérification automatiques comme la *vérification de modèles* pour les propriété temporelles ont également été prouvées correctes à l'aide de l'assistant de preuve

Isabelle/HOL [ELN+13]. Ces projets démontrent qu'il est possible de prouver formellement la correction d'outils de vérification automatique réalistes.

Vérifier les détecteurs de bugs. Des travaux récents se sont intéressés à la formalisation des méthodes de test. On peut notamment citer les travaux sur *incorrectness logic* [O'H19], qui proposent d'utiliser des logiques de programmes pour prouver l'existence d'erreurs plutôt que de prouver leur absence. D'autres approches, plus automatiques, on également été explorées. Par exemple, la plateforme Gillian a une fonction de recherche de bugs qui a été formalisée rigoureusement [FSMAG20]. En revanche, les preuves n'ont pas été mécanisées dans un assistant à la démonstration. À notre connaissance, le seul effort de mécanisation d'un détecteur de *bugs* automatique dans un assistant à la démonstration est *QuickChick*, une bibliothèque logicielle pour tester la correction de programmes Coq en utilisant des méthodes de génération d'entrées aléatoires [PHD+14].

3 Détection automatique de bugs par exécution symbolique

Qu'est-ce qu'un bug ? Selon le contexte, il existe plusieurs manières de définir ce qu'est un "*bug*" dans un programme informatique. Par exemple, il peut s'agir d'une simple erreur à l'execution comme une division par zéro ou un accès hors des bornes d'un tableau. Plus généralement, les erreurs à l'exécution sont toutes les situations dans lesquelles un programme est contraint d'être interrompu prématurément. Un *bug* peut aussi désigner une erreur de programmation qui n'entraîne pas nécessairement d'erreur à l'exécution mais qui conduit un programme à produire des résultats incorrects ou inattendus. Dans la suite de cet article, nous nous concentrons sur la détection des erreurs à l'exécution. Cela n'est pas forcément une limitation car on peut détecter certaines erreurs de correction en les ramenant à des erreurs à l'exécution en utilisant des *assertions* (c'est à dire, en forçant l'interruption d'un programme lorsque certaines conditions de correction de ne sont pas satisfaites pendant l'exécution).

Trouver les bugs en exécutant les programmes. Pour détecter des erreurs à l'exécution, une méthode naïve est d'employer la force brute. On peut exécuter un programme un très grand nombre de fois en essayant un maximum de combinaisons d'entrées possibles (cette méthode est appelée *fuzzing* [FMEH20]). Si le programme part en erreur pour certains choix d'entrées, il faut le mettre à jour en conséquence. Cette approche peut s'avérer très efficace pour détecter rapidement des cas particuliers qui ont été oubliés lors du développement d'un programme. En revanche, la force brute a plusieurs limitations. Tout d'abord, il est impossible de couvrir toutes les entrées possibles en pratique. Par ailleurs, certaines erreurs sont presque impossible à identifier simplement en testant des combinaisons d'entrées au hasard. Considérons par exemple le programme suivant :

```
if (x == 42) {
  fail();
}
return 2 * x;
```

Ce programme peut partir en erreur si on choisit $\texttt{x} = 42$. En revanche, la probabilité de choisir la bonne valeur de `x` sans inspecter le code source est presque nulle (si on considère des entiers 64 bits, il y a seulement $\frac{1}{2^{64}}$ chances de trouver l'erreur !).

Trouver les bugs par exécution *symbolique*. Une solution aux limites des approches de test par force brute est d'avoir recours à des techniques d'*exécution symbolique* pour essayer de prédire l'ensemble de toutes les issues possibles d'un programme [Kin76]. L'idée

est d'exécuter les programmes en traitant les entrées comme des *symboles opaques*. Lors de l'exécution symbolique d'un programme, quand un branchement gardé par une condition (par exemple un `if` ou une boucle `while`) est exécuté, on envisage tous les scénarios possibles selon que la condition est supposée satisfaite ou non. On prend soin de mémoriser les conditions que l'on a supposées satisfaites tout au long de l'exécution. Le résultat d'une telle *exécution symbolique* est un ensemble représentant toutes les issues possibles d'un programme. Chaque issue est exprimée comme une fonction des entrées et est annotée par l'ensemble des hypothèses qui doivent être satisfaites pour atteindre le résultat associé. Les issues possibles sont typiquement "le programme part en erreur" (on note `fail`) ou bien "le programme termine et retourne un résultat" (on note `return`). On note $\langle H, r\rangle$ une issue r étiquetée par un ensemble d'hypothèses H. Par exemple, l'exécution symbolique de l'exemple précédent donne les deux issues $\{\langle \mathtt{x} = 42, \mathtt{fail}\rangle, \langle \mathtt{x} \neq 42, \mathtt{return}(2 * \mathtt{x})\rangle\}$.

