Spécification et vérification de systèmes paramétrés

Alain Giorgetti

Ecole doctorale SPIM : Sciences Pour l'Ingénieur et Microtechniques

HABILITATION À DIRIGER DES RECHERCHES
DE L'ÉTABLISSEMENT UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

PRÉPARÉE À L'UNIVERSITÉ DE FRANCHE-COMTÉ

Spécialité : Informatique

présentée par

Alain Giorgetti

Spécification et vérification de systèmes paramétrés

soutenue à Besançon le 1er décembre 2017 devant le jury composé de :

Mme Sandrine BLAZY	Professeur à l'Université de Rennes 1	Rapporteur
Mme Pascale LE GALL	Professeur à l'Université Paris-Saclay	Rapporteur
M. Pascal SCHRECK	Professeur à l'Université de Strasbourg	Rapporteur
M. Jean-Luc BARIL	Professeur à l'Université de Bourgogne	Examinateur
M. Stephan MERZ	Directeur de recherche Inria à Nancy	Examinateur
M. Pierre-Cyrille HEAM	Professeur à l'Université de Franche-Comté	Directeur

Spécification et vérification de systèmes paramétrés

Mémoire d'habilitation à diriger des recherches

Alain Giorgetti
FEMTO-ST/DISC
Université de Franche-Comté

Soutenue le 1er décembre 2017 devant le jury composé de :

Mme Sandrine Blazy	Rapporteur	Professeur à l'Université de Rennes 1
Mme Pascale Le Gall	Rapporteur	Professeur à l'Université Paris-Saclay
M. Pascal Schreck	Rapporteur	Professeur à l'Université de Strasbourg
M. Jean-Luc Baril	Examinateur	Professeur à l'Université de Bourgogne
M. Stephan Merz	Examinateur	Directeur de recherche Inria à Nancy
M. Pierre-Cyrille Héam	Directeur	Professeur à l'Université de Franche-Comté

Remerciements

Je remercie vivement Sandrine Blazy, Pascale Le Gall et Pascal Schreck d'avoir accepté d'être rapporteurs de ce mémoire. Je remercie également Stephan Merz, Jean-Luc Baril et Pierre-Cyrille Héam d'avoir participé au jury.

Mes travaux sont les fruits de collaborations avec de nombreux chercheurs, que je tiens à remercier collectivement ici. L'aventure a commencé avec Didier Arquès, professeur à l'université de Marne-la-Vallée, qui a accepté de diriger ma thèse à distance, lorsque j'étais enseignant, d'abord en lycée à Besançon et Vesoul, puis à l'université Henri Poincaré à Nancy. Merci au LORIA (Laboratoire lOrrain de Recherche en Informatique et ses Applications) et à son équipe PolKA (Polynômes, Combinatoire et Arithmétique) de m'avoir offert un environnement scientifique de qualité durant mon séjour à Nancy. Merci à l'université de Franche-Comté, à l'U.F.R. Sciences et Techniques, au département informatique et au LIFC (Laboratoire d'Informatique de l'université de Franche-Comté) de m'avoir accueilli ensuite. Merci aux membres des autres disciplines de l'institut FEMTO-ST de m'offrir des opportunités de recherches pluri-disciplinaires.

Je suis particulièrement reconnaissant à tous mes co-auteurs de publications et communications de recherche d'avoir partagé ses moments intenses de découverte, de rédaction et de diffusion de nos résultats. Merci aussi aux doctorants et stagiaires de recherche qui ont choisi de découvrir la recherche en ma compagnie.

J'adresse ma reconnaissance à tous les membres du DISC (Département d'Informatique des Systèmes Complexes) de l'institut FEMTO-ST et du département d'enseignement d'informatique, pour avoir allégé mes responsabilités collectives durant mon CRCT (Congé pour Recherches ou Conversions Thématiques) et toute la période de rédaction de ce mémoire. En particulier, merci à Hervé Guyennet, Bruno Tatibouët et Jacques Julliand de m'avoir remplacé comme responsable de la deuxième année du master informatique. Je suis aussi reconnaissant à tous les personnels administratifs et techniques pour la qualité de leurs services durant toutes ces années.

Un grand merci à Jocelyne et Joseph Rouyer, pour leur accueil et leur amitié indéfectible dès mon arrivée à Nancy. Merci à Jacques Julliand pour son amitié, son soutien et sa confiance en mes recherches et initiatives en enseignement. Je dédie ce mémoire à Françoise Bellegarde, qui m'a transmis sa passion et son savoir-faire d'enseignement des méthodes formelles, et qui nous a malheureusement quittés le 28 août 2016.

Dans l'ordre chronologique de notre première rencontre, merci aussi à Catherine Oriat, Marie-Christine et Jean-Paul Haton, Jean-Pierre Jacquot, Brigitte Jaray, Lucette Aubry, Paul Zimmermann, Michaël Rusinowitch, Jean-Michel Hufflen, Pierre-Etienne Moreau, Catherine Dubois, Timothy Walsh, Henri Lombardi, Guillaume Chapuy et Pierre Lescanne. Toutes mes excuses à celles et ceux que je n'ai pas cités. Je ne vous oublie pas et je vous adresse une pensée amicale.

Enfin, rien n'aurait été possible sans le soutien de ma famille et de mes amis, que je remercie mille fois d'être à mes côtés.

Table des matières

Chapitre 1

Introduction

Sommaire

Ce mémoire est issu des activités de recherche que j'ai menées depuis 2001 à l'Université de Franche-Comté, dans le domaine de la vérification formelle de spécifications et de programmes.

Dans la partie 1.1, je présente mes motivations pour étudier ce sujet. La partie 1.2 précise le cadre de ce mémoire et la partie 1.3 en décrit la structure.

1.1 Motivations

Les notions de "vérification", "preuve de programmes" et "procédure de décision" évoquées dans cette partie seront définies précisément dans la partie suivante.

La motivation principale de mes recherches est de faciliter la vérification de programmes par preuve, dans des domaines où sa primauté est la plus manifeste, en contribuant aux défis suivants :

— Combiner la preuve de programmes avec d'autres méthodes de vérification, plus faciles à appliquer et à analyser.
— Démontrer formellement la correction des méthodes de vérification.
— Étendre la portée de la preuve automatique, par la conception de nouvelles procédures de décision.
— Isoler des "niches" de programmation et de spécification où la preuve fonctionne le mieux, au prix d'une éventuelle adaptation.

1.2 Cadre et contenu du mémoire

En quelques mots, ce mémoire porte sur l'automatisation de la preuve de programmes. Comme cette expression courte est trompeuse et recouvre des réalités multiples, j'ai choisi un titre plus long. Les parties 1.2.1 à 1.2.4 définissent soigneusement chaque terme employé dans ce titre. Ces définitions s'appuient sur d'autres notions (soulignées) qui sont définies ensuite. Les systèmes et les propriétés de ces systèmes que nous voulons vérifier sont respectivement détaillés dans les parties 1.2.3 et 1.2.5. Les méthodes de vérification adaptées à ces systèmes et propriétés sont décrites dans la partie 1.2.6. Le tout, et plus particulièrement ce qui est écrit en **gras**, fixe le cadre de ce mémoire.

1.2.1 Spécifier

Spécifier, qu'est-ce que c'est ?

Le verbe "spécifier" et le substantif "spécification" ne sont pas des mots du langage le plus courant. Ils désignent une activité technique essentielle dans divers domaines, en particulier en informatique, qui doit être bien distinguée de celle de développement de logiciel, et associée à elle.

Dans le cadre de l'informatique, on désigne par "spécification" la description initiale d'un problème informatique. Une spécification guide le développement d'un logiciel en mettant en avant ses principales caractéristiques attendues, sans entrer dans les détails. Une spécification est dite *fonctionnelle* si elle décrit les fonctionnalités du logiciel. D'autres spécifications, dites non fonctionnelles, peuvent par exemple décrire les performances ou l'architecture du logiciel. Une spécification est dite *formelle* si elle est exprimée dans un langage exploitable par une machine. Ce langage est appelé *langage de spécification formelle*. Certains de ces langages seront présentés dans le chapitre 4.

Sous le nom de "spécification", ce mémoire ne traite que de spécifications fonctionnelles et formelles.

Pourquoi spécifier ?

Spécifier est une activité essentielle pour améliorer la qualité du logiciel. On peut voir la spécification comme l'expression d'un problème informatique, qu'on souhaite résoudre. Résoudre un problème informatique, c'est presque toujours en donner une solution algorithmique.

Ce point de vue des spécifications comme problèmes et des programmes comme solutions est celui de Carroll Morgan :

> "One can [...] regard computer programming, or program development as it is known these days, in terms of 'solving' and 'solutions'. [...Some] programs will be regarded as solving others (we will say 'refining') ; and some programs will be so simple we will regard them as solutions (we will say 'code')." [Mor98, Preface].

Le degré de précision d'une solution peut être très variable. Il peut s'agir d'une description succincte du principe d'un algorithme, dans un article scientifique, ou aller jusqu'à la diffusion d'une implémentation au sein d'un logiciel. Dans mes activités de recherche, je ne souhaite pas me spécialiser dans une certaine forme de solution, mais au contraire travailler sur un même problème à plusieurs niveaux de précision. En effet, je suis convaincu de l'intérêt majeur de connecter entre elles toutes ces formes de solutions, par un lien le plus explicite possible. Ce lien peut prendre la forme d'un chemin menant d'une description informelle d'un problème à un logiciel qui en implémente une solution exécutable. Tracer et maintenir un tel chemin est une tâche ardue et un enjeu majeur du génie logiciel. Le chemin le plus direct est le *raffinement* [Mor98], qui propose de dériver un code exécutable à partir d'une spécification abstraite. Le raffinement est une méthode séduisante mais souvent trop exigeante pour être appliquée efficacement. C'est pourquoi nous ne traitons pas de raffinement, et considérons l'élaboration relativement indépendante de deux systèmes : un système abstrait, proche du problème initial, et un système concret, proche d'une solution exécutable. Tous deux sont considérés comme deux spécifications formelles. Ainsi, **la notion de spécification formelle considérée ici rassemble les notions de spécification déclarative (logique) et opérationnelle, car exécutable (algorithme séquentiel, processus en parallèle).**

1.2.2 Vérifier

Lorsque deux systèmes décrivent ou traitent le même problème, la question qui se pose naturellement est celle de la correction du système concret vis-à-vis du système abstrait. On parle aussi de "cohérence" entre les deux systèmes. Nous décrivons dans cette partie les enjeux (le pourquoi) et les principes (le comment) de la vérification du logiciel, brièvement et de manière générale.

Pourquoi vérifier ?

La vérification nécessite un effort et a un coût qui ne sont souvent justifiés que pour des systèmes dits "critiques", qui mettent en jeu des vies humaines ou de lourds enjeux économiques. Mais une technique de vérification mise au point dans un cadre critique, qui serait fortement automatisée et outillée, peut ensuite être appliquée à moindre coût à des systèmes non critiques. Inversement, une technique de vérification initialement mise au point sur des systèmes quelconques peut servir de base à son adaptation à des systèmes critiques. De ce point de vue, la criticité des systèmes n'est pas la motivation principale pour la conception de techniques de vérification.

Un fort besoin de vérification se fait sentir lorsque la description d'un système, même non critique, est devenue trop complexe pour convaincre seule que le système satisfait ses propriétés attendues, celles pour lesquelles il est en cours de réalisation. La vérification intervient alors pour rétablir la confiance. Elle opère souvent par confrontation de points de vue sur ce système. En pratique, la vérification s'inscrit dans la partie "évaluation" d'un processus de développement composé d'étapes de production et d'étapes d'évaluation qui contrôlent la qualité de cette production. C'est avant tout la complexité des systèmes étudiés qui motive les travaux de spécification formelle et de vérification présentés dans ce document.

Comment vérifier ?

On trouve dans la littérature deux grandes classifications des techniques de vérification. La première classification distingue les techniques de test, de model-checking et de preuve [Dad06, chapitre 2]. La seconde classification distingue les analyses statiques (sans exécution du système) et les analyses dynamiques (par exécution du système) [Pet15, partie 1.1]. Certaines techniques se situent à la frontière entre ces catégories, ou sont des combinaisons de techniques [Pet15, partie 2.1]. Plutôt que de disserter longuement sur ces catégories et la pertinence de ces classifications, nous inscrivons clairement notre travail parmi les analyses statiques, sous le nom de "**vérification déductive**". Cette expression sera définie et justifiée dans la partie 1.2.6.

Mécanisation/automaticité. Pour réduire le coût de vérification et rendre les méthodes de vérification accessibles au plus grand nombre, nous étudions essentiellement des solutions automatisables. Nous ne nous interdisons néanmoins pas d'utiliser des méthodes interactives, à travers des assistants de preuve, dans certains contextes difficiles.

1.2.3 Système

Mon travail de recherche s'est souvent situé en amont de la programmation, dans les étapes de conception où les spécifications sont décrites soit de manière logique, déclarative, soit par des modèles à états, comme les systèmes de transitions.

J'ai choisi le mot "système" pour unifier les notions de système réactif, d'automate, de programme impératif et de système d'inférence que j'ai successivement étudiés. Ce mémoire ne porte que sur des **systèmes à états**. La partie suivante décrit l'aspect paramétré de ces systèmes.

1.2.4 Paramétrer

De nombreux facteurs peuvent rendre un système complexe. Parmi eux, nous avons retenu la paramétrisation du système. Le plus souvent, le système n'aura qu'un paramètre, qui sera un entier naturel n, qu'on pourra considérer comme étant la *taille* du système.

Dans tous mes travaux, les spécifications sont paramétrées, et au moins un paramètre est dans un domaine infini. Sous cette hypothèse, seules des méthodes symboliques permettent de vérifier en une seule fois tous les systèmes finis obtenus en instanciant les paramètres d'un unique système paramétré sur un domaine infini.

En 2007, avec F. Bouquet, J.-F. Couchot et F. Dadeau, j'ai étudié l'impact des paramètres sur la génération automatique de tests à partir de modèles, pour satisfaire certains critères de

couverture du modèle [BCDG07]. Nous avons proposé une combinaison originale de techniques de preuve et de test adaptée aux structures de données paramétrées.

Ce mémoire ne porte que sur des systèmes paramétrés, dont la vérification requiert une approche symbolique.

1.2.5 Propriétés d'états

La vérification du logiciel est fondée sur un large éventail de méthodes formelles. Chacune d'elles est dédiée à un certain paradigme de programmation (programmation impérative, programmation fonctionnelle, systèmes concurrents, etc.) et se focalise sur certains aspects de la correction (terminaison, correction partielle, raffinement, etc.). Pour les systèmes à états, la plupart de ces aspects se réduisent à la question suivante : Comment un programme donné modifie la valeur de vérité de propriétés portant sur son état ? Sous le nom de "*propositions*", Robert W. Floyd [Flo67] a proposé d'utiliser ces propriétés d'états pour spécifier les ensembles d'états atteints lors de l'exécution d'un "*flowchart*". Plus tard, ces propriétés ont été insérées dans le code source des programmes impératifs, pour former des **programmes annotés**. Ainsi, la "cohérence d'un programme impératif avec sa spécification" est devenue la "correction du programme annoté avec cette spécification". L'idée suivante a été de considérer chaque annotation comme une instruction et de définir cette correction comme une **sémantique**, dite "**bloquante**", qui fait échouer l'exécution lorsqu'elle rencontre une annotation fausse.

Lorsque les programmes sont ainsi annotés, leur vérification s'inscrit dans le cadre de leur analyse statique, en prolongement de leur analyse syntaxique.

1.2.6 Vérification déductive

Les expressions "preuve de programmes" et "vérification de programmes" sont pratiques, mais abusives et imprécises. On ne vérifie pas des programmes. On vérifie qu'ils satisfont certaines propriétés attendues. De plus, la notion de programme n'est pas claire : Vérifie-t'on un code source ou un code compilé ? un algorithme ou une implémentation ? Quelles sont les méthodes de vérification utilisées ? Comme ces questions – et bien d'autres – se posent de manière récurrente dans ce domaine, je commence par fixer ici un vocabulaire précis, qui permettra ensuite de mieux décrire le sujet de ce mémoire.

Après avoir spécifié des systèmes paramétrés de deux manières différentes, d'une part en décrivant leur comportement, d'autre part en formalisant leurs propriétés, nous souhaitons que la cohérence entre ces deux formes de spécifications soit vérifiée de manière générique (une fois et une seule pour toutes les valeurs de leurs paramètres) et automatique. Dans ce but, nous suivons la démarche la plus répandue dans ce domaine, qui décompose la vérification en deux étapes : La première étape est une traduction automatique des conditions de correction en formules dans une logique adaptée. La seconde étape est l'application d'une procédure de décision (voir partie 1.2.7) de la validité des formules d'un fragment logique incluant tous les problèmes issus de la première étape. Dans toute la suite, nous désignons cette démarche en deux étapes par **vérification déductive**, ou seulement par **vérification** car c'est la seule méthode de vérification étudiée dans ce mémoire.

Comment populariser la preuve de programmes ? La preuve de programmes deviendra plus accessible lorsqu'elle sera intégrée aux outils et environnements de développement les plus répandus, qui évalueront sa faisabilité en analysant automatiquement la nature et la complexité des programmes à vérifier. **Nous contribuons à cet objectif en identifiant de manière plus ou moins précise des familles de programmes auxquelles il serait raisonnable d'appliquer cette méthode de vérification.** Ces niveaux de précision sont décrits dans le chapitre 3.

1.2.7 Procédure de décision

Un *problème de décision* est une question qui admet une réponse par "oui" ou par "non". Une *procédure de décision* est un algorithme qui termine toujours et établit toujours la réponse correcte, pour tous les problèmes de décision d'une même famille. Nous ne considérons ici que des procédures de décision de la validité ou de la satisfaisabilité des formules d'un fragment d'une logique formelle.

Concevoir, décrire et valider une nouvelle procédure de décision est une tâche délicate. D'une part, il faut connaître l'existant, déjà vaste. D'autre part, on dispose de peu de moyens de détecter des erreurs. Avant de s'y engager, il faut se munir des bons outils conceptuels. Nos contributions à la découverte de nouvelles procédures de décision sont présentées dans la partie 2.5.

1.3 Plan

Le chapitre 2 synthétise mes activités de recherche. Les chapitres suivants approfondissent le sujet de l'automatisation de la preuve de programmes.

Le chapitre 3 distingue trois niveaux de finesse dans la démarche de vérification déductive, et situe mes travaux dans cette classification. Le niveau le plus fin requiert d'expliciter la traduction d'un système spécifié en formulation logique de sa correction. La suite du mémoire présente cette traduction pour des programmes impératifs spécifiés par des annotations formelles.

Le chapitre 4 introduit les notions classiques utilisées dans la suite. Le chapitre 5 définit formellement diverses sémantiques des programmes impératifs, et leur correction partielle. Le chapitre 6 étend cette sémantique formelle aux programmes impératifs annotés. Le chapitre 7 synthétise le contenu de ce document et présente diverses perspectives de recherche.

Chapitre 2

Activités de recherche

Sommaire

Ce chapitre présente une synthèse de mes activités de recherche depuis l'obtention de mon doctorat en décembre 1998.

2.1 Résumé de thèse

Ma thèse, encadrée par D. Arquès, portait sur la combinatoire bijective et énumérative des cartes pointées sur une surface [Gio98a]. Une carte est le plongement d'un graphe dans une surface, à un homéomorphisme près. Ainsi, une carte est un objet topologique, énumérable en fonction du nombre de ses sommets, de ses arêtes et de ses faces. Les cartes admettent des symétries internes qui rendent leur énumération difficile. Mon travail traitait l'énumération des cartes pointées, le pointage supprimant toutes les symétries. Avant cette thèse le nombre exact de cartes pointées sur une surface donnée n'était connu que pour les surfaces de petit genre, comme la sphère (genre 0), le tore ou le plan projectif (genre 1). En effet, la complexité des méthodes de calcul de ces nombres augmente rapidement avec le genre des surfaces.

Un travail important de cette thèse a été de convertir l'une de ces méthodes de calcul en une preuve de l'existence d'une structure commune à toutes les séries génératrices de cartes pointées de genre non nul. Pour chaque surface orientable, j'ai réduit le problème à la détermination d'un polynôme, dont le degré est majoré par une fonction simple du genre de la surface. Un résultat analogue a été obtenu pour les cartes pointées sur les surfaces non orientables.

Des conséquences pratiques et une implémentation logicielle de tous ces résultats ont été décrites. De nouvelles formules explicites d'énumération ont été données. Indépendamment, une bijection géométrique nouvelle a été exposée, entre certaines cartes 2-coloriables et les partitions de polygones, énumérées par les nombres de Schröder.

Ce travail a donné lieu à trois rapports de recherche [AG97a, AG98, BG98], un article de vulgarisation [Gio98b], et trois publications internationales [AG99, AG00a, AG00b], dont l'une prolonge une communication internationale [AG97b].

2.2 Thématiques de recherche

Mes recherches s'inscrivent dans le domaine des méthodes formelles de spécification et de vérification de modèles et de programmes. Comme ce domaine est très vaste, je me suis focalisé sur certaines formes de vérification, pour certains modèles et certains programmes. Parmi les méthodes de vérification, j'ai choisi l'approche logique, dite "par preuve formelle", où la vérification s'apparente à une démonstration mathématique. Dans cette approche, qui combine raisonnement et calcul, j'ai privilégié les approches automatiques, en particulier celles fondées sur la réécriture.

Dans mon cheminement de recherche dans ce domaine, on peut distinguer les cinq thématiques suivantes, détaillées dans les parties 2.3 à 2.7 :

1. Vérification et synthèse d'invariants de modèles logico-ensemblistes, appliquées aux systèmes distribués uniformes, de 2001 à 2005 (partie 2.3).
2. Spécification par contrat et vérification de ces contrats, pour des langages impératifs (Java, C, PHP), depuis 2004 (partie 2.4).
3. Aide à la conception de procédures de décision, de 2008 à 2012 (partie 2.5).
4. Stratégies de réécriture et calcul formel, appliqués à la formalisation des méthodes multi-échelles, depuis 2010 (partie 2.6).
5. Application des méthodes formelles à l'étude de problèmes de combinatoire énumérative, depuis 2010 (partie 2.7).

Quelques travaux et résultats qui n'entrent pas dans cette classification, mais demeurent dans le domaine des méthodes formelles, sont décrits dans la partie 2.8.

2.3 Vérification et synthèse d'invariants de systèmes distribués uniformes

A mon arrivée au LIFC (Laboratoire d'Informatique de Franche-Comté) en 2001, de nombreux travaux de l'équipe TFC (Techniques Formelles et à Contraintes) portaient sur la spécification d'algorithmes et de systèmes par des machines abstraites B, basées sur la théorie des ensembles. En particulier, l'équipe proposait un outil de génération automatique de tests fonctionnels aux limites, à partir de ces spécifications logico-ensemblistes. Mais il manquait à cet outil un composant de vérification d'atteignabilité des objectifs de tests générés.

Lors de mon intégration dans l'équipe, avec F. Bellegarde, J.-F. Couchot et F. Dadeau, j'ai d'abord inscrit ce besoin dans le cadre du model-checking des systèmes infinis. Dans un cadre aussi général, la question de l'atteignabilité d'états est indécidable. Cependant, certaines familles de systèmes concrets peuvent être spécifiés dans des fragments décidables de la théorie des ensembles. Nous avons alors choisi d'orienter nos recherches vers l'identification de ces familles et de procédures de décision adaptées.

En 2002, lors du montage de l'équipe-projet Inria CASSIS (Combinaison d'Approches pour la Sécurité des Systèmes InfiniS) au LORIA (Laboratoire lOrrain de Recherche en Informatique et ses Applications), j'ai pris connaissance des travaux de S. Ranise et al. sur le prouveur haRVey [DR02] et sur la conception de nouvelles procédures de décision pour des théories du premier ordre fondées sur le calcul de superposition [ARR01, ARR03]. Ensemble, nous avons alors envisagé d'utiliser ces procédures pour prouver et déboguer des spécifications ensemblistes. Sur ce sujet, j'ai encadré le D.E.A. [1] de J.-F. Couchot, puis j'ai co-encadré sa thèse, avec F. Bellegarde. Parallèlement, en 2003,

1. Diplôme d'Etudes Approfondies. A partir de 2004, selon la réforme LMD (Licence, Master, Doctorat), le D.E.A. devient le master recherche.

j'ai encadré le D.E.A. de F. Dadeau, qui a défini et implémenté une traduction d'une famille de spécifications B vers haRVey, et une autre vers l'outil Mona, qui implémente la logique monadique du second ordre, décidable [DG03].

La thèse de J.-F. Couchot considère une famille de spécifications particulières, dite des "systèmes distribués uniformes", qui sont des systèmes composés de n programmes identiques, exécutés en parallèle et partageant un espace mémoire global. Dans le cadre de cette thèse, le problème d'atteignabilité est dual de celui du renforcement d'invariant. Comme J.-F. Couchot dans sa thèse, je présente ici nos résultats selon ce point de vue dual de la synthèse d'invariant. En 2005, j'ai co-encadré avec J.-F. Couchot le D.E.A. de X. Feng, qui a enrichi cette thèse avec d'autres exemples de systèmes distribués uniformes, et synthétisé des invariants prouvant leur propriété d'exclusion mutuelle ou de cohérence de cache.

Sur cette thématique, mes résultats sont :

— Une méthode efficace de vérification de machines abstraites B de grande taille [CDD+04]. Son exposé par S. Ranise a reçu le prix du meilleur exposé de ce workshop.
— La publication en revue d'une version détaillée et enrichie de cette méthode [CDGR03].
— Une présentation d'une implémentation libre de cette méthode et de son interface graphique lors de la conférence AFADL [CDGR04].
— Une méthode de vérification de propriétés de sûreté [CG04] pour des spécifications paramétrées sur des données tabulées.
— L'application d'une extension de cette méthode à la vérification uniforme de la sûreté de protocoles distribués synchrones [CGK05].

2.4 Contrats pour Java, C et PHP

La modélisation selon la méthode B n'est pratiquée que par une partie limitée de la communauté d'informatique. Pour diffuser plus largement mes résultats précédents, et les étendre, je me suis orienté, à partir de 2003, vers la spécification par contrat et la vérification de ces contrats pour des langages impératifs.

De 2004 à 2008, j'ai participé à la conception et à l'implémentation d'une méthode de vérification de propriétés temporelles par génération d'annotations JML (*Java Modeling Language*), pour la vérification d'applications en langage Java [GG07]. J'ai co-encadré la thèse de J. Groslambert (à 50% en 2004, puis à 30%) sur la vérification de programmes Java annotés en JML. Cette thèse, dirigée par J. Julliand et également co-encadrée par O. Kouchnarenko, était financée par l'ACI SI (Action Concertée Incitative - Sécurité Informatique) GECCOO (GÉnération de Code Certifié pour des applications Orientées Objet). J'ai aidé J. Groslambert à implémenter le prototype JAG (*JML Assertion Generator*), qui transforme des propriétés temporelles en annotations JML équivalentes [GG06a, GG06b].

Sur cette thématique, mes premiers résultats définissent une méthode de vérification de propriétés temporelles par génération automatique d'annotations JML. Le cas des propriétés de sûreté a été traité dans [GG06c], celui des propriétés de vivacité dans [GGJK08].

Mon travail avec J. Groslambert a été adapté au langage C et à son langage de spécification ACSL par N. Stouls, puis par V. Prevosto, et intégré dans le greffon Aoraï [SP17] de la plateforme Frama-C [KKP+12] du CEA.

Avec N. Stouls et J.-F. Couchot, j'ai élaboré une stratégie de sélection d'hypothèses qui automatise davantage la vérification de propriétés de sûreté. Cette stratégie a été appliquée avec succès à une étude de cas industrielle de *Trusted Computing* [CGS09].

En 2009, avec C. Marché, O. Kouchnarenko et E. Tushkanova, j'ai ajouté la généricité au langage KML de spécification de programmes Java [TGK09, TGMK09, TGMK10]. J'ai ainsi contribué à l'Action de Recherche Collaborative (ARC) Inria CeProMi (Certification de Programmes manipulant la Mémoire (Modularité, Invariants, Raffinement, Effets Cachés et Ordre Supérieur)).

De 2008 à 2011, en collaboration avec le CEA, j'ai participé à la conception de méthodes automatiques de recherche d'erreurs dans des programmes C [CKGJ10b, CKGJ12]. J'ai co-encadré (à 30%), avec Nikolai Kosmatov, ingénieur-chercheur au CEA, la thèse CEA d'O. Chebaro [Che11],

dirigée par J. Julliand, sur des méthodes de combinaison d'analyse statique, de simplification syntaxique (*slicing*) et de test structurel, pour lever des menaces d'erreurs à l'exécution dans les programmes C. Nous avons prouvé informellement la correction de la méthode proposée. Ces travaux sont implémentés dans l'outil SANTE [CKGJ10a, CKGJ11], qui relie l'outil de génération de test PathCrawler et la plate-forme d'analyse statique Frama-C. Des expérimentations ont montré, d'une part, que notre combinaison est plus performante que chaque technique utilisée indépendamment et, d'autre part, que la vérification devient plus rapide avec l'utilisation du slicing. De plus, la simplification du programme par le slicing rend les erreurs détectées et les alarmes restantes plus faciles à analyser.

De 2012 à 2015, j'ai participé à l'élaboration d'une méthode originale d'aide à la vérification déductive de programmes C annotés en ACSL, par génération de tests sur une version instrumentée du programme d'origine. Cette méthode est détaillée dans la thèse CEA de Guillaume Petiot [Pet15] (dirigée par J. Julliand) que j'ai co-encadrée (à 30 %) avec N. Kosmatov. J'ai participé à 4 des 6 communications sur ce sujet, présentées dans des conférences internationales [KPS13, PKGJ14b, PBJ+14, GGP15b, PKB+16] et nationales [PKGJ14a].

Enfin, à partir de 2010, j'ai mis mon expérience sur les langages de spécification au service de la communauté PHP, en participant à la conception d'un nouveau langage de spécification pour PHP, appelé Praspel (*PHP Realistic Annotation and SPEcification Language*) [EBODG10, EDGO11, EDGB12, EGB13], avec I. Enderlin, F. Dadeau, F. Bouquet et A. Ben Othman. J'ai introduit le concept de domaine réaliste, pour décrire le type précis des données traitées par des fonctions PHP. Leur spécification formelle est utilisée par des générateurs de données pour produire et exécuter automatiquement des tests unitaires pertinents. L'approche a été entièrement outillée. L'outil inclut un modèle objet du langage de spécification, un compilateur, une bibliothèque extensible de types de données complexes, et un générateur de test unitaire pour PHP.

2.5 Procédures de décision

De nombreux problèmes de vérification peuvent se réduire à un problème de Satisfaisabilité Modulo Théorie (SMT). Pour construire des procédures de décision de satisfaisabilité, Armando et al. ont proposé en 2001 une approche basée sur la réécriture [ARR01]. Cette approche utilise un calcul général pour le raisonnement équationnel appelé *paramodulation* ou *superposition* (PC, pour *paramodulation calculus* en anglais). En général, une application équitable et exhaustive des règles du calcul de superposition PC conduit à une procédure de semi-décision qui termine sur les entrées insatisfaisables (la clause vide est alors engendrée), mais qui peut diverger sur les entrées satisfaisables. Mais ce calcul peut aussi terminer pour des théories intéressantes en vérification, et devient ainsi une procédure de décision.

Mes recherches sur les systèmes distribués uniformes, détaillées dans la partie 2.3, ont été freinées par les limitations de portée des procédures de décision connues fondées sur le calcul de superposition. Cet obstable m'a conduit à proposer et obtenir en 2009 une bourse CORDI de l'Inria sur ce sujet. Avec C. Ringeissen, chercheur Inria Nancy, j'ai co-encadré (à 40%) la thèse d'E. Tushkanova (dirigée par O. Kouchnarenko) sur la spécification et la certification formelle de (combinaisons de) procédures de décision.

Pour découvrir de nouvelles procédures de décision, un *calcul de superposition schématique* (SPC, pour *schematic paramodulation calculus* en anglais) a été conçu [LM02]. Il permet en particulier de prouver automatiquement la décidabilité de théories particulières et de leurs combinaisons. L'avantage de ce calcul SPC est le suivant : s'il termine sur une seule entrée abstraite, alors PC termine pour toutes les entrées concrètes correspondantes. Jusqu'à notre contribution, ce calcul SPC n'était appliqué que sur papier. Avec E. Tushkanova, C. Ringeissen et O. Kouchnarenko, j'ai conçu et développé une implémentation du calcul SPC, qui automatise la découverte de nouvelles procédures de décision fondées sur le calcul de superposition [TGRK12, TRGK13, TGRK15].

2.6 Stratégies de réécriture pour les méthodes multi-échelles

En 2009, j'ai proposé à Michel Lenczner, professeur au département Temps-Fréquence de FEMTO-ST (Franche-Comté Electronique Mécanique Thermique et Optique - Sciences et Technologies, UMR 6174 associée au CNRS), d'appliquer mon expérience des méthodes formelles à sa problématique de dérivation outillée de modèles de matrices de microsystèmes (MEMS) par des méthodes multi-échelles. J'ai ainsi initié une recherche pluri-disciplinaire entre le LIFC et l'institut FEMTO-ST, sur un axe prioritaire du pôle de compétitivité de la Région de Franche-Comté. En janvier 2011, j'ai demandé et obtenu un Bonus Qualité Recherche (BQR) de l'Université de Franche-Comté (5000 euros) pour soutenir ce travail de conception et l'implémentation de procédures d'approximation multi-échelles pour les matrices de micro-systèmes.

J'ai co-encadré à 50% la seconde année de thèse des doctorants Bin Yang et Raj Narayan Dhara du département Temps-Fréquence de FEMTO-ST, dirigés par M. Lenczner. Le quatrième membre de ce groupe de travail sur la formalisation des méthodes asymptotiques multi-échelles est Walid Belkhir, recruté par mes soins en septembre 2010, sur un contrat post-doctoral de 6 mois, financé par le programme pluri-formations (PPF) MIDi (Microsystèmes Intelligents Distribués), qui soutient les projets transversaux entre le laboratoire LIFC et l'institut FEMTO-ST. Nous avons écrit et transféré un package de stratégies de réécriture pour Maple. Ce travail a donné lieu à une communication dans un workshop [BG11] et à un article de revue [BGL14]. Les résultats ont également été présentés dans des conférences internationales du domaine [YBD+11, YDB+11, BGL+12].

