Design de langage dédié orienté vers la preuve pour le logiciel critique

Denis Merigoux [1]

Cet article est le résumé de la thèse de doctorat [2] de l'auteur ayant reçu le prix de thèse Gilles Kahn 2022 décerné par la Société Informatique de France.

© Inria / Photo B. Fourrier.

La vérification de programmes consiste en l'analyse de programmes informatiques vus comme des artefacts formels, afin de prouver l'absence de certaines catégories de *bugs* avant l'exécution. Mais pour utiliser un cadriciel de vérification de programmes, il faut auparavant traduire le code source originel du programme dans le langage formel du cadriciel. De plus, il est possible d'utiliser plusieurs cadriciels de vérification pour prouver des propriétés de plus en plus spécialisées à propos du programme. Ceci pose un véritable défi au champ de recherche de la vérification de programme, qui a vu ces dernières décennies ses efforts se concentrer dans le perfectionnement de quelques assistants de preuve généralistes (Coq, Isabelle, etc.). En effet, les projets de vérification de programmes dans le monde réel impliquent souvent du code écrit dans des langages non propices à la formalisation et des spécifications ambiguës. Si de nombreux travaux de recherche ont permis d'élaborer des outils très puissants pour prouver la correction de programmes écrits dans ces outils, peu d'attention a été portée à la manière dont on amène les programmes originaux à l'intérieur des outils de vérification. Sur la chaîne de confiance présentée en figure 1,

1. Inria Paris, denis.merigoux@inria.fr.
2. https://theses.hal.science/tel-03622012.

DOI:10.48556/SIF.1024.21.167 **1024** – Bulletin de la SIF, numéro 21, avril 2023, p. 167–170

cela revient à dire que si les maillons (1), (2) et (3) sont solides, le maillon (4) est généralement peu considéré et peut devenir le cheval de Troie de futurs bugs. Le manque d'attention de la communauté des méthodes formelles porté à ce maillon et à l'encodage correct d'un domaine vers une spécification formelle est justement lié au manque de formalisme et à l'ambiguïté auxquels les experts du domaine sont habitués. Par exemple, un programme informatique appliquant automatiquement le droit possède une spécification extrêmement informelle qui doit d'être interprétée intensivement par des juristes avant d'en tirer quelque chose de formel.

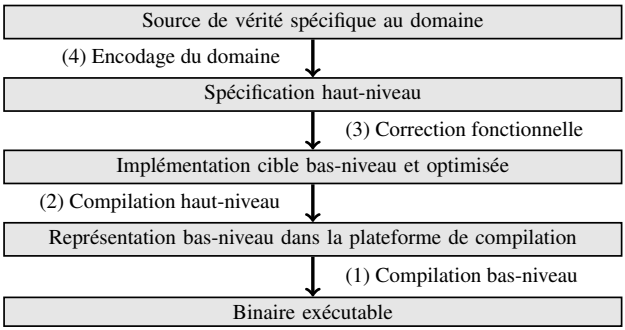

FIGURE 1. Schéma sommaire d'une chaîne typique de vérification de programmes utilisant un assistant de preuve.

Pareillement, même si les cryptographes décrivent leurs primitives cryptographiques à l'aide de notations mathématiques, il n'est pas rare de trouver des bugs dans les spécifications formelles dérivées de ces primitives cryptographiques par des informaticiens.

Pour répondre à cette problématique, ainsi qu'au besoin de traductions multiples du programme source vers différents cadriciels de vérification ayant chacun leur paradigme de preuve, nous défendons l'utilisation de langages dédiés orientés vers la preuve. Ces langages dédiés (DSL) devraient être pensés comme une surcouche au-dessus des cadriciels de preuves, avec un design qui incorpore et distribue les obligations de preuves entre les prouveurs. De plus, le programme originel est souvent déjà traduit depuis des spécifications d'exigences informelles liées au domaine d'activité afférent. Afin de raffermir le maillon le plus haut de la chaîne de confiance, nous soutenons que les langages dédiés orientés vers la preuve peuvent aider les experts du domaine à relire la spécification du programme, spécification à la base d'ultérieurs développements d'implémentations vérifiées.

L'utilisation de langages dédiés orientés vers la preuve doit, selon nous, s'inscrire dans une méthodologie plus globale présentée en figure 2 dont le but est d'augmenter le niveau d'assurance d'un logiciel critique existant.