Si une issue possible est une erreur et est étiquetée par un ensemble d'hypothèses *satisfiables*, alors le programme contient un *bug*. Cela donne une méthode systématique pour tester un programme automatiquement. On commence par exécuter le programme symboliquement, puis on parcours exhaustivement les issues obtenues. Pour chaque issue représentant une erreur, on peut faire appel à un solveur de contraintes pour vérifier que les hypothèses conduisant à l'erreur en question sont réalisables.

$E \leftarrow$ exécution-symbolique(P)
for all $\langle H, r\rangle \in E$ **do**
  if r est une erreur et H est satisfiable **then**
    return P part en erreur pour toutes entrées satisfiant H
  end if
end for
return P ne part jamais en erreur

Figure 1. Un algorithme simple pour détecter les bugs dans un programme P

Si l'exécution symbolique de P est effectuée rigoureusement (nous reviendrons sur ce point dans la partie 4) et sous l'hypothèse que le test de satisfiabilité des hypothèses est correct, cet algorithme ne détecte que des vrais bugs. On dit qu'il est *précis*. En revanche, l'ensemble des issues symboliques d'un programme n'est pas toujours fini. Par ailleurs, l'exécution symbolique d'un programme qui ne termine pas peut diverger. En général, on ne peut donc calculer qu'un sous-ensemble des issues possibles. Par conséquent, l'algorithme 1 n'est pas *exhaustif* et il peut manquer des bugs. Toutefois, en calculant l'ensemble des issues de façon *paresseuse*, il est possible d'obtenir une implémentation *relativement exhaustive* : toute erreur à l'exécution sera éventuellement découverte pourvu que l'on laisse l'algorithme tourner suffisamment longtemps et avec suffisamment de ressources mémoire. Dans la suite de cet article, nous proposons une telle implémentation de cet algorithme en Coq. Nous effectuons aussi la preuve formelle de précision et d'exhaustivité relative.

4 Formaliser l'exécution symbolique

Dans cette partie, nous nous penchons sur la formalisation de la notion d'exécution symbolique. Nous commençons par introduire un langage de programmation simple que nous équipons avec une sémantique formelle pour décrire son modèle d'exécution. Puis, nous introduisons une deuxième sémantique formelle pour décrire un modèle d'exécution symbolique pour ce langage. Enfin, nous établissons une connection entre la sémantique concrète et la sémantique symbolique. Cette connection servira de fondation pour justifier la correction d'un interprète symbolique.

Le langage d'étude BUG. On se donne un langage impératif à contrôle structuré (avec des constructions `if` et `while`) pouvant manipuler des expressions arithmétiques simples. Nous étendons ce langage avec une instruction `Error` qui provoque l'interruption volontaire de l'exécution. Cette instruction peut-être utilisée pour instrumenter les programmes. Par exemple, pour modéliser des régions inaccessibles ou des assertions logiques qui ne devraient jamais être invalidées à l'exécution. La syntaxe complète de ce langage, que l'on appellera BUG dans la suite, est définie comme suit.

Définition 1 (Syntaxe du langage BUG)**.**

Variables
$$\mathit{var} \in \mathbb{V}$$

Arithmétique
$$\begin{aligned}\mathbb{A}\mathit{expr} \ni \mathit{aexpr} ::= {} & c \in \mathbb{Z} \mid \mathit{var} \\ \mid {} & \mathit{aexpr}\ (\texttt{+} \mid \texttt{-})\ \mathit{aexpr}\end{aligned}$$