2.7 Combinatoire énumérative formelle

En 2009, j'ai été sollicité par Timothy R. S. Walsh, professeur à l'UQAM (Université du Québec à Montréal) et par Alexander Mednykh, professeur au Sobolev Institute of Mathematics (Université d'état de Novosibirsk, Russie), pour mon expertise de thèse [Gio98b] en énumération des cartes pointées. J'ai appliqué des méthodes de test pour valider une implémentation optimisée, développée par T. Walsh, d'un algorithme que j'avais conçu et implémenté en Maple durant ma thèse de doctorat. Cette optimisation et sa validation ont été publiées dans une revue internationale [WG14]. En 2010, j'ai démontré formellement [Gio10] une conjecture nouvelle proposée quelques mois plus tôt par T. Walsh. Puis j'ai collaboré avec A. Mednykh au dénombrement des cartes de genre 4 [MG11]. Une synthèse de ces collaborations a été publiée dans la revue *Discrete Mathematics* en 2012 [WGM12]. Enfin, j'ai élaboré et validé avec V. Senni des algorithmes efficaces de génération exhaustive de mots codant les cartes planaires pointées [GS12].

Depuis 2012, je travaille au rapprochement de mes thématiques de vérification formelle et de combinatoire, pour contribuer au domaine de la *combinatoire formelle*. Je propose d'adapter des méthodes de vérification formelle à des problèmes et algorithmes de combinatoire énumérative ou bijective [BGGP16, GG16, DGG16]. De 2012 à 2016, j'ai co-encadré très fortement (à 90 %) la thèse de Richard Genestier, dirigée par Olga Kouchnarenko, sur la vérification formelle de programmes de génération de données structurées. Ce travail a donné lieu à une publication en revue internationale [BGGP16], deux communications en conférence internationale [GGP15b, DGG16], deux communications en conférence nationale [GGP15a, GG16] et à de nombreux exposés. Nous détaillons ces travaux dans la partie 3.3.1. L'article [GGP15b] montre l'intérêt de la spécification formelle, de la validation par test et de la vérification déductive, dans l'implémentation d'algorithmes connus et la conception de nouveaux algorithmes dans le domaine de la combinatoire énumérative.

2.8 Autres résultats

Cette partie présente mes autres résultats de recherche, qui n'entrent pas exactement dans la classification thématique précédente.

De 1999 à 2001, je me suis familiarisé avec les thématiques de recherches du LIFC. En collaboration avec M. Tréhel, de l'équipe SDR (Systèmes Distribués et Réseaux) et P. Gradit, alors doctorant au LAAS (Laboratoire d'Analyse et d'Architecture des Systèmes) à Toulouse, j'ai contribué à résoudre une conjecture sur l'efficacité d'un algorithme distribué d'exclusion mutuelle [TGG01, Gio03].

Lors d'une collaboration avec une équipe du département d'automatique de FEMTO-ST, j'ai co-encadré avec C. Lang une thèse sur la modélisation par systèmes multi-agents, dirigée par F. Bouquet, dont l'une des études de cas était une micro-usine composée de cellules de production de micro-composants. Cette thèse n'a pas été soutenue mais a donné lieu à une communication [BBGL08].

Dans le projet ANR "Smart surface" de l'ACI Systèmes Interactifs et Robotique, j'ai participé à la modélisation générique de l'interaction entre un micro-objet et une surface de convoyage constituée d'une matrice de MEMS [GHT10], ainsi qu'à l'étude des langages d'images pour abstraire ces modèles et les vérifier. J'ai ainsi pu contribuer à la problématique du model-checking régulier pour les langages d'images, en étudiant la simulation des 2OTA (*Two-dimensional On-line Tessellation Automata*), avec G. Cécé [CG11].

2.9 Liste de publications et communications

Cette partie énumère toutes mes publications et communications scientifiques avec comité de lecture. Elles sont classées par nature, puis dans chaque rubrique dans l'ordre alphabétique des auteurs.

2.9.1 Publications internationales

[AG99] D. Arquès and A. Giorgetti. Énumération des cartes pointées de genre quelconque en fonction des nombres de sommets et de faces. *Journal of Combinatorial Theory Series B*, 77(1) :1–24, 1999.

[AG00a] D. Arquès and A. Giorgetti. Counting rooted maps on a surface. *Theoretical Computer Science*, 234 :255–272, 2000.

[AG00b] D. Arquès and A. Giorgetti. Une bijection géométrique entre une famille d'hypercartes et une famille de polygones énumérées par la série de Schröder. *Discrete Mathematics*, 217 :17–32, 2000.

[BGGP16] J.-L. Baril, R. Genestier, A. Giorgetti, and Armen Petrossian. Rooted planar maps modulo some patterns. *Discrete Mathematics*, 339 :1199–1205, 2016.

[BGL14] W. Belkhir, A. Giorgetti, and M. Lenczner. A symbolic transformation language and its application to a multiscale method. *Journal of Symbolic Computation*, 65 :49–78, 2014.

[CDGR03] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society*, 9(2) :17–36, 2003.

[GGJK08] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java modeling language. *IET Software*, 2(6) :500–514, 2008.

[Gio03] A. Giorgetti. An asymptotic study for path reversal. *Theoretical Computer Science*, 299(1-3) :585–602, 2003.

[MG11] A. Mednykh and A. Giorgetti. Enumeration of genus four maps by number of edges. *Ars Mathematica Contemporanea*, 4 :351–361, 2011.

[PGHS15] M. Planat, A. Giorgetti, F. Holweck, and M. Saniga. Quantum contextual finite geometries from dessins d'enfants. *Int. J. of Geometric Methods in Modern Physics*, 2015.

[TGRK15] E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based system for automatic decidability and combinability. *Science of Computer Programming*, 99 :3–23, 2015.

[WG14] T. R. S. Walsh and A. Giorgetti. Efficient enumeration of rooted maps of a given orientable genus by number of faces and vertices. *Ars Mathematica Contemporanea*, 7 :263–280, 2014.

[WGM12] T. R. S. Walsh, A. Giorgetti, and A. Mednykh. Enumeration of unrooted orientable maps of arbitrary genus by number of edges and vertices. *Discrete Mathematics*, 312(17) :2660 – 2671, 2012.

[ZG15] N. Zeilberger and A. Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Logical Methods in Computer Science*, volume 11, issue 3, 2015.

2.9.2 Publications nationales

[TGG01] M. Tréhel, P. Gradit, and A. Giorgetti. Performances d'un algorithme distribué d'exclusion mutuelle en cas de non-équiprobabilité des requêtes des processus. *RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles), Numéro spécial Evaluation quantitative des performances des réseaux et systèmes*, 13(6) :557–573, 2001.

2.9.3 Communications à des colloques internationaux

[AG97b] D. Arquès and A. Giorgetti. Une bijection géométrique entre une famille d'hypercartes et une famille de polygones énumérées par la série de Schröder. In *FPSAC'97 (Formal Power Series and Algebraic Combinatorics)*, pages 14–25, 1997.

[BCDG07] F. Bouquet, J.-F. Couchot, F. Dadeau, and A. Giorgetti. Instantiation of parameterized data structures for model-based testing. In *B'07 (B Conference)*, volume 4355 of *LNCS*, pages 96–110. Springer, 2007.

[BG11] W. Belkhir and A. Giorgetti. Lazy rewriting modulo associativity and commutativity. In *WRS'11 (Workshop on Reduction Strategies in Rewriting and Programming)*, pages 17–21, 2011.

[BGL+12] W. Belkhir, A. Giorgetti, M. Lenzner, R. N. Dhara, and B. Yang. Rewriting strategies for a two-scale method : Application to combined thin and periodic structures. In *dMEMS'12 (Workshop on Design, Control and Software Implementation for Distributed MEMS)*, pages 82–89. IEEE Computer Society, 2012.

[CDD+04] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and debugging set-based specifications. In *WMF'03 (Brazilian Workshop on Formal Methods)*, volume 95 of *Electronic Notes in Theoretical Computer Science*, pages 189–208, 2004.

[CG11] G. Cécé and A. Giorgetti. Simulations over two-dimensional on-line tessellation automata. In *DLT'11 (Developments in Language Theory)*, volume 6795 of *LNCS*, pages 141–152. Springer, 2011.

[CGK05] J.-F. Couchot, A. Giorgetti, and N. Kosmatov. A uniform deductive approach for parameterized protocol safety. In *ASE'05 (Automated Software Engineering)*, pages 364–367. IEEE Computer Society, 2005.

[CGS09] J.-F. Couchot, A. Giorgetti, and N. Stouls. Graph-based reduction of program verification conditions. In *AFM'09 (Automated Formal Methods (colocated with CAV'09))*, pages 40–47. ACM Press, 2009. http://hal.inria.fr/inria-00402204.

[CKGJ10b] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Combining static analysis and test generation for C program debugging. In *TAP'10 (Tests and Proofs)*, volume 6143 of *LNCS*, pages 94–100. Springer, 2010.

[CKGJ11] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool : Value analysis, program slicing and test generation for C program debugging. In *TAP'11 (Tests and Proofs)*, volume 6706 of *LNCS*, pages 78–83. Springer, 2011.

[CKGJ12] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC'12 (Symposium On Applied Computing)*, pages 1284–1291. ACM, 2012.

[DGG16] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In *TAP'16 (Tests and Proofs)*, volume 6792 of *LNCS*, pages 57–75. Springer, 2016.

[EDGB12] I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet. Grammar-based testing using realistic domains in PHP. In *A-MOST'12 (Advances in Model-Based Testing), workshop ICST*, pages 509–518. IEEE Computer Society, 2012.

[EDGO11] I. Enderlin, F. Dadeau, A. Giorgetti, and A. Ben Othman. Praspel : A specification language for contract-based testing in PHP. In *ICTSS'11 (Int. Conf. on Testing Software and Systems)*, volume 7019 of *LNCS*, pages 64–79. Springer, 2011.

[EGB13] I. Enderlin, A. Giorgetti, and F. Bouquet. A constraint solver for PHP arrays. In *CSTVA'13 (Constraints in Software Testing, Verification and Analysis, workshop ICST)*, pages 218–223. IEEE, 2013.

[GG06c] A. Giorgetti and J. Groslambert. JAG : JML Annotation Generation for verifying temporal properties. In *FASE'06 (Fundamental Approaches to Software Engineering)*, volume 3922 of *LNCS*, pages 373–376. Springer, 2006.

[GGP15b] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP'15 (Tests and Proofs)*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[GHT10] A. Giorgetti, A. Hammad, and B. Tatibouët. Using SysML for smart surface modeling. In *dMEMS'10 (workshop on design, control and software implementation for distributed MEMS)*, pages 100–107. IEEE, 2010.

[Gio10] A. Giorgetti. Guessing a conjecture in enumerative combinatorics and proving it with a computer algebra system. In *SCSS'10 (Symposium on Symbolic Computation in Software Science)*, pages 5–18, 2010. https://hal.archives-ouvertes.fr/hal-00563330.

[GS12] A. Giorgetti and V. Senni. Specification and validation of algorithms generating planar Lehman words. In *GASCom'12 (random generation of combinatorial structures)*, 2012. https://hal.inria.fr/hal-00753008.

[PKB+16] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. Your proof fails ? testing helps to find the reason. In *TAP'16 (Tests and Proofs)*, volume 6792 of *LNCS*, pages 130–150. Springer, 2016.

[PKGJ14b] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand. How test generation helps software specification and deductive verification in Frama-C. In *TAP'14 (Tests and Proofs)*, volume 8570 of *LNCS*, pages 204–211. Springer, 2014.

[TGMK10] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Specifying generic Java programs : Two case studies. In *LDTA'10 (Language Descriptions, Tools and Applications)*, pages 92–106. ACM, 2010.

[TGRK12] E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In *RLA'12 (Rewriting Logic and its Applications)*, volume 7571 of *LNCS*, pages 221–239. Springer, 2012.

[TRGK13] E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic decidability for theories with counting operators. In *ADDCT'13 (Automated Deduction : Decidability, Complexity, Tractability)*, 2013.

[YBD+11] B. Yang, W. Belkhir, R. N. Dhara, M. Lenczner, and A. Giorgetti. Computer-aided multiscale model derivation for MEMS arrays. In *EuroSimE'11 (Int. Conf. on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems)*. IEEE Computer Society, 2011.

[YDB+11] B. Yang, R. N. Dhara, W. Belkhir, M. Lenczner, and A. Giorgetti. Formal methods for multiscale models derivation. In *CFM'11 (Congrès Français de Mécanique)*, 2011.

2.9.4 Communications à des colloques nationaux

[CDGR04] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Barvey : Vérification automatique de consistance de machines abstraites B. In *AFADL'04 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 369–372, 2004.

[CG04] J.-F. Couchot and A. Giorgetti. Analyse d'atteignabilité déductive. In *AFADL'04 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 269–283, 2004.

[GG06a] A. Giorgetti and J. Groslambert. JAG : Génération d'annotations JML pour vérifier des propriétés temporelles. In *AFADL'06 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, 2006. Session outils.

[GG07] A. Giorgetti and J. Groslambert. Un programme annoté en vaut deux. In *JFLA'07 (Journées francophones des langages applicatifs)*, pages 87–101. Inria, 2007.

[GG16] R. Genestier and A. Giorgetti. Spécification et vérification formelle d'opérations sur les permutations. In *AFADL'16 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 72–78. FEMTO-ST, 2016. http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf.

[GGP15a] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In *JFLA'15 (Journées Francophones des Langages Applicatifs)*. Inria, 2015. https://hal.inria.fr/hal-01099135.

[PKGJ14a] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand. Comment la génération de tests facilite la spécification et la vérification déductive des programmes dans Frama-C. In *AFADL'14 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 133–133, 2014. Résumé étendu de [PKGJ14b]. http://afadl2014.lacl.fr/actes_AFADL2014-HAL.pdf.

Chapitre 3

Trois approches de la vérification déductive

Sommaire

La vérification déductive de la correction d'un système est une méthode en deux étapes :

1. La première étape dérive du système une formule, appelée *condition de vérification*, dont la validité implique la correction du système :

   > "the proof methods [...] achieve [...] a reduction of claims about the program to claims about the domain of computation [...] [appelées "conditions de vérification"]. Indeed, this is the essence of program verification in general." [Fra92, page 29].

   Il est plus fréquent de parler de plusieurs "conditionS de vérification", au pluriel, mais par commodité nous réduisons ces conditions à leur conjonction, et nous parlons d'une unique formule, nommée "condition de vérification" au singulier. Cette formule est exprimée dans une certaine logique formelle.

2. La seconde étape de la vérification déductive tente de démontrer cette formule à l'aide d'un prouveur de théorèmes, qui applique les principes de raisonnement et de calcul de cette logique. Ces prouveurs de théorèmes peuvent être automatiques (ATP, pour *Automated Theorem Prover* en anglais) ou interactifs (voir partie 4.3.2). Actuellement, les prouveurs automatiques de théorèmes les plus répandus sont les solveurs SMT, présentés dans la partie 4.3.1.

Le principal obstacle à l'automatisation de la méthode réside dans la seconde étape. En effet, pour de nombreuses familles de systèmes intéressantes en pratique, on ne connait pas d'algorithme capable de déterminer la validité de la condition de vérification de tous les systèmes dans cette famille, autrement dit de procédure de décision de leur correction.

Dans ce chapitre, nous distinguons trois approches de la vérification déductive, selon le niveau de finesse de l'analyse menée pour comprendre et, si possible, surmonter ou contourner ce problème de non-automaticité de la preuve. Nous proposons de qualifier chaque approche par un niveau de gris, parmi "noir", "gris" et "blanc". Nous empruntons les expressions "boîte blanche", "boîte grise" et "boîte noire" au domaine du test, selon l'analogie suivante : En test, la boîte est le système sous

test. Ici, la boîte est le prouveur de théorèmes. Dans les deux cas, le niveau de gris qualifie le degré de connaissance du contenu de la boîte. L'analogie s'arrête là, nous ne parlons pas ici d'utilisation de tests pour aider la preuve.

En quelques mots, l'approche *boîte noire* de la vérification déductive, détaillée dans la partie 3.1, est l'utilisation d'un outil de preuve de programmes sans se préoccuper des limites théoriques et pratiques des prouveurs de théorèmes utilisés pour décharger la condition de vérification. En pratique cette approche à l'aveuglette aboutit rarement. Pour démontrer la correction des systèmes qui résistent à cette approche "magique", on peut suivre l'approche "boîte grise" ou "boîte blanche".

L'approche *boîte grise* consiste à n'envisager la preuve automatique que de systèmes aux caractéristiques adaptées aux capacités actuelles des prouveurs automatiques de théorèmes. Ceci suppose que l'utilisateur dispose d'une culture minimale sur les principes de démonstration automatique, les logiques décidables et le lien entre le système sous vérification et la condition de vérification produite par la première étape de la méthode déductive. Nous détaillons et illustrons cette approche dans la partie 3.3.

Poussée à l'extrême, l'approche boîte grise devient l'approche *boîte blanche*, qui explicite totalement la condition de vérification de chaque système, établit son appartenance à un fragment logique décidable, et identifie avec certitude un prouveur automatique qui implémente une procédure dé décision pour ce fragment. Cette approche est détaillée et discutée dans la partie 3.2.

À notre connaissance, cette classification est inédite. Elle nous permet de situer nos travaux de recherche dans ce domaine, et de justifier leur évolution dans le temps. Nous présentons l'approche extrême boîte blanche avant l'approche boîte grise pour faciliter la compréhension, mais aussi parce que nous l'avons suivie dans nos premiers travaux, avant de nous orienter vers l'approche boîte grise.

3.1 Boîte noire, ou approche générale

La première approche, qualifiée ici de *boîte noire*, ou d'*approche générale*, soumet des problèmes de vérification à un prouveur automatique de programmes sans s'enquérir des limitations des prouveurs automatiques de théorèmes qu'il utilise.

Les conditions de vérification sont généralement des formules de la logique du premier ordre. Cette logique étant indécidable, les prouveurs automatiques de théorèmes qui acceptent son langage en entrée sont par nature incomplets : Ils n'offrent aucune garantie de terminaison. De plus, ces prouveurs peuvent implémenter des procédures de décision de forte complexité algorithmique. Ainsi, la durée d'une démonstration peut être très élevée, sans fournir d'indication sur son progrès ou sa divergence.

Cependant, diverses recherches sur les (combinaisons de) procédures de décision ou sur la synthèse d'invariants de boucle, et l'intégration de leurs résultats dans les prouveurs de programmes, rendent progressivement cette approche de plus en plus effective. Pour comprendre ces recherches et y contribuer, il est nécessaire d'analyser les ressorts de la méthode déductive, c'est-à-dire de suivre les approches boîte blanche ou grise détaillées ci-dessous. C'est pourquoi aucune de mes activités de recherche ne s'inscrit dans l'approche boîte noire.

3.2 Boîte blanche, ou vérification analytique

Par définition, l'approche générale boîte noire accorde peu d'attention à la nature de la condition de vérification et à la manière dont elle est calculée. Elle se contente de produire cette condition de vérification dans le langage d'entrée d'un prouveur automatique. Il en résulte souvent la non-terminaison du prouveur, suspectée par une durée d'exécution excessive.

L'approche *boîte blanche* de la vérification déductive consiste à analyser finement la structure de la condition de vérification. Dans certains cas, l'analyse permet d'établir que la condition de vérification appartient à un fragment décidable d'une logique indécidable. Dans d'autres cas, l'analyse peut identifier un nouveau fragment sur lequel on peut entreprendre la recherche d'une

nouvelle procédure de décision. Nous analysons successivement ces deux cas dans les deux sous-parties suivantes. La partie 3.2.3 décrit ensuite l'objectif majeur de cette approche, et la partie 3.2.4 présente mes activités de recherche qui ont visé et atteint cet objectif.

3.2.1 Justification de décidabilité

L'identification qu'une condition de vérification est dans un fragment décidable est en elle-même un travail conséquent. En effet, le rapide développement des solveurs SMT a l'efficacité comme objectif principal, et comme second objectif une meilleure couverture de familles plus larges de problèmes, par implémentation de procédures plus spécialisées. Tout ceci s'est effectué au détriment de la transparence sur leurs fonctionnalités précises.

Depuis 2002, la communauté SMT s'est organisée autour de la "*Satisfiability Modulo Theories Library*" (SMT-LIB, http://smtlib.cs.uiowa.edu/), pour définir un format commun permettant de comparer les solveurs SMT entre eux. Après plusieurs années de compétition (*Satisfiability Modulo Theories Competition*, SMT-COMP) pour identifier les solveurs les plus performants, la communauté lance une évaluation en 2013, en particulier pour mieux identifier les capacités de chaque solveur :

> "in 2013 we are conducting an evaluation (SMT-EVAL) of the state of SMT solvers and benchmarks, rather than a head-to-head competition as at previous SMT-COMPs." [*SMT-EVAL : Call for comments, solvers, and benchmarks.* D. Cok, A. Stump, T. Weber, 12/02/2013, message sur la liste de diffusion smt-lib@cs.nyu.edu].

Il en résulte un rapport [CSW15] qui contient une synthèse des logiques et des théories supportées par chaque solveur SMT acceptant la version SMT-LIB v2 du format d'entrée. Cette information est essentielle pour l'approche boîte blanche, qui peut ensuite chercher à décrire des familles de systèmes dont la correction s'exprime dans ces logiques et théories. Cependant, au-delà de ces logiques et théories nommées et décrites dans la SMT-LIB, les solveurs SMT implémentent aussi des heuristiques (par exemple pour instancier des quantifications existentielles) difficiles à décrire et à comparer entre elles.

Il serait souhaitable que les solveurs SMT explicitent complètement leur portée exacte. Il ne suffit pas que la documentation du solveur décrive chaque fragment logique sur lequel sa terminaison résulte de l'existence d'une procédure de décision connue (publique), comme c'est actuellement le cas. En effet, grâce à ses heuristisques complémentaires, le solveur peut aussi terminer "par chance" pour une entrée en dehors de tous ces fragments. Il faut donc que le solveur indique, le cas échéant, dans quel fragment décidable connu se situe chaque problème qu'on lui soumet, et sinon, si possible, quelles heuristiques il a utilisées. Ainsi, l'utilisateur sait si le succès de sa vérification est fortuit (un petit miracle) ou a une explication rationnelle.

3.2.2 Conception de nouvelles procédures de décision

Un échec dans la vérification déductive automatique de la correction d'un système peut être dû soit à une incohérence entre le code et la spécification du système, soit à l'incomplétude des prouveurs de théorèmes sous-jacents. Pour certains systèmes, nous avons contribué à une méthode permettant souvent de discriminer ces deux cas, comme détaillé dans la partie 2.4. Lorsqu'une telle méthode établit (ou suggère fortement) que la cause de l'échec est l'incomplétude des prouveurs, l'approche boîte blanche continue en comparant la condition de vérification non prouvée avec le champ d'application de toutes les procédures de décision connues. Lorsque cette condition de vérification n'est dans aucun fragment décidable connu, l'approche boîte blanche envisage d'élaborer pour elle une nouvelle procédure de décision. C'est cependant une activité de recherche très pointue qui requiert beaucoup d'expérience, de temps et d'efforts, comme j'ai pu le constater lors de ma pratique de cette approche relatée dans la partie 3.2.4.

3.2.3 Systèmes dont la correction est décidable

Une justification de décidabilité au niveau des logiques et théories des solveurs SMT est précieuse, mais pas suffisante, pour deux raisons. La première raison est que l'utilisateur n'étudie pas un système unique, mais aussi ses variantes (parmi lesquelles peuvent se trouver, par exemple, des améliorations d'un système initialement prouvé incorrect), et plus généralement toute une famille de problèmes dans un même domaine. La correspondance avec les procédures de décision ne doit donc pas être établie pour un système, mais pour toute une famille de systèmes, ce qui soulève le problème des moyens de description de telles familles. La deuxième raison est que l'utilisateur ne doit pas avoir à maîtriser lui-même la traduction de la correction de son système en condition de vérification : C'est une sorte de compilation, et la plupart des développeurs ignorent les détails des mécanismes des compilateurs qu'ils utilisent, et leur langage cible.

Pour que l'effort d'utilisation de la vérification déductive devienne raisonnable, il faut "faire remonter" la question de décidabilité au niveau des langages de programmation et de spécification auxquels on l'applique. Autrement dit, il s'agit de décrire syntaxiquement des langages de systèmes spécifiés dont la correction est décidable, comme illustré par l'exemple de la partie 3.2.4. C'est un défi ambitieux, du fait d'une part de la multiplicité des langages, et d'autre part de la diversité de leurs sémantiques. Cette "remontée" exige, entre autres, une forte maîtrise de ces sémantiques.

Ainsi, un objectif majeur de la recherche en boîte blanche, ou *vérification analytique*, est de démontrer la décidabilité de la correction d'une famille infinie de systèmes dynamiques formels.

3.2.4 Exemple contributif

De 2003 à 2005, avec J-F. Couchot, F. Dadeau, D. Deharbe et S. Ranise, j'ai appliqué l'approche boîte blanche à des systèmes distribués, qualifiés d'"uniformes" car composés de n processus identiques exécutés en parallèle [CDGR03, CG04, CGK05]. Il s'agissait de démontrer des propriétés de sûreté comme l'exclusion mutuelle ou la cohérence de cache. Ces systèmes et leur propriété étaient respectivement décrits par une machine B [Abr96] et un invariant pour cette machine. La cible était un solveur SMT fondé sur la logique équationnelle, nommé haRVey, aujourd'hui remplacé par le solveur veriT [Ver17].

Nous avons remplacé l'analyse de sûreté par une synthèse d'invariant équivalente. Puis nous avons établi que les conditions de progression et de terminaison d'un algorithme de synthèse de ces invariants appartenaient à un fragment décidable, pour une famille de tels systèmes et de propriétés de sûreté caractérisés syntaxiquement [CGK05].

Nous avons d'abord présenté les règles formelles de génération de conditions de vérification, dans une logique ensembliste, puis dans une logique équationnelle [CDD+04]. Ce travail a été honoré du titre de meilleur article de la conférence. Il a été étendu par une démonstration informelle de la correction de la démarche de traduction en logique équationnelle [CDGR03].

Nous avons ensuite défini un langage de substitutions généralisées sur des tableaux pour décrire la famille des systèmes distribués uniformes se synchronisant par rendez-vous [CG04].

Nous avons étendu cette famille aux systèmes uniformes distribués se synchronisant par envoi multiple (*broadcast*). La décidabilité des conditions de vérification est établie pour une sous-famille de ces systèmes. Ceci a nécessité l'élaboration d'une nouvelle procédure de décision [CGK05]. Dans un cadre plus vaste, aucune procédure n'a pu être établie dans un délai raisonnable. Nous avons cependant observé la terminaison du solveur SMT sur de nombreux exemples. Ceci nous a orienté vers l'approche boîte grise décrite dans la partie suivante.

Avec cet exemple, nous avons relevé le défi de "faire remonter" la question de décidabilité au niveau des syntaxes des langages de programmation et de spécification. Ainsi, un concepteur de systèmes distribués uniformes dispose d'une garantie de preuve automatique de leur correction, simplement en respectant la syntaxe de leur langage de description.

Ces travaux ont été repris et approfondis par d'autres chercheurs, d'abord d'un point de vue logique [GR09, GR10], puis du point de vue du langage d'entrée, avec l'outil Cubicle [CGK+12, Meb14].

3.3 Boîte grise, ou vérification expérimentale

L'approche *boîte grise*, ou *vérification expérimentale*, est un compromis raisonnable entre l'approche boîte noire, trop aveugle, et l'approche boîte blanche, souvent trop ambitieuse.

C'est une approche empirique, pragmatique, en trois phases :

1. La première phase consiste à choisir une famille de systèmes dont la correction est susceptible d'être démontrée automatiquement par les prouveurs automatiques actuels.
2. La deuxième phase consiste à collecter de nombreux exemples de succès et d'échec de preuve automatique de systèmes appartenant à cette famille.
3. La troisième phase consiste à analyser ces résultats pour déterminer s'il est pertinent ou non d'appliquer une approche boîte blanche à cette famille de systèmes.

L'objectif de la première phase est de choisir des systèmes d'une complexité qui sollicite les limites des capacités actuelles des prouveurs automatiques. Ce choix est fondé sur une intuition résultant de notre expérience en preuve de programmes et de nos connaissances imparfaites des procédures de décision et des capacités des prouveurs, en perpétuelle progression. Par exemple, si un système est spécifié par une propriété inductive, il est peu raisonnable d'envisager la preuve automatique de sa correction avec des prouveurs de théorèmes qui n'automatisent pas le raisonnement par induction.

Les résultats de la deuxième phase peuvent être surprenants, car les prouveurs automatiques tels que les solveurs SMT évoluent rapidement et deviennent de plus en plus généraux et efficaces. Il s'avère qu'en pratique ces solveurs résolvent davantage de problèmes que les procédures de décision qu'ils implémentent ne le laissent supposer. Ceci justifie cette approche empirique, qui soumet à ces prouveurs automatiques des problèmes de plus en plus complexes, afin d'évaluer leurs capacités réelles.

Les objectifs de la troisième phase sont d'une part d'expliquer rationnellement les succès de preuve observés, et d'autre part d'exprimer clairement un besoin de conception d'une nouvelle procédure de décision pour les cas d'échec de preuve observés. Selon la proportion d'exemples de succès et d'échec, on pourra choisir de poursuivre un seul de ces deux objectifs.

Il est important de rappeler ici qu'on ne veut pas savoir vérifier un système particulier, mais tous les systèmes d'une famille infinie, souvent décrite par des paramètres. Par conséquent, il n'est pas suffisant d'apporter une réponse *ad hoc* pour démontrer tel ou tel système de cette famille. Il faut que cette "astuce" devienne une recette applicable à tous les systèmes de la famille. Autrement dit, elle doit être aussi générale/paramétrée que la famille elle-même. Dans l'exemple ci-dessus d'une propriété inductive et de prouveurs sans induction automatique, une astuce serait, si possible, d'étendre la spécification avec un lemme, admis ou démontré séparément, exprimant le principe d'induction requis. Ce n'est qu'une astuce si des systèmes différents requièrent des lemmes trop différents pour être unifiés en un seul, qui serait la clé d'une recette applicable à tous les systèmes de la famille. Une autre recette serait de choisir un prouveur adapté. Ceci peut nécessiter une extension de l'outil de preuve ou une (interface de) traduction des exemples dans un autre langage.

3.3.1 Exemple contributif

J'ai pratiqué l'approche boîte grise à partir de 2012, d'abord avec R. Genestier [Gen16a, Gen16b], puis aussi avec G. Petiot [Pet15, GGP15a, GGP15b] et C. Dubois [DGG16], selon les trois phases détaillées ci-dessous.

Phase 1 : Domaine d'application

Dans la première phase, j'ai choisi d'étudier des familles d'algorithmes de génération et de transformation de tableaux d'entiers de longueur [1] quelconque, d'une part parce qu'ils me sem-

1. La *longueur* (*length* en anglais) d'un tableau est son nombre d'éléments, tandis que sa taille (*size* en anglais) est l'espace occupé par le tableau, qui dépend de la longueur du tableau et de l'espace occupé par chacun de ses éléments.

blaient satisfaire le critère de criticité par rapport aux capacités des prouveurs automatiques, et d'autre part parce qu'ils ont des applications en test aléatoire ou exhaustif borné [DGG16]. Chaque algorithme est paramétré par la longueur non bornée des tableaux qu'il génère ou transforme.

Il s'agissait plus précisément d'algorithmes d'énumération combinatoire, qui construisent séquentiellement tous les tableaux structurés de même longueur, selon un ordre total prédéfini. Nous disons qu'un tableau est *structuré* lorsque son contenu satisfait certaines contraintes paramétrées par la longueur du tableau, comme être trié ou ne contenir que des éléments deux à deux distincts. Un tel algorithme est nommé *générateur séquentiel* lorsqu'il est composé de deux fonctions : la première fonction construit la plus petite structure de longueur donnée selon l'ordre prédéfini, et la seconde fonction modifie une structure quelconque pour construire la structure suivante (de même longueur) selon cet ordre.

Nous avons considéré trois propriétés comportementales pour ces fonctions de génération. La propriété de *correction* affirme que les deux fonctions génèrent des tableaux satisfaisant leur contrainte structurelle. La propriété de *progression* affirme que la deuxième fonction génère un tableau supérieur à son tableau d'entrée selon l'ordre prédéfini. Cette propriété implique la terminaison des appels répétés à la seconde fonction. La propriété d'*exhaustivité* affirme que le générateur n'omet aucune solution.

Phase 2 : Collecte d'exemples

Nous avons rapidement constaté que la plupart des conditions de vérification de ces systèmes étaient des formules du premier ordre avec égalité, tableaux et arithmétique linéaire sur des entiers. Cette logique est indécidable [BMS06], mais certains fragments décidables sont connus. Notre intuition était que nos problèmes dépassaient le cadre de ces fragments et pouvaient motiver la recherche de nouvelles procédures de décision. Nous avons donc entamé la deuxième phase, de collecte d'exemples de succès et d'échec de preuve automatique de ces systèmes.

Nous avons choisi de programmer ces algorithmes en C, de les spécifier avec ACSL et de les vérifier avec le greffon WP de Frama-C (ce langage de spécification et cet outil sont présentés dans le chapitre 4). Nous avons conçu et développé une bibliothèque *open source* de générateurs séquentiels de tableaux structurés formellement spécifiés et vérifiés [GGP15a, GGP15b]. Les fonctions C de génération de tableaux sont annotées avec des invariants et variants de boucle, afin que leurs propriétés soient automatiquement prouvées.

Au total, nous disposons actuellement d'une quarantaine d'exemples de preuve de tels programmes, de complexité croissante. Leurs propriétés de correction et de progression sont prouvées automatiquement, mais pas leur propriété d'exhaustivité.

Phase 3 : Sélection d'objectifs de recherche

La troisième phase de l'approche boîte grise consiste à sélectionner les objectifs de recherche les plus prometteurs, parmi les suivants :

1. Expliquer le succès des preuves des propriétés de correction et de progression.
2. Comprendre les causes de l'échec des preuves d'exhaustivité.
3. Caractériser (syntaxiquement ou par d'autres moyens) une large classe de programmes annotés partageant les mêmes résultats de preuve automatique.

Je n'ai pas suffisamment avancé dans cette troisième phase pour pouvoir la détailler ici. Pour l'objectif 2 j'envisage de recourir à la preuve interactive.