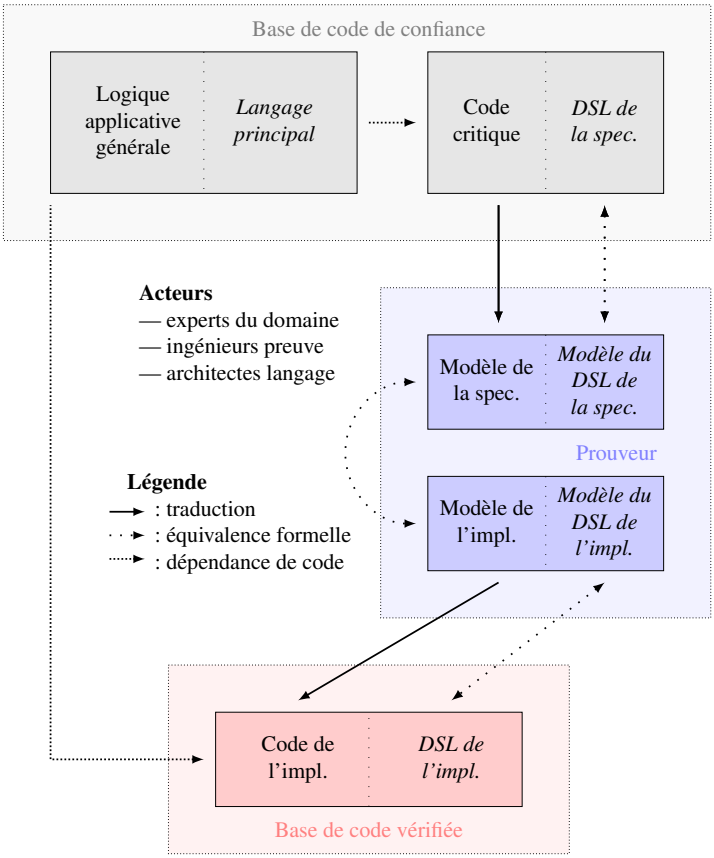

FIGURE 2. Illustration d'ensemble de la méthodologie proposée dans la thèse. Dans chaque bloc représentant un morceau du logiciel, la partie gauche décrit le code tandis que la partie droite décrit le langage dans lequel le code est écrit. Les traductions ou preuves d'équivalence peuvent être faites manuellement ou mécaniquement.

Cette méthodologie que nous avons éprouvée expérimentalement commence par découper dans un logiciel existant la partie réellement critique sur laquelle le travail va porter. Puis, nous simplifions dans cette partie critique en enlevant au maximum les artefacts d'implémentation et d'optimisation afin de constituer une spécification qui sera le point de départ de la suite. Vient alors le travail central de conception d'un langage dédié orienté vers la preuve pour capturer de manière concise et adaptée la

logique à l'œuvre dans la spécification du logiciel critique. C'est là que nous faisons relire la spécification par des experts du domaine afin d'en assurer la validité. Ensuite, nous transposons le code critique ainsi qu'un modèle du langage dédié dans un prouveur (ou plusieurs) dans lequel sera réalisée la preuve de correction fonctionnelle entre la spécification et une implémentation de plus bas-niveau et optimisée selon les besoins en performance ou en portabilité. Enfin, cette implémentation est transformée en du code exécutable qui peut être relié au reste de l'application originale.

La thèse traite du design et de l'utilité des langages dédiés orientés vers la preuve, ainsi que de la méthodologie présentée ci-dessus, au travers de cinq études de cas. Ces études de cas portent sur des domaines allant des implémentations cryptographiques aux systèmes experts légaux, et sur des logiciels à haut niveau d'assurance actuellement utilisés en production. Chaque étude de cas donne son nom à l'un des chapitres de cette dissertation. LibSignal* est une implémentation vérifiée du protocole cryptographique Signal à destination du Web. Hacspec est un langage dédié pour les spécifications cryptographiques en Rust. Steel est un cadriciel de vérification de programmes utilisant la logique de séparation à l'intérieur de l'assistant de preuve F*. Mlang est un compilateur pour un langage dédié aux calculs fiscaux utilisé par la DGFiP. Enfin, Catala est un nouveau langage qui permet l'encodage de spécifications législatives dans un code source exécutable et analysable. Vous pouvez retrouver une description vulgarisée de Catala dans le précédent numéro de 1024 [3].

3. https://doi.org/10.48556/SIF.1024.20.77.