Booléens
$$\begin{aligned}\mathbb{B}\mathit{expr} \ni \mathit{bexpr} ::= {} & \texttt{true} \mid \texttt{false} \\ \mid {} & \mathit{aexpr}\ (\texttt{<} \mid \texttt{=})\ \mathit{aexpr} \\ \mid {} & \mathit{bexpr}\ (\texttt{and} \mid \texttt{or})\ \mathit{bexpr} \\ \mid {} & \texttt{not}\ \mathit{bexpr}\end{aligned}$$

Instructions
$$\begin{aligned}\mathbb{I}\mathit{nstr} \ni \mathit{instr} ::= {} & \texttt{skip} \mid \texttt{fail} \\ \mid {} & \mathit{var}\ \texttt{=}\ \mathit{aexpr} \\ \mid {} & \mathit{instr}\ \texttt{;}\ \mathit{instr} \\ \mid {} & \texttt{if}\ \mathit{bexpr}\ \texttt{then}\ \mathit{instr}\ \texttt{else}\ \mathit{instr} \\ \mid {} & \texttt{while}\ \mathit{bexpr}\ \texttt{do}\ \mathit{instr}\end{aligned}$$

Sémantique concrète. Les programmes BUG opèrent sur une mémoire qui associe à chaque variables une valeur entière. Dans la suite, on notera $\mathbb{M} = \mathbb{V} \to \mathbb{Z}$ l'ensemble des états mémoires. Étant donnée une mémoire $M \in \mathbb{M}$, une variable $v \in \mathbb{V}$ et une constante $z \in \mathbb{Z}$, on note $M[x \leftarrow z]$ la mémoire M où la valeur de v à été remplacée par z. Par ailleurs, étant donnée une expression $e \in \mathbb{A}\mathit{expr} \cup \mathbb{B}\mathit{expr}$ on note $[\![e]\!]_M$ la valeur de e dans la mémoire M. Dans le cas où e est une expression arithmétique on a $[\![e]\!]_M \in \mathbb{Z}$. Si e est une expression booléenne on a $[\![e]\!]_M \in \{\texttt{true}, \texttt{false}\}$.

La sémantique des instructions et donnée dans un style *à petits pas*. Un état d'exécution est une paire $\langle M, i\rangle \in \mathbb{M} \times \mathbb{I}\mathit{nstr}$ où M est l'état courant de la mémoire et i est la prochaine instruction à exécuter. On note $\langle M_1, i_1\rangle \hookrightarrow \langle M_2, i_2\rangle$ pour exprimer que l'exécution de l'instruction i_1 dans la mémoire M_1 mène à la mémoire M_2 et i_2 est la prochaine instruction à exécuter. Les transitions valides entre états sont définies par les règles d'inférence suivantes.

Définition 2 (Sémantique concrète de BUG)**.**

$$\frac{}{\langle M, x\ \texttt{=}\ e\rangle \hookrightarrow \langle M[x \leftarrow [\![e]\!]_M], \texttt{skip}\rangle}$$

$$\frac{}{\langle M, \texttt{skip ; } i\rangle \hookrightarrow \langle M, i\rangle} \qquad \frac{\langle M, i_1\rangle \hookrightarrow \langle M, i_2\rangle}{\langle M, i_1\ \texttt{;}\ i_3\rangle \hookrightarrow \langle M, i_2\ \texttt{;}\ i_3\rangle}$$

$$\frac{[\![b]\!]_M = \texttt{true}}{\langle M, \texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2\rangle \hookrightarrow \langle M, i_1\rangle} \qquad \frac{[\![b]\!]_M = \texttt{false}}{\langle M, \texttt{if}\ b\ \texttt{then}\ i_1\ \texttt{else}\ i_2\rangle \hookrightarrow \langle M, i_2\rangle}$$

$$\frac{[\![b]\!]_M = \texttt{true}}{\langle M, \texttt{while}\ b\ \texttt{do}\ i\rangle \hookrightarrow \langle M, i\ \texttt{;}\ \texttt{while}\ b\ \texttt{do}\ i\rangle} \qquad \frac{[\![b]\!]_M = \texttt{false}}{\langle M, \texttt{while}\ b\ \texttt{do}\ i\rangle \hookrightarrow \langle M, \texttt{skip}\rangle}$$