Avec C. Dubois et R. Genestier, j'ai d'ores et déjà appliqué avec succès la preuve interactive en Coq, pour démontrer la correction d'opérations de construction de cartes combinatoires [DGG16]. Les cartes sont des objets combinatoires complexes. Nous proposons un codage de ces cartes dans un tableau structuré, soumis à une contrainte de transitivité. Nous avons démontré formellement que les opérations de construction préservent cette contrainte. Nous présentons dans le même article des outils pour tester la validité de conjectures Coq avant de tenter de les démontrer. Ceci contribue à l'objectif 2, en permettant d'écarter l'hypothèse d'une erreur de spécification, au profit de celle d'incomplétude des prouveurs automatiques.

3.4 Synthèse

Nous souhaitons automatiser la vérification déductive de familles de systèmes paramétrés, lorsque la décidabilité de leur correction est inconnue. Notre expérience antérieure nous suggère la stratégie suivante : Suivre d'abord l'approche expérimentale (boîte grise), qui évalue empiriquement la probabilité de décidabilité. Si cette probabilité est assez forte et la famille de systèmes est d'un intérêt suffisant, suivre ensuite l'approche analytique (boîte blanche) d'explication de cette décidabilité.

Dans cette approche analytique, nous devons disposer d'une description rigoureuse et précise de la traduction d'un système spécifié en sa condition de vérification. La suite de ce mémoire est consacrée à ce sujet.

Chapitre 4

Prérequis

Sommaire

Dans le chapitre 3, nous avons qualifié d'approche "boîte blanche" de la vérification déductive la démarche de recherche qui vise à identifier des familles de systèmes paramétrés dont la correction est décidable, en établissant la décidabilité de leurs conditions de vérification, puis en transférant ce résultat sur une description formelle du langage de ces systèmes.

Le chapitre 6 contient une contribution à cette approche "boîte blanche", lorsque ces systèmes sont des programmes impératifs annotés par des spécifications fonctionnelles. Cette contribution est une description formelle précise des conditions de vérification de la correction partielle des programmes d'un langage impératif Turing-complet, annotés par des assertions et des invariants de boucle. Pour justifier la correction de cette description et élargir le sujet, nous l'inscrivons dans le cadre plus large des sémantiques des programmes annotés. Nous définissons formellement ces sémantiques et nous démontrons formellement leurs corrections et complétudes relatives, avec un assistant de preuve, dans le chapitre 6. Cette partie technique du mémoire commence avec le chapitre 5, qui décrit et complète l'existant en sémantique formelle, pour le même langage impératif, mais en ignorant les annotations de spécification.

Le présent chapitre sert d'introduction à cette partie technique du mémoire, en fournissant tous ses prérequis, selon le plan suivant. Les parties 4.1 et 4.2 présentent brièvement les langages de spécification fonctionnelles, puis, parmi eux, les langages de contrats, qui intègrent la spécification dans les programmes, sous forme d'annotations formelles. La partie 4.3 présente quelques outils de vérification déductive des programmes écrits dans ces langages de contrats. La partie 4.4 présente la syntaxe des annotations les plus usuelles dans le langage de spécification ACSL, sur un exemple de programme annoté. La partie 4.5 définit les notions de correction totale et partielle. La suite du mémoire ne porte que sur la correction partielle, reliée à la sémantique des programmes dans la partie 4.6. Les parties 4.7 et 4.8 fournissent les ingrédients de base d'une approche formelle de la sémantique, avec un assistant de preuve. La partie 4.9 résume ce chapitre.

4.1 Langages de spécification fonctionnelle

Il existe de nombreux langages de spécification formelle. Il serait trop long et peu contributif de les présenter ici. Le lecteur intéressé est invité à consulter la synthèse collective intitulée "*Logics of Specification Languages*" [BH08]. L'extrait suivant donne un aperçu de l'étendue du sujet :

> "By a specification language we understand a formal system of syntax, semantics and proof rules. The syntax and semantics define a language ; the proof rules define a proof system. Specifications are expressions in the language, and reasoning over properties of these specifications is done within the proof system. This book presents comprehensive studies on nine specification languages and their logics of reasoning. The editors and authors are authorities on these specification languages and their application. Dedicated chapters address : the use of ASM (Abstract State Machines) in the classroom ; the Event-B modelling method ; a methodological guide to CafeOBJ logic ; CASL, the Common Algebraic Specification Language ; the Duration Calculus ; the logic of the RAISE specification language (RSL) ; the specification language TLA+ ; the typed logic of partial functions and the Vienna Development Method (VDM) ; and Z logic and its applications." [BH08]

Les méthodes de vérification de programmes sont plus simples à mettre en œuvre lorsque le langage de spécification est proche de celui des conditions du langage de programmation. Par exemple, Francez suggère que ces conditions soient des combinaisons booléennes des prédicats atomiques de la logique de spécification :

> "There is [...] an implicit assumption connecting [the programming language and the specification language], in that tests occuring in programs are quantifier-free Boolean combinations of the primitive predicates." [Fra92, page 16]

Cette remarque de Francez suppose que le langage de programmation est conçu **après** le langage de spécification. La tendance actuelle est plutôt d'ajouter un langage de spécification à un langage de programmation **pré-existant**, de sorte qu'il faut inverser la suggestion, en conseillant que les formules de spécification étendent les conditions booléennes du langage de programmation. Cette extension inclut souvent les quantifications universelle et existentielle de la logique du premier ordre.

D'un point de vue pratique, l'idée a surgi très tôt d'intégrer les spécifications dans le code source des programmes spécifiés. Dans son mémoire de thèse, Julien Groslambert attribue cette idée à Owicki et Gries :

> "Un point de vue pour spécifier et vérifier du code consiste à travailler directement au niveau du code source en insérant des annotations logiques (invariants, pré et post-conditions). L'origine des annotations de code revient à Owicki et Gries [OG76], qui proposent d'insérer des assertions à l'intérieur même des programmes afin de rendre la preuve de programme, basée sur la logique de Hoare, applicable en pratique." [Gro07].

Cette approche d'annotation de programmes a été implémentée dès 1978 dans des langages comme Euclid [LGH+78] et Eiffel [MNM87]. Elle a été reprise dans des travaux plus récents, notamment par le langage LARCH [GH93], un langage d'annotation pour C, et dans le concept de *conception par contrat* (*Design by Contract*, DbC, en anglais) défini par B. Meyer dans le langage Eiffel [Mey97]. Un autre exemple récent de cette approche de conception par contrat est le langage JML (*Java Modeling Language*) [LBR06], initié par Gary T. Leavens et son équipe à l'Université de l'Iowa. JML permet d'intégrer au code Java des spécifications formelles (précondition, postcondition, invariant, etc).

Nous adoptons dorénavant cette idée d'intégrer la spécification dans le langage de programmation, pour son intérêt simplificateur. En effet, selon ce point de vue, il ne s'agit plus d'établir une relation de cohérence entre **deux ingrédients**, un programme et une spécification extérieure, mais d'établir la correction d'un unique ingrédient, appelé *programme annoté*, écrit dans un **langage unique** incluant des annotations formelles de spécification.

4.2 Langages de contrats

Les contrats sont apparus dans le langage Eiffel [MNM87], basé sur le paradigme de la programmation orienté objet. La *programmation par contrat* est un paradigme de programmation dans lequel le comportement d'un programme est décrit par des *clauses* formelles (parfois appelées *contraintes formelles*), afin de réduire le nombre d'erreurs dans les programmes [LG86]. Le contrat précise ce qui doit être vrai à un moment donné de l'exécution d'un programme.

On peut distinguer trois sortes de clauses :

— Les *préconditions* spécifient l'état du système avant son exécution.
— Les *postconditions* spécifient l'état du système après son exécution.
— Les *invariants* spécifient la préservation d'une partie de l'état du système durant une partie de son exécution.

Les langages décrivant une spécification formelle du comportement des programmes à travers des contrats sont appelés des *langages de contrats*.

Il existe de nombreux langages de contrats. La plupart sont basés sur la logique du premier ordre. Certains sont dédiés à un langage de programmation, comme par exemple Spec# [BFL+11] pour C#, ACSL [BCF+13] et VCC [DMS+09] pour C, JML [ABB+14] pour Java, ou Praspel [EDGO11] pour PHP. D'autres ne sont pas liés à un langage de programmation pré-existant, mais sont intégrés dans un langage de programmation conçu pour la preuve, comme par exemple WhyML pour Why3 [BFMP11, FP13] ou Dafny [Lei10].

J'ai participé à la création de Praspel, à partir de 2010, avec Ivan Enderlin, Frédéric Dadeau, Fabrice Bouquet et Abdallah Ben Othman [EBODG10, EDGO11, EDGB12, EGB13].

4.3 Outils de vérification déductive

Il existe de nombreux logiciels d'aide à la vérification déductive. Nous présentons les prouveurs automatiques dans la partie 4.3.1. La partie 4.3.2 décrit le principe des prouveurs interactifs. L'assistant de preuve Coq, utilisé dans la partie 4.8 et les chapitres 5 et 6, est détaillé dans la partie 4.3.3. La partie 4.3.4 décrit des plate-formes d'analyse de programmes.

4.3.1 Solveurs SMT

Les prouveurs automatiques les plus utilisés actuellement pour décharger les conditions de vérification sont des *solveurs SMT* (SMT est l'acronyme de "*Satisfiability Modulo Theories*"). Ces outils implémentent des procédures de décision de satisfaisabilité, pour des fragments de la logique du premier ordre, dans des théories sur l'arithmétique, les tableaux, les types de données algébriques, etc. En preuve de programmes, on les applique à la négation des conditions de vérification dont on veut démontrer la validité. Ils sont entièrement automatiques, et une durée de calcul par défaut est allouée pour chaque formule logique traitée. Ils valident (ou *déchargent*) les conditions de vérification, ou échouent, soit parce qu'ils ne disposent pas d'assez de temps de calcul, soit parce que les procédures de décision qu'ils implémentent ne sont pas adaptées pour les décharger. De nombreux solveurs SMT existent, comme par exemple Alt-Ergo [LO13], CVC3 [BT07], CVC4 [BT12], veriT [Ver17], Yices [Dut14] ou Z3 [DMB08].

Ces outils peuvent être utilisés de façon autonome, ou bien au sein de plate-formes d'analyse de programmes.

4.3.2 Prouveurs interactifs

D'autres systèmes, appelés *prouveurs interactifs* ou *assistants de preuve* sont fondées sur une logique non décidable. L'action de l'utilisateur est alors nécessaire pour mener certaines preuves. Des exemples de tels systèmes sont Coq [BC04], PVS [ORR+96], ACL2 [KM04] et Isabelle [CMc16].

Une amélioration notoire dans ce domaine est l'introduction de prouveurs automatiques dans les assistants de preuve. Par exemple, Isabelle/HOL avec son outil Sledgehammer [MP08], et HOL4

qui intègre des solveurs SMT [Web11], épargnent à l'utilisateur de traiter certaines obligations de preuve.

Dans le cadre de cette étude, nous utilisons l'assistant de preuve Coq, présenté dans la partie 4.3.3.

4.3.3 Coq

Coq [BC04] est un assistant de preuve développé par l'équipe PI.R2 d'Inria. Coq est fondé sur le calcul des constructions inductives (CoC, pour *Calculus of Constructions*), une théorie des types d'ordre supérieur. Son langage de spécification, Gallina, est une forme de lambda-calcul typé. Coq peut être vu aussi bien comme un langage de programmation que comme un système de preuves mathématiques. Il permet de synthétiser des programmes certifiés à partir de preuves constructives de leurs spécifications.

Notre utilisation de l'assistant de preuve Coq commence dans la partie 4.8 et se poursuit dans les chapitres 5 et 6.

4.3.4 Plate-formes d'analyse de programmes

Certaines plate-formes d'analyse de programmes automatisent la preuve de programmes via l'implémentation d'un générateur de conditions de vérification de programmes spécifiés. On trouve parmi elles KeY [BHS07], Why3 [BFM+13] et Frama-C [KKP+12].

Les conditions de vérification générées par le système KeY sont traitées par un solveur interne auquel on peut adjoindre certains solveurs SMT. Sa particularité réside dans le fait qu'en cas d'échec de la preuve automatique, il est possible d'ajouter manuellement des règles de déduction, nommées "taclets", et de relancer la preuve automatique ensuite.

L'outil Why3 [BFMP11, FP13] offre une large palette de possibilités, en ce sens que les conditions de vérification qu'il génère respectent le format d'entrée SMT-LIB de très nombreux solveurs SMT, mais aussi des assistants de preuve Coq, PVS et Isabelle/HOL. Son langage de programmation et de spécification, WhyML, est un dialecte ML.

La plateforme d'analyse de programmes C Frama-C [KKP+12], développée par le CEA LIST et Inria Saclay, utilise le langage de spécification ACSL (*ANSI C Specification Language*) [BCF+13]. ACSL est un langage dédié à l'analyse statique de programmes C. Il permet de spécifier formellement des contrats qui doivent être vérifiés par le programme. Un exemple est détaillé dans la partie 4.4. Le greffon WP [BBCD15] implémente un générateur de conditions de vérification pour Why3, ce qui donne accès à de nombreux prouveurs SMT et à des assistants de preuve.

Dans le cadre de leur utilisation avec Frama-C, les solveurs déchargent une condition de vérification générée en renvoyant le résultat "Valid" (assorti d'une durée de preuve), ou échouent en précisant qu'ils ne disposent pas d'assez de temps de calcul (et renvoient le résultat "Timeout") ou que les heuristiques qu'ils incorporent et les procédures de décision qu'ils implémentent ne leur permettent pas de prouver le but dans le temps imparti (et renvoient le résultat "Unknown"). La durée de calcul par défaut allouée à chaque condition de vérification est de 10 secondes. Il est possible d'augmenter cette durée dans le cas où certaines conditions de vérification complexes nécessitent davantage de temps pour être déchargées.

4.4 Exemple d'annotations ACSL

Cette partie présente les principales annotations d'un langage de contrats, sur l'exemple du langage de spécification ACSL pour le langage C.

Le listing 4.1 présente une fonction C qui stocke dans le tableau `b[0..nb-1]` les éléments du tableau `a[0..na-1]`, supposé trié dans l'ordre croissant, en ignorant les doublons (valeurs identiques). La fonction est paramétrée par la taille `na` quelconque (non nulle) du tableau `a`. Le code C est spécifié formellement par des annotations ACSL.

```
1  /*@ predicate sorted(int *a, int n) =
2       ∀ ℤ i, j; 0 ⩽ i < j < n ⇒ a[i] ⩽ a[j];
3      predicate strict(int *b, int n) =
4       ∀ ℤ i; 1 ⩽ i < n ⇒ b[i-1] < b[i];
5      predicate includes(int *a, int na, int *b, int nb) =
6       ∀ ℤ i; 0 ⩽ i < nb ⇒ (∃ ℤ j; 0 ⩽ j < na ∧ b[i] == a[j]); */
7
8  /*@ requires r1: 0 < na ⩽ nb;
9      requires r2: \valid(a+(0..na-1)) ∧ \valid(b+(0..nb-1));
10     requires r3: \separated(a+(0..na-1),b+(0..nb-1));
11     requires r4: sorted(a,na);
12     ensures e1: 1 ⩽ \result ⩽ nb;
13     ensures e2: strict(b,\result);
14     ensures e3: includes(a,na,b,\result);
15     assigns b[0..nb-1]; */
16 int remove_duplicates(int a[], int na, int b[], int nb) {
17  int m, k;
18  b[0] = a[0];
19  m = 1;
20  /*@ loop invariant i1: 1 ⩽ m ⩽ k ⩽ na ∧ strict(b,m);
21      loop invariant i2: includes(a,na,b,m);
22      loop assigns k, m, b[1..nb-1];
23      loop variant na-k; */
24  for (k = 1; k < na; k++) {
25   if (a[k] > b[m-1]) {
26    b[m] = a[k];
27    /*@ assert a1: b[m] == a[k]; */
28    m = m+1;
29   }
30   /*@ assert a2: includes(a,na,b,m); */
31  }
32  return(m);
33 }
```

Listing 4.1 – Élimination des doubles dans un tableau d'entiers trié, en C/ACSL.

Le code C de la fonction `remove_duplicates` décrit un parcours du tableau `a`, qui compare chaque élément de ce tableau avec l'élément suivant, et place les éléments distincts du tableau `a` dans le tableau `b`, dans le même ordre. La fonction retourne le nombre d'éléments différents dans le tableau `a`. Ce nombre est aussi le nombre d'éléments placés dans le tableau `b`. Il est utile pour spécifier certaines postconditions de la fonction.

Nous détaillons à présent les annotations ACSL de cette fonction C, dans l'ordre de leur apparition dans le listing. Les annotations de la fonction `remove_duplicates` sont composées du *contrat de la fonction*, dans un commentaire ACSL délimité par les balises `/*@` et `*/` (lignes 8 à 15), du *contrat de la boucle*, dans les lignes 20 à 23, et de deux *assertions*, lignes 27 et 30. Pour désigner certaines annotations, nous utilisons la possibilité offerte par le langage ACSL de les étiqueter.

4.4.1 Prédicats nommés

Le listing commence par la définition de trois prédicats nommés, qui facilitent l'expression des annotations ACSL. Le prédicat `sorted` défini dans les lignes 1 et 2 spécifie qu'un tableau d'entiers est trié dans l'ordre croissant. Le prédicat `strict` défini dans les lignes 3 et 4 spécifie de plus que tous ses éléments sont deux à deux distincts. Le prédicat `includes` défini dans les lignes 5 et 6 spécifie que tous les éléments du tableau `b[0..nb-1]` sont dans le tableau `a[0..na-1]`.

4.4.2 Préconditions

Dans le contrat d'une fonction, une annotation `requires R;` spécifie que la *précondition* `R` doit être vérifiée par les paramètres de la fonction lorsqu'elle est appelée.

Les préconditions de la fonction `remove_duplicates` sont étiquetées de `r1` à `r4`.

La précondition `r1` exige que la taille `na` du tableau d'entrée `a` soit non nulle et que la taille `nb` du tableau de sortie `b` lui soit supérieure ou égale, afin que le tableau `b` puisse accueillir tous les éléments du tableau `a`, s'ils sont tous différents.

Les tableaux `a` et `b` sont supposés alloués avec ces tailles (précondition `r2`), dans des zones mémoire disjointes (précondition `r3`).

La précondition `r4` exige que le tableau `a` soit trié dans l'ordre croissant.

4.4.3 Postconditions

Une *postcondition* d'un programme décrit les états attendus après exécution du programme. Une annotation ACSL de la forme `ensures E;` affirme que la *postcondition* `E` est vraie en sortie de fonction. Dans cette postcondition ACSL la variable dédiée `\result` désigne la donnée retournée par la fonction.

Les postconditions de la fonction `remove_duplicates` sont étiquetées de `e1` à `e3`. La postcondition `e1` affirme que le nombre d'éléments placés dans le tableau `b` est non nul et inférieur ou égal à la taille allouée pour ce tableau. Selon la postcondition `e2` le sous-tableau `b[0..\result-1]` doit être trié dans l'ordre croissant et ne pas contenir de doubles. Selon la postcondition `e3` tous les éléments de ce sous-tableau doivent être dans le tableau d'entrée `a`.

Les expressions présentes dans ces postconditions sont toutes évaluées **après** exécution de la fonction. Pour simplifier la spécification et la vérification de cette fonction, nous devons ajouter qu'elle ne modifie pas le contenu du tableau `a`. On peut le faire avec une postcondition et une étiquette (*label* en anglais) prédéfinie `Old` pour désigner l'état **avant** exécution de la fonction. Nous ne détaillons pas cet aspect du langage ACSL, expliqué dans son manuel de documentation, puisqu'on peut aussi spécifier ceci à l'aide d'une clause `assigns`.

Une *clause de modification mémoire* ACSL est une clause de la forme `assigns A;`, où `A` est une liste de zones mémoire. Dans le contrat d'une fonction, cette clause déclare les paramètres de la fonction qui peuvent être modifiés lors de son exécution. Ainsi, la ligne 15 déclare que tous les éléments du tableau `b` peuvent être modifiés. Par conséquent, cette clause spécifie implicitement qu'aucun élément du tableau `a` ne peut être modifié.

4.4.4 Contrat de boucle

La boucle dans les lignes 24 à 29 du corps de la fonction `remove_duplicates` est spécifiée par le contrat de boucle délimité par les balises `/*@` et `*/` dans les lignes 20 à 23, composé de deux invariants de boucle, d'une clause d'assignation (de boucle) et d'un variant de boucle.

Invariant de boucle

Une annotation `loop invariant I;` immédiatement avant une boucle déclare que la formule `I` est un *invariant (inductif)* de cette boucle, c'est-à-dire une propriété qui doit être établie avant exécution de la boucle et préservée à chaque passage dans la boucle.

L'invariant `i1` borne avec précision le nombre `m` d'éléments distincts du tableau `a` déjà parcourus. Il spécifie aussi que le sous-tableau `b[0..m-1]` est trié et sans doublons. L'invariant `i2` affirme que toutes les valeurs de ce sous-tableau sont dans le tableau `a`.

Affectation par une boucle

Une annotation `loop assigns` avant une boucle est une clause de modification mémoire pour cette boucle. Elle définit un sur-ensemble des variables dont la valeur peut être modifiée par une

itération de la boucle. Ainsi la ligne 22 stipule que la boucle ne peut modifier que la variable de boucle `k`, la variable entière `m` et les éléments du tableau `b` entre les indices `1` et `nb-1` inclus.

Variant de boucle

Une annotation `loop variant V;` définit un *variant de boucle* `V` qui peut être utilisé pour établir la terminaison de la boucle. Cette expression entière doit rester positive ou nulle et décroître strictement entre deux itérations successives de la boucle. Par exemple, l'expression `na-k` est un variant de la boucle, spécifié dans la ligne 23 du listing 4.1.

Assertions

Les assertions `a1` et `a2` sont deux propriétés d'état qui facilitent la preuve automatique, comme détaillé dans la partie suivante.

4.4.5 Vérification déductive avec Frama-C

Supposons que le listing 4.1 soit le contenu d'un fichier `duplicates.c`. La vérification statique de la fonction `remove_duplicates` avec Frama-C assisté de WP s'effectue par exécution de la commande `frama-c -wp duplicates.c`. Frama-C indique si chaque condition de vérification générée par WP est prouvée par le solveur SMT Alt-Ergo, en indiquant la durée de chaque preuve.

Avec Frama-C Aluminium et Alt-Ergo 0.99.1, on obtient les résultats suivants :

— Sans les assertions `a1` et `a2`, la preuve de préservation de l'invariant de boucle `i2` échoue.

— Avec les assertions `a1` et `a2`, la correction du contrat de la fonction `remove_duplicates` est démontrée.

Ces résultats peuvent être expliqués comme suit : Une démonstration de préservation de l'invariant de boucle `i2` doit isoler le cas de l'élément `m` dans la quantification universelle

$(\forall\ \mathbb{Z}\ \texttt{i};\ 0 \leqslant \texttt{i} < \texttt{m+1} \Rightarrow \ldots)$

du prédicat `includes`, puis doit justifier que `b[m]` est dans le tableau `a`. L'assertion `a1` aide le prouveur en lui donnant l'indice dans le tableau `a` de cet élément, placé dans le tableau `b` à l'indice `m`. Ceci permet au prouveur d'instancier la quantification existentielle

$(\exists\ \mathbb{Z}\ \texttt{j};\ 0 \leqslant \texttt{j} < \texttt{na} \wedge \ldots)$

du prédicat `includes`. Cette assertion a été ajoutée car on sait que l'instantiation des quantifications existentielles est un aspect difficile de l'automatisation des démonstrations.

L'assertion `a2` impose au prouveur de démontrer l'inclusion de `b[0..m-1]` dans `a[0..na-1]`, puis permet au prouveur d'utiliser ce fait pour démontrer la préservation de l'invariant `i2`. Elle réduit ainsi la difficulté de cette démonstration.

Ces résultats illustrent l'intérêt des assertions pour alléger la preuve de programmes.

Les améliorations apportées dans les outils de vérification déductive permettent de réduire l'effort de spécification. Ainsi, le même exemple sans assertions est prouvé automatiquement avec la version Silicon de Frama-C, plus récente, et la version 1.30 du prouveur Alt-Ergo.

4.5 Correction des systèmes dynamiques

Un système est *dynamique*, ou *exécutable*, s'il évolue dans le temps. Nous n'étudions que des systèmes dynamiques dont l'exécution peut être décrite rigoureusement. Pour certains systèmes nous définirons formellement cette notion d'exécution dans les chapitres suivants. Pour l'instant, nous décrivons la notion de correction d'un système exécutable à partir de la notion intuitive d'exécution, sans formaliser.

La notion de correction d'un système dépend de ses caractéristiques. Les deux caractéristiques qui nous intéressent ici sont la possibilité d'**échec** de certaines exécutions de certains systèmes et la présence potentielle d'une **spécification** au sein du système, posant la question de la correction du reste du système par rapport à cette spécification. Nous définissons successivement la correction de trois sortes de systèmes, de complexité croissante : des systèmes sans spécification et qui n'échouent

jamais (partie 4.5.1), des systèmes qui ne peuvent pas échouer mais qui doivent satisfaire une spécification (partie 4.5.2), et des systèmes qui doivent satisfaire une spécification et peuvent échouer (partie 4.5.3).

Les caractéristiques d'échec et de spécification sont différentes : L'échec est un défaut du système, qui s'observe lors de son exécution, et qu'on veut éviter, tandis que la spécification contribue à la qualité du système, puisqu'elle permet sa vérification, éventuellement statiquement, c'est-à-dire sans exécuter le système. Cependant, ces deux caractéristiques sont liées : Par exemple, on peut parfois spécifier un système avec assez de précision pour qu'il n'échoue jamais s'il satisfait sa spécification. Nous voulons approfondir l'étude de cette relation entre échec et spécification.

4.5.1 Systèmes sans échec ni spécification

Commençons par considérer les systèmes sans spécification ni cause d'échec. Ainsi, leur seule propriété comportementale est leur *terminaison*. On dit qu'un système *termine* si toutes ses exécutions s'effectuent en un temps fini.

Il faut distinguer deux sortes de systèmes, selon que leur terminaison est considérée comme une qualité ou un défaut :

— Les *systèmes réactifs*, comme les services web ou les démons d'un système d'exploitation, doivent toujours répondre à leur environnement. Ainsi, leur terminaison est vue comme un défaut, une propriété à éviter. Nous souhaitons au contraire qu'il satisfassent une *propriété de vivacité*, qui exprime la permanence du service rendu. J'ai étudié la preuve de sûreté de certains systèmes réactifs distribués jusqu'en 2005, puis j'ai étudié la traduction de propriétés de vivacité dans le langage JML d'annotation du langage de programmation objet Java [GGJK08].

— Les *programmes impératifs* sont des exemples de systèmes dont la terminaison est souhaitée, lorsqu'ils implémentent des algorithmes de *calcul*. En effet, un calcul doit délivrer un résultat dans un temps fini, de préférence dans un délai raisonnable, donc ces programmes doivent toujours terminer, pour délivrer ce résultat. J'étudie la preuve de programmes impératifs depuis 2004.

Ainsi, un système réactif est dit *correct* s'il ne termine pas et un programme impératif (sans annotations) qui implémente un calcul est dit *correct* s'il termine.

Remarque 4.5.1. Dès qu'une famille de systèmes contient des systèmes qui peuvent ne pas terminer, la théorie s'écarte de la pratique, en ce sens qu'il est théoriquement possible de raisonner sur la (non-)terminaison, et de la formaliser, par exemple à l'aide de traces infinies, mais qu'il est en pratique impossible de la distinguer d'une exécution très longue (on désigne ici par "pratique" une exécution concrète du système, dans un temps limité). La suite de cette étude écarte cette difficulté, en se plaçant dans le cadre théorique.

4.5.2 Systèmes sans échec avec spécification

A partir d'ici et dans toute la suite de ce mémoire, nous ne considérons plus que des systèmes dont la terminaison est une qualité, une propriété attendue, avec comme principal exemple les **programmes impératifs qui implémentent un calcul**.

Considérons à présent de tels systèmes munis d'une spécification, par exemple un programme impératif spécifié par une postcondition. La correction d'un tel système (par rapport à sa spécification) est la propriété qu'il satisfait cette spécification. Il faut cependant combiner cette propriété avec la propriété de terminaison. Ceci est fait en distinguant la *correction partielle* (*partial correctness* en anglais) du système, qui est sa correction sous hypothèse de terminaison, et sa *correction totale* (*total correctness* en anglais), qui est la conjonction de sa correction partielle et de sa terminaison.

Cette classification des propriétés comportementales des systèmes sans échec est résumée dans l'extrait suivant :

> "The relationship between partial and total correctness can be informally expressed by the equation :

Total correctness = Termination + Partial correctness.

> Total correctness is what we are ultimately interested in, but it is usually easier to prove it by establishing partial correctness and termination separately. [...] Floyd-Hoare logic [...] only deals with partial correctness. Theories of total correctness can be found in the texts by Dijkstra [Dij76] and Gries [Gri81].” [Gor88, Chapitre 1].

Dans la suite de ce mémoire, nous écartons la question de terminaison et nous étudions principalement la **correction partielle**.

4.5.3 Systèmes avec échec et spécification

Nous considérons à présent des systèmes exécutables dont certaines exécutions peuvent échouer, en supposant que chaque échec stoppe brutalement leur exécution. Ainsi la propriété de terminaison se raffine en terminaison normale, dite *terminaison propre*, ou “sans échec” (*clean termination* en anglais) et *terminaison brutale*, ou “avec échec” (*abrupt termination* en anglais). Le cas complémentaire est la non-terminaison, ou *divergence*.

Comme nous l'avons fait pour des programmes C annotés avec ACSL [CKGJ12], pour des causes d'échec comme une division par zéro ou un accès hors des limites d'un tableau, nous supposons ici qu'il est possible d'ajouter au système des spécifications exécutables qui seraient violées avant que le système n'échoue selon une de ces causes d'échec. Ainsi, d'une part, la correction du système par rapport à ces spécifications implique l'absence de tels échecs à l'exécution. D'autre part une vérification statique de cette correction est envisageable. Nous introduisons une notion d'exécution des spécifications du système, au cours de laquelle le système échoue dès qu'une spécification est évaluée comme fausse.

En ignorant la question de distinguabilité entre divergence et exécution longue, discutée dans la remarque 4.5.1, il reste à choisir un moyen de distinguer entre la violation et le respect de la spécification. Pour simplifier, nous considérons que le système doit s'arrêter sur la première spécification violée, dans un état spécial, dit “état d'erreur”, distinguable de tous les autres états, dits “normaux”. Le système ne peut pas être exécuté à partir de l'état d'erreur. Nous détaillerons plus loin ce choix à travers la notion de “sémantique bloquante”.

Cette sémantique bloquante est une sémantique dynamique, d'exécution, qui tente d'évaluer la spécification au cours de l'exécution. Pour simplifier, nous supposons que l'évaluation de la spécification est toujours possible (on dit alors que “la spécification est exécutable”) et qu'elle termine toujours.

A titre d'exemple, nous formalisons la terminaison propre, la terminaison avec erreur et la non-terminaison de programmes impératifs annotés dans le chapitre 6.

4.6 Correction partielle et sémantiques

La correction des programmes est liée à leur sémantique. Parmi les diverses sémantiques formelles d'un programme, on peut distinguer les sémantiques *dynamiques*, qui incluent une notion d'état et définissent une certaine forme d'exécution du programme, et les sémantiques *statiques*, qui définissent la correction des programmes sans se fonder sur une notion d'état.

Parmi les sémantiques dynamiques, la théorie des programmes distingue classiquement les sémantiques “à petites étapes” (*small step semantics* en anglais) et “à grandes étapes” (*big step semantics* en anglais). On parle aussi de “petits pas” et de “grands pas”, mais nous préférons le substantif “étape”, emprunté à P. Lescanne [Les07].

La correction partielle de programmes impératifs spécifiés par une précondition et une postcorrection est connue sour le nom de logique de Hoare [Hoa69]. Historiquement, elle a été présentée comme une sémantique statique, appelée *sémantique axiomatique*. Elle est détaillée dans la partie 5.7.

4.7 Formalisation avec Coq

Pour rendre la preuve de programmes plus populaire, la recherche doit améliorer les méthodes de spécification formelle et de vérification déductive des programmes impératifs. La démarche scientifique exige que la correction de ces nouvelles méthodes soit démontrée avec rigueur, ou à défaut clairement justifiée. En tant que spécialistes de la preuve formelle, nous pouvons envisager d'appliquer nos outils à nos méthodes, en démontrant formellement leur correction. Nous suivons systématiquement cette démarche, en formalisant toute notre étude avec l'assistant de preuve Coq.

Cette formalisation Coq est également appelée *formalisation machine*, ou *mécanisation*, pour la distinguer de la *formalisation papier* qu'elle accompagne. Elle commence dans la partie 4.8 et se poursuit dans les chapitres 5 et 6.

Parmi les environnements de formalisation et de preuve interactive, j'ai choisi l'assistant de preuve Coq pour plusieurs raisons : (a) je me suis initié à cet outil ces dernières années ; (b) cotoyant régulièrement la communauté qui développe, maintient ou utilise fortement cet outil, je peux solliciter de l'aide en cas de problème ; et (c) de nombreux travaux de théorie des programmes existent dans cet environnement. Je me suis efforcé de réduire au minimum les connaissances requises pour lire le code Coq commenté dans ce mémoire.

Les notations de la formalisation Coq ont été choisies pour coïncider maximalement avec celles de la formalisation papier. Ainsi, une explication de l'une d'entre elles convient pour les deux et certaines parties de la formalisation ne sont présentées qu'en Coq, car leur version papier serait quasi-identique. La lecture de la formalisation Coq ne requiert que des connaissances minimes de cet assistant de preuve, distillées au fur et à mesure de la présentation. Enfin, une partie de cette formalisation Coq a déjà été présentée dans la littérature, comme indiqué dans la partie 5.8 de notes bibliographiques.

4.8 États et prédicats

4.8.1 États du système étudié

Nous formalisons les états du système étudié par un type abstrait Coq nommé state :

```
Parameter state : Set.
```

Par exemple, on peut instancier cet ensemble d'états avec le type Coq prédéfini nat. Ainsi, notre étude est paramétrée par ce type abstrait d'états.

4.8.2 Prédicats

Nous formalisons les propriétés d'états présentes dans les spécifications par le type Coq

```
Definition pred := state → Prop.
```

des fonctions des états dans les propositions Coq. Des opérations sur ces prédicats seront définies dans la partie 5.7.1, lorsqu'ils seront utilisés pour formaliser des préconditions et postconditions.

Permettez-moi ici de marquer une pause et de justifier dans la suite de cette partie le choix du mot "prédicat" (*predicate* en anglais) pour désigner ces propriétés d'états. Il s'agissait de choisir un couple composé d'une expression en français et d'une expression en anglais, parmi les expressions parfois utilisées avec cette signification dans la littérature, en privilégiant celles qui ne correspondent pas (trop) avec d'autres notions liées à cette étude. Pour chaque couple candidat, une sous-partie indique quelles références l'utilisent, en citant parfois un extrait significatif, puis justifie l'élimination de ce choix, jusqu'à la dernière partie qui justifie le choix du couple (prédicat, *predicate*).

(propriété d'état, *state property*) :

L'expression "propriété d'état" (*state property* en anglais) est clairement la plus précise, mais nous l'excluons pour sa lourdeur, mais aussi pour éviter une confusion avec les "propriétés des

programmes" définies dans la partie 4.5.

(proposition, *proposition*) :

C'est le nom qu'utilise Floyd dans son article fondateur [Flo67]. Nous l'excluons pour éviter toute confusion avec les propositions du langage Coq (type Prop).

(assertion, *assertion*) :

C'est le mot le plus souvent utilisé pour désigner ces propriétés d'état. Il est absent de [Flo67], mais présent dans [Gor88, Hoa02].

> "an assertion language is a first-order language over interpreted symbols for expressing functions and relations over some concrete domains such as integers, arrays, and lists of integers [and] a formula in [an] assertion language [is referred] as a *state formula*, or simply as an *assertion*." [MP95].

Nous excluons ce mot car nous lui donnons un sens plus précis dans le chapitre 6.

(condition, *condition*) :

"Condition" [Gor88, Chapitre 1] est exclu, car on désigne souvent par "condition" une expression booléenne de contrôle dans les commandes conditionnelles et de répétition du langage de programmation.

(déclaration, *statement*) :

Gordon parle d'abord de "condition", puis choisit le mot "statement" (et "assertion" [Gor88, page 44]) :

> "Here the term 'statement' is used for conditions on program variables that occur in correctness specifications [...]. There is a potential for confusion here because some writers use this word for commands (as in 'for-statement')" [Gor88, chapitre 1, page 4].

Ce choix ne convient pas, car le mot "statement" s'est répandu pour désigner toute instruction du langage de programmation.

(prédicat, *predicate*) :

Dijkstra parle indifféremment de "prédicat" et de "condition", après la réserve suivante :

> "I could, and perhaps should, [reserve] the term "predicate" for the formal expression denoting the "condition" : [...] Knowing myself I do not expect to indulge very much in such a mannerism. [...] I shall use synonymously expressions such as "a state for which a predicate is true" and "a state that satisfies a condition" [...] and "a state in which a condition holds", etc." [Dij76, page 14].

Nous serons plus "maniéristes" que Dijkstra, en n'utilisant que des expressions telles que "un état qui satisfait un prédicat" et "un état tel qu'un prédicat est vrai".

4.9 Synthèse

Après un bref état de l'art sur les langages de spécification fonctionnelle, de contrats et les outils de vérification associés, nous avons défini la correction d'un système exécutable par rapport à sa spécification. Puis nous avons focalisé notre étude sur la correction partielle et nous avons justifié la nécessité de définir une sémantique formelle dynamique et une sémantique formelle statique des langages de contrats.

La suite de ce document définit ces sémantiques pour un langage impératif réduit à un jeu d'instructions Turing-complet spécifiées par une précondition, une postcondition, des assertions et des invariants de boucle, qui sont des propriétés d'états, dorénavant appelées "prédicats".

Chapitre 5

Sémantique et correction partielle des programmes impératifs

Sommaire

Qu'on le veuille ou non, la programmation impérative est le paradigme de programmation le plus populaire. C'est vrai depuis plusieurs décennies et devrait le rester encore longtemps. Pour s'en convaincre, il suffit de consulter le TOP 20 des langages de programmation les plus utilisés, résultant d'une analyse des besoins de l'industrie du logiciel menée depuis 2001 par la communauté TIOBE [TIO17]. En 2017, les huit premiers langages dans ce classement sont fondés sur la programmation impérative (parfois à travers la programmation orientée objet).

Nous voulons définir rigoureusement la correction partielle des programmes impératifs formellement spécifiés. Ce chapitre traite le cas d'une spécification limitée à une précondition et une postcondition. Le chapitre suivant généralise la spécification à des annotations dans les programmes.

Comme expliqué dans la partie 4.6, il s'agit de formaliser au moins une sémantique dynamique et une sémantique statique de ces programmes, correspondant respectivement à leur exécution et à leur vérification déductive. Nous allons plus loin, en présentant plusieurs sémantiques dynamiques et statiques et en démontrant toutes les équivalences entre elles. Comme justifié dans les parties 4.7 et 5.1, dans un souci de rigueur et de re-utilisabilité, nous encodons toutes ces définitions en Coq et nous utilisons cet assistant de preuve pour mécaniser toutes ces démonstrations.

La partie 5.1 présente mes motivations pour mener et rédiger la présente étude, dans le contexte de certains de mes travaux de recherche. Le sujet de cette étude et la suite du plan du chapitre sont précisés dans la partie 5.2.

5.1 Contexte et motivations

A partir de 2003, j'ai étudié la spécification fonctionnelle (par des contrats formels) de programmes impératifs (écrits en langage Java, C et PHP) et la vérification déductive de ces contrats.

Avec J. Groslambert j'ai conçu une méthode de vérification de propriétés temporelles par génération automatique d'annotations JML [GG07]. Le cas des propriétés de sûreté a été traité dans [GG06c], celui des propriétés de vivacité dans [GGJK08]. J'ai aidé J. Groslambert à implémenter un prototype, nommé JAG (pour *JML Assertion Generator*), de transformation de propriétés temporelles en annotations JML équivalentes [GG06a, GG06b].

En collaboration avec le CEA, j'ai élaboré des méthodes de combinaison d'analyse statique, de simplification syntaxique et de test structurel, pour lever des menaces d'erreurs à l'exécution dans les programmes C [CKGJ10b, CKGJ12] et assister la compréhension des échecs de preuve de programmes [PKGJ14a, PKGJ14b].

Durant ces travaux de recherche j'ai rencontré à plusieurs reprises la même difficulté, qui était de justifier rigoureusement la correction des méthodes formelles proposées.

Dans ce but, j'ai encouragé et aidé les doctorants à formaliser la méthode sur papier, à énoncer un théorème de correction et à rédiger une démonstration de ce théorème. Mais cette démarche sur papier, dite *informelle*, était loin de me satisfaire, car les problèmes traités étaient suffisamment complexes pour que des erreurs se glissent dans cette formalisation papier et échappent à des relectures attentives. Durant chaque co-encadrement de thèse, j'ai péniblement vécu la période de mise au point de cette formalisation avec le doctorant, son directeur et ses autres co-encadrants. En effet, il n'est jamais facile de détecter des erreurs de formalisation, et encore moins d'expliquer ces erreurs, quand on dispose d'un temps très limité pour cette vérification et cette interaction.

L'extrait suivant du rapport de Pascale Le Gall sur la thèse de G. Petiot [Pet15] témoigne de la complexité d'une formalisation :

> "Comme il ne s'agit pas d'un langage défini à des fins pédagogiques, mais d'un langage réel, la sémantique présentée est particulièrement riche, au sens qu'elle nécessite un grand nombre de règles."

L'idéal serait que chaque auteur de formalisation identifie lui-même ses erreurs, même si, comme le doctorant, il n'est pas expert en la matière. Il existe à ce jour des langages de formalisation très expressifs et des outils d'aide à la démonstration très puissants (les assistants de preuve), qui pourraient rationaliser ce travail et en améliorer la qualité. Malheureusement, la complexité de ces outils s'ajoutant à celle du problème étudié, une telle mécanisation du raisonnement est le plus souvent trop lourde à mener dans la durée d'une thèse et sur un langage complet, à moins bien sûr que ce ne soit le sujet principal de cette thèse.

Pour aller dans cette direction de l'étude *formelle* des méthodes de vérification, il faut choisir un niveau de formalisation des langages et méthodes permettant d'atteindre les deux objectifs suivants :

1. lever toute ambiguité sur les notions traitées et les méthodes appliquées, et
2. démontrer de manière rigoureuse de bonnes propriétés de ces méthodes, en commençant par leur correction.

Ces deux objectifs sont en partie contradictoires : L'objectif (2) exige une grande précision dans la formalisation, pour augmenter la profondeur des propriétés démontrées, tandis que l'objectif (1) exige que la formalisation reste suffisamment simple, pour que les démonstrations soient rapidement réalisées. La formalisation ne doit pas ensevelir l'intuition sous des montagnes de notions et de notations. Il s'agit toujours de trouver un point d'équilibre entre ces deux objectifs.

Une partie des difficultés de formalisation de la théorie des programmes provient de l'abondance de détails qui encombrent l'esprit du formalisateur et le texte de sa formalisation, lorsqu'il traite un langage concret – comme le langage C – et des aspects délicats de ce langage – comme les pointeurs, les exceptions, les entiers machine, etc. C'est pourquoi je recommande d'effectuer ce travail formel sur un langage le plus simplifié possible, comme celui de la logique de Hoare. Un second facteur de réussite d'une telle formalisation est de ne pas partir de rien, mais d'adapter et d'étendre une formalisation pré-existante dont on s'est assuré la validité. En suivant ces deux

recommandations, je propose dans ce mémoire une formalisation Coq de la correction d'un langage de programmes impératifs annotés avec des spécifications formelles. Le pari est que cette formalisation soit suffisamment claire et légère pour être assimilée par un doctorant en début de thèse, dans la perspective qu'il l'adapte et l'étende aux spécificités et subtilités de son sujet de recherche, sans y consacrer tout son temps. Pour simplifier, la question de la terminaison des programmes est ignorée.

5.2 Sujet traité et mode opératoire

Ce chapitre définit formellement la correction partielle de programmes impératifs vis-à-vis d'une précondition et d'une postcondition. C'est essentiellement une synthèse de l'existant, conçue et rédigée pour être étendue dans le chapitre suivant. La référence principale est le support du cours donné par Xavier Leroy à l'école d'été VSTA (*Verification Technology, Systems & Applications summer school*) en 2013 [Ler13], qui synthétise et étend de nombreuses contributions antérieures. L'apport de chaque référence utilisée est détaillé après les parties techniques, dans la partie 5.8 de notes bibliographiques.

Le maître-mot ici est "simplicité". Nous proposons une formalisation aussi légère que possible d'un langage de programmes impératifs Turing-complet, de ses sémantiques et de leurs propriétés. Pour simplifier, nous faisons abstraction des parties du langage qui n'ont pas d'impact direct sur ces propriétés, en les remplaçant par leur version abstraite. Ceci concerne les notions d'état et de propriété d'état présentées dans la partie 4.8, mais aussi les notions d'affectation et de condition booléenne présentées plus loin.

Nous proposons plusieurs définitions de la correction partielle, fondées sur plusieurs sémantiques des programmes impératifs, comme expliqué dans la partie 5.2.1. Nous démontrons ensuite des équivalences entre définitions, selon la progression décrite dans la partie 5.2.2. La suite du plan du chapitre est présentée dans la partie 5.2.3.

Le lien avec le titre du mémoire est le suivant : Les systèmes considérés dans ce chapitre sont des programmes impératifs spécifiés par une précondition et une postcondition. Ils sont paramétrés par le type abstrait state défini dans la partie 4.8.

5.2.1 Sémantiques

Avant de détailler plus loin chaque sémantique considérée, nous introduisons dès à présent (quelques-uns de) ses noms usuels, le nom choisi pour le prédicat (co-)inductif Coq correspondant, et son type. Ce type est fondé sur les types state des états et pred des prédicats présentés dans la partie 4.8, sur le type Prop des propositions Coq, mais aussi sur le type cmd des commandes du langage de programmation, présenté dans la partie 5.3.

Nous considérons d'abord deux sémantiques dynamiques. La *sémantique de réduction*, dite "à petites étapes" (*small step semantics* en anglais), est formalisée par le prédicat inductif

```
red : (cmd * state) → (cmd * state) → Prop
```

où * est le symbole du produit cartésien. La *sémantique naturelle*, dite "à grandes étapes" (*big step semantics* en anglais) est formalisée par le prédicat inductif

```
big : cmd → (state → state → Prop)
```

Pour toute commande c, (big c) est la relation binaire sur les états qui associe à tout état s tous les états s' dans lesquels la commande c peut terminer, lorsqu'elle est exécutée à partir de l'état s.

Quoique la propriété de correction partielle porte sur toutes les exécutions du programme, l'enjeu est de l'établir sans exécuter le programme, de manière statique. Nous verrons une sémantique statique, appelée "sémantique axiomatique" qui permet cette vérification statique. Elle est formalisée par le prédicat inductif

```
triple : pred → cmd → pred → Prop
```

tel que la proposition Coq (triple P c Q) encode le triplet de Hoare $\{P\}\ c\ \{Q\}$.

5.2.2 Progression

> "[The] most useful definitions are those that are given in many different forms together with proof that all of them are equivalent" [HH98, Chapitre 3].

Nous allons montrer (formellement) des équivalences entre ces sémantiques, plus exactement entre des caractérisations de la correction des programmes fondées sur elles, en suivant une démarche systématique expliquée ici.

Le tableau de la figure 5.1 récapitule les sémantiques considérées, leur nom en Coq et leur type Coq. Comme indiqué dans la première colonne de ce tableau, nous attribuons un niveau à chaque sémantique. La sémantique à petites étapes est au niveau 1. La sémantique à grandes étapes est au niveau 2. Elle est complétée par la sémantique de divergence, à laquelle nous attribuons le niveau parallèle 2'. La sémantique axiomatique occupe le niveau 3.

La colonne "Nature" du tableau de la figure 5.1 indique la nature de chaque définition Coq. La nature "Inductive" (resp. "Co-inductive") signale un prédicat inductif (resp. co-inductif), qui formalise un système d'inférence avec une sémantique de plus petit (resp. grand) point fixe. A partir du niveau 2, la nature "Spécification" signale une sémantique spécifiée par rapport à une sémantique du niveau précédent. Ainsi, la définition terminates spécifie la sémantique à grandes étapes big (niveau 2) à l'aide de la sémantique à petites étapes red (niveau 1). De même, la définition triple_big spécifie la sémantique axiomatique triple (niveau 3) à l'aide de la sémantique naturelle big (niveau 2).

Niveau	Sémantique	Nature	nom en Coq	type Coq
1	réduction	Inductive	red	cmd * state → cmd * state → Prop
2	naturelle	Spécification	terminates	cmd → state → state → Prop
		Inductive	exec	state → cmd → state → Prop
		Inductive	big	cmd → state → state → Prop
2'	divergence	Spécification	diverges	cmd → state → Prop
		Co-inductive	execinf	state → cmd → Prop
		Co-inductive	biginf	cmd → state → Prop
3	axiomatique	Spécification	triple_big	pred → cmd → pred → Prop
		Inductive	triple	pred → cmd → pred → Prop

FIGURE 5.1 – Sémantiques des programmes impératifs.

Les définitions big et biginf ne font qu'échanger l'ordre des deux premiers paramètres des définitions exec et execinf de [Ler13], pour que leur type corresponde à celui de leur spécification respective terminates et diverges. L'ordre des paramètres de la définition

```
big : cmd → (state → state → Prop)
```

facilite aussi sa mise en correspondance avec la sémantique axiomatique. En effet, considérons la variante (triple_rel : cmd → pred → pred → Prop) de la sémantique axiomatique (triple : pred → cmd → pred → Prop) obtenue en échangeant l'ordre des deux premiers paramètres, et rappelons que pred est le type state → Prop des propriétés sur les états. Alors, la variante (triple_rel c) de la sémantique axiomatique de la commande c, de type

```
triple_rel c : (state → Prop) → (state → Prop) → Prop
```

est simplement la généralisation de la sémantique naturelle

```
big c : state → state → Prop
```

de cette commande, d'un état de type state à un ensemble d'états, représenté par son prédicat caractéristique de type state → Prop.

Lorsqu'on propose un système d'inférence, comme chaque prédicat (co-)inductif de la figure 5.1, on doit démontrer sa correction et sa complétude par rapport à une autre définition de ce qu'il

caractérise, elle-même admise comme correcte. Nous considérons cette référence comme une "spécification" dont le système d'inférence est une "implémentation", entre guillemets car cette "implémentation" est non déterministe, au niveau du choix de la prochaine règle à appliquer et au niveau des différentes manières d'appliquer la même règle à chaque pas d'exécution de ce système d'inférence.

Dans la suite de ce chapitre nous étudions les niveaux dans l'ordre croissant. A chaque niveau supérieur ou égal à 2, nous procédons en trois étapes. Nous spécifions d'abord la sémantique de ce niveau à partir d'une sémantique du niveau précédent. Ensuite, nous proposons un système d'inférence qui axiomatise cette sémantique. Enfin, nous démontrons formellement l'équivalence entre ce système et sa spécification, ce qui revient à établir sa correction et sa complétude. Ainsi, au niveau 2, nous démontrons que la définition inductive big de la sémantique à grandes étapes est équivalente à sa spécification terminates à l'aide de la sémantique à petites étapes red du niveau 1. La spécification triple_big du niveau 3 étend directement aux prédicats (propriétés d'états) notre version big de la sémantique naturelle (niveau 2), définie sur les états. Nous démontrons la correction et la complétude de la sémantique axiomatique triple par rapport à la sémantique naturelle big en montrant que le système d'inférence triple définit la même notion de correction partielle que la spécification triple_big.

Les définitions red, terminates, exec, diverges, execinf et triple sont issues de [Ler13]. Les autres définitions, la structuration en niveaux et la systématisation de la démarche sont des contributions.

5.2.3 Plan du chapitre

La partie 5.3 introduit le langage de programmes impératifs étudié. La partie 5.4 définit sa sémantique de réduction, à petites étapes. Les parties 5.5 et 5.6 définissent sa sémantique à grandes étapes, respectivement dans les cas de terminaison et de divergence. La partie 5.7 définit la sémantique axiomatique de ce langage, proche de la logique de Hoare. La partie 5.8 retrace l'historique des travaux théoriques sur le sujet de ce chapitre, depuis les articles fondateurs jusqu'aux formalisations récentes. Enfin, la partie 5.9 résume et commente ce chapitre préparatoire au chapitre suivant.

5.3 Syntaxe du langage

Nous désignons par CMD un langage impératif réduit au jeu d'instructions impératives présenté dans la partie 5.3.2. Nous simplifions les présentations usuelles de tels langages comme détaillé dans la partie 5.3.1.

5.3.1 Simplifications

Les présentations usuelles d'un jeu d'instructions impératives [Win93, Ler13] sont composées d'une syntaxe pour les expressions de calcul (constantes, variables et opérateurs de calcul), d'une syntaxe pour les conditions booléennes présentes dans les instructions de contrôle, et enfin d'une syntaxe des instructions. Or, les sémantiques et propriétés de ces instructions, qui nous intéressent ici, sont totalement indépendantes de la nature de ces expressions booléennes et de calcul.

Pour expliciter cette indépendance et aller à l'essentiel, nous adoptons pour les expressions le point de vue abstrait de leur sémantique : les variables sont remplacées par une notion d'état abstraite (type abstrait state défini dans la partie 4.8). La commande d'affectation est vue comme un changement d'état (déterministe), comme détaillé dans la partie 5.3.2. Ainsi il n'est pas nécessaire de définir des expressions de calcul.

Pour éviter d'avoir à définir la syntaxe des expressions booléennes présentes dans les instructions conditionnelles (If) et de répétition (While) du langage, nous les identifions avec leur sémantique de fonction booléenne sur les états, en définissant pour elles le type Coq

```
Definition bexpr := state → bool.
```

Par rapport à [Ler13] ces simplifications éliminent quatre pages de présentation et 84 lignes de code Coq.

5.3.2 Commandes

Pour coïncider avec la formalisation [Ler13] que nous adaptons, les instructions sont appelées "commandes".

Le langage CMD est composé des commandes définies par la grammaire

$$c, c' ::= \mathsf{Skip} \mid \mathsf{Update}\ u \mid \mathsf{Seq}\ c\ c' \mid \mathsf{If}\ b\ c\ c' \mid \mathsf{While}\ b\ c$$

où u est une fonction totale sur les états et b est une expression booléenne. Chaque commande est détaillée dans une partie ci-dessous.

Cette grammaire est formalisée par le type inductif cmd du listing 5.1.

```
Inductive cmd :=                          (* Commands: *)
| Skip : cmd                              (* - do-nothing command      *)
| Update : (state → state) → cmd          (* - state transformer       *)
| Seq : cmd → cmd → cmd                   (* - sequential composition  *)
| If : bexpr → cmd → cmd → cmd            (* - conditional statement   *)
| While : bexpr → cmd → cmd.              (* - repetition statement    *)
```

Listing 5.1 – Syntaxe du langage CMD en Coq.

Notre langage CMD diffère de celui de [Ler13] sur quatre points :

1. Les noms des constructeurs sont simplifiés, par exemple If au lieu de Cifthenelse.
2. L'affectation explicite Cassign de [Ler13] est remplacée par la mise à jour déterministe Update.
3. Nous n'introduisons pas les notations usuelles (syntaxe concrète) $c; c'$, if b then c_1 else c_2 et while b do c done, pour les commandes Seq $c\ c'$, If $b\ c_1\ c_2$ et While $b\ c$, pour au moins deux raisons : (i) nous ne traitons pas ici de syntaxe concrète, mais de syntaxe abstraite, où chaque commande est un arbre dont les nœuds sont étiquetés par Skip, Update, Seq, If et While (ce qui élimine toute question d'ambiguité syntaxique), et (ii) la syntaxe concrète des commandes If et While est plus lourde que leur syntaxe abstraite.
4. Le langage CMD porte le même nom que le type inductif cmd, au lieu du nom IMP [Ler13], car il se réduit à ses commandes.

Lorsque nous indiquons dans ce chapitre que nous reprenons telle ou telle partie de [Ler13], il faut comprendre que nous parlons de notre adaptation du code de [Ler13] à ces modifications du langage CMD.

Dans toute la suite les mots "programme", "instruction" et "commande" sont considérés comme synonymes.

La commande qui ne fait rien

La commande Skip ne modifie pas l'état du système. Elle a peu d'intérêt pratique, mais elle a un intérêt théorique, pour définir la sémantique de réduction (dans la partie 5.4).

Mise à jour

Les présentations usuelles du langage CMD [Ler13, MT13] définissent une commande d'affectation de la forme $x := e$, où x est une variable et e est une expression. Ceci nécessite de définir la syntaxe des variables et des expressions, mais aussi la notion sémantique de substitution, qui doit s'appliquer à toute expression et à toute formule.

Pour éviter ces formalisations, nous adoptons ici un point de vue un peu plus abstrait, selon lequel l'affectation est vue comme une *transformation d'état* (*state transformer* en anglais), formalisée par une fonction totale u sur les états, de type state → state, usuellement proposée comme sémantique dénotationnelle de l'affectation.

La commande de *mise à jour* est notée Update u, pour la distinguer de l'affectation usuelle Assign : var → expr → cmd.

Composition séquentielle

La *composition séquentielle* (ou *séquence*) (Seq c c') des deux commandes c et c', usuellement notée $(c; c')$, est la commande qui exécute d'abord la commande c, puis la commande c' si la commande c termine. Sinon, la séquence ne termine pas. C'est une construction essentielle en programmation impérative, qui permet de contrôler l'ordre d'exécution de ses ingrédients.

Conditionnelle

La commande conditionnelle (If b c c') permet un traitement par cas : La commande c est exécutée si la condition booléenne b est vraie. Sinon, la commande c' est exécutée. Dans les langages C et Java, cette instruction s'écrit

`if(`b`)` c `else` c'

et des accolades ouvrantes et fermantes débutent et terminent c et c' si ce sont des séquences d'instructions.

Répétition

L'exécution de la commande de *répétition* (While b c), plus communément appelée commande de *boucle*, répète la commande c tant que l'expression booléenne b est vraie. Elle termine dès que b devient fausse. Si b ne devient jamais fausse, l'exécution de cette commande ne termine pas. La commande c est appelée *corps de boucle*. L'expression b est appelée *condition de boucle*.

La syntaxe concrète du terme (While b c) dans les langages C et Java est l'instruction

`while(`b`)` c

avec la même condition d'accolades que pour l'instruction `if`.

5.4 Sémantique de réduction

> "A standard way to give semantics to languages such as IMP, where programs may not terminate, is reduction semantics, popularized by Plotkin under the name "structural operational semantics" [Plo04], and also called "small-step semantics"." [Ler13, partie 2.2].

Sous le même nom nous adaptons la relation de réduction red définie dans [Ler13, partie 2.2]. C'est une relation de réécriture de couples (c, s) composés d'une commande c et d'un état s. La réécriture $c, s \to c', s'$, formalisée par la proposition Coq (red (c, s) (c', s')), définit une exécution élémentaire de la commande c à partir de l'état s, qui "réduit" la commande c à la commande résiduelle c' et place le système dans l'état s'.

5.4.1 Système d'inférence

La figure 5.2 présente un système d'inférence qui définit inductivement cette relation de réduction. Dans la règle [red_update], la notation $u\ s$ désigne l'application de la transformation d'état (u : state → state) à l'état s. Dans toute la suite toute application d'une fonction unaire f à un argument x est notée $f\ x$, comme le veut l'usage en programmation fonctionnelle, au lieu de la notation $f(x)$ des langages de programmation impérative comme C ou Java.

Les autres règles de la figure 5.2 sont classiques et correspondent à la sémantique informelle des commandes donnée dans la partie 5.3.2. C'est pourquoi elles ne sont pas détaillées ici.

Outre les adaptations déjà signalées, le système de la figure 5.2 diffère de son analogue dans [Ler13] par la correction c'_1, s' au lieu de c'_1, s dans la règle [red_seq_left], et par la suppression des parenthèses dans la notation $(c, s) \to (c', s')$, remplacée par la notation $c, s \to c', s'$.

$$\mathsf{Update}\ u, s \to \mathsf{Skip}, u\ s \quad [\mathsf{red\_update}]$$

$$\frac{c_1, s \to c_1', s'}{\mathsf{Seq}\ c_1\ c_2, s \to \mathsf{Seq}\ c_1'\ c_2, s'}\ [\mathsf{red\_seq\_left}] \qquad \mathsf{Seq\ Skip}\ c, s \to c, s\ [\mathsf{red\_seq\_skip}]$$

$$\frac{b\ s = \mathsf{true}}{\mathsf{If}\ b\ c_1\ c_2, s \to c_1, s}\ [\mathsf{red\_if\_true}] \qquad \frac{b\ s = \mathsf{false}}{\mathsf{If}\ b\ c_1\ c_2, s \to c_2, s}\ [\mathsf{red\_if\_false}]$$

$$\frac{b\ s = \mathsf{true}}{\mathsf{While}\ b\ c, s \to \mathsf{Seq}\ c\ (\mathsf{While}\ b\ c), s}\ [\mathsf{red\_while\_true}]$$

$$\frac{b\ s = \mathsf{false}}{\mathsf{While}\ b\ c, s \to \mathsf{Skip}, s}\ [\mathsf{red\_while\_false}]$$

FIGURE 5.2 – Sémantique de réduction.

5.4.2 Formalisation par un prédicat inductif Coq

La définition inductive du listing 5.2 formalise ce système d'inférence en Coq.

```
Inductive red: cmd * state → cmd * state → Prop :=
| red_update: ∀ u s,
   red (Update u, s) (Skip, u s)
| red_seq_left: ∀ c1 c2 s c1' s',
   red (c1, s) (c1', s')
   → red (Seq c1 c2, s) (Seq c1' c2, s')
| red_seq_skip: ∀ c s,
   red (Seq Skip c, s) (c, s)
| red_if_true: ∀ s b c1 c2,
   b s = true
   → red (If b c1 c2, s) (c1, s)
| red_if_false: ∀ s b c1 c2,
   b s = false
   → red (If b c1 c2, s) (c2, s)
| red_while_true: ∀ s b c,
   b s = true
   → red (While b c, s) (Seq c (While b c), s)
| red_while_false: ∀ b c s,
   b s = false
   → red (While b c, s) (Skip, s).
```

Listing 5.2 – Sémantique de réduction en Coq.

5.4.3 Traduction des systèmes d'inférence en Coq

Plus généralement, une vaste classe de systèmes d'inférence se formalise simplement en Coq, selon les règles suivantes :

1. Associer un *prédicat inductif* Coq au système d'inférence. Un prédicat inductif est une fonction Coq définie inductivement et à valeurs dans le type Prop des propositions Coq. Ici, le prédicat inductif red est associé au système d'inférence de la sémantique de réduction.
2. Associer à chaque axiome et règle du système d'inférence un constructeur de ce type inductif. Dans tout ce document, nous donnons le même nom à chaque axiome ou règle et au constructeur correspondant, par exemple red_update pour le premier axiome de la figure 5.2 et pour le premier constructeur du prédicat inductif red.
3. Pour obtenir le type de chaque constructeur, ajouter à l'axiome ou à la règle correspondante une quantification universelle pour chaque variable libre et une implication après chaque prémisse (s'il y en a).

Ce principe de traduction s'applique à tous les systèmes présentés dans ce document. Son explication est inspirée par l'extrait suivant :

> "The Coq translation of such a definition by inference rules is called an inductive predicate. [...] The general recipe for translating inference rules to an inductive predicate is as follows. First, write each axiom and rule as a proper logical formula, using implications and universal quantification over free variables. [...] Second, give a name to each rule. (These names are called "constructors" by analogy with data type constructors.) Last, wrap these named rules in an inductive predicate definition" [Ler13, partie 2.2].

Pour maximiser la ressemblance entre un axiome ou une règle et son code Coq, nous appliquons systématiquement les règles d'indentation suivantes, illustrées par le listing 5.3 : Toutes les quantifications universelles sont placées sur la même ligne que le nom du constructeur ; la conclusion de l'axiome ou de la règle est codée seule sur la dernière ligne ; pour une règle, cette dernière ligne commence par le symbole d'implication →, qui suggère le trait horizontal de la règle ; les prémisses sont placées sur les lignes intermédiaires, séparées par →, avec des retours à la ligne pour contrôler la largeur de ces lignes.

```
Inductive system_name: ... → Prop :=
| constructor_1: ∀ x1 .. xn,
   premisses separated by →
   → conclusion
| constructor_2: ...
...
```

Listing 5.3 – Forme générale de la formalisation Coq d'un système d'inférence.

Tous les systèmes d'inférence présentés dans ce document sont codés en Coq de cette manière. Le codage Coq d'une règle d'inférence étant quasiment aussi lisible que la règle elle-même, nous nous autorisons à présenter plus loin certains systèmes d'inférence uniquement par leur code en Coq.

5.5 Sémantique naturelle

La *sémantique naturelle* (*natural semantics* en anglais), ou sémantique à grandes étapes (*big step semantics* en anglais) d'une commande c est la relation binaire (big c) sur les états telle que (big c s s') est vraie si et seulement si l'exécution de la commande c dans l'état s termine dans l'état s'.

Leroy [Ler13, partie 2.3] formalise la sémantique naturelle avec le prédicat inductif Coq

exec : state → cmd → state → Prop.

Nous préférons placer la commande en premier paramètre de notre formalisation de cette sémantique, en définissant le prédicat inductif Coq

big : cmd → state → state → Prop,

car cet ordre des paramètres permet de considérer simplement la relation binaire

big c : state → state → Prop

sur les états, comme dans la définition informelle ci-dessus, pour toute commande c du langage.

Selon la démarche générale introduite dans la partie 5.2.2, commençons par **spécifier** cette sémantique naturelle à partir de la sémantique de réduction. Informellement, l'exécution de la commande c dans l'état s termine dans l'état s' si et seulement s'il existe une séquence finie de réductions entre la configuration (c, s) et la configuration (Skip, s'). Cette spécification est la

```
Definition terminates (c: cmd) (s s': state) : Prop := star red (c, s) (Skip, s').
```

de [Ler13, page 8], où le prédicat inductif star défini par

```
Inductive star: A → A → Prop :=
| star_refl: ∀ a, star a a
| star_step: ∀ a b c, R a b → star b c → star a c.
```

est tel que (star r) formalise la fermeture réflexive et transitive r^* (étoile de Kleene) de la relation binaire r sur un type A quelconque.

Ensuite, nous proposons le système d'inférence de la figure 5.3. Sa traduction en Coq du listing 5.4 regroupe les deux règles [big_if_true] et [big_if_false] en une seule nommée [big_if].

$$\mathsf{big\ Skip}\ s\ s\ \ [\mathsf{big\_skip}] \qquad \mathsf{big}\ (\mathsf{Update}\ u)\ s\ (u\ s)\ \ [\mathsf{big\_update}]$$

$$\frac{\mathsf{big}\ c_1\ s\ s_1 \qquad \mathsf{big}\ c_2\ s_1\ s'}{\mathsf{big}\ (\mathsf{Seq}\ c_1\ c_2)\ s\ s'}\ [\mathsf{big\_seq}]$$

$$\frac{b\ s = \mathsf{true} \qquad \mathsf{big}\ c_1\ s\ s'}{\mathsf{big}\ (\mathsf{If}\ b\ c_1\ c_2)\ s\ s'}\ [\mathsf{big\_if\_true}] \qquad \frac{b\ s = \mathsf{false} \qquad \mathsf{big}\ c_2\ s\ s'}{\mathsf{big}\ (\mathsf{If}\ b\ c_1\ c_2)\ s\ s'}\ [\mathsf{big\_if\_false}]$$

$$\frac{b\ s = \mathsf{true} \qquad \mathsf{big}\ c\ s\ s_1 \qquad \mathsf{big}\ (\mathsf{While}\ b\ c)\ s_1\ s'}{\mathsf{big}\ (\mathsf{While}\ b\ c)\ s\ s'}\ [\mathsf{big\_while\_loop}]$$

$$\frac{b\ s = \mathsf{false}}{\mathsf{big}\ (\mathsf{While}\ b\ c)\ s\ s}\ [\mathsf{big\_while\_stop}]$$

FIGURE 5.3 – Sémantique naturelle.

```
Inductive big: cmd → state → state → Prop :=
| big_skip: ∀ s,
   big Skip s s
| big_update: ∀ s u,
   big (Update u) s (u s)
| big_seq: ∀ s c1 c2 s1 s',
   big c1 s s1 → big c2 s1 s'
   → big (Seq c1 c2) s s'
| big_if: ∀ s (b: bexpr) c1 c2 s',
   big (if b s then c1 else c2) s s'
   → big (If b c1 c2) s s'
| big_while_loop: ∀ s b c s1 s',
   b s = true → big c s s1 → big (While b c) s1 s'
   → big (While b c) s s'
| big_while_stop: ∀ s b c,
   b s = false
   → big (While b c) s s.
```

Listing 5.4 – Sémantique naturelle en Coq.