À l'aide de cette sémantique, on peut définir formellement la notion d'erreur à l'exécution. Étant donné un état $\langle M_1, i_1 \rangle$, on note $\langle M_1, i_1 \rangle \hookrightarrow^* \langle M_2, i_2 \rangle$ si le nouvel état $\langle M_2, i_2 \rangle$ peut-être atteint en 0, une ou plusieurs étapes d'exécution. En partant d'un état $\langle M_1, i_1 \rangle$, on dit qu'il y a une erreur à l'exécution s'il existe un état $\langle M_2, i_2 \rangle$ tel que $\langle M_1, i_1 \rangle \hookrightarrow^* \langle M_2, i_2 \rangle$ et $\langle M_2, i_2 \rangle$ est "bloqué" (c'est à dire, n'a aucun successeur valide d'après la relation $\hookrightarrow$). Les seuls états bloqués sont ceux de la forme $\langle M, \texttt{skip} \rangle$ et $\langle M, \texttt{fail ; } \ldots \rangle$. Bien sûr, les états $\langle M, \texttt{skip} \rangle$ ne doivent pas être considérés comme des états d'erreur. On définit donc la notion de bug comme suit.

Définition 3. Un programme p possède un bug s'il existe des états mémoire M_1, M_2 tels que $\langle M_1, p \rangle \hookrightarrow^* \langle M_2, \texttt{fail ; } \ldots \rangle$.

Sémantique symbolique. D'après la définition 3, trouver un bug revient à exposer un état mémoire M_1 conduisant à une erreur. Pour automatiser la recherche d'une telle mémoire initiale, on effectue l'exécution de p en suivant des règles d'exécution symbolique plutôt que la sémantique concrète donnée par la relation $\hookrightarrow$. Lors de l'exécution symbolique d'un programme, on remplace les états mémoire $\mathbb{M}$ par des états mémoire symboliques $\mathbb{M}_{\text{sym}} = \mathbb{V} \to \mathbb{A}expr$ qui associent chaque variable à une expression arithmétique. Cela permet d'affecter aux variables des valeurs qui dépendent des entrées du programme. Pour évaluer une expression arithmétique ou booléenne dans une mémoire symbolique, il suffit de substituer chaque variable par son expression associée. On note $[\![e]\!]^{\texttt{sym}}_{\hat{M}}$ la valeur symbolique d'une expression dans une mémoire symbolique $\hat{M} \in \mathbb{M}_{\text{sym}}$. Au début de l'exécution symbolique d'un programme, on choisit comme mémoire symbolique $\hat{M}_{\text{init}} = x \mapsto x$ (chaque variable contient son propre nom comme valeur symbolique). Par exemple, si l'on exécute symboliquement la séquence d'instructions `x = 2 + y ; x = x + 1`, la mémoire symbolique obtenue en sortie associe l'expression `(2 + y) + 1` à la variable `x`.

On formalise cette intuition en donnant une nouvelle sémantique aux programmes. Dans le contexte de l'exécution symbolique, les états sont des triplets $\langle \varphi, \hat{M}, i \rangle$ où $\varphi \in \mathbb{B}expr$ est une expression booléenne représentant un ensemble d'hypothèses qui sont faites sur les entrées du programme, $\hat{M} \in \mathbb{M}_{\text{sym}}$ est une mémoire symbolique, et $i \in \mathbb{I}nstr$ est une instruction à exécuter. On note $\mathbb{S}_{\text{sym}}$ l'ensemble des états symboliques et les transitions valides entres états sont données par une relation $\hookrightarrow_{\text{sym}} \subseteq \mathbb{S}_{\text{sym}} \times \mathbb{S}_{\text{sym}}$. Cette sémantique suit exactement la même structure que celle de la définition 2. Toutefois, une différence notable est que les instructions `if` et `while` sont exécutées de manière *non-déterministe*. L'exécution de ces instructions peut se poursuivre dans l'une ou l'autre branche au choix. Dans tous les cas, les conditions booléennes (ou leurs négations) sont évaluées de manière symbolique avant d'être ajoutées à l'ensemble des hypothèses courantes.