Nous démontrons d'abord que notre sémantique naturelle big : cmd → state → state → Prop ne diffère de la sémantique naturelle exec : state → cmd → state → Prop de [Ler13] que par l'ordre des paramètres, comme formalisé par le lemme

```
Lemma big_exec_eq: ∀ c s s', big c s s' ↔ exec s c s'.
```

Puis nous utilisons ce lemme big_exec_eq pour adapter à big tout ce que Leroy démontre pour exec, ce qui permet en particulier de démontrer facilement que la définition inductive big et sa spécification terminates sont équivalentes :

```
Lemma terminates_big_eq: ∀ c s s', terminates c s s' ↔ big c s s'.
```

En complément de [Ler13], nous démontrons que la relation binaire (big c) est déterministe, pour toutes les commandes c :

```
Lemma big_deterministic: ∀ c x y, big c x y → ∀ z, big c x z → y = z.
```

5.6 Sémantique de divergence

La sémantique naturelle d'une commande c exige sa terminaison. Nous la complétons ici avec une sémantique qui traite les cas de non-terminaison.

La *sémantique de divergence* est le prédicat biginf tel que (biginf c s) est vrai si et seulement si l'exécution de la commande c à partir de l'état s ne termine pas. À l'ordre des paramètres près ce prédicat correspond au prédicat execinf : state → cmd → Prop défini par Leroy sous le titre "natural semantics for divergence" [Ler13, partie 2.4].

Leur spécification commune est l'existence d'une séquence infinie de réductions à partir de la configuration (c, s). Cette propriété est (diverges c s) avec la définition suivante

```
Definition diverges (c: cmd) (s: state) : Prop := infseq red (c, s).
```

où infseq est le prédicat co-inductif

```
CoInductive infseq: A → Prop :=
| infseq_step: ∀ a b, R a b → infseq b → infseq a.
```

des séquences infinies par la relation binaire R sur un type quelconque A [Ler13].

L'adaptation de la définition co-inductive d'execinf donne la définition co-inductive de biginf du listing 5.5.

```
CoInductive biginf: cmd → state → Prop :=
| biginf_seq_left: ∀ s c1 c2,
   biginf c1 s
   → biginf (Seq c1 c2) s
| biginf_seq_right: ∀ s c1 c2 s1,
   big c1 s s1 → biginf c2 s1
   → biginf (Seq c1 c2) s
| biginf_if: ∀ s (b : bexpr) c1 c2,
   biginf (if b s then c1 else c2) s
   → biginf (If b c1 c2) s
| biginf_while_body: ∀ s b c,
   b s = true → biginf c s
   → biginf (While b c) s
| biginf_while_loop: ∀ s b c s1,
   b s = true → big c s s1 → biginf (While b c) s1
   → biginf (While b c) s.
```

Listing 5.5 – Sémantique de divergence du langage CMD, en Coq.

L'équivalence entre la définition co-inductive de biginf et sa spécification

```
Lemma diverges_biginf_eq: ∀ c s, diverges c s ↔ biginf c s.
```

se démontre facilement à partir de [Ler13] et du lemme

```
Lemma biginf_execinf_eq: ∀ c s, biginf c s ↔ execinf s c.
```

d'échange de l'ordre des deux paramètres.

5.7 Sémantique axiomatique

Nous considérons à présent des programmes spécifiés par une condition sur leur entrée (*précondition*) et une condition sur leur sortie (*postcondition*). Nous désignons parfois par *programme pré-post(conditionné)* un programme ainsi annoté par une précondition et une postcondition.

La *sémantique axiomatique* considère que la correction partielle d'un programme pré-post est la sémantique de ce programme. Tony Hoare a introduit la notation [1] $\{P\}\ c\ \{Q\}$, souvent nommée "triplet de Hoare", pour la propriété de correction partielle de la commande c par rapport à la précondition P et à la postcondition Q [Hoa69]. Autrement dit, le triplet de Hoare $\{P\}\ c\ \{Q\}$ est valide si et seulement si, lorsque la précondition P est satisfaite, la commande c établit la postcondition Q, si cette commande termine. Nous formalisons cette correction partielle par la proposition Coq (triple P c Q).

Pour formaliser les formules manipulées et produites par les sémantiques statiques (préconditions et postconditions ici, assertions, invariants et conditions de vérification dans le chapitre 6), nous devons choisir entre l'*approche syntaxique*, qui consiste à introduire un langage de formules, et l'*approche sémantique*, qui les considère comme des prédicats, du type

```
Definition pred := state → Prop.
```

défini dans la partie 4.8.2. Nous retenons l'approche sémantique, car elle est plus légère que l'approche syntaxique et permet d'utiliser plus directement les tactiques de raisonnement sur les propositions de l'assistant de preuve Coq.

La partie 5.7.1 définit toutes les opérations requises pour construire ces formules. Nous spécifions ensuite la sémantique axiomatique dans la partie 5.7.2. Puis nous en donnons une définition inductive, appelée logique de Hoare (partie 5.7.3). Enfin, nous démontrons séparément la correction et la complétude de cette logique par rapport à sa spécification, dans les parties 5.7.4 et 5.7.5.

1. Plus exactement, la notation originelle proposée par Hoare était $P\ \{c\}\ Q$, mais elle a été rapidement remplacée par $\{P\}\ c\ \{Q\}$, où les accolades suggèrent que P et Q sont des commentaires de la commande c.

5.7.1 Opérations sur les prédicats

Le listing 5.6 définit des opérations qui permettent de construire les termes suivants :
— La mise à jour (pupd p u) d'un prédicat p par composition avec une transformation d'états u,
— les conversions (ptrue b) et (pfalse b) d'une expression booléenne b et de sa négation en prédicat,
— les prédicats (pand p q) et (por p q) de conjonction et de disjonction de deux prédicats p et q, et
— la proposition (pimp p q) d'implication entre deux prédicats p et q.

```
Definition pupd (p: pred) (u: state → state) := fun s ⇒ p (u s).

Definition ptrue  (b: bexpr) : pred := fun s ⇒ b s = true.
Definition pfalse (b: bexpr) : pred := fun s ⇒ b s = false.

Definition pand (p q: pred) : pred := fun s ⇒ p s ∧ q s.
Definition por  (p q: pred) : pred := fun s ⇒ p s ∨ q s.

Definition pimp (p q: pred) : Prop := ∀ s, p s → q s.
```

Listing 5.6 – Opérations sur les prédicats.

Les définitions pupd, ptrue, pfalse, pand, por et pimp sont essentiellement des renommages des définitions aupdate, atrue, afalse, aand, aor et aimp de [Ler13], où la lettre a est l'initiale du type assertion défini comme pred.

Dans la perspective d'étendre cette étude avec un langage de formules (approche syntaxique des prédicats), il est possible de définir l'implication sur les prédicats, par

```
Definition pimp (p q: pred) : pred := fun s ⇒ p s → q s.
```

et de remplacer la définition actuelle de (pimp p q) par prop_of_pred (pimp p q), avec

```
Definition prop_of_pred (p : pred) := ∀ s, p s.
```

5.7.2 Spécification de la sémantique axiomatique

La fonction Coq

```
Definition must (r : state → state → Prop) (Q : pred) : pred := fun x ⇒ (∀ y, r x y → Q y).
```

associe à toute relation binaire r entre états (appelée *relation de transition* dans un système de transitions) et tout prédicat Q le prédicat de ses *précédesseurs nécessaires*, qui caractérise les états à partir desquels toute transition selon r mène dans un état qui satisfait le prédicat Q.

Avec cette fonction, nous spécifions la sémantique axiomatique par rapport au niveau 2 de la sémantique naturelle, par la définition

```
Definition triple_big P c Q := ∀ s, P s → must (big c) Q s.
```

En effet, (big c) est la relation de transition établie par la commande c entre ses états d'entrée et de sortie. En particulier, un état à partir duquel c ne termine pas n'a pas d'image par cette relation. Ainsi, la proposition (triple_big P c Q) spécifie bien que l'exécution de c à partir d'un état qui satisfait P soit ne termine pas, soit termine dans un état qui satisfait Q.

5.7.3 Logique de Hoare

La logique de Hoare axiomatise la correction partielle d'un programme par rapport à une précondition et une postcondition. Cette logique proposée par Hoare [Hoa69] est souvent appelée "logique de Floyd-Hoare" car elle doit beaucoup au travail antérieur de Floyd [Flo67].

Les triplets de Hoare valides sont définis par le système d'inférence de la figure 5.4, qui ne diffère de celui de [Ler13, partie 3] que pour la règle [triple_update]. Ce système d'inférence est formalisé en Coq par le prédicat inductif triple tel que (triple P c Q) est vrai si et seulement si $\{P\}$ c $\{Q\}$ est dérivable dans la logique de Hoare. Sa définition est reproduite dans le listing 5.7.

$$\{P\}\ \mathsf{Skip}\ \{P\}\ \ [\mathsf{triple\_skip}] \qquad \{u\ P\}\ \mathsf{Update}\ u\ \{P\}\ \ [\mathsf{triple\_update}]$$

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ \mathsf{Seq}\ c_1\ c_2\ \{R\}}\ [\mathsf{triple\_seq}]$$

$$\frac{\{b \wedge P\}\ c_1\ \{Q\} \qquad \{\neg b \wedge P\}\ c_2\ \{Q\}}{\{P\}\ \mathsf{If}\ b\ c_1\ c_2\ \{Q\}}\ [\mathsf{triple\_if}]$$

$$\frac{\{b \wedge P\}\ c\ \{P\}}{\{P\}\ \mathsf{While}\ b\ c\ \{\neg b \wedge P\}}\ [\mathsf{triple\_while}]$$

$$\frac{P \to P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \to Q}{\{P\}\ c\ \{Q\}}\ [\mathsf{triple\_consequence}]$$

FIGURE 5.4 – Logique de Hoare.

Pour alléger la figure 5.4, les notations $u\ P$, b, $\neg b$, $\wedge$ et $\to$ remplacent – abusivement mais sans risque de confusion – les termes et opérations (pupd P u), (ptrue b), (pfalse b), pand et pimp présentés dans la partie 5.7.1.

Le prédicat P dans la règle [triple_while] est un *invariant inductif* de la boucle (While b c) : Pour qu'il soit préservé par la boucle, la prémisse de cette règle exige qu'il soit préservé par le corps de boucle c lorsque la condition de boucle b est vraie. La postcondition de la conclusion $\{P\}$ While b c $\{\neg b \wedge P\}$ de la règle contient aussi $\neg b$ car cette condition de boucle b devient fausse lorsque la boucle termine.

```
Inductive triple: pred → cmd → pred → Prop :=
| triple_skip: ∀ P,
   triple P Skip P
| triple_update: ∀ P u,
   triple (pupd P u) (Update u) P
| triple_seq: ∀ c1 c2 P Q R,
   triple P c1 Q → triple Q c2 R
   → triple P (Seq c1 c2) R
| triple_if: ∀ b c1 c2 P Q,
   triple (pand (ptrue b) P) c1 Q → triple (pand (pfalse b) P) c2 Q
   → triple P (If b c1 c2) Q
| triple_while: ∀ b c P,
   triple (pand (ptrue b) P) c P
   → triple P (While b c) (pand (pfalse b) P)
| triple_consequence: ∀ c P Q P' Q',
   pimp P P' → triple P' c Q' → pimp Q' Q
   → triple P c Q.
```

Listing 5.7 – Logique de Hoare en Coq.

5.7.4 Correction de la logique de Hoare

Le théorème

```
Theorem triple_sound: ∀ P c Q, triple P c Q → triple_big P c Q.
```

stipule que la logique de Hoare est *correcte* par rapport à la spécification triple_big de la correction partielle fondée sur la sémantique naturelle big. Sa démonstration est une adaptation au langage CMD et à l'assistant de preuve Coq de celle du lemme 12.2 en Isabelle de la monographie "Concrete semantics" de Nipkow et Klein [NK14, page 204].

En 1998 Nipkow a démontré en Isabelle la correction de la logique de Hoare par rapport à la "*relational semantics*", qui est une version ensembliste de la sémantique naturelle [Nip98]. Pour la commande While cette sémantique relationelle utilise une fonction d'ordre supérieur qui calcule le plus petit point fixe d'une fonction monotone sur des ensembles. C'est plus complexe que notre définition de la sémantique naturelle.

En 2009 Bertot démontre cette correction en Coq [Ber09, partie 4.3]. Cette démonstration passe par une réduction du non-déterminisme dans la logique de Hoare, qui impose une application unique de la règle triple_consequence lors de chaque application d'une autre règle de cette logique. Notre démonstration n'a pas besoin de cet intermédiaire. Nous reviendrons dans la partie 6.10 sur la réduction du non-déterminisme dans la logique de Hoare.

En 2013 cette démonstration n'est pas reprise par Leroy, qui préfère démontrer formellement en Coq la correction de la sémantique axiomatique triple (niveau 3) par rapport à sa spécification avec la sémantique de réduction red à petites étapes (niveau 1) [Ler13, partie 3.2]. Cette spécification utilise un prédicat co-inductif nommé finally.

5.7.5 Complétude de la logique de Hoare

La logique de Hoare est *complète* si elle permet de dériver tout triplet de Hoare $\{P\}\ c\ \{Q\}$ valide. Comme pour la correction, nous utilisons la spécification triple_big de cette validité à l'aide de la sémantique naturelle big.

Le théorème de complétude

```
Theorem triple_complete: ∀ c (P Q: pred), triple_big P c Q → triple P c Q.
```

se démontre facilement par induction sur la commande c.

5.7.6 Logique de Hoare et sémantique axiomatique

Il y a une nuance de sens entre les expressions "logique de Hoare" et "sémantique axiomatique", bien expliquée par les citations suivantes :

> "Floyd-Hoare logic [...] is [...] the basis for axiomatic semantics, in which the meaning of a programming language is specified by requiring that all programs written in it satisfy the rules and axioms of a formal logic." [Gor88, Preface].
>
> "There are two approaches to ensure this :
>
> (i) Define the language by the axioms and rules of the logic.
>
> (ii) Prove that the logic fits the language.
>
> Approach (i) is called *axiomatic semantics*. The idea is to *define* the semantics of the language by requiring that it make the axioms and rules of inference true. It is then up to implementers to ensure that the logic matches the language. One snag with this approach is that most existing languages have already been defined in some other way. [...] Approach (ii) requires that the axioms and rules of the logic be proved valid." [Gor88, Chapitre 2].

La logique de Hoare est une spécification de la correction partielle d'une commande par rapport à une précondition et une postcondition. On l'appelle *sémantique axiomatique* quand on l'adopte sans vérification comme sémantique du langage de commandes (approche (i) de Gordon). Nous préférons l'approche (ii), qu'on peut qualifier de *sceptique*, qui vérifie la correction de la logique de Hoare par rapport à d'autres sémantiques formelles du langage de programmation. Ceci dit, nous ignorons dorénavant cette nuance de sens entre logique de Hoare et sémantique axiomatique et utiliserons indifféremment l'une de ces expressions pour l'autre.

5.8 Notes bibliographiques

Cette partie présente des contributions majeures du domaine de la sémantique des langages impératifs. La partie 5.8.1 est une tentative de structuration de ces contributions en quatre grandes périodes. Chaque partie suivante commente une contribution particulière, dans l'ordre chronologique.

5.8.1 Quatre grandes périodes

Pour faciliter le repérage dans la littérature du domaine, qui est particulièrement riche, je propose d'y distinguer quatre grandes périodes, décrites et datées approximativement dans cette partie. Ces dates sont imprécises par nature, et quelque peu arbitraires, car elles correspondent aux documents les plus anciens et les plus récents que j'ai étudiés sur le sujet indiqué. Elles n'ont pas d'autre objectif que de fixer des repères sur les dépendances temporelles entre travaux et d'illustrer le rythme de progression de la recherche.

Les précurseurs (1967-76)

De manière approximative, on peut considérer que les pères fondateurs sont Floyd, Hoare et Dijkstra [Flo67, Hoa69, Dij75, Dij76]. Leurs contributions essentielles sont détaillées dans les parties 5.8.2 à 5.8.4.

Monographies de sémantique formelle (1978-93)

Dès 1967, Floyd a relié la correction des programmes avec la question de la définition rigoureuse de leur sémantique. Cette idée essentielle a été affinée ensuite. La sémantique formelle des langages de programmation est ensuite devenue le sujet d'ouvrages de synthèse conséquents [Coo78, Bak80, Win93].

Théorie des programmes (1988-98)

Des monographies moins générales, présentant plus précisément la théorie des programmes, de leur correction et de leur vérification, sont apparues peu de temps après [TZ88, Fra92, Rey98].

Formalisation et preuve sur machine (1998-)

Peu de temps après l'avènement de la sémantique formelle sur papier, les premières formalisations de sémantiques et preuves formelles de leurs propriétés avec un assistant de preuve sont apparues [Nip98, Ber09, Her13, Ler13, MT13, NK14, PAC+17]. Certains de ces travaux sont présentés ici. Les autres sont présentés dans la partie 6.14.

5.8.2 Floyd (1967)

L'article "*Assigning meanings to programs*" [Flo67] publié par R. W. Floyd en 1967 est une contribution incontournable. Nous résumons ses apports en citant quelques fragments.

L'article [Flo67] propose "a rigorous standard [...] for proofs [...] of correctness, equivalence and termination.". L'introduction définit des principes de vérification d'un programme formalisé par un *flowchart* (un graphe orienté dont certains sommets sont des commandes). Les deux parties suivantes considèrent deux langages de programmation plus particuliers : La deuxième partie considère un langage de *flowchart* limité à cinq "*statement types*", "*having the usual interpretations as an assignment operation, a conditional branch, a join of control, a starting point for the program, and a halt for the program.*". La troisième partie considère un fragment du langage ALGOL. La quatrième et dernière partie traite le problème de la terminaison.

Floyd associe une proposition à chaque transition (arc) d'un *flowchart* et impose une condition pour chacune de ses commandes (instructions). Floyd parle ensuite de

> "semantics of a programming language [...] defined [...] by establishing standards [...] for proofs about programs in the language, [that] appear to be novel" [Flo67, pages 19-20].

Cette sémantique construit une condition de vérification par commande.

L'extrait suivant d'un témoignage de Hoare sur cette époque reconnait l'importance de l'apport de Floyd :

> "By 1968, [...] I [moved] to the Queen's University in Belfast as professor of computer science. During this time, I came across a preprint of Robert Floyd's paper, "Assigning Meanings to Programs." Floyd adopted the same philosophy as I had, that the meaning of a programming language is defined by the rules that can be used for reasoning about programs in the language. These could include not only equations but also rules of inference. In his paper, Floyd presented an effective method of proving the total correctness of programs, not just their equality to other programs [...]" [Hoa02].

5.8.3 Hoare (1969)

En 1969, C. A. R. Hoare présente une définition axiomatique de la propriété de correction partielle d'un programme spécifié par une précondition et une postcondition [Hoa69]. Ce système déductif est depuis appelé "logique de Hoare" ou "sémantique axiomatique". Sa présentation actuelle (voir partie 5.7) est très proche de sa première présentation dans cet article fondateur de la théorie des programmes.

Hoare formalise complètement et simplement le lien entre correction et sémantique pressenti par Floyd :

> "the ultimate goal of a good formal semantics for a good programming language [is] the complete avoidance of programming error." [Hoa02].

5.8.4 Dijkstra (1975 et 1976)

Floyd a défini la correction de programmes dont chaque commande est précédée et suivie par une propriété d'état [Flo67]. Cependant, il indique aussi que certaines de ces propriétés peuvent être calculées à partir d'autres. Dans ce but, il introduit la notion de "*strongest verifiable consequent*" $T_c(P)$ d'une commande c, étant donnée une propriété P vraie avant son exécution.

Dijkstra prend le point de vue inverse, en considérant la *plus faible précondition* (*weakest precondition* en anglais) d'une commande c, étant donnée une propriété Q vraie **après** son exécution [Dij75] :

> "The way in which we use predicates (as a tool for defining sets of initial or final states) for the definition of the semantics of programming language constructs has been directly inspired by Hoare [Hoa69], the main difference being that we have tightened things up a bit : while Hoare introduces sufficient pre-conditions such that the mechanisms will not produce the wrong result (but may fail to terminate), we shall introduce necessary and sufficient–i.e, so-called "weakest"–pre-conditions such that the mechanisms are guaranteed to produce the right result." [Dij75, partie 3.1].

Le calcul de cette propriété se révèle être plus simple que celui de plus forte postcondition de Floyd. Comme détaillé dans la partie 6.11, ce calcul de plus faible précondition va aussi permettre de remplacer les démonstrations interactives en logique de Hoare par des démonstrations automatisées (sous certaines conditions précisées plus loin).

L'article de Dijkstra [Dij75] est très rapidement suivi par une monographie essentielle [Dij76], décrite par Hoare en ces termes :

> "the starting point for a software project should always be the specification, and the program proof should be developed along with the program. [...] This philosophy has been beautifully illustrated in Dijkstra's *A Discipline of Programming*" [Hoa02, page 18].

5.8.5 Winskel (1993)

L'ouvrage "*The Formal Semantics of Programming Languages*" de Glynn Winskel [Win93] est un manuel universitaire sur la sémantique formelle des langages de programmation. Il a servi de référence pour la formalisation de Nipkow [Nip98].

5.8.6 Nipkow (1998)

L'article de Tobias Nipkow intitulé "Winskel is (almost) right : Towards a mechanized semantics" [Nip98] est reconnu comme un effort majeur de formalisation machine des sémantiques des programmes :

> "The most impressive experiment is described in [Nip98], who approximately formalizes the first 100 pages of Winskel's book" [Ber09, Page 8].

Nipkow simplifie la formalisation en écartant le sujet annexe d'une syntaxe pour les expressions :

> "Winskel also treats syntax and semantics of arithmetic and boolean expressions [...]. Because expressions add nothing new, we have taken a semantic view, i.e. we have identified expressions with their semantics. The central semantic concept is that of a state, i.e. a mapping from locations to values." [Nip98, page 173].

Nous avons poussé l'abstraction un cran plus loin, en remplaçant l'affectation concrète des variables par une mise à jour abstraite des états.

Pour les assertions-prédicats, nous avons suivi l'approche sémantique de Nipkow :

> "we diverge most significantly from Winskel's treatment : [...] we take a short cut by not formalizing the syntax of the assertion language [...] A complete formalization of syntax and semantics of assertions [...] is a project in its own right. Therefore we have taken the semantic way out :
>
> **types** assn = state ⇒ bool" [Nip98, page 177].

5.8.7 Projet CompCert (2005-)

Le projet CompCert (`http://compcert.inria.fr`), dont le principal architecte et développeur est Xavier Leroy, a relevé le défi de la création d'un compilateur certifié, lancé par Hoare [Hoa03] peu de temps auparavant. Le compilateur C du projet [LB08, Ler09a, BL09, Ler09b, BJLM15] est correct par construction, car il est obtenu par extraction, à partir d'une spécification Coq qui certifie la préservation de la sémantique, entre le code source et le code compilé pour plusieurs processeurs. Ce compilateur couvre la quasi-totalité du langage C selon la norme ISO/IEC9899 :1999.

5.8.8 Appel et Blazy (2007)

Pour le principal langage intermédiaire Cminor du compilateur C du projet CompCert, A. Appel et S. Blazy ont formalisé en Coq une sémantique à petits pas, avec des continuations, et une sémantique axiomatique (sous forme de logique de séparation, pour prendre en compte les pointeurs) [AB07]. Ils ont démontré formellement la correction de cette sémantique axiomatique par rapport à cette sémantique à petits pas. À ma connaissance c'est la première preuve de correction de ce genre.

5.8.9 Blazy (2008)

Dans son mémoire d'HDR sur les sémantiques formelles, Sandrine Blazy vérifie formellement, avec l'assistant de preuve Coq, que des transformations de programmes préservent la sémantique de ces programmes [Bla08]. Ces travaux contribuent au projet CompCert de vérification formelle d'un compilateur du langage C.

5.8.10 Bertot (2009)

Dans un hommage à Gilles Kahn, père de la sémantique naturelle, Yves Bertot [Ber09] adapte à Coq la formalisation Isabelle de Nipkow, et la complète avec un outil d'analyse basé sur l'interprétation abstraite. Il démontre la correction de la sémantique axiomatique par rapport à la sémantique naturelle, alors que Nipkow [Nip98] l'avait démontrée par rapport à la sémantique dénotationnelle.

5.8.11 Leroy (2013)

Lors de l'école d'été VSTA (*Verification Technology, Systems & Applications*) en 2013, Xavier Leroy présente une riche synthèse des formalisations Coq précédentes des sémantiques de langages impératifs [Nip98, Ber09], et leurs applications, notamment à la preuve de programmes [Ler13]. Il ajoute ses propres contributions, en particulier une définition co-inductive de la divergence [LG09], adaptée dans la partie 5.6 sous le nom de "sémantique de divergence".

Cette contribution étant particulièrement complète et claire, nous l'avons choisie comme point de départ pour nos travaux.

5.8.12 Nipkow et Klein (2014)

La partie II de la monographie "Concrete Semantics" de Tobias Nipkow et Gerwin Klein [NK14] formalise la sémantique des programmes impératifs avec l'assistant de preuve Isabelle. Le chapitre 7 formalise la sémantique naturelle (*Big-Step Semantics*) et la sémantique de réduction (*Small-Step Semantics*). Le chapitre 12 est consacré à la logique de Hoare. Les démonstrations d'équivalence sont similaires à celles de ce chapitre.

5.9 Conclusion

Nous avons défini trois sémantiques d'un langage impératif Turing-complet : à petites étapes, à grandes étapes et axiomatique. Puis nous avons démontré formellement leur équivalence pour définir la correction partielle d'un programme impératif par rapport à une précondition et une postcondition.

Ce chapitre est principalement une compilation de résultats classiques de sémantique formelle, mais quelques originalités sont introduites. Pour simplifier, les états sont abstraits, l'instruction d'affectation concrète est remplacée par une mise à jour abstraite (transformation d'état quelconque), les expressions booléennes et les formules sont remplacées par leur sémantique de fonctions sur les états. Les trois sémantiques sont structurées en niveaux. La correction et la complétude des sémantiques de niveau 2 et 3 est démontrée par rapport à la sémantique du niveau précédent. Dans un souci de rigueur, toutes les démonstrations sont effectuées avec l'assistant de preuve formelle Coq.

Ce travail prépare sa généralisation dans le chapitre suivant : Ici, les propriétés vérifiées sont des postconditions, qui sont des assertions en fin de programme. Dans le chapitre 6 le sujet est élargi à des assertions placées à n'importe quel endroit dans le programme.

Chapitre 6

Sémantiques bloquante et inductive des programmes impératifs annotés

Sommaire

L'objectif principal de ce chapitre est de définir formellement la correction partielle de programmes impératifs annotés avec des spécifications fonctionnelles formelles. Parmi les diverses sortes d'annotations des langages de spécification comme JML ou ACSL, présentés dans le chapitre 4, nous avons sélectionné les assertions et les invariants de boucle explicites, pour leur rôle central dans la vérification déductive.

Les assertions sont des conditions sur l'état du programme qui peuvent être placées entre ses instructions. Elles généralisent les postconditions étudiées dans le chapitre 5, qui ne peuvent être placées qu'en fin de programme. Les assertions facilitent la vérification automatique, comme nous l'avons montré sur un exemple dans la partie 4.4.

Les origines historiques de la notion d'assertion en informatique sont brièvement retracées dans la partie 6.14.1 de notes bibliographiques. Nous réduisons ici la notion d'assertion à une commande d'assertion (Aassert a) et au prédicat a inclus dans cette commande, qui décrit une propriété d'état qui doit être vraie lorsque l'exécution du programme atteint cette commande. Nous étudions la question de l'exécution de ces nouvelles commandes, mais aussi leur impact sur la preuve des programmes qu'elles annotent.

Les invariants de boucle explicites sont nécessaires pour automatiser la vérification déductive, par la méthode dite de "génération de conditions de vérification" formalisée dans la partie 6.11.

Avec des invariants de boucle, nous verrons qu'on doit distinguer deux sémantiques des programmes annotés, non équivalentes, selon qu'on s'intéresse à l'exécution des annotations ou à leur vérification déductive. La sémantique d'exécution, dynamique, est dite *bloquante*. La sémantique statique est appelée *sémantique inductive*, car elle impose que chaque annotation de boucle soit un invariant inductif. Il y a ainsi deux notions de correction (partielle) des programmes annotés, l'une dynamique et l'autre statique.

Une sémantique d'exécution de programmes annotés est dite *bloquante* si l'exécution s'arrête dès qu'une annotation (assertion ou invariant de boucle) rencontrée est fausse. La méthode de vérification correspondante est appelée "validation à l'exécution" (*runtime assertion checking* en anglais) [CR06].

6.1 Contexte et motivations

Comme détaillé dans la partie 5.1, j'ai étudié des sémantiques de programmes impératifs annotés depuis 2004, dans le cadre des thèses de Julien Groslambert [Gro07], d'Omar Chebaro [Che11] et de Guillaume Petiot [Pet15]. Il s'agissait de programmes Java annotés en JML dans le premier cas, et de programmes C annotés en ACSL dans les deuxième et troisième cas. Il s'agissait de sémantique de trace dans les deux premiers cas, et de sémantique opérationnelle dans le troisième cas.

Dans [CKGJ10b, CKGJ11, CKGJ12] nous avons montré comment réduire à des annotations formelles les causes d'erreur à l'exécution (comme la division par zéro ou l'accès dans un tableau hors de ses limites). Nous considérons donc ici des programmes annotés, dont les annotations sont les seules causes d'erreur lors de l'exécution de ces programmes. Ces annotations sont intégrées dans la syntaxe d'un langage de programmation.

Nous avons aussi montré l'intérêt d'exécuter ces annotations, ce qui a conduit à la notion de sémantique bloquante, sous-jacente dans notre article sur la vérification de propriétés de vivacité pour les programmes Java [GGJK08] et approfondie par Paolo Herms dans sa thèse [HMM12] en 2012. L'intérêt de cette sémantique a été reconnu très rapidement :

> "The correctness of our algorithm (see Theorem 1) requires a blocking semantics which is usual in semantics of annotated programs (see for instance [GGJK08, HMM12])." [CS12].

Or, les langages étudiés dans ces travaux antérieurs étaient trop complexes pour donner une vision simple et claire de la sémantique bloquante et de ses relations avec la preuve de programmes. Dès lors, ma motivation principale a été de définir rigoureusement la sémantique bloquante et la vérification déductive de programmes annotés. Ce chapitre présente l'état actuel de ce travail de formalisation, pour un langage impératif annoté réduit.

6.2 Plan du chapitre

La partie 6.3 définit la syntaxe d'un langage de programmes impératifs annotés, nommé ACMD. Les parties 6.4 à 6.9 étendent au langage ACMD les sémantiques de réduction, naturelle, et axiomatique définies dans le chapitre 5 pour le langage CMD. La partie 6.10 introduit une sémantique intermédiaire pour faciliter certains raisonnements sur la génération de conditions de vérification pour les prouveurs de théorèmes, définie dans la partie 6.11.

La figure 6.1 présente ces sémantiques par niveau croissant. Elle indique la partie où elle est définie, le nom de la sémantique, la nature de sa définition (parmi spécification, définition inductive et définition co-inductive), son nom en Coq et son type Coq. Les lettres a, s, f, p et P abrègent respectivement les types acmd, state, flag, pred et Prop. Le nouveau type flag est assimilable au type booléen à deux valeurs. Il est détaillé dans la partie 6.4.

La séparation entre les sémantiques bloquantes (dynamiques) et inductives (statiques) est matérialisée par un double trait horizontal. Les prédicats inductifs et co-inductifs small, clean,

Niveau, partie	Sémantique	Nature	nom Coq	type Coq
1, 6.4	réduction	Inductive	small	a * s * f → a * s * f → P
2, 6.5	terminaison propre	Spécif. Inductive	terminates_cleanly clean	a → s → s → P
2', 6.6	erreur	Spécif. Inductive	goes_wrong err	a → s → P
2", 6.8	divergence	Spécif. 1 Spécif. 2 Co-inductive	adiverges adiverges_fine div	a → s → P
3, 6.9	inductive (axiomatique)	Spécif. Inductive	ax_term ax	p → a → p → P
4, 6.10	commande	Inductive	acommand	p → a → p → P
5, 6.11	conditions de vérification	Fonctions	wp cc vcgen	a → p → p a → p → P p → a → p → P

FIGURE 6.1 – Sémantiques des programmes impératifs annotés.

err et div participent à des définitions équivalentes de la sémantique bloquante. Mais on veut aussi établir la correction de tout programme annoté sans l'exécuter, statiquement. Les prédicats inductifs ax et acommand, ainsi que la fonction vcgen (définie à l'aide des fonctions wp et cc) sont des définitions équivalentes de la sémantique inductive.

Les niveaux 2* et 3 introduisent une sémantique d'abord spécifiée par rapport au niveau précédent, puis définie inductivement ou co-inductivement. L'équivalence entre définition et spécification(s) est démontrée. Le niveau 3 généralise la logique de Hoare aux programmes annotés. Les niveaux 4 et 5 définissent des sémantiques de même type que le niveau 3, mais en réduisant (niveau 4) et en éliminant totalement (niveau 5) le non-déterminisme des règles du niveau 3.

La partie 6.12 démontre la correction de la sémantique inductive par rapport à la sémantique bloquante. La partie 6.13 explique pourquoi les sémantiques dynamiques et statiques de programmes annotés par des invariants de boucle explicites ne peuvent pas être équivalentes. La partie 6.14 présente quelques travaux antérieurs sur les assertions et les sémantiques formelles de programmes annotés. Enfin, la partie 6.15 résume ce chapitre et présente quelques perspectives.

6.3 Syntaxe du langage des programmes annotés

Nous considérons un langage impératif annoté, nommé ACMD ('A' pour A*nnotated*), qui ajoute des annotations formelles au langage CMD du chapitre 5. Ce langage n'a pas vocation à devenir un langage de programmation. Il sert uniquement à définir formellement diverses sémantiques et la correction partielle de certains programmes annotés.

La syntaxe des commandes du langage ACMD est définie par le type inductif acmd suivant :

```
Inductive acmd :=                          (* Commands:                  *)
| Askip : acmd                             (* - do-nothing command       *)
| Aupdate : (state → state) → acmd         (* - state transformer        *)
| Aseq : acmd → acmd → acmd                (* - sequential composition   *)
| Aif : bexpr → acmd → acmd → acmd         (* - conditional command      *)
| Awhile : bexpr → pred → acmd → acmd      (* - repetition with invariant *)
| Aassert : pred → acmd.                   (* - assertion                *)
```

Ce langage ajoute au langage CMD du chapitre 5 la commande d'assertion (Aassert a) et ajoute un invariant i dans la commande de boucle (While b c), pour former la commande de *boucle avec invariant explicite* (Awhile b i c).