Définition 4 (Sémantique symbolique de BUG)**.**

$$\frac{}{\langle \varphi, \hat{M}, x \texttt{ = } e\rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}[x \leftarrow [\![e]\!]^{\text{sym}}_{\hat{M}}], \texttt{skip}\rangle}$$

$$\frac{}{\langle \varphi, \hat{M}, \texttt{skip ; } s\rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}, s\rangle} \qquad \frac{\langle \varphi, \hat{M}, s_1\rangle \hookrightarrow_{\text{sym}} \langle \varphi, \hat{M}, s_2\rangle}{\langle \varphi, \hat{M}, s_1 \texttt{ ; } s_3\rangle \hookrightarrow \langle \varphi, \hat{M}, s_2 \texttt{ ; } s_3\rangle}$$

$$\frac{}{\langle \varphi, \hat{M}, \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and } [\![b]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, i_1\rangle}$$

$$\frac{}{\langle \varphi, \hat{M}, \texttt{if } b \texttt{ then } i_1 \texttt{ else } i_2\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and not } [\![b]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, i_2\rangle}$$

$$\frac{}{\langle \varphi, \hat{M}, \texttt{while } b \texttt{ do } i\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and } [\![b]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, i \texttt{ ; while } b \texttt{ do } i\rangle}$$

$$\frac{}{\langle \varphi, M, \texttt{while } b \texttt{ do } i\rangle \hookrightarrow_{\text{sym}} \langle \varphi \texttt{ and not } [\![b]\!]^{\text{sym}}_{\hat{M}}, \hat{M}, \texttt{skip}\rangle}$$

Correction et complétude. Il existe un lien formel entre les sémantiques concrète et symbolique. D'une part, toute exécution symbolique simule un ensemble d'exécution concrètes. D'autre part, toute exécution concrète peut être simulée par une exécution symbolique. On dit que la sémantique symbolique est une abstraction correcte et complète de la sémantique concrète. Plus précisément, la correction assure que toute exécution symbolique peut être *transformée* en exécution concrète en choisissant une mémoire initiale qui satisfait les hypothèses collectées durant l'exécution symbolique.

Théorème 1 (Correction)**.** *Soit i_1 et i_2 deux instructions, $\hat{M}$ une mémoire symbolique, et φ une expression booléenne tels que $\langle \texttt{true}, \hat{M}_{init}, i_1\rangle \hookrightarrow^*_{sym} \langle \varphi, \hat{M}, i_2\rangle$. Alors, pour toute mémoire concrète M telle que $[\![\varphi]\!]_M = true$, on a $\langle M, i_1\rangle \hookrightarrow^* \langle M \circ \hat{M}, i_2\rangle$*

Ici, la notation $M \circ \hat{M}$ désigne la mémoire concrète obtenue en remplaçant toutes les variables libres dans la mémoire symbolique $\hat{M}$ par leurs valeurs dans M. En corollaire du théorème 1, on obtient également que l'exécution symbolique peut être utilisée pour détecter précisément les bugs.

Corollaire 1. *Soit p un programme, $\hat{M}$ une mémoire symbolique, et φ une expression booléenne tels que $\langle \texttt{true}, \hat{M}_{init}, p\rangle \hookrightarrow^*_{sym} \langle \varphi, \hat{M}, \texttt{fail ; } \ldots\rangle$. Alors p part en erreur pour toute mémoire initiale qui satisfait φ.*

La complétude assure que toute exécution concrète à partir d'une mémoire donnée peut être simulée par une exécution symbolique de sorte que les hypothèses collectées *généralisent* la mémoire en question.

Théorème 2 (Complétude)**.** *Soient i_1 et i_2 deux instructions et M_1 et M_2 deux mémoires tels que $\langle i_1, M_1\rangle \hookrightarrow^* \langle i_2, M_2\rangle$. Alors il existe une expression booléenne φ et une mémoire symbolique $\hat{M}$ tels que $\langle \texttt{true}, i_1, \hat{M}_{init}\rangle \hookrightarrow^*_{sym} \langle \varphi, i_2, \hat{M}\rangle$ et $M_2 = M_1 \circ \hat{M}$ et $[\![\varphi]\!]_{M_1} = \texttt{true}$.*

En corollaire de 2, on obtient que toute exécution qui part en erreur peut être simulée par exécution symbolique.