Ce langage ne diffère du langage acmd de [Ler13] que sur deux points :

1. L'affectation explicite Aassign de [Ler13] est remplacée par la mise à jour déterministe Aupdate, comme pour CMD dans le chapitre 5.
2. Le type (state → Prop) des assertions et invariants est nommé pred au lieu d'assertion.

Dans toute la suite, nous désignons par *annotation* d'un programme du langage ACMD tout prédicat d'une commande d'assertion et tout invariant explicite de ce programme.

6.4 Sémantique de réduction bloquante

Nous définissons ici une sémantique "à petites étapes" des programmes annotés, appelée *sémantique de réduction* et notée small en Coq, qui est bloquante en ce sens qu'elle décrit une exécution qui s'arrête dès qu'une annotation rencontrée est fausse.

$$\mathsf{Aupdate}\ u, s, \mathsf{fine} \to \mathsf{Askip}, u\ s, \mathsf{fine}\ \text{[small\_update]}$$

$$\frac{c_1, s, \mathsf{fine} \to c_1', s', f}{\mathsf{Aseq}\ c_1\ c_2, s, \mathsf{fine} \to \mathsf{Aseq}\ c_1'\ c_2, s', f}\ \text{[small\_seq\_left]}$$

$$\mathsf{Aseq}\ \mathsf{Askip}\ c, s, \mathsf{fine} \to c, s, \mathsf{fine}\ \text{[small\_seq\_skip]}$$

$$\frac{b\ s = \mathsf{true}}{\mathsf{Aif}\ b\ c_1\ c_2, s, \mathsf{fine} \to c_1, s, \mathsf{fine}}\ \text{[small\_if\_true]} \qquad \frac{b\ s = \mathsf{false}}{\mathsf{Aif}\ b\ c_1\ c_2, s, \mathsf{fine} \to c_2, s, \mathsf{fine}}\ \text{[small\_if\_false]}$$

$$\frac{\neg\ (i\ s)}{\mathsf{Awhile}\ b\ i\ c, s, \mathsf{fine} \to \mathsf{Awhile}\ b\ i\ c, s, \mathsf{abort}}\ \text{[small\_while\_inv\_false]}$$

$$\frac{i\ s \qquad b\ s = \mathsf{true}}{\mathsf{Awhile}\ b\ i\ c, s, \mathsf{fine} \to \mathsf{Aseq}\ c\ (\mathsf{Awhile}\ b\ i\ c), s, \mathsf{fine}}\ \text{[small\_while\_true]}$$

$$\frac{i\ s \qquad b\ s = \mathsf{false}}{\mathsf{Awhile}\ b\ i\ c, s, \mathsf{fine} \to \mathsf{Askip}, s, \mathsf{fine}}\ \text{[small\_while\_false]}$$

$$\frac{\neg\ (a\ s)}{\mathsf{Aassert}\ a, s, \mathsf{fine} \to \mathsf{Aassert}\ a, s, \mathsf{abort}}\ \text{[small\_assert\_false]}$$

$$\frac{a\ s}{\mathsf{Aassert}\ a, s, \mathsf{fine} \to \mathsf{Askip}\ a, s, \mathsf{fine}}\ \text{[small\_assert\_true]}$$

FIGURE 6.2 – Sémantique de réduction des programmes annotés.

L'évaluation d'une annotation fausse doit (a) mettre fin au programme, et (b) cet arrêt sur erreur doit être distinguable de la terminaison normale du programme. Dans le premier cas, on dira que le programme *échoue* ou *bloque*, dans le second cas on dira qu'il *termine* (normalement, c'est-à-dire sans erreur). Pour bien expliciter la condition (a) nous disons que cette sémantique est *bloquante*.

Une alternative à la condition (a) serait que le programme poursuive son exécution après évaluation d'une annotation fausse. Si l'exécution se poursuit après une assertion fausse, avec la commande séquentiellement suivante, la notion d'assertion se réduit à celle de commande conditionnelle, modifiant une partie de l'état du programme pour indiquer qu'elle n'est pas respectée. Si l'exécution normale est remplacée par une exécution exceptionnelle associée à l'assertion, alors la commande d'assertion se comporte comme une levée d'exception. Nous excluons ces deux sémantiques, qui correspondent à d'autres commandes.

Pour distinguer l'arrêt sur erreur de la terminaison normale (condition (b)), nous introduisons un état d'erreur, bloquant, en ce sens qu'aucune commande ne peut être exécutée à partir de cet état.

Nous complétons le couple (commande, état) avec un "drapeau" booléen, qui indique si l'état est normal ou d'erreur. Pour plus de lisibilité, nous ne formalisons pas ce drapeau avec le type bool des booléens Coq, mais nous introduisons un type énuméré

```
Inductive flag := fine | abort.
```

exclusivement dédié à cet usage. La constante fine identifie les états normaux. La constante abort signale un état d'erreur. Nous appelons *configuration* tout triplet (commande, état, drapeau).

La sémantique de réduction bloquante est une relation binaire sur le produit cartésien acmd $*$ state $*$ flag. Elle est définie par le système d'inférence de la figure 6.2, qui étend la sémantique de réduction red du langage CMD au drapeau fine/abort et considère tous les cas d'erreur.

Le cas particulier de la règle [small_seq_left] lorsque f est le drapeau d'erreur abort correspond à une erreur survenue lors de l'exécution de la première commande c_1 d'une séquence de commandes (Aseq c_1 c_2).

Une erreur peut aussi survenir lorsqu'un invariant de boucle i n'est pas satisfait, comme décrit par la règle [small_while_inv_false], ou lorsque le prédicat a d'une commande d'assertion est faux, comme décrit par la règle [small_assert_false].

Ce système est formalisé par le prédicat inductif Coq du listing 6.1.

```
Inductive small: acmd * state * flag → acmd * state * flag → Prop :=
| small_update: ∀ u s,
  small (Aupdate u, s, fine) (Askip, u s, fine)
| small_seq_left: ∀ c1 c2 s c1' s' f,
   small (c1, s, fine) (c1', s', f)
   → small (Aseq c1 c2, s, fine) (Aseq c1' c2, s', f)
| small_seq_skip: ∀ c s,
   small (Aseq Askip c, s, fine) (c, s, fine)
| small_if_true: ∀ s b c1 c2,
   b s = true
   → small (Aif b c1 c2, s, fine) (c1, s, fine)
| small_if_false: ∀ s b c1 c2,
   b s = false
   → small (Aif b c1 c2, s, fine) (c2, s, fine)
| small_while_inv_false: ∀ b (i: pred) c s,
   ~ (i s)
   →  small (Awhile b i c, s, fine) (Awhile b i c, s, abort)
| small_while_true: ∀ b (i: pred) c s,
   i s → b s = true
   → small (Awhile b i c, s, fine) (Aseq c (Awhile b i c), s, fine)
| small_while_false: ∀ b (i: pred) c s,
   i s → b s = false
   → small (Awhile b i c, s, fine) (Askip, s, fine)
| small_assert_false: ∀ a s,
   ~ (a s)
   → small (Aassert a, s, fine) (Aassert a, s, abort)
| small_assert_true: ∀ (a: pred) s,
   a s
   → small (Aassert a, s, fine) (Askip, s, fine).
```

Listing 6.1 – Sémantique de réduction des programmes annotés, en Coq.

Nous démontrons ensuite que la relation (small c) est déterministe pour toute commande annotée c, et divers lemmes non détaillés ici, qui caractérisent les configurations irréductibles comme étant de la forme (Askip, _, fine) ou (_, _, abort).

6.5 Sémantique de terminaison propre

On appelle *terminaison propre* (*clean termination* en anglais) la propriété de terminaison sans erreur des programmes annotés [Bli81].

Nous spécifions d'abord la terminaison propre à l'aide de la sémantique de réduction, puis nous définissons inductivement une *sémantique de terminaison propre* "à grandes étapes" (naturelle) des programmes annotés, limitée aux cas de terminaison sans erreur.

6.5.1 Spécification

Informellement, l'exécution de la commande c dans l'état s termine proprement si et seulement s'il existe une séquence finie de réductions entre la configuration (c, s, fine) et une configuration $(\mathsf{Askip}, s', \mathsf{fine})$ pour un certain état s'.

Le prédicat Coq

```
Definition terminates_cleanly (c: acmd) (s s': state) :=
 star small (c, s, fine) (Askip, s', fine).
```

permet de spécifier par (terminates_cleanly c) la terminaison propre de tout programme annoté c. Cette spécification de la terminaison sans erreur est fondée sur la sémantique de réduction (small). Comme détaillé dans la partie 5.5 le prédicat inductif star définit la fermeture réflexive-transitive de toute relation binaire.

6.5.2 Définition inductive

$$\mathsf{clean\ Askip}\ s\ s\ \ [\mathsf{clean\_skip}] \qquad \mathsf{clean}\ (\mathsf{Aupdate}\ u)\ s\ (u\ s)\ \ [\mathsf{clean\_update}]$$

$$\frac{\mathsf{clean}\ c_1\ s\ s_1 \qquad \mathsf{clean}\ c_2\ s_1\ s'}{\mathsf{clean}\ (\mathsf{Aseq}\ c_1\ c_2)\ s\ s'}\ \ [\mathsf{clean\_seq}]$$

$$\frac{b\ s = \mathsf{true} \qquad \mathsf{clean}\ c_1\ s\ s'}{\mathsf{clean}\ (\mathsf{Aif}\ b\ c_1\ c_2)\ s\ s'}\ \ [\mathsf{clean\_if\_true}] \qquad \frac{b\ s = \mathsf{false} \qquad \mathsf{clean}\ c_2\ s\ s'}{\mathsf{clean}\ (\mathsf{Aif}\ b\ c_1\ c_2)\ s\ s'}\ \ [\mathsf{clean\_if\_false}]$$

$$\frac{i\ s \qquad b\ s = \mathsf{true} \qquad \mathsf{clean}\ c\ s\ s_1 \qquad \mathsf{clean}\ (\mathsf{Awhile}\ b\ i\ c)\ s_1\ s'}{\mathsf{clean}\ (\mathsf{Awhile}\ b\ i\ c)\ s\ s'}\ \ [\mathsf{clean\_while\_loop}]$$

$$\frac{i\ s \qquad b\ s = \mathsf{false}}{\mathsf{clean}\ (\mathsf{Awhile}\ b\ i\ c)\ s\ s}\ \ [\mathsf{clean\_while\_stop}]$$

$$\frac{a\ s}{\mathsf{clean}\ (\mathsf{Aassert}\ a)\ s\ s}\ \ [\mathsf{clean\_assert\_true}]$$

FIGURE 6.3 – Sémantique de terminaison propre.

La sémantique de terminaison propre est définie inductivement dans la figure 6.3 et dans le listing 6.2. Elle généralise aux programmes annotés la sémantique naturelle big de la partie 5.5.

```
Inductive clean: acmd → state → state → Prop :=
| clean_skip: ∀ s,
   clean Askip s s
| clean_update: ∀ u s,
   clean (Aupdate u) s (u s)
| clean_seq: ∀ s c1 c2 s1 s',
   clean c1 s s1 → clean c2 s1 s'
   → clean (Aseq c1 c2) s s'
| clean_if: ∀ s (b: bexpr) c1 c2 s',
   clean (if b s then c1 else c2) s s'
   → clean (Aif b c1 c2) s s'
| clean_while_loop: ∀ s b (i: pred) c s1 s',
   i s → b s = true → clean c s s1 → clean (Awhile b i c) s1 s'
   → clean (Awhile b i c) s s'
| clean_while_stop: ∀ s b (i: pred) c,
   i s → b s = false
   → clean (Awhile b i c) s s
| clean_assert_true: ∀ (a: pred) s,
   a s → clean (Aassert a) s s.
```

Listing 6.2 – Sémantique de terminaison propre en Coq.

6.5.3 Propriétés

Cette définition inductive et sa spécification sont équivalentes :

```
Theorem terminates_cleanly_clean_eq: ∀ c s s', terminates_cleanly c s s' ↔ clean c s s'.
```

Par ailleurs, pour toute commande annotée c, la relation binaire (clean c) est déterministe.

6.6 Sémantique d'erreur

La sémantique de terminaison propre ne décrit ni les cas d'erreur, ni les cas de non-terminaison des exécutions des programmes annotés. Ceci requiert un approfondissement traité dans cette partie et les deux suivantes.

Nous appelons *sémantique d'erreur* une sémantique à grandes étapes des programmes annotés qui décrit les cas de terminaison avec erreur.

La monographie "*Program Correctness over Abstract Data Types, with Error-state Semantics*" de Tucker et Zucker traite la sémantique de programmes avec des erreurs particulières, dues à des variables non initialisées :

> "programs with the semantical feature that using an uninitialized variable leads to an error message" [TZ88, Preface].
>
> "we redesign the semantics of our basic constructs in order to model *errors* which may arise in a computation from uninitialized variables and may cause the computation to halt in an error state. This error semantics is very well known in practice [...] However [it] receives its first mathematical analysis here." [TZ88, page 4].

Nous spécifions d'abord l'échec à l'aide de la sémantique de réduction, puis nous définissons la sémantique d'erreur par un prédicat inductif.

6.6.1 Spécification

Informellement, l'exécution de la commande c dans l'état s termine avec erreur, ou "échoue", si et seulement s'il existe une séquence finie de réductions entre la configuration (c, s, fine) et une configuration (c', s', abort) pour une certaine commande c' et un certain état s'.

Avec le prédicat Coq

```
Definition goes_wrong (c: acmd) (s: state) :=
 ∃ c' s', star small (c, s, fine) (c', s', abort).
```

la sémantique d'erreur de tout programme annoté c est spécifiée par rapport à sa sémantique de réduction par (goes_wrong c).

6.6.2 Définition inductive

La sémantique d'erreur est définie inductivement dans le listing 6.3.

```
Inductive err: acmd → state → Prop :=
| err_seq_left: ∀ s c1 c2,
   err c1 s
   → err (Aseq c1 c2) s
| err_seq_right: ∀ s c1 c2 s1,
   clean c1 s s1 → err c2 s1
   → err (Aseq c1 c2) s
| err_if: ∀ s (b: bexpr) c1 c2,
   err (if b s then c1 else c2) s
   → err (Aif b c1 c2) s
| err_while_inv: ∀ s b i c,
   ~ (i s)
   → err (Awhile b i c) s
| err_while_body: ∀ s b (i: pred) c,
   i s → b s = true → err c s
   → err (Awhile b i c) s
| err_while_loop: ∀ s b (i: pred) c s1,
   i s → b s = true → clean c s s1 → err (Awhile b i c) s1
   → err (Awhile b i c) s
```

```
| err_assert: ∀ (a: pred) s,
   ~ (a s)
   → err (Aassert a) s.
```

Listing 6.3 – Sémantique d'erreur en Coq.

6.6.3 Propriétés

Cette définition inductive et sa spécification sont équivalentes :

```
Theorem goes_wrong_err_eq: ∀ c s, goes_wrong c s ↔ err c s.
```

6.7 Sémantique de terminaison

On appelle *terminaison* (*termination* en anglais) la propriété qu'un programme annoté termine avec ou sans erreur. Nous spécifions d'abord cette propriété de terminaison à l'aide de la sémantique de réduction, puis nous définissons inductivement une *sémantique de terminaison* "à grandes étapes" (naturelle) des programmes annotés. Pour distinguer les cas de blocage, cette spécification et cette définition inductive ne portent pas seulement sur les états, mais sur les couples (état, drapeau), de type state × flag, appelés *états complets*.

6.7.1 Spécification

Informellement, l'exécution de la commande annotée c dans l'état complet (s, f) termine sans erreur si et seulement s'il existe une séquence finie de réductions entre la configuration (c, s, f) et une configuration $(\mathsf{Askip}, s', \mathsf{fine})$ pour un certain état s'. Elle termine avec erreur si et seulement s'il existe une séquence finie de réductions entre la configuration (c, s, f) et une configuration (c', s', abort) pour une certaine commande c' et un certain état s'.

Ces deux cas sont regroupés dans la définition suivante :

```
Definition aterminates (c: acmd) (k k': state * flag) :=
 (snd k' = fine ∧ star small (c, fst k, snd k) (Askip, fst k', fine)) ∨
 (snd k' = abort ∧ ∃ c', star small (c, fst k, snd k) (c', fst k', snd k')).
```

Ce prédicat Coq permet de spécifier par (aterminates c) la terminaison de tout programme annoté c.

6.7.2 Définition inductive

La sémantique de terminaison est définie inductivement, sur les états complets, dans le listing 6.4. Elle regroupe les cas de la sémantique clean de terminaison propre et de la sémantique d'erreur err.

```
Inductive term: acmd → state * flag → state * flag → Prop :=
| term_skip: ∀ s,
   term Askip (s, fine) (s, fine)
| term_update: ∀ u s, term (Aupdate u) (s, fine) (u s, fine)
| term_seq_left_error: ∀ s c1 c2 s1,
   term c1 (s, fine) (s1, abort)
   → term (Aseq c1 c2) (s, fine) (s1, abort)
| term_seq_left_fine: ∀ s c1 c2 s1 s2 f2,
   term c1 (s, fine) (s1, fine) → term c2 (s1, fine) (s2, f2)
   → term (Aseq c1 c2) (s, fine) (s2, f2)
| term_if: ∀ s (b: bexpr) c1 c2 s' f',
   term (if b s then c1 else c2) (s, fine) (s', f')
   → term (Aif b c1 c2) (s, fine) (s', f')
| term_while_inv_false: ∀ s b (i: pred) c,
   ~ (i s)
   → term (Awhile b i c) (s, fine) (s, abort)
| term_while_first_error: ∀ s b (i: pred) c s1,
   i s → b s = true → term c (s, fine) (s1, abort)
   → term (Awhile b i c) (s, fine) (s1, abort)
| term_while_loop: ∀ s b (i: pred) c s1 s2 f2,
   i s → b s = true → term c (s, fine) (s1, fine) →
```

```
    term (Awhile b i c) (s1, fine) (s2, f2)
  → term (Awhile b i c) (s, fine) (s2, f2)
| term_while_stop: ∀ s b (i: pred) c,
    i s → b s = false
  → term (Awhile b i c) (s, fine) (s, fine)
| term_assert_false: ∀ (a: pred) s,
    ~ (a s)
  → term (Aassert a) (s, fine) (s, abort)
| term_assert_true: ∀ (a: pred) s,
    a s
  → term (Aassert a) (s, fine) (s, fine)
| term_error: ∀ c s,
    term c (s, abort) (s, abort).
```

Listing 6.4 – Sémantique de terminaison en Coq.

Cette définition inductive et sa spécification sont équivalentes :

```
Theorem aterminates_term_eq: ∀ c k k', aterminates c k k' ↔ term c k k'.
```

Par ailleurs, pour toute commande annotée c, la relation binaire (term c) sur les états complets est déterministe.

6.8 Sémantique de divergence des programmes annotés

6.8.1 Spécifications

La *sémantique de divergence* de tout programme annoté c, pour tout état s, peut être spécifiée par l'existence d'une séquence infinie de réductions à partir de c et s, par l'une des deux définitions suivantes, qui sont équivalentes :

```
Definition adiverges (c: acmd) (s: state) := ∃ f, infseq small (c, s, f).
Definition adiverges_fine (c: acmd) (s: state) := infseq small (c, s, fine).
Lemma adiverges_fine_eq : ∀ c s, adiverges c s ↔ adiverges_fine c s.
```

Dans ces définitions infseq est le prédicat co-inductif

```
CoInductive infseq: A → Prop :=
| infseq_step: ∀ a b, R a b → infseq b → infseq a.
```

des séquences infinies par la relation binaire R sur A [Ler13].

6.8.2 Définition co-inductive équivalente

Le listing 6.5 définit co-inductivement la sémantique de divergence, par le prédicat div qui adapte simplement au langage ACMD le prédicat co-inductif biginf de la sémantique de divergence du langage CMD, en remplaçant la sémantique naturelle big par la sémantique de terminaison propre clean.

```
CoInductive div: acmd → state → Prop :=
| div_seq_left: ∀ s c1 c2,
    div c1 s
  → div (Aseq c1 c2) s
| div_seq_right: ∀ s c1 c2 s1,
    clean c1 s s1 → div c2 s1
  → div (Aseq c1 c2) s
| div_if: ∀ s (b :bexpr) c1 c2,
    div (if b s then c1 else c2) s
  → div (Aif b c1 c2) s
| div_while_body: ∀ s b (i: pred) c,
    i s → b s = true → div c s
  → div (Awhile b i c) s
| div_while_loop: ∀ s b (i: pred) c s1,
    i s → b s = true → clean c s s1 → div (Awhile b i c) s1
  → div (Awhile b i c) s.
```

Listing 6.5 – Sémantique de divergence des programmes annotés, en Coq.

Les propositions (adiverges c s), (adiverges_fine c s) et (div c s) sont vraies si et seulement si l'exécution du programme annoté c à partir de l'état s ne termine pas.

Ensuite, nous démontrons l'équivalence entre la définition co-inductive de la sémantique de divergence et l'une de ses deux spécifications (équivalentes entre elles) :

```
Theorem adiverges_div_eq: ∀ c s, adiverges c s ↔ div c s.
```

La démonstration de ce théorème est une généralisation simple de celle du résultat analogue pour les programmes sans annotations.

6.9 Sémantique axiomatique inductive

Après avoir défini la sémantique d'exécution bloquante du langage ACMD, nous définissons ici sa sémantique axiomatique, en généralisant la logique de Hoare aux assertions et aux invariants de boucle explicites. Puisqu'une commande peut aussi échouer, il faut réviser comme suit la définition d'un triplet de Hoare valide :

Pour toute commante annotée c et tous les prédicats P et Q, le triplet de Hoare $\{P\}\ c\ \{Q\}$ est valide si et seulement si, lorsque la précondition P est satisfaite, la commande c établit la postcondition Q, si cette commande termine **sans erreur**.

$$\{P\}\ \mathsf{Askip}\ \{P\}\ \ [\mathsf{ax\_skip}] \qquad \{u\ P\}\ \mathsf{Aupdate}\ u\ \{P\}\ \ [\mathsf{ax\_update}]$$

$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ \mathsf{Aseq}\ c_1\ c_2\ \{R\}}\ [\mathsf{ax\_seq}]$$

$$\frac{\{b \wedge P\}\ c_1\ \{Q\} \qquad \{\neg b \wedge P\}\ c_2\ \{Q\}}{\{P\}\ \mathsf{Aif}\ b\ c_1\ c_2\ \{Q\}}\ [\mathsf{ax\_if}]$$

$$\frac{\{b \wedge i\}\ c\ \{i\}}{\{i\}\ \mathsf{Awhile}\ b\ i\ c\ \{\neg b \wedge i\}}\ [\mathsf{ax\_while}]$$

$$\frac{P \to a}{\{P\}\ \mathsf{Aassert}\ a\ \{P\}}\ [\mathsf{ax\_assert}]$$

$$\frac{P \to P' \qquad \{P'\}\ c\ \{Q'\} \qquad Q' \to Q}{\{P\}\ c\ \{Q\}}\ [\mathsf{ax\_consequence}]$$

FIGURE 6.4 – Sémantique axiomatique inductive des programmes annotés.

La figure 6.4 présente une axiomatisation des triplets de Hoare $\{P\}\ c\ \{Q\}$ valides, pour toute commande annotée c, toute précondition P et toute postcondition Q.

Cette sémantique axiomatique est dite *inductive* car la règle ax_while exige que l'annotation i de chaque boucle en soit un invariant inductif.

Avec la définition inductive du listing 6.6, la proposition (ax P c Q) formalise la dérivabilité du triplet de Hoare $\{P\}\ c\ \{Q\}$ dans cette logique.

```
Inductive ax: pred → acmd → pred → Prop :=
| ax_skip: ∀ P,
   ax P Askip P
| ax_update: ∀ P u,
   ax (pupd P u) (Aupdate u) P
| ax_seq: ∀ c1 c2 P Q R,
   ax P c1 Q → ax Q c2 R
   → ax P (Aseq c1 c2) R
| ax_if: ∀ b c1 c2 P Q,
   ax (pand (ptrue b) P) c1 Q → ax (pand (pfalse b) P) c2 Q
   → ax P (Aif b c1 c2) Q
| ax_while: ∀ b (i: pred) c,
   ax (pand (ptrue b) i) c i
   → ax i (Awhile b i c) (pand (pfalse b) i)
| ax_assert: ∀ a P,
   pimp P a
   → ax P (Aassert a) P
| ax_consequence: ∀ c P Q P' Q',
   pimp P P' → ax P' c Q' → pimp Q' Q
```

```
→ ax P c Q.
```

Listing 6.6 – Sémantique axiomatique inductive en Coq.

6.10 Logique de commande

La règle ax_consequence de la logique axiomatique ax complique certaines démonstrations portant sur le calcul de plus faible précondition et le générateur de conditions de vérification définis dans la partie 6.11. Nous introduisons ici un raffinement de cette logique axiomatique, sans règle de conséquence, puis nous montrons son équivalence avec la logique axiomatique. Puisque ce système d'inférence propose une règle par sorte de commandes dans le langage, nous l'appelons *logique de commande* ou *sémantique de commande* des programmes annotés.

$$\frac{P \to Q}{(P)\ \mathsf{Askip}\ (Q)}\ [\mathsf{acommand\_skip}] \qquad \frac{P \to u\ Q}{(P)\ \mathsf{Aupdate}\ u\ (Q)}\ [\mathsf{acommand\_update}]$$

$$\frac{(P)\ c_1\ (Q) \qquad (Q)\ c_2\ (R)}{(P)\ \mathsf{Aseq}\ c_1\ c_2\ (R)}\ [\mathsf{acommand\_seq}]$$

$$\frac{(b \wedge P)\ c_1\ (Q) \qquad (\neg b \wedge P)\ c_2\ (Q)}{(P)\ \mathsf{Aif}\ b\ c_1\ c_2\ (Q)}\ [\mathsf{acommand\_if}]$$

$$\frac{P \to i \qquad (b \wedge i)\ c\ (i) \qquad (\neg b \wedge i) \to Q}{(P)\ \mathsf{Awhile}\ b\ i\ c\ (Q)}\ [\mathsf{acommand\_while}]$$

$$\frac{P \to a \qquad P \to Q}{(P)\ \mathsf{Aassert}\ a\ (Q)}\ [\mathsf{acommand\_assert}]$$

FIGURE 6.5 – Sémantique de commande du langage ACMD.

Ce système est présenté sous forme de règles de déduction dans la figure 6.5 et sous forme de prédicat inductif Coq dans le listing 6.7. Pour matérialiser que ce système est différent de celui de la partie 6.9, les parenthèses remplacent les accolades dans ses "triplets de Hoare", dans la figure 6.5.

La règle acommand_while établit (P) Awhile $b\ i\ c\ (Q)$ avec les obligations de preuve bien connues sur l'invariant de boucle i : L'invariant i est impliqué par la précondition P, il est préservé par le corps de boucle c, et sa conjonction avec la négation de la condition de boucle b implique la postcondition Q.

```
Inductive acommand: pred → acmd → pred → Prop :=
| acommand_skip: ∀ P Q,
   pimp P Q
   → acommand P Askip Q
| acommand_update: ∀ st P Q,
   pimp P (pupd Q st)
   → acommand P (Aupdate st) Q
| acommand_seq: ∀ s1 s2 P Q R,
   acommand P s1 Q → acommand Q s2 R
   → acommand P (Aseq s1 s2) R
| acommand_if: ∀ b s1 s2 P Q,
   acommand (pand (ptrue b) P) s1 Q → acommand (pand (pfalse b) P) s2 Q
   → acommand P (Aif b s1 s2) Q
| acommand_while: ∀ b (i: pred) c P Q,
   pimp P i → acommand (pand (ptrue b) i) c i → pimp (pand (pfalse b) i) Q
   → acommand P (Awhile b i c) Q
| acommand_assert: ∀ a P Q,
   pimp P a → pimp P Q
   → acommand P (Aassert a) Q.
```

Listing 6.7 – Sémantique de commande des programmes annotés, en Coq.

Cette sémantique est équivalente à la sémantique axiomatique ax décrite dans la partie 6.9 :

```
Theorem ax_acommand_eq: ∀ c P Q, ax P c Q ↔ acommand P c Q.
```

Par conséquent, elle peut être adoptée comme définition alternative de la sémantique inductive.

6.11 Génération de conditions de vérification

Au cœur des outils de vérification déductive se trouve un *générateur de conditions de vérification* (VCGEN, pour *Verification Condition GENerator* en anglais), qui prend en entrée un système et sa spécification et produit une liste de conditions logiques appelées *conditions de vérification*. La satisfaction de toutes ces conditions garantit la correction du système par rapport à sa spécification.

À partir du calcul de plus faible précondition wp défini dans la partie 6.11.1, un VCGEN pour la correction partielle des programmes annotés du langage ACMD est défini dans la partie 6.11.2. Nous y démontrons l'équivalence entre cette sémantique et la sémantique axiomatique inductive ax. La partie 6.11.3 justifie le nom de "calcul de plus faible précondition" pour la fonction wp.

6.11.1 Calcul de plus faible précondition

La fonction wp définie dans le listing 6.8 adapte au langage ACMD le calcul de plus faible précondition introduit par Dijkstra [Dij76]. Le prédicat (wp c Q) est la plus faible précondition qui doit être satisfaite pour que la commande c établisse la postcondition Q lorsqu'elle termine sans erreur. Le nom de cette fonction wp vient des initiales de l'expression *weakest precondition* en anglais.

```
Fixpoint wp (c: acmd) (Q: pred) : pred :=
 match c with
 | Askip ⇒ Q
 | Aupdate u ⇒ pupd Q u
 | Aseq c1 c2 ⇒ wp c1 (wp c2 Q)
 | Aif b c1 c2 ⇒ por (pand (ptrue b) (wp c1 Q)) (pand (pfalse b) (wp c2 Q))
 | Awhile b i c ⇒ i
 | Aassert a ⇒ pand a Q
 end.
```

Listing 6.8 – Calcul de plus faible précondition pour les programmes annotés, en Coq.

Pour toute commande annotée c, le transformateur de prédicat (wp c) (*predicate tranformer* en anglais) est monotone :

```
Lemma wp_mono : ∀ c Q Q', pimp Q Q' → pimp (wp c Q) (wp c Q').
```

Ce lemme est démontré par induction sur la commande annotée c.

6.11.2 Générateur de conditions de correction partielle

Le listing 6.9 présente une définition d'un générateur vcgen de conditions de vérification de la correction partielle des programmes annotés du langage ACMD, fondée sur le calcul de plus faible précondition wp et une fonction récursive cc de collecte de conditions.

La proposition (vcgen P c Q) est une autre définition de la correction partielle de la commande annotée c par rapport à la précondition P et à la postcondition Q. Nous le confirmons en démontrant sa correction et sa complétude par rapport à la sémantique axiomatique inductive :

```
Theorem vcgen_sound_ax: ∀ P a Q, vcgen P a Q → ax P a Q.
Theorem vcgen_complete_ax: ∀ a P Q, ax P a Q → vcgen P a Q.

Fixpoint cc (c: acmd) (Q: pred) {struct c} : Prop :=
 match c with
 | Askip ⇒ True
 | Aupdate u ⇒ True
 | Aseq c1 c2 ⇒ cc c1 (wp c2 Q) ∧ cc c2 Q
 | Aif b c1 c2 ⇒ cc c1 Q ∧ cc c2 Q
 | Awhile b i c ⇒ cc c i ∧ pimp (pand (pfalse b) i) Q ∧ pimp (pand (ptrue b) i) (wp c i)
 | Aassert a ⇒ True
 end.
```

```
Definition vcgen (P: pred) (c: acmd) (Q: pred) : Prop := pimp P (wp c Q) ∧ cc c Q.
```

Listing 6.9 – Générateur de conditions de vérification pour les programmes annotés.

Les démonstrations de ces théorèmes passent par l'intermédiaire de théorèmes similaires vcgen_sound et vcgen_complete entre vcgen et la sémantique de commande acommand.

Par conséquent, on peut indifféremment choisir ax, acommand ou vcgen (mutuellement équivalents) comme définition de la sémantique inductive des programmes annotés.

6.11.3 Justification de plus faible précondition

Nous justifions le nom de "calcul de plus faible précondition" pour la fonction wp, en démontrant formellement les lemmes

```
Lemma precond_ax: ∀ c Q, cc c Q → ax (wp c Q) c Q.
```

et

```
Lemma weakest_ax: ∀ c P Q, ax P c Q → pimp P (wp c Q).
```

Le lemme precond_ax formalise que (wp c Q) est une précondition pour que la condition annotée c satisfasse Q, lorsque les conditions collectées sont satisfaites. Il est démontré par induction sur la commande annotée c.

Le lemme weakest_ax formalise que toute précondition P pour c et Q est plus faible que (wp c Q). Il est démontré par l'intermédiaire d'un lemme analogue avec acommand au lieu de ax.

Le lemme precond_acommand, analogue du lemme precond_ax, mais pour la logique de commande acommand, participe à la démonstration du théorème sus-mentionné vcgen_sound de correction du générateur de conditions de vérification par rapport à la logique de commande.

6.11.4 Autres propriétés

Pour toute commande annotée c, le transformateur de prédicat (cc c) est monotone :

```
Lemma cc_mono: ∀ c P Q, pimp P Q → cc c P → cc c Q.
```

Ce lemme est démontré par induction sur la commande annotée c.

Le lemme suivant établit que les conditions de vérification sont correctes par rapport à la logique de commande :

```
Lemma acommand_cc: ∀ c P Q, acommand P c Q → cc c Q.
```

Ces deux lemmes interviennent dans une démonstration du théorème vcgen_complete présenté plus haut.

6.12 Correction de la sémantique inductive par rapport à la sémantique bloquante

Cette partie établit la correction de la sémantique inductive (parties 6.9 à 6.11) par rapport à la sémantique bloquante (parties 6.4 à 6.8).

Plus précisément, nous démontrons la correction de la sémantique axiomatique inductive ax (partie 6.9) par rapport à la sémantique de terminaison à grandes étapes term (partie 6.7). D'autres théorèmes de correction entre les autres définitions de ces sémantiques (acommand et vcgen pour la sémantique inductive, small pour la sémantique bloquante) s'en déduisent facilement par les équivalences déjà démontrées. Ils ne sont pas présentés ici.

Pour spécifier la sémantique inductive à partir de la sémantique bloquante, nous introduisons la définition suivante :

```
Definition cmust (r : state * flag → state * flag → Prop) (Q : pred) : pred :=
 fun x ⇒ (∀ y f, r (x, fine) (y, f) → Q y ∧ f = fine).
```

La fonction cmust généralise aux états complets la fonction must définie dans la partie 5.7.2, en n'acceptant que les transitions vers les états complets (y, f) sans erreur (f = fine).

Avec cette fonction, nous pouvons spécifier le calcul de plus faible précondition par

```
Definition wp_term c : pred → pred := cmust (term c).
```

et la sémantique inductive par

```
Definition ax_term P c Q := ∀ s, P s → wp_term c Q s.
```

Le théorème de correction

```
Theorem ax_sound: ∀ P c Q, ax P c Q → ax_term P c Q.
```

se démontre par induction sur les règles de la sémantique axiomatique ax.

Nous avons également démontré le théorème de correction

```
Theorem vcgen_sound_term : ∀ P c Q, vcgen P c Q → ax_term P c Q.
```

pour la sémantique inductive définie par le générateur de conditions de vérification vcgen. La principale clé de cette démonstration est le lemme

```
Lemma wp_and_cc_imply_wp_term: ∀ c Q, cc c Q → pimp (wp c Q) (wp_term c Q).
```

qui est intéressant pour lui-même, car il établit que le calcul de plus faible précondition est correct par rapport à sa spécification wp_term à l'aide de la sémantique de terminaison term, lorsque les conditions collectées sont satisfaites.

6.13 Non-complétude de la sémantique inductive

Cette partie montre que la sémantique inductive est incomplète par rapport à la sémantique bloquante. Plus précisément, nous donnons deux contre-exemples pour la fausse conjecture

```
Conjecture ind_complete : ∀ (P: pred) c Q, ax_term P c Q → vcgen P c Q.
```

de complétude de la sémantique inductive par rapport à sa spécification ax_term avec la sémantique bloquante à grandes étapes term. Pour simplifier la justification que cette conjecture est fausse, elle est énoncée avec la version calculatoire vcgen de la sémantique inductive, mais les conjectures correspondantes avec ax ou acommand à la place de vcgen sont également fausses, par équivalence entre ces trois définitions de la sémantique inductive.

La non-complétude de la sémantique inductive provient exclusivement de la commande de boucle annotée. Nous l'expliquons ici en comparant successivement chaque prémisse de la règle

$$\frac{P \to i \qquad (b \wedge i)\ c\ (i) \qquad (\neg b \wedge i) \to Q}{(P)\ \mathsf{Awhile}\ b\ i\ c\ (Q)}\ \text{[acommand\_while]}$$

de la sémantique de commande acommand avec les règles des sémantiques bloquantes small et term à petites et grandes étapes.

Rappelons que la première prémisse $(P \to i)$ abrège la condition $(\forall\ s, P\ s \to i\ s)$. Cette prémisse n'est pas liée au problème de non-complétude, puisque la sémantique bloquante exige $(i\ s)$ avant exécution de la boucle, et la spécification ax_term – qui commence par $\forall\ s, P\ s \to$ – limite cette condition aux états s qui satisfont P.

Le premier (resp. second) contre-exemple suivant montre que la non-complétude peut être due à la deuxième prémisse $(b \wedge i)\ c\ (i)$ (resp. troisième prémisse $(\neg b \wedge i \to Q)$).

6.13.1 Contre-exemple 1 : "invariant" non inductif

Considérons la commande annotée

$$c_1 = \mathsf{Aseq}\ a\ (\mathsf{Awhile}\ b\ i\ c)$$

où les commandes a (appelée *prélude*) et c (appelée *corps de boucle*) sont deux mises à jour (Aupdate).

états	1	2	3	4
a	2	3	3	3
c	4	3	3	4
b	T	T	F	F
i	T	T	T	F

FIGURE 6.6 – Données du premier contre-exemple.

La deuxième prémisse $(b \wedge i)\ c\ (i)$ de la règle acommand_while exige que l'annotation i soit un invariant inductif du corps de boucle c. Pour vérifier cette prémisse dynamiquement, il faudrait exécuter c dans tous les états qui satisfont $b \wedge i$. Or, la sémantique bloquante n'exécute c que dans les états accessibles après le prélude a. Il en résulte que cette sémantique dynamique peut considérer la commande c_1 comme correcte, tandis que la sémantique inductive la considère comme incorrecte, car sa proposition d'invariant n'est pas un invariant inductif de sa boucle.

Nous illustrons cette affirmation par un système à quatre états seulement. Ces états sont nommés 1, 2, 3 et 4. Ils sont codés par st1, st2, st3 et st4 en Coq. Le tableau de la figure 6.6 donne une sémantique de fonction totale aux mises à jour a et c, une sémantique de fonction booléenne à la condition de boucle b et une sémantique de fonction propositionnelle à l'invariant de boucle i, en notant indifféremment par T et F les constantes booléennes true et false et les constantes propositionnelles True et False de Coq. Le listing 6.10 traduit ces définitions en Coq.

Cet exemple ignore les questions de précondition et de postcondition, en considérant qu'elles sont toutes deux vraies dans tous les états, avec la définition

```
Definition truep (s: state) : Prop := True.
```

pour le prédicat toujours vrai T.

```
(** *** Prelude [a]: *)
Function upd1 (s: state) : state :=
 match s with
 | st1 ⇒ st2
 | _ ⇒ st3
 end.

(** *** Loop body [c]: *)
Function upd2 (s: state) : state :=
 match s with
 | st1 ⇒ st4
 | st4 ⇒ st4
 | _ ⇒ st3
 end.

(** *** Loop condition [b]: *)
Function cond1 (s: state) : bool :=
 match s with
 | st1 ⇒ true
 | st2 ⇒ true
 | _ ⇒ false
 end.

(** *** Loop invariant candidate: *)
Function inv1 (s: state) : Prop :=
 match s with
 | st4 ⇒ False
 | _ ⇒ True
 end.

(** *** Program [a; while b i c]: *)
Definition prgm1 := Aseq (Aupdate upd1) (Awhile cond1 inv1 (Aupdate upd2)).
```

Listing 6.10 – Données du premier contre-exemple en Coq.

Pour une précondition et une postcondition toujours vraies, la correction du programme c_1 (codé en Coq par prgm1) selon la sémantique bloquante est formalisée par le lemme

```
Lemma ax_term1: ax_term truep prgm1 truep.
```

qui se démontre facilement par énumération des états.

La même propriété de correction du programme c_1 selon la sémantique inductive est formalisée par le lemme

```
Lemma vcgen1: vcgen truep prgm1 truep.
```

Après simplification on obtient la condition de vérification

$$\mathsf{vcgen}\ T\ c_1\ T = \mathsf{pupd}\ i\ \mathsf{upd1} \wedge ((b \wedge i) \rightarrow \mathsf{pupd}\ i\ \mathsf{upd2}).$$

avec $\mathsf{pupd}\ i\ u\ s = i\ (u\ s)$ pour tout état s et toute fonction totale sur les états u. La table de vérité de cette condition et de quelques-unes de ses sous-expressions est :

s	1	2	3	4
i (upd1 s)	T	T	T	T
i (upd2 s)	F	T	T	F
$b\ s$	T	T	F	F
$i\ s$	T	T	T	F
$b\ s \wedge i\ s$	T	T	F	F
vcgen $T\ c_1\ T$	F	T	T	T

Ainsi, le lemme vcgen1 est faux pour l'état 1. Lorsqu'on essaie de le démontrer, après simplification et énumération des états, on obtient pour l'état 1 le sous-but

```
true = true, True ⊢ False
```

évidemment non prouvable. En effet l'état 1 satisfait b et i mais la mise à jour c le remplace par l'état 4 qui ne satisfait pas i.

Ainsi, le programme annoté c_1 contredit la conjecture ind_complete de complétude de la sémantique inductive par rapport à la sémantique bloquante.

6.13.2 Contre-exemple 2 : troisième prémisse fausse

Il est encore plus simple de donner un contre-exemple de complétude portant sur la troisième prémisse $(\neg b \wedge i) \rightarrow Q$ de la règle acommand_while. Cette condition exige que l'annotation i et la négation de la condition de boucle b impliquent la postcondition Q.

Considérons deux états 1 et 2 (codés par st1 et st2 en Coq), la commande annotée

$$c_2 = \mathsf{Awhile}\ b\ i\ c$$

et les définitions de la figure 6.7 pour le corps de boucle c, la condition de boucle b, l'annotation de boucle i, la précondition P et la postcondition Q, avec les mêmes conventions que précédemment pour T et F.

états	1	2
c	1	2
b	F	F
i	T	T
P, Q	T	F

FIGURE 6.7 – Données du second contre-exemple.

Dans cet exemple, la condition de boucle b est toujours fausse. Il en résulte que le corps de boucle n'est jamais exécuté et que le candidat-invariant i est un invariant inductif de la boucle. Par contre, la troisième prémisse $(\neg b \wedge i) \rightarrow Q$ n'est pas satisfaite dans l'état 2. Pourtant, l'exécution de la commande c_2 à partir de l'état 1, seul état qui satisfait la précondition P, termine dans l'état 1 qui satisfait la postcondition Q. Pour ces pré- et postconditions, ce programme est donc

correct selon la sémantique dynamique et incorrect selon la sémantique statique. Ce raisonnement est illustré par le code Coq du listing 6.13.2.

```
(** *** Loop body [c]: *)
Function upd1 (s: state) : state :=
 match s with
 | st1 ⇒ st1
 | st2 ⇒ st2
 end.

(** *** Loop condition [b]: *)
Function cond1 (s: state) : bool :=
 match s with
 | _ ⇒ false
 end.

(** *** Loop invariant candidate [i]: *)
Function inv1 (s: state) : Prop :=
 match s with
 | _ ⇒ True
 end.

(** *** Precondition [P] and postcondition [Q]: *)
Function pred1 (s: state) : Prop :=
 match s with
 | st1 ⇒ True
 | st2 ⇒ False
 end.

(** *** Annotated program [c2 = while b i c]: *)
Definition c2 := Awhile cond1 inv1 (Aupdate upd1).

(** *** [c2] is correct for blocking semantics: *)

Lemma ax_term2: ax_term pred1 c2 pred1.

(** *** [c2] is not correct for inductive semantics: *)

Lemma vcgen2: vcgen pred1 c2 pred1.

  (** Unclosable subgoal:

   false = false, True ⊢ False *)
```

Ainsi, le programme annoté c_2 contredit aussi la conjecture ind_complete de complétude de la sémantique inductive par rapport à la sémantique bloquante.

La sémantique inductive est dite “forte”, car elle implique la sémantique bloquante, dite “faible”. La réciproque est fausse.

6.14 Notes bibliographiques

Cette partie complète les notes bibliographiques de la partie 5.8 avec des références sur les assertions (partie 6.14.1) et sur les formalisations de sémantiques de programmes annotés (partie 6.14.2). A l’intérieur de chaque sujet les références sont classées dans l’ordre chronologique.

6.14.1 Assertions

Dans la littérature du domaine, la notion d’assertion a dépassé sa définition initiale de propriété d’état. Voici par exemple comment Hoare élargit progressivement la signification du mot “assertion” :

> “[Assertions] started as simple Boolean expressions in a sequential programming language, testing a property of a single machine state at the point that control reaches the assertion. By adding dashed variables to stand for the values of variables at program termination, an assertion is generalized to a complete specification of an arbitrary fragment of a sequential program. By adding variables that record the history of interactions between a program and its environment, assertions specify the interfaces

> between concurrent programs. By defining a program's semantics as the strongest assertion that describes all its possible behaviors, we give a complete method for proving the total correctness of all programs expressed in the language." [Hoa02, page 23].

La littérature du domaine désigne parfois aussi par assertion une propriété qui doit être admise (*assume*), ou considère qu'un invariant de boucle est une assertion. Nous excluons ces généralisations et nous donnons ici au mot "assertion" la signification précise de commande particulière dans un langage de programmes annotés. Comme justifié dans la partie 4.8.2, nous désignons une propriété d'état par "prédicat" plutôt que par "assertion".

Cette partie présente quelques références sur les assertions, en tant qu'annotations ou décorations de programmes, avec les objectifs suivants :

1. retracer l'histoire de l'idée d'inclure des prédicats comme des annotations dans les programmes à vérifier, puis l'idée de considérer ces annotations comme des commandes de ces programmes annotés,
2. présenter des alternatives aux assertions vues comme des commandes, et
3. montrer comment les quantités d'assertions exigées pour automatiser la vérification ont été progressivement réduites.

Floyd (1967)

L'idée d'annoter des programmes avec des propriétés d'état est déjà présente dans l'article fondateur de Floyd [Flo67], comme l'idée d'évaluer ces propositions (pas encore nommées "assertions") au cours de l'exécution :

> "Let us say that each statement [...] is tagged with an antecedent and a consequent proposition [...] said to hold whenever control enters and leaves the statement in the normal sequential mode of control." [Flo67, partie 3].

Igarashi, London et Luckham (1973)

L'origine de l'idée de commande d'assertion est difficile à établir, mais elle est certainement antérieure à 1973, puisqu'elle est clairement décrite cette année-là dans un rapport de l'université de Stanford :

> "Assertions are added to programs as additional statements beginning with the special symbol ASSERT" [ILL73, page 8]

Ce rapport définit un générateur de conditions de vérification pour un fragment du langage Pascal.

Owicki et Gries (1976)

Owicki et Gries suggèrent d'annoter le programme avec les assertions qui permettent de le démontrer à l'aide du système déductif de Hoare :

> "proof is made much more understandable by giving a *proof outline*, in which the program is given with assertions interleaved at appropriate places" [OG76, pages 321-322].

Pour qu'on puisse voir des triplets de Hoare dans un programme ainsi annoté, une assertion est ajoutée **avant** et **après** chaque commande. Les prémisses $P \to P'$ des applications de la règle de conséquence de la logique de Hoare correspondent aux séquences $\{P\}\{P'\}$ de deux assertions consécutives P et P' dans cette *proof outline* complète.

Puis Owicki et Gries autorisent l'élimination des annotations pour les séquences d'affectations et les instructions conditionnelles simples (If) :

> "We may [...] leave out assertions entirely for a sequence of assignments or simple conditionals, since the necessary weakest precondition of the sequence can always be derived from the postcondition." [OG76, pages 322].

Gordon (1988)

Gordon définit le langage de programmes annotés suivant :

> "An annotated command is a command with statements (called *assertions*) embedded within it. A command is said to be properly annotated if statements have been inserted at the following places :
>
> (i) Before each command C_i (where $i > 1$) in a sequence $C_1; C_2; \ldots; C_n$ which is not an assignment command,
>
> (ii) After the word `DO` in `WHILE` and `FOR` commands." [Gor88, partie 3.3, page 44].

Gordon définit ensuite un générateur de conditions de vérification [Gor88, pages 44 à 51] pour ces commandes annotées par un prédicat pour chaque boucle et avant chaque commande de chaque séquence, si cette commande n'est ni une affectation ni la première commande de la séquence.

Ainsi, le langage de Gordon est moins général que notre langage ACMD, notamment parce que ses séquences doivent être fortement annotées et parce qu'il exclut la possibilité d'une annotation après la dernière commande d'une séquence, pouvant jouer un rôle de postcondition.

Winskel (1993)

Dans sa monographie [Win93] déjà présentée dans le chapitre 5, Winskel reprend le langage de commandes annotées de Gordon, en ajoutant **skip** et en remplaçant les séquences $C_1; \ldots; C_n$ de longueur quelconque de Gordon par un opérateur de séquence binaire :

> "Define the syntactic set of annotated commands by :
>
> $$c ::= \textbf{skip} \mid X := a \mid c_0; (X := a) \mid c_0; \{D\}c_1 \mid \textbf{if } b \textbf{ then } c_0 \textbf{ else } c_1 \mid \textbf{while } b \textbf{ do } \{D\}c$$
>
> where X is a location, a an arithmetic expression, b is a boolean expression, c, c_0, c_1 are annotated commands and D is an assertion such that in $c_0; \{D\}c_1$, the annotated command c_1, is *not* an assignment." [Win93, partie 7.4, page 112].

L'ajout de **skip** permet d'écrire la commande annotée $(c; \{Q\}\textbf{skip})$ qui place Q en position de postcondition de c.

Fraer (1996)

Ranan Fraer définit un générateur de conditions de vérification pour un langage impératif simple, composé de l'instruction `skip`, de la séquence binaire (`;`), de l'affectation, d'une instruction conditionnelle `if then else` et d'une instruction de boucle `while` annotée avec un invariant [Fra96]. Chaque instruction dans une séquence, une conditionnelle ou une boucle peut être optionnellement précédée par une assertion.

Nipkow (1998)

Dans sa formalisation de [Win93], Nipkow réduit à leur minimum les assertions requises pour la génération automatique des conditions de vérification :

> "Winskel follows Gordon [Gor88] in his treatment of verification conditions, who inserts rather more annotations than strictly speaking necessary. This complicates the syntax of annotated commands. We go for the minimal amount of annotation, namely loop invariants." [Nip98, page 180].

Après cette constatation, Nipkow définit un type de commandes où seule l'instruction de boucle est annotée, par un invariant.

Reynolds (1998)

Reynolds [Rey98, chapitre 5, p. 97] ajoute à son langage impératif

> "a command **fail** that causes a program to cease execution".

La commande **fail** présentée dans [Rey98] est plus simple que notre proposition de commande d'assertion. Au lieu de la commande (Aassert b), nous pourrions étendre le langage CMD uniquement avec une commande Afail et considérer (Aassert b) comme une notation pour (Aif b Afail Askip). Inversement, nous pouvons coder la commande **fail** avec la commande (Aassert pfalse).

Ensuite, Reynolds pose le problème de faire passer cette interruption au contexte, en particulier à travers la composition séquentielle. Il propose d'ajouter un marqueur d'échec (**abort**) et de mémoriser l'état dans lequel **fail** est exécuté. Nous procédons de même avec la commande (Aassert b), qui s'arrête dans une configuration (Aassert b, s, abort) qui mémorise la commande qui a échoué et l'état s dans lequel elle a échoué.

Hoare (2002)

Dans une perspective historique, Hoare nous donne en 2002 son point de vue personnel sur les assertions [Hoa02]. Je ne résiste pas au plaisir de citer et de commenter quelques extraits de ce document, très éclairant sur la pensée de ce grand chercheur.

> "It's the concept of an assertion that links my earlier research with current industrial software engineering practice and provides the basis for hopes of future improvement. Assertions figure strongly in Microsoft code. A recent count discovered more than a quarter million of them in the code for its Office product suite. **The primary role of an assertion today is as a test oracle**, defining the circumstances under which a program under test is considered to fail. A collection of aptly placed assertions is what permits a massive suite of test cases to be run overnight, without human intervention. Failure of an assertion triggers a dump of the program state, to be analyzed by the programmer on the following morning. [...] So assertions have already found their major application, not to the proof of program correctness, but to the diagnosis of their errors." [Hoa02, page 23].

Effectivement, les assertions, initialement introduites pour la preuve de programmes, ont acquis leur popularité comme objectifs et oracles pour les outils de génération et d'exécution automatique de tests.

Hoare présente ainsi le premier programme qu'il a prouvé correct :

> "This was the first program that I proved correct. It performs positive integer division by the lengthy process of counting the number of times the divisor can be subtracted from the dividend. The assertion at the beginning states the precondition, which the user of the routine must guarantee before entry. The final assertion describes the postcondition that the routine will make true on exit." [Hoa02, Figure 2].

En attribuant ainsi en 2002 le même statut d'assertion à une précondition et à une postcondition (et en leur associant le même mot-clé ASSERT), Hoare indique qu'il ne s'intéresse qu'à la preuve de correction (de ce programme) vis-à-vis d'une pré- et d'une postcondition, pas à la validité de ces assertions lors des exécutions du programme (une précondition ne doit pas être validée, mais doit être admise comme valide). D'autres travaux et divers langages font la distinction, en utilisant par exemple le mot-clé ASSERT pour les assertions à valider, et le mot-clé ASSUME pour les préconditions.

Bertot (2009)

Bertot [Ber09, page 13] considère aussi un langage annoté avec des assertions et des invariants de boucles, mais une assertion n'y est pas une instruction, mais une *garde* facultative pour les instructions Skip, d'affectation, de séquence et de boucle.

Leroy (2013)

Leroy annote les boucles avec un invariant et ajoute une commande d'assertion :

> "In this section, we enrich the syntax of [...] commands with an annotation on `while` loops (to give the loop invariant) and an `assert`(P) command to let the user provide assertions. [acmd]" [Ler13, partie 3.3].

Notre langage ACMD de programmes annotés ne diffère de ce langage [acmd] que sur des détails décrits dans la partie 6.3.

Marché et Tafat (2013)

L'article [MT13] formalise le même langage avec l'outil Why3, en définissant un langage de formules pour les assertions.

Software foundations (2017)

Les ouvrages de la série "*Software Foundations*" sont des scripts de preuve avec Coq. Le volume 2 [PAC+17] est consacré à la théorie des programmes.

Cette synthèse va aussi loin que nous dans le langage. Cependant, les assertions n'y sont pas des commandes mais des décorations dans les commandes :

> "The first thing we need to do is to formalize a variant of the syntax of commands with embedded assertions. We call the new commands decorated commands [...] decorations are added corresponding to each postcondition. A separate type [...] is used to add the precondition for the entire program. [...] The choice of exactly where to put assertions in the definition [...] is a bit subtle." [PAC+17, `Hoare2.html`]

En effet, cette proposition est plus complexe que celles de Bertot [Ber09] (où les assertions sont des gardes des instructions) et de [ILL73, Ler13, MT13] (où les assertions sont des instructions).

6.14.2 Formalisations de la correction des programmes annotés

Leroy (2013)

Leroy définit un calcul de plus faible précondition wp sur son langage acmd de programmes annotés et un générateur de conditions de vérification vcgen pour toute commande annotée c.

Leroy démontre ensuite la correction de ce VCGEN par rapport à la logique de Hoare :

> "**Theorem 19** [vcgen_correct] If `vcgen`(P, c, Q) holds, then $\{P\}\ c\ \{Q\}$ can be derived by the rules of axiomatic semantics." [Ler13, page 15]

En réalité, comme le montre le code Coq correspondant

```
Theorem vcgen_correct: ∀ P a Q, vcgen P a Q → triple P (erase a) Q.
```

tandis que le VCGEN porte sur une commande annotée c, le triplet de Hoare dans ce théorème porte sur la commande c' sans annotations obtenue en effaçant toutes les assertions dans c. Gordon [Gor88] et Nipkow [Nip98] on démontré un théorème analogue.

Dans les parties 6.9 et 6.11.2 nous allons plus loin, en étendant la logique de Hoare aux commandes annotées et en démontrant la correction du VCGEN par rapport à cette logique sur le même langage.

Herms (2013)

Dans sa thèse, Paolo Herms développe et certifie avec Coq un générateur de conditions de vérification pour des programmes C annotés avec ACSL [Her13]. Ce VCGEN est d'abord défini sur un langage impératif intermédiaire Turing-complet avec exceptions et fonctions récursives, et prouvé correct par rapport à la sémantique bloquante du langage. Le VCGEN est ensuite étendu aux langages C et ACSL et relié au compilateur certifié Compcert.

La définition du VCGEN sur le langage intermédiaire et sa preuve de correction dépassent de loin ce que nous proposons dans ce mémoire, mais elles sont si complexes qu'il est difficile de les comparer. C'est d'ailleurs peu pertinent, puisque notre formalisation sur un langage impératif élémentaire poursuit essentiellement un objectif pédagogique, tandis que la formalisation très complète de Herms poursuit l'objectif d'extraire un outil de vérification déductive 100% certifié pour C et ACSL.

Marché et Tafat (2013)

C. Marché et A. Tafat formalisent avec Why3 une sémantique bloquante à petites étapes `one_step` et un calcul de plus faible précondition `wp` [MT13]. Ils profitent ainsi d'un haut niveau d'automatisation des preuves offert par l'environnement Why3. Puis ils démontrent la correction du calcul de plus faible précondition par rapport à cette sémantique bloquante à petits pas.

En Coq, nous avons complété ce travail, notamment avec une sémantique bloquante à grandes étapes et une sémantique inductive.

6.15 Conclusion

6.15.1 Synthèse

Nous avons défini un langage Turing-complet de programmes impératifs *annotés* par une instruction d'assertion et un invariant de boucle dans chaque instruction de boucle. L'instruction d'assertion déclare une propriété d'état – elle-même appelée assertion – qui doit être vraie lors de l'exécution de cette instruction. Ces assertions et invariants de boucle explicites sont conjointement appelés *annotations*.

Un programme annoté ne doit être considéré comme correct que si ses annotations sont respectées. Pour formaliser cette notion de correction, nous avons défini diverses sémantiques des instructions et annotations de ce langage.

La présence d'invariants de boucle explicites divise ces sémantiques en deux groupes, correspondant aux points de vue statique et dynamique :

— Les sémantiques *inductives* correspondent au point de vue de l'analyse statique des programmes annotés, sans exécution. Ces sémantiques exigent que les invariants soient *inductifs*, c'est-à-dire préservés par le corps de leur boucle à partir de tout état. Ce groupe est composé de sémantiques axiomatiques et de générateurs de condition de vérification pour un prouveur de théorèmes.

— En définissant l'exécution de chaque programme annoté, les sémantiques *bloquantes* correspondent au point de vue dynamique sur ces programmes. L'exécution échoue dans un état d'erreur dès qu'elle rencontre une assertion fausse ou un invariant de boucle faux lors de l'exécution de sa boucle.

Ainsi, les sémantiques dynamiques n'exigent la préservation d'un invariant de boucle qu'à partir de tout état accessible avant exécution de sa boucle. Par conséquent, elles ne sont pas équivalentes aux sémantiques statiques, mais seulement impliquées par elles. En ce sens, les sémantiques *inductives*, statiques, sont dites "fortes", tandis que les sémantiques *bloquantes*, dynamiques, sont dites "faibles".

A l'intérieur de chaque groupe, toutes les sémantiques sont équivalentes entre elles.

6.15.2 Discussion

Cette partie rassemble diverses remarques inspirées par le contenu du chapitre. Elle commente certains choix et ouvre la voie pour diverses perspectives d'améliorations, d'adaptation et d'extension de cette étude.

Séparation des sujets

Pour bien mettre en évidence l'impact relatif d'ajout d'assertions et d'invariants explicites dans un langage impératif, nous aurions pu effectuer ces deux ajouts dans deux études disjointes. Nous avons choisi de les ajouter simultanément par souci de compacité.

Du statique au dynamique

Nous avons présenté les diverses sémantiques des programmes annotés selon la même progression par niveaux que dans le chapitre 5, qui va des sémantiques dynamiques aux sémantiques statiques. Ceci nous a conduit à une démonstration de correction et à une justification de non-complétude des sémantiques statiques par rapport aux sémantiques dynamiques.

Cependant, la "bonne" sémantique d'un invariant de boucle est plutôt sa sémantique statique, qui exige qu'il soit inductif. C'est ce qu'on cherche à assurer quand on le spécifie. Ainsi, en prenant le point de vue inverse, nous avons démontré la **complétude de la sémantique bloquante par rapport à la sémantique inductive** (tout programme annoté qui est inductivement correct s'exécute sans blocage), et justifié la **non-correction de la sémantique bloquante par rapport à la sémantique inductive** (il existe des programmes annotés qui s'exécutent sans blocage, mais qui sont inductivement incorrects).

Limites de l'analyse dynamique

Nous avons démontré que les sémantiques dynamiques sont "moins précises" que les sémantiques statiques, en ce sens qu'elles n'imposent pas et ne vérifient pas que chaque invariant de boucle soit inductif. C'est une limitation connue de l'exécution concrète pour aider la preuve de programmes : elle ne peut pas toujours détecter qu'un invariant est inductif.

Vérification des assertions à l'exécution

On parle de *vérification des assertions à l'exécution* (RAC, *Runtime Assertion Checking* en anglais) lorsque l'exécution vérifie chaque annotation à l'exécution. Les sémantiques bloquantes formalisées ici correspondent à la forme la plus élémentaire de RAC, qui arrête le programme dès que la première annotation fausse est rencontrée.

6.15.3 Perspectives

Cette partie énumère quelques-unes des nombreuses perspectives offertes par la présente étude.

Vers la vérification boîte blanche

La vérification déductive en boîte blanche, définie dans le chapitre 3, exige d'expliciter la traduction de la correction d'un système et de sa spécification en formules logiques soumises à un prouveur de théorèmes.

Dans la littérature du domaine cette réduction est définie et implémentée par un générateur de conditions de vérification. Nous avons nommé `vcgen` une fonction Coq qui génère ces formules sous la forme de propositions Coq, pour tout programme annoté du langage ACMD. Cette approche dite *superficielle* (*shallow* en anglais) ou *sémantique* est adaptée à notre objectif intermédiaire, qui est d'établir des correspondances entre des sémantiques et entre des définitions possibles de la correction (partielle) de ces programmes annotés. Cependant, l'objectif de vérification boîte blanche est plus exigeant : If faut pouvoir observer finement la structure des formules produites, voire transformer souplement ces formules. Dans ce but, il est préférable d'adopter une approche *profonde* (*deep* en anglais) ou *syntaxique* selon laquelle les conditions de vérification sont générées dans un langage défini par un type inductif Coq de formules dédié à cet usage.

Sémantique formelle avec Why3

Why3 [BFMP11, FP13] est une plateforme pour la vérification déductive de programmes, composée d'un langage de programmation et de spécification, appelé WhyML, et d'un outil de génération de conditions de vérification de ces programmes spécifiés, dans le format d'entrée de nombreux prouveurs externes automatiques (solveurs SMT) et interactifs.

Des travaux récents revisitent la sémantique formelle avec Why3, en particulier pour accroitre l'automaticité des preuves [MT13, CG15]. Quoique l'interactivité des preuves avec Coq n'ait pas été un obstacle majeur dans la présente étude, il serait intéressant de la reprendre en Why3, en la comparant ou en l'intégrant avec ces travaux.

Formalisation de l'instrumentation de StaDy

Au cours de sa thèse [Pet15] G. Petiot a développé une méthode et un outil (nommé StaDy) de diagnostic automatique des échecs de preuve, dans le cadre de la plate-forme Frama-C d'analyse des programmes C annotés en ACSL. Cette méthode consiste à appliquer une génération de tests structurels sur une version instrumentée du programme d'origine.

Notre formalisation Coq est une base solide pour démontrer formellement la correction de la méthode de diagnostic de G. Petiot.

Concrétisations et extensions

Le langage ACMD peut être concrétisé par le choix de langages pour les expressions et les mises à jour. Il peut être étendu à d'autres annotations, comme les hypothèses (*assume* en anglais), qui généralisent les préconditions, ou les clauses de modification mémoire (*assigns* en anglais). Après ajout de variants pour les boucles, la présente étude de la correction partielle de ces programmes annotés peut être étendue à la question de leur correction totale.

Chapitre 7

Conclusion

Ce chapitre conclut ce mémoire par une synthèse de son contenu (partie 7.1) et un exposé de mon projet de recherche (partie 7.2).

7.1 Synthèse

Ce mémoire résulte de mes activités de recherche jusqu'en 2016, dans le domaine des méthodes formelles de spécification et de vérification de programmes. Après une présentation chronologique complète de mes travaux (chapitre 2), il traite d'un sujet central dans mes recherches : la vérification déductive.

Le problème général de vérification de la correction des systèmes paramétrés est indécidable. Le défi est d'identifier des familles de systèmes paramétrés dont la correction est décidable. Dans ce but, le chapitre 3 définit une méthode de vérification déductive, distingue trois niveaux de finesse dans son analyse, et classifie mes travaux sur ce sujet selon ces trois niveaux.

Le niveau d'analyse le plus fin, qualifié de "boîte blanche", requiert d'expliciter la traduction d'un système spécifié en formulation logique de sa correction. Le chapitre 6 présente cette traduction comme une sémantique formelle, pour un langage simple de programmes impératifs annotés. Cette sémantique dite *inductive* est reliée à une sémantique d'exécution de ces programmes et annotations, dite *bloquante*. Cette étude est précédée d'une présentation de toutes les notions et notations utiles (chapitre 4) et d'une synthèse de sémantiques formelles du même langage impératif, mais sans annotations (chapitre 5).

Cette base formelle doit permettre de décrire formellement diverses méthodes de vérification, que nous proposons d'appliquer aux programmes de ce langage, avec assez de précision pour pouvoir démontrer de bonnes propriétés de ces méthodes, en particulier leur correction. En deçà de cet objectif de démonstration, cette description est assez précise pour lever toute ambiguité sur les notions traitées, afin d'améliorer la communication entre chercheurs, concepteurs et utilisateurs de ces langages et méthodes.

7.2 Projets de recherche

Mes projets de recherche s'articulent autour de deux axes, présentés dans les deux parties suivantes.

7.2.1 Méthodes de vérification

Pour populariser les méthodes de vérification par preuve, il faut les inscrire avec pragmatisme en complémentarité avec les autres outils du génie logiciel et les appliquer dans les situations où leur apport est le plus évident.

La vérification de programmes par preuve fait partie des méthodes du génie logiciel, mais elle n'occupe pas encore toute la place qu'elle mérite, si l'on compare son degré d'utilisation avec son potentiel. Lorsqu'elle est connue, elle est souvent considérée comme trop coûteuse et trop technique pour être appliquée à de "vrais" programmes. Il est vrai qu'elle est difficile à appliquer aux langages de programmation les plus populaires, dont la complexité logique est grande.

Cependant, de nombreux signes laissent penser que la situation évolue favorablement et que la preuve de programmes sera de plus en plus enseignée et utilisée. Son degré d'automatisation s'est considérablement accru ces dernières décennies, par la découverte de nouvelles procédures de décision et par la combinaison et l'optimisation de procédures existantes. Similairement, l'accessibilité de la preuve interactive s'accroît également, par l'amélioration des langages de tactiques des assistants de preuve. Enfin, on voit émerger des environnements de preuve plus conviviaux, mieux intégrés dans les environnements de développement et couplés avec d'autres outils d'analyse statique ou dynamique.

Mon premier projet de recherche est de poursuivre mes contributions à cet élan de popularisation de la preuve de programmes, à travers les actions suivantes :

Combinaison d'analyses

Nous avons montré comment le slicing et le test pouvaient faciliter la preuve de programmes ou de théorèmes [CKGJ12, PKB+16, DGG16]. Je souhaite poursuivre l'effort d'élaboration de telles combinaisons, selon trois directions :

1. Formalisation de ces combinaisons pour démontrer leur correction.
2. Étude de l'apport d'autres analyses dynamiques ou statiques, comme l'interprétation abstraite.
3. Diffusion de ces méthodes par leur intégration dans les environnements de développement et d'analyse les plus utilisés, comme eclipse, Frama-C ou Coq.