Corollaire 2. *Soit p un programme qui part en erreur si il est exécuté depuis une certaine mémoire M. Alors, il existe une expression booléenne φ et une mémoire symbolique $\hat{M}$ tels que $\langle \texttt{true}, \hat{M}_{init}, p\rangle \hookrightarrow^*_{sym} \langle \varphi, \hat{M}, \texttt{fail ; } \ldots\rangle$ et M satisfait φ.*

L'ensemble de ces résultats de correction et de complétude peuvent être énoncés et prouvés en Coq. Un schéma de preuve est présenté dans l'article [dBB19] et une formalisation Coq est détaillée dans [CS23].

5 Implémentation vérifiée d'un détecteur de bugs

Un interpréteur symbolique correct et complet. La sémantique symbolique décrite dans la partie précédente n'est pas directement exécutable. En particulier, cette sémantique est non-déterministe puisque l'on peut choisir de poursuivre l'exécution de deux manières différentes lorsqu'un branchement est rencontré. Pour rendre la sémantique symbolique exécutable, il faut résoudre ce non-déterminisme. Pour se faire, on commence par définir une fonction `expand` : $\mathbb{S}_{\text{sym}} \to$ `list` $\mathbb{S}_{\text{sym}}$ qui associe à chaque état symbolique la liste de tous ses successeurs possibles. L'implémentation d'une telle fonction est immédiate si l'on suit les règles décrites à la définition 4. Cette fonction d'*expansion* des états symboliques reflète la sémantique symbolique au sens du théorème suivant.

Théorème 3 (Correction et complétude de la fonction d'expansion)**.** *Soient* $s_1, s_2 \in \mathbb{S}_{sym}$*,* $s_1 \hookrightarrow_{sym} s_2$ *si et seulement si* $s_2 \in$ ***`expand`*** s_1*.*

Pour calculer l'ensemble des états (symboliques) accessibles d'un programme p, on itère simplement la fonction `expand` en partant de l'état $\langle \texttt{true}, \hat{M}_{\text{init}}, p \rangle$. En revanche, ce processus peut ne jamais terminer. En effet, il se peut qu'un état génère une suite infinie de successeurs si le programme analysé ne termine pas.

Une approche paresseuse pour résoudre le problème de la terminaison. Pour résoudre le problème de non-terminaison, on se contente d'énumérer l'ensemble des états accessibles sous la forme d'une séquence infinie évaluée de manière *paresseuse*. En langage OCaml, les séquences infinies peuvent être implémentées en utilisant des *suspensions* (grâce au module `Lazy` par exemple). En Coq, on utilise des listes co-inductives, aussi appelées *streams*. Ci-dessous, nous proposons une implémentation possible des *stream* en OCaml (à gauche) et en Coq (à droite).

```
type 'a cell =
  | snil
  | scons of 'a * 'a stream
with 'a stream = 'a cell Lazy.t
```

```
CoInductive stream A :=
  | snil
  | scons (x : A) (xs : stream A)
  .
```

En utilisant le type `stream`, on peut écrire en Coq une fonction récursive `reachable` : `list` $\mathbb{S}_{\text{sym}} \to$ `stream` $\mathbb{S}_{\text{sym}}$ qui énumère tous les états symboliques accessibles en partant d'une liste d'états initiaux.

```
CoFixpoint reachable l :=
  match l with
  | [] => snil
  | s::l => scons s (reachable (l ++ expand s))
  end.
```

A l'aide du théorème 3, on peut prouver que le *stream* `reachable` l énumère bien exactement l'ensemble des états symboliques $\hookrightarrow_{\text{sym}}$-accessibles depuis l. Notons toutefois que cela n'est pas vrai si la liste des états courants est étendue à gauche au lieu d'être étendue à droite d'un appel récursif à l'autre ! En effet, en concaténant les successeurs à droite, on réalise un parcours en largeur de l'espace d'états. Cet argument joue un role crucial dans la preuve et garantit l'exhaustivité du parcours, même lorsqu'il existe une infinité d'états accessibles.

Théorème 4. *Soit* l *une liste d'états symboliques et* $s_2 \in \mathbb{S}_{sym}$*.* s_2 *apparaît dans la séquence* ***`reachable`*** l *si et seulement s'il existe un état* $s_1 \in l$ *tel que* $s_1 \hookrightarrow^*_{sym} s_2$*.*

En conséquence de ce théorème et des théorèmes de correction et de complétude de la sémantique symbolique, on obtient le corollaire suivant.

Corollaire 3. *Un programme p part en erreur si et seulement s'il existe une mémoire symbolique $\hat{M}$ et une expression booléenne satisfiable φ tels qu'un état de la forme $\langle \varphi, \hat{M}, \texttt{fail ; } \ldots \rangle$ apparaît dans la séquence* `reachable` $[\langle \texttt{true}, \hat{M}_{init}, p \rangle]$

Implémentation finale d'un détecteur de bug. Le corollaire 3 suggère une manière fiable d'implémenter l'algorithme de détection de bug 1. Il suffit de balayer le *stream* des états accessibles jusqu'à trouver un état symbolique dont la condition associée est satisfiable. Dans ce cas, le programme analysé contient nécessairement un bug. Si aucun état d'erreur n'est rencontré et que le *stream* est épuisé, le programme analysé ne contient pas de bugs. Si la recherche n'aboutie pas, on peut laisser l'utilisateur interrompre l'algorithme après un certain temps.

En pratique, on se contente de vérifier la fonction `reachable` en Coq. On génère ensuite automatiquement une version OCaml de cette fonction en utilisant le mécanisme d'extraction de Coq [Let08]. La lecture d'un fichier source, l'appel à la fonction `reachable` et son parcours sont programmés en OCaml. Dans notre implémentation, nous utilisons le solveur de contrainte Z3 [dMB08] pour tester la satisfiabilité des conditions booléennes. La question de vérifier la correction du solveur de contrainte sort du cadre de cet article. Toutefois, une solution plus fiable serait de prouver la correction d'un solveur de contraintes en Coq et de faire en sorte que la fonction `reachable` filtre les états dont la condition est non-satisfiable. Prouver la correction des solveurs de contraintes est un sujet de recherche à part entière qui a déjà reçu beaucoup d'attention [AFG+11, Les11, CDG12, BFW17]. Exploiter l'état de l'art dans ce domaine est une piste de réflexion intéressante pour de futurs travaux.

6 Conclusion

S'il est admis qu'il est utile voire nécessaire de vérifier la fiabilité des outils de preuve, les outils de test ont reçu moins d'attention. Dans cet article, nous avons proposé de vérifier la correction d'un détecteur de *bugs* automatique à l'aide de l'assistant de preuve Coq. En particulier, nous avons prouvé qu'il énumère toutes les erreurs à l'exécution. En supposant l'accès à un solveur de contraintes fiable, nous avons également prouvé que notre algorithme ne détecte que des vraies erreurs. Ce premier prototype ouvre la voie à de nombreuses possibilités d'extensions futures. En particulier, il serait intéressant de lever l'hypothèse d'accès à un solveur correct. Un autre angle de recherche, orthogonal, serait de se concentrer sur l'analyse de langages plus réalistes pour lesquels il existe une sémantique formalisée en Coq comme par exemple C [BL09] ou JavaScript [BCF+14].

Accessibilité des sources. Les théorèmes et des définitions énoncés dans cet article ont tous été formalisés en Coq. Les preuves sont disponibles à l'adresse `https://github.com/acorrenson/minibug`. Une version plus complète comprenant l'extraction vers OCaml peut être consultée librement à l'adresse `https://github.com/acorrenson/WiSE`.

Remerciements. Je remercie Dominic Steinhöfel et Sébastien Bardin pour les nombreuses discussions que nous avons eu sur la *sémantique des bugs* et sur la correction des interprètes symboliques. Je remercie également mes camarades de promotion Charles de Haro et Naïm Moussaoui Remil pour les débats agités mais constructifs que nous avons eu sur les critères de *soundness* des outils d'analyse statique. Enfin, merci au commité des JFLA pour les commentaires détaillés qui ont guidés la rédaction de la version finale de cet article.

Financements. This work was supported by the European Research Council (ERC) Grant HYPER (No. 101055412). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Références

[AFG+11] Michael Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry et Benjamin Werner : A modular integration of sat/smt solvers to coq through proof witnesses. *In* Jean-Pierre Jouannaud et Zhong Shao, éditeurs : *Certified Programs and Proofs*, pages 135–150, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[BCBdODF09] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe et Pascal Fontaine : verit : An open, trustable and efficient smt-solver. *In* Renate A. Schmidt, éditeur : *Automated Deduction – CADE-22*, pages 151–156, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BCD+11] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds et Cesare Tinelli : CVC4. *In* Ganesh Gopalakrishnan et Shaz Qadeer, éditeurs : *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 de *Lecture Notes in Computer Science*, pages 171–177. Springer, 2011.

[BCF+14] Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt et Gareth Smith : A trusted mechanised javascript specification. *SIGPLAN Not.*, 49(1):87–100, jan 2014.

[BFW17] Jasmin Christian Blanchette, Mathias Fleury et Christoph Weidenbach : A verified sat solver framework with learn, forget, restart, and incrementality. *In Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 4786–4790, 2017.

[BL09] Sandrine Blazy et Xavier Leroy : Mechanized semantics for the clight subset of the c language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[CDG12] Matthieu Carlier, Catherine Dubois et Arnaud Gotlieb : A certified constraint solver over finite domains. *In* Dimitra Giannakopoulou et Dominique Méry, éditeurs : *FM 2012 : Formal Methods*, pages 116–131, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[CS23] Arthur Correnson et Dominic Steinhöfel : Engineering a formally verified automated bug finder. *In* Satish Chandra, Kelly Blincoe et Paolo Tonella, éditeurs : *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1165–1176. ACM, 2023.

[dBB19] Frank S. de Boer et Marcello Bonsangue : On the nature of symbolic execution. *In* Maurice H. ter Beek, Annabelle McIver et José N. Oliveira, éditeurs : *Formal Methods – The Next 30 Years*, pages 64–80, Cham, 2019. Springer International Publishing.

[DKFW10] Klaus Dräger, Andrey Kupriyanov, Bernd Finkbeiner et Heike Wehrheim : Slab : A certifying model checker for infinite-state concurrent systems. *In* Javier Esparza et Rupak Majumdar, éditeurs : *Tools and*

Algorithms for the Construction and Analysis of Systems, pages 271–274, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[dMB08] Leonardo de Moura et Nikolaj Bjørner : Z3 : An efficient smt solver. *In* C. R. Ramakrishnan et Jakob Rehof, éditeurs : *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[ELN+13] Javier Esparza, Peter Lammich, René Neumann, Tobias Nipkow, Alexander Schimpf et Jan-Georg Smaus : A fully verified executable ltl model checker. *In* Natasha Sharygina et Helmut Veith, éditeurs : *Computer Aided Verification*, pages 463–478, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[FMEH20] Andrea Fioraldi, Dominik Maier, Heiko Eissfeldt et Marc Heuse : Afl++ : Combining incremental steps of fuzzing research. *In Proceedings of the 14th USENIX Conference on Offensive Technologies*, WOOT'20, USA, 2020. USENIX Association.

[FSMAG20] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun et Philippa Gardner : Gillian, part i : A multi-language platform for symbolic execution. *In Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 927–942, New York, NY, USA, 2020. Association for Computing Machinery.

[God05] Patrice Godefroid : The soundness of bugs is what matters (position statement). *In BUGS'2005 (PLDI'2005 Workshop on the Evaluation of Software Defect Detection Tools)*, 2005.

[JLB+15] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy et David Pichardie : A formally-verified c static analyzer. *In Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 247–259, New York, NY, USA, 2015. Association for Computing Machinery.

[Kin76] James C. King : Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[LBK+16] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister et Christian Ferdinand : Compcert – a formally verified optimizing compiler. *In ERTS 2016 : Embedded Real Time Software and Systems*. SEE, 2016.

[Les11] Stephane Lescuyer : *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. Theses, Université Paris Sud - Paris XI, janvier 2011.

[Let08] Pierre Letouzey : Extraction in coq : An overview. *In* Arnold Beckmann, Costas Dimitracopoulos et Benedikt Löwe, éditeurs : *Logic and Theory of Algorithms*, pages 359–369, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[Nam01] Kedar S. Namjoshi : Certifying model checkers. *In* Gérard Berry, Hubert Comon et Alain Finkel, éditeurs : *Computer Aided Verification*, pages 2–13, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[O'H19] Peter W. O'Hearn : Incorrectness logic. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[PHD+14] Zoe Paraskevopoulou, Catalin Hritcu, Maxime Dénès, Leonidas Lampropoulos et Benjamin C. Pierce : A coq framework for verified property-based testing (extended abstract). 2014.