Conception de procédures de décision

Nous avons vu comment la vérification de systèmes paramétrés se ramène à la démonstration de la validité de certaines formules dans une certaine logique. Ainsi, les progrès dans l'automatisation de cette vérification sont liés à la découverte de nouvelles procédures de décision. Cette tâche est suffisamment ardue et l'enjeu est assez grand pour justifier une approche formelle. Nous recommandons d'utiliser des méthodes formelles pour étudier ces procédures.

Comme détaillé dans la partie 2.5, j'ai suivi cette recommandation pour étudier les procédures de décision fondées sur le calcul de superposition. J'envisage de faire de même avec d'autres familles de procédures de décision, par exemple pour la théorie des tableaux, sous-jacente à de nombreux algorithmes et programmes dont la correction mérite d'être démontrée.

Lorsque certaines spécifications ne sont pas exprimables en logique du premier ordre, ou lorsque la vérification automatique de certaines propriétés est problématique, il est possible d'utiliser des prouveurs interactifs, aux possibilités plus étendues. D'autres extensions à la logique du premier ordre incluent des fonctions récursives et des types de données algébriques (dans Dafny [Lei10] et Why3 [FP13] par exemple) ainsi que du polymorphisme [BP11] et des prédicats inductifs [BFMP11].

7.2.2 Applications scientifiques

Parmi les nombreuses applications des méthodes formelles du génie logiciel (comme la sécurité) je souhaite approfondir les applications scientifiques, soit dans une autre branche de l'informatique que le génie logiciel (comme les systèmes distribués), soit dans une autre discipline scientifique que l'informatique. Le *calcul formel* est un exemple de méthode formelle mise à la disposition de toutes les sciences, avec succès, depuis de nombreuses années. Je souhaite contribuer à l'utilisation scientifique d'autres méthodes formelles, notamment celles fondées sur la logique, beaucoup moins répandues.

Depuis janvier 2012, mon laboratoire, le LIFC, a intégré l'institut FEMTO-ST, sous la forme du département DISC (Département d'Informatique des Systèmes Complexes). Dans ce cadre j'ai participé aux applications suivantes :

1. Modélisation par des systèmes multi-agents d'une micro-usine composée de cellules de production de micro-composants, en collaboration avec une équipe du département d'automatique de FEMTO-ST [BBGL08].
2. Dans le projet ANR "Smart surface" de l'ACI Systèmes Interactifs et Robotique, modélisation de l'interaction entre un micro-objet et une surface de convoyage constituée d'une matrice de MEMS, **paramétrée** par le nombre de cellules de la matrice [GHT10].
3. Transfert du concept et d'une librairie de stratégies de réécriture, pour un logiciel d'automatisation des méthodes multi-échelles pour les matrices de MEMS, combinant des méthodes de calcul formelles et numériques [BGL14]. Ce logiciel est développé dans l'équipe Temps-Fréquence de FEMTO-ST.
4. Partage de mes connaissances en combinatoire des cartes avec Michel Planat, professeur de physique du département MN2S de FEMTO-ST, qui nous a permis de révéler une connexion nouvelle entre hypercartes et physique quantique [PGHS15].

Je souhaite poursuivre et intensifier ma participation aux deux derniers sujets, qui offrent toujours l'opportunité d'appliquer d'autres compétences en méthodes formelles et combinatoire des cartes, plus généralement en génie logiciel et mathématiques discrètes.

Les clés du succès de ces activités pluri-disciplinaires sont une bonne compréhension mutuelle du domaine et de ses problématiques, et des délais suffisants pour acquérir cette compréhension. Or, je possède depuis ma thèse cette compréhension sur un autre sujet scientifique, à l'interface entre les mathématiques appliquées et l'informatique : la combinatoire énumérative et bijective. Ainsi, sur ce sujet, je peux sans risque pratiquer la pluri-disciplinarité en autonomie, en jouant alternativement les rôles de chercheur en génie logiciel et combinatoire. En partenariat avec d'autres spécialistes, j'ai montré depuis 2010 diverses manières d'appliquer les méthodes formelles en combinatoire [Gio10, GS12, ZG15, DGG16].

Ces expériences m'ont pleinement convaincu de l'originalité de cette démarche et de son importance pour relever les défis de raisonnement et de calcul rigoureux. J'envisage de poursuivre ces applications, pour obtenir beaucoup d'autres résultats similaires, et d'y associer d'autres chercheurs, comme les combinatoriciens du laboratoire LE2I de Dijon, membres de la même nouvelle communauté d'universités appelée "Université de Bourgogne Franche-Comté".

7.2.3 Résumé

Les deux projets ci-dessus sont fondés sur la même culture en informatique théorique, logique et combinatoire, que j'ai acquise et développée depuis 1994. J'opère déjà des transferts d'expérience entre diverses thématiques. Je souhaite intensifier ces transferts, les systématiser et les diffuser en produisant des documents scientifiques de synthèse.

Je souhaite aussi prochainement diriger des étudiants en thèse, sur des sujets dans les thématiques ci-dessus. Je suis plus que jamais motivé par les collaborations avec mes collègues de l'institut FEMTO-ST et avec mes confrères, sur le plan national et international, en particulier à travers des montages de projets. De plus, je souhaite accompagner la plupart de mes travaux par des développements logiciels d'une plus large portée, afin d'en faire profiter la communauté.

Chapitre 8

Bibliographie

[AB07] A. W. Appel and S. Blazy. Separation logic for small-step Cminor. In *TPHOL'07 (Theorem Proving in Higher Order Logics)*, pages 5–21. Springer, 2007.

[ABB+14] W. Ahrendt, B. Beckert, D. Bruns, R. Bubel, C. Gladisch, S. Grebing, R. Hähnle, M. Hentschel, M. Herda, V. Klebanov, W. Mostowski, C. Scheben, P. H. Schmitt, and M. Ulbrich. The KeY platform for verification and analysis of Java programs. In *VSTTE'14 (Verified Software : Theories, Tools, and Experiments)*, volume 8471 of *LNCS*, pages 55–71. Springer, 2014.

[Abr96] J.-R. Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, 1996.

[AG97a] D. Arquès and A. Giorgetti. Énumération des cartes pointées de genre quelconque en fonction des nombres de sommets et de faces. Rapport de recherche, Institut Gaspard Monge, 1997.

[AG97b] D. Arquès and A. Giorgetti. Une bijection géométrique entre une famille d'hypercartes et une famille de polygones énumérées par la série de Schröder. In *FPSAC'97 (Formal Power Series and Algebraic Combinatorics)*, pages 14–25, 1997.

[AG98] D. Arquès and A. Giorgetti. Counting rooted maps on a surface. Rapport de recherche, Institut Gaspard Monge, 1998.

[AG99] D. Arquès and A. Giorgetti. Énumération des cartes pointées de genre quelconque en fonction des nombres de sommets et de faces. *Journal of Combinatorial Theory Series B*, 77(1) :1–24, 1999.

[AG00a] D. Arquès and A. Giorgetti. Counting rooted maps on a surface. *Theoretical Computer Science*, 234 :255–272, 2000.

[AG00b] D. Arquès and A. Giorgetti. Une bijection géométrique entre une famille d'hypercartes et une famille de polygones énumérées par la série de Schröder. *Discrete Mathematics*, 217 :17–32, 2000.

[ARR01] A. Armando, S. Ranise, and M. Rusinowitch. Uniform derivation of decision procedures by superposition. In *CSL'01 (Computer Science Logic)*, volume 2142 of *LNCS*, pages 513–527. Springer, 2001.

[ARR03] A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Information and Computation*, 183(2) :140–164, 2003.

[Bak80] J. W. de Bakker. *Mathematical Theory of Program Correctness*. Prentice-Hall, 1980.

[BBCD15] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Plug-in Manual, version 0.9 for Magnesium-20151002*, 2015. http://frama-c.com/download/wp-manual-Magnesium-20151002.pdf.

[BBGL08] T. Brocard, F. Bouquet, A. Giorgetti, and C. Lang. Agent based modelling of complex systems with AML and the situation calculus. In *ABS2, Int. workshop on Agent Based Spatial Simulation*, 2008.

[BC04] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development. Coq'Art : the Calculus of Inductive Constructions*. Springer, 2004.

[BCDG07] F. Bouquet, J.-F. Couchot, F. Dadeau, and A. Giorgetti. Instantiation of parameterized data structures for model-based testing. In *B'07 (B Conference)*, volume 4355 of *LNCS*, pages 96–110. Springer, 2007.

[BCF+13] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2013. http://frama-c.com/acsl.html.

[Ber09] Y. Bertot. Theorem proving support in programming language semantics. In *From Semantics to Computer Science - Essays in Honour of Gilles Kahn*, pages 337–361. Cambridge University Press, 2009.

[BFL+11] M. Barnett, M. Fähndrich, K. Rustan M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification : The Spec# experience. *Commun. ACM*, 54(6) :81–91, 2011.

[BFM+13] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 platform 0.81, tutorial and reference manual*, 2013. https://hal.inria.fr/hal-00822856.

[BFMP11] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3 : Shepherd your herd of provers. In *Boogie'11 (International Workshop on Intermediate Verification Languages)*, pages 53–64, 2011.

[BG98] J.-F. Béraud and A. Giorgetti. MAP : un package Maple pour compter les cartes pointées. Rapport de recherche 98-R-357, LORIA, 1998.

[BG11] W. Belkhir and A. Giorgetti. Lazy rewriting modulo associativity and commutativity. In *WRS'11 (Workshop on Reduction Strategies in Rewriting and Programming)*, pages 17–21, 2011.

[BGGP16] J.-L. Baril, R. Genestier, A. Giorgetti, and Armen Petrossian. Rooted planar maps modulo some patterns. *Discrete Mathematics*, 339 :1199–1205, 2016.

[BGL+12] W. Belkhir, A. Giorgetti, M. Lenzner, R. N. Dhara, and B. Yang. Rewriting strategies for a two-scale method : Application to combined thin and periodic structures. In *dMEMS'12 (Workshop on Design, Control and Software Implementation for Distributed MEMS)*, pages 82–89. IEEE Computer Society, 2012.

[BGL14] W. Belkhir, A. Giorgetti, and M. Lenczner. A symbolic transformation language and its application to a multiscale method. *Journal of Symbolic Computation*, 65 :49–78, 2014.

[BH08] D. Bjorner and M. Henson. *Logics of Specification Languages*. Springer, 2008.

[BHS07] B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of object-oriented software : The KeY approach*, volume 4334 of *LNCS*. Springer, 2007.

[BJLM15] S. Boldo, J.-H. Jourdan, X. Leroy, and G. Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2) :135–163, 2015.

[BL09] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3) :263–288, 2009.

[Bla08] S. Blazy. *Sémantiques formelles*. Habilitation à diriger les recherches, Université Évry Val d'Essone, 2008.

[Bli81] A. Blikle. The clean termination of iterative programs. *Acta Informatica*, 16(2) :199–217, 1981.

[BMS06] A. R. Bradley, Z. Manna, and H. B. Sipma. What's Decidable About Arrays ? In *VMCAI'06 (Verification, Model Checking, and Abstract Interpretation)*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[BP11] F. Bobot and A. Paskevich. Expressing polymorphic types in a many-sorted language. In *FroCoS'11 (Frontiers of Combining Systems)*, volume 6989 of *LNAI*, pages 87–102. Springer, 2011.

[BT07] C. Barrett and C. Tinelli. CVC3. In *CAV'07*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[BT12] C. Barrett and C. Tinelli. CVC4, 2012. http://cvc4.cs.nyu.edu/web/.

[CDD+04] J.-F. Couchot, F. Dadeau, D. Déharbe, A. Giorgetti, and S. Ranise. Proving and debugging set-based specifications. In *WMF'03 (Brazilian Workshop on Formal Methods)*, volume 95 of *Electronic Notes in Theoretical Computer Science*, pages 189–208, 2004.

[CDGR03] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Scalable automated proving and debugging of set-based specifications. *Journal of the Brazilian Computer Society*, 9(2) :17–36, 2003.

[CDGR04] J.-F. Couchot, D. Déharbe, A. Giorgetti, and S. Ranise. Barvey : Vérification automatique de consistance de machines abstraites B. In *AFADL'04 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 369–372, 2004.

[CG04] J.-F. Couchot and A. Giorgetti. Analyse d'atteignabilité déductive. In *AFADL'04 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 269–283, 2004.

[CG11] G. Cécé and A. Giorgetti. Simulations over two-dimensional on-line tessellation automata. In *DLT'11 (Developments in Language Theory)*, volume 6795 of *LNCS*, pages 141–152. Springer, 2011.

[CG15] M. Clochard and L. Gondelman. Double WP : Vers une preuve automatique d'un compilateur. In *JFLA'15 (Journées Francophones des Langages Applicatifs)*, 2015. https://hal.inria.fr/hal-01094488.

[CGK05] J.-F. Couchot, A. Giorgetti, and N. Kosmatov. A uniform deductive approach for parameterized protocol safety. In *ASE'05 (Automated Software Engineering)*, pages 364–367. IEEE Computer Society, 2005.

[CGK+12] S. Conchon, A. Goel, S. Krstić, A. Mebsout, and F. Zaïdi. Cubicle : A parallel SMT-based model checker for parameterized systems. In *CAV'12 (Computer Aided Verification)*, volume 7358 of *LNCS*, pages 718–724. Springer, 2012.

[CGS09] J.-F. Couchot, A. Giorgetti, and N. Stouls. Graph-based reduction of program verification conditions. In *AFM'09 (Automated Formal Methods (colocated with CAV'09))*, pages 40–47. ACM Press, 2009. http://hal.inria.fr/inria-00402204.

[Che11] O. Chebaro. *Classification of errors threats by static analysis, program sclicing and structural testing of programs.* Thèse de doctorat, Université de Franche-Comté, 2011.

[CKGJ10a] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Combining Frama-C and PathCrawler for C program debugging. In *GDR GPL'10 (journées nationales du Groupement de recherche CNRS du Génie de la programmation et du logiciel)*, pages 217–218, 2010. Résumé étendu.

[CKGJ10b] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Combining static analysis and test generation for C program debugging. In *TAP'10 (Tests and Proofs)*, volume 6143 of *LNCS*, pages 94–100. Springer, 2010.

[CKGJ11] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. The SANTE tool : Value analysis, program slicing and test generation for C program debugging. In *TAP'11 (Tests and Proofs)*, volume 6706 of *LNCS*, pages 78–83. Springer, 2011.

[CKGJ12] O. Chebaro, N. Kosmatov, A. Giorgetti, and J. Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *SAC'12 (Symposium On Applied Computing)*, pages 1284–1291. ACM, 2012.

[CMc16] U. Cambridge, T. U. München, and contributors. Isabelle, 2016. http://isabelle.in.tum.de/.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM J. Comput.*, 7(1) :70–90, 1978.

[CR06] L. A. Clarke and D. S. Rosenblum. A historical perspective on runtime assertion checking in software development. *SIGSOFT Softw. Eng. Notes*, 31(3) :25–37, 2006.

[CS12] L. Correnson and J. Signoles. Combining analyses for C program verification. In *FMICS'12 (Formal Methods for Industrial Case Studies)*, volume 7437 of *LNCS*, pages 108–130. Springer, 2012.

[CSW15] D. R. Cok, A. Stump, and T. Weber. The 2013 evaluation of SMT-COMP and SMT-LIB. *Journal of Automated Reasoning*, 55(1) :61–90, 2015.

[Dad06] F. Dadeau. *Évaluation symbolique à contraintes pour la validation – Application à Java/JML*. Thèse de doctorat, Université de Franche-Comté, 2006.

[DG03] F. Dadeau and A. Giorgetti. Vérification de machines abstraites B en logique monadique du second ordre. Rapport de recherche RR2003-01, LIFC, 2003.

[DGG16] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In *TAP'16 (Tests and Proofs)*, volume 6792 of *LNCS*, pages 57–75. Springer, 2016.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[Dij76] E. W. Dijkstra. *A Discipline of Programming*. In : Series in Automatic Computation. Prentice Hall, 1976.

[DMB08] L. De Moura and N. Björner. Z3 : An efficient SMT solver. In *TACAS'08 (Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[DMS+09] M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC : Contract-based modular verification of concurrent C. In *ICSE'09 (International Conference on Software Engineering)*, pages 429–430. IEEE Computer Society, 2009.

[DR02] D. Déharbe and S. Ranise. BDD-driven first-order satisfiability procedures (extended version). Rapport de recherche 4630, LORIA, 2002.

[Dut14] B. Dutertre. Yices 2.2. In *CAV'14 (Computer Aided Verification)*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[EBODG10] I. Enderlin, A. Ben Othman, F. Dadeau, and A. Giorgetti. Realistic domains for unit tests generation. Rapport de recherche RR2010-01, LIFC, 2010.

[EDGB12] I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet. Grammar-based testing using realistic domains in PHP. In *A-MOST'12 (Advances in Model-Based Testing), workshop ICST*, pages 509–518. IEEE Computer Society, 2012.

[EDGO11] I. Enderlin, F. Dadeau, A. Giorgetti, and A. Ben Othman. Praspel : A specification language for contract-based testing in PHP. In *ICTSS'11 (Int. Conf. on Testing Software and Systems)*, volume 7019 of *LNCS*, pages 64–79. Springer, 2011.

[EGB13] I. Enderlin, A. Giorgetti, and F. Bouquet. A constraint solver for PHP arrays. In *CSTVA'13 (Constraints in Software Testing, Verification and Analysis), workshop ICST*, pages 218–223. IEEE, 2013.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Symposia in Applied Mathematics*, volume 19, pages 19–32, 1967.

[FP13] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *ESOP'13 (European Symposium on Programming)*, volume 7792 of *LNCS*, pages 125–128. Springer, 2013.

[Fra92] N. Francez. *Program verification*. International computer science series. Addison-Wesley, 1992.

[Fra96] R. Fraer. Tracing the origins of verification conditions. In *AMAST'96 (Algebraic Methodology and Software Technology)*, pages 241–255. Springer, 1996.

[Gen16a] R. Genestier. Vérification formelle de programmes de génération de données structurées. In *AFADL'16 (Approches Formelles dans l'Assistance au Développement Logiciel)*, pages 67–71, 2016. Résumé de thèse, http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf.

[Gen16b] R. Genestier. *Vérification formelle de programmes de génération de données structurées*. Thèse de doctorat, Université de Franche-Comté, 2016.

[GG06a] A. Giorgetti and J. Groslambert. JAG : Génération d'annotations JML pour vérifier des propriétés temporelles. In *AFADL'06 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, 2006. Session outils.

[GG06b] A. Giorgetti and J. Groslambert. JAG : Génération d'annotations JML pour vérifier des propriétés temporelles. Rapport de recherche RT2006-02, LIFC, 2006.

[GG06c] A. Giorgetti and J. Groslambert. JAG : JML Annotation Generation for verifying temporal properties. In *FASE'06 (Fundamental Approaches to Software Engineering)*, volume 3922 of *LNCS*, pages 373–376. Springer, 2006.

[GG07] A. Giorgetti and J. Groslambert. Un programme annoté en vaut deux. In *JFLA'07 (Journées francophones des langages applicatifs)*, pages 87–101. INRIA, 2007.

[GG16] R. Genestier and A. Giorgetti. Spécification et vérification formelle d'opérations sur les permutations. In *AFADL'16 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 72–78. FEMTO-ST, 2016. http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf.

[GGJK08] A. Giorgetti, J. Groslambert, J. Julliand, and O. Kouchnarenko. Verification of class liveness properties with Java modeling language. *IET Software*, 2(6) :500–514, 2008.

[GGP15a] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In *JFLA'15 (Journées Francophones des Langages Applicatifs)*. INRIA, 2015. https://hal.inria.fr/hal-01099135.

[GGP15b] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP'15 (Tests and Proofs)*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[GH93] J. V. Guttag and J. J. Horning, editors. *Larch : Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer, 1993. With S. J. Garland, K. D. Jones, A. Modet, and J. M. Wing.

[GHT10] A. Giorgetti, A. Hammad, and B. Tatibouët. Using SysML for smart surface modeling. In *dMEMS'10 (workshop on design, control and software implementation for distributed MEMS)*, pages 100–107. IEEE, 2010.

[Gio98a] A. Giorgetti. *Combinatoire bijective et énumérative des cartes pointées sur une surface*. Thèse de doctorat, Université de Marne-la-Vallée, Institut Gaspard Monge, 1998.

[Gio98b] A. Giorgetti. Maple V release 5 pour Windows 95, analyse de logiciel. *Pour la Science*, 249 :110–111, 1998.

[Gio03] A. Giorgetti. An asymptotic study for path reversal. *Theoretical Computer Science*, 299(1-3) :585–602, 2003.

[Gio10] A. Giorgetti. Guessing a conjecture in enumerative combinatorics and proving it with a computer algebra system. In *SCSS'10 (Symposium on Symbolic Computation in Software Science)*, pages 5–18, 2010. https://hal.archives-ouvertes.fr/hal-00563330.

[Gor88] M. C. J. Gordon. *Programming Language Theory and its Implementation*. Prentice-Hall, 1988.

[GR09] S. Ghilardi and S. Ranise. Goal-directed invariant synthesis for model checking modulo theories. In *TABLEAUX (Automated Reasoning with Analytic Tableaux and Related Methods)*, volume 5607 of *LNCS*, pages 173–188. Springer, 2009.

[GR10] S. Ghilardi and S. Ranise. Backward reachability of array-based systems by SMT solving : Termination and invariant synthesis. *Logical Methods in Computer Science*, volume 6, issue 4, 2010.

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[Gro07] J. Groslambert. *Vérification de propriétés temporelles par génération d'annotations*. Thèse de Doctorat, Université de Franche-Comté, 2007.

[GS12] A. Giorgetti and V. Senni. Specification and validation of algorithms generating planar Lehman words. In *GASCom'12 (random generation of combinatorial structures)*, 2012. https://hal.inria.fr/hal-00753008.

[Her13] P. Herms. *Certification of a tool chain for deductive program verification*. Thèse de doctorat, Université Paris Sud - Paris XI, 2013.

[HH98] C. A. R. Hoare and J. He. *Unifying theories of programming*. Prentice Hall, 1998.

[HMM12] P. Herms, C. Marché, and B. Monate. *A certified multi-prover verification condition generator*, pages 2–17. Springer, 2012.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10) :576–580, 1969.

[Hoa02] C. A. R. Hoare. Assertions : A personal perspective. In *Software Pioneers : Contributions to Software Engineering*, pages 356–366. Springer, 2002.

[Hoa03] T. Hoare. The verifying compiler : A grand challenge for computing research. In *JMLC'03 (Joint Modular Languages Conference)*, pages 25–35. Springer, 2003.

[ILL73] S. Igarashi, R. L. London, and D. C. Luckham. Automatic program verification I : A logical basis and its implementation. Rapport de recherche RR-73-11, Stanford Artificial Intelligence Laboratory, 1973.

[KKP+12] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a software analysis perspective. *Formal Aspects of Computing*, 27(3) :573–609, 2012.

[KM04] M. Kaufmann and J. S. Moore. The ACL2 home page, 2004. http://www.cs.utexas.edu/users/moore/acl2/.

[KPS13] N. Kosmatov, G. Petiot, and J. Signoles. An optimized memory monitoring for runtime assertion checking of C programs. In *RV'13 (Runtime Verification)*, volume 8174 of *LNCS*, pages 167–182. Springer, 2013.

[LB08] X. Leroy and S. Blazy. Formal verification of a C-like memory model and its uses for verifying program transformations. *Journal of Automated Reasoning*, 41(1) :1–31, 2008.

[LBR06] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML : A behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3) :1–38, 2006.

[Lei10] K. R. M. Leino. Dafny : An automatic program verifier for functional correctness. In *LPAR'10 (Logic for Programming, Artificial intelligence, and Reasoning)*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.

[Ler09a] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7) :107–115, 2009.

[Ler09b] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4) :363–446, 2009.

[Ler13] X. Leroy. Mechanized semantics. École d'été VSTA (*Verification Technology, Systems & Applications*), http://gallium.inria.fr/~xleroy/courses/VTSA-2013/notes.pdf, 2013.

[Les07] P. Lescanne. Différentes sémantiques d'un langage impératif (support de cours). http://perso.ens-lyon.fr/pierre.lescanne/ENSEIGNEMENT/PROG2/06-07/IMP.pdf, 2007.

[LG86] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[LG09] X. Leroy and H. Grall. Coinductive big-step operational semantics. *Information and Computation*, 207(2) :284–304, 2009.

[LGH+78] R. L. London, J. V. Guttag, J. J. Horning, B. W. Lampson, J. G. Mitchell, and G. J. Popek. Proof rules for the programming language Euclid. *Acta Informatica*, 10(1) :1–26, 1978.

[LM02] C. Lynch and B. Morawska. Automatic decidability. In *LICS'02 (Logic in Computer Science)*, pages 7–16. IEEE Computer Society, 2002.

[LO13] LRI/VALS and OCamlPro. The Alt-Ergo SMT solver, 2013. http://alt-ergo.lri.fr.

[Meb14] A. Mebsout. *Invariants inference for model checking of parameterized systems*. Thèse de doctorat, Université Paris Sud - Paris XI, 2014.

[Mey97] B. Meyer. *Object-oriented software construction, 2^{nd} edition*. Prentice Hall, 1997.

[MG11] A. Mednykh and A. Giorgetti. Enumeration of genus four maps by number of edges. *Ars Mathematica Contemporanea*, 4 :351–361, 2011.

[MNM87] B. Meyer, J.-M. Nerson, and M. Matsuo. EIFFEL : Object-oriented design for software engineering. In *ESEC'87 (European Software Engineering Conference)*, volume 289 of *LNCS*, pages 221–229. Springer, 1987.

[Mor98] C. C. Morgan. *Programming from specifications, 2^{nd} edition*. International series in computer science. Prentice Hall, 1998.

[MP95] Z. Manna and A. Pnueli. *Temporal verification of reactive systems : safety*. Springer, 1995.

[MP08] J. Meng and L. C. Paulson. Translating higher-order clauses to first-order clauses. *Journal of Automated Reasoning*, 40(1) :35–60, 2008.

[MT13] C. Marché and A. Tafat. Calcul de plus faible précondition, revisité en Why3. In *JFLA'13 (Journées Francophones des Langages Applicatifs)*. INRIA, 2013. http://hal.inria.fr/hal-00778791.

[Nip98] T. Nipkow. Winskel is (almost) right : Towards a mechanized semantics textbook. *Formal Aspects of Computing*, 10(2) :171–186, 1998.

[NK14] T. Nipkow and G. Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6 :319–340, 1976.

[ORR+96] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS : Combining specification, proof checking, and model checking. In *CAV'96 (Computer Aided Verification)*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.

[PAC+17] B. C. Pierce, A. Azevedo de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, A. Tolmach, and B. Yorgey. *Software foundations*, volume 2 : Programming Language Foundations. 2017. https://softwarefoundations.cis.upenn.edu/.

[PBJ+14] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. Instrumentation of annotated C programs for test generation. In *SCAM'14 (Source Code Analysis and Manipulation)*, pages 105–114. IEEE, 2014.

[Pet15] G. Petiot. *Contribution à la vérification de programmes C par combinaison de tests et de preuves.* Thèse de doctorat, Université de Franche-Comté, 2015.

[PGHS15] M. Planat, A. Giorgetti, F. Holweck, and M. Saniga. Quantum contextual finite geometries from dessins d'enfants. *Int. J. of Geometric Methods in Modern Physics*, 2015.

[PKB+16] G. Petiot, N. Kosmatov, B. Botella, A. Giorgetti, and J. Julliand. Your proof fails ? testing helps to find the reason. In *TAP'16 (Tests and Proofs)*, volume 6792 of *LNCS*, pages 130–150. Springer, 2016.

[PKGJ14a] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand. Comment la génération de tests facilite la spécification et la vérification déductive des programmes dans Frama-C. In *AFADL'14 (Approches Formelles dans l'Assistance au Développement de Logiciels)*, pages 133–133, 2014. Résumé étendu de [PKGJ14b]. http://afadl2014.lacl.fr/actes_AFADL2014-HAL.pdf.

[PKGJ14b] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand. How test generation helps software specification and deductive verification in Frama-C. In *TAP'14 (Tests and Proofs)*, volume 8570 of *LNCS*, pages 204–211. Springer, 2014.

[Plo04] G. D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60–61 :17–139, 2004.

[Rey98] J. C. Reynolds. *Theories of Programming Languages, 1st edition.* Cambridge University Press, 1998.

[SP17] N. Stouls and V. Prevosto. *Aoraï Plugin Tutorial*, 2017. https://frama-c.com/download/frama-c-aorai-manual.pdf.

[TGG01] M. Tréhel, P. Gradit, and A. Giorgetti. Performances d'un algorithme distribué d'exclusion mutuelle en cas de non-équiprobabilité des requêtes des processus. *RSRCP (Réseaux et Systèmes Répartis, Calculateurs Parallèles), Numéro spécial Evaluation quantitative des performances des réseaux et systèmes*, 13(6) :557–573, 2001.

[TGK09] E. Tushkanova, A. Giorgetti, and O. Kouchnarenko. Specifying and proving a sorting algorithm. Rapport de recherche RR2009-03, LIFC, 2009.

[TGMK09] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Modular specification of Java programs. Rapport de recherche RR-7097, INRIA, 2009.

[TGMK10] E. Tushkanova, A. Giorgetti, C. Marché, and O. Kouchnarenko. Specifying generic Java programs : Two case studies. In *LDTA'10 (Language Descriptions, Tools and Applications)*, pages 92–106. ACM, 2010.

[TGRK12] E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based framework for building superposition-based decision procedures. In *RLA'12 (Rewriting Logic and its Applications)*, volume 7571 of *LNCS*, pages 221–239. Springer, 2012.

[TGRK15] E. Tushkanova, A. Giorgetti, C. Ringeissen, and O. Kouchnarenko. A rule-based system for automatic decidability and combinability. *Science of Computer Programming*, 99 :3–23, 2015.

[TIO17] TIOBE : The Importance of Being Earnest. TIOBE Programming Community index, 2017. https://www.tiobe.com/tiobe-index/.

[TRGK13] E. Tushkanova, C. Ringeissen, A. Giorgetti, and O. Kouchnarenko. Automatic decidability for theories with counting operators. In *ADDCT'13 (Automated Deduction : Decidability, Complexity, Tractability)*, 2013.

[TZ88] J. V. Tucker and J. I. Zucker. *Program Correctness over Abstract Data Types, with Error-state Semantics.* Elsevier, 1988.

[Ver17] Équipe VeriDis. veriT - An open, trustable and efficient SMT-prover, 2017. http://www.verit-solver.org/.

[Web11] T. Weber. SMT solvers : New oracles for the HOL theorem prover. *Int. J. Softw. Tools Technol. Transf.*, 13(5) :419–429, 2011.

[WG14] T. R. S. Walsh and A. Giorgetti. Efficient enumeration of rooted maps of a given orientable genus by number of faces and vertices. *Ars Mathematica Contemporanea*, 7 :263–280, 2014.

[WGM12] T. R. S. Walsh, A. Giorgetti, and A. Mednykh. Enumeration of unrooted orientable maps of arbitrary genus by number of edges and vertices. *Discrete Mathematics*, 312(17) :2660 – 2671, 2012.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. MIT Press, 1993.

[YBD+11] B. Yang, W. Belkhir, R. N. Dhara, M. Lenczner, and A. Giorgetti. Computer-aided multiscale model derivation for MEMS arrays. In *EuroSimE'11 (Int. Conf. on Thermal, Mechanical and Multi-Physics Simulation and Experiments in Microelectronics and Microsystems)*. IEEE Computer Society, 2011.

[YDB+11] B. Yang, R. N. Dhara, W. Belkhir, M. Lenczner, and A. Giorgetti. Formal methods for multiscale models derivation. In *CFM'11 (Congrès Français de Mécanique)*, 2011.

[ZG15] N. Zeilberger and A. Giorgetti. A correspondence between rooted planar maps and normal planar lambda terms. *Logical Methods in Computer Science*, volume 11, issue 3, 2015.

école doctorale sciences pour l'ingénieur et microtechniques SPIM

Titre : Spécification et vérification de systèmes paramétrés

Mots clés : Méthodes formelles, programmation impérative, spécification fonctionnelle, méthode déductive, sémantique formelle.

Résumé : Ce mémoire résume mes travaux de recherche jusqu'en 2016, dans le domaine des méthodes formelles de spécification et de vérification de modèles et de programmes.
Mes premiers travaux ont porté sur la vérification et la synthèse d'invariants de systèmes distribués, paramétrés par le nombre de processus exécutés en parallèle. Puis j'ai étudié la correction de programmes impératifs par rapport à des propriétés temporelles et des spécifications fonctionnelles formelles. Plus récemment, j'ai outillé une méthode d'aide à la conception de procédures de décision, appliqué la vérification déductive à des algorithmes de combinatoire énumérative, et adapté des stratégies de réécriture à la formalisation de méthodes multi-échelles.
Une part importante de mes travaux porte sur la méthode de vérification dite déductive, qui formalise la correction d'un système par une formule logique, qui doit ensuite être démontrée rigoureusement. Le problème général de vérification de ces formules est indécidable.
Trois niveaux de finesse sont distingués dans l'analyse de la méthode déductive, pour étendre ses résultats de décidabilité et accroître l'automaticité des outils de preuve de programmes. Pour contribuer au niveau d'analyse le plus fin, ce mémoire explicite la traduction logique de la correction partielle de programmes impératifs annotés par des assertions et des invariants de boucle. Cette traduction est présentée comme une sémantique formelle d'un langage simple de programmes impératifs annotés.

Title : Specification and verification of parameterized systems

Keywords : Formal methods, imperative programming, functional specification, deductive method, formal semantics.

Abstract : This document summarizes my research until 2016, in the domain of formal methods of specification and verification of models and programs.
My first work focused on the verification and synthesis of invariants for distributed systems parameterized by the number of processes run in parallel. Then I studied the correction of imperative programs with respect to temporal properties and formal functional specifications. More recently, I have implemented a method to discover new decision procedures, applied deductive verification to enumerative combinatorics, and adapted rewriting strategies to the formalization of multiscale methods.
An important part of my work is about the deductive method, which formalizes the correction of a system by a logical formula, which must then be rigorously proved. The general problem of verifying these formulas is undecidable.
Three levels of fineness are distinguished in the deductive method analysis, to extend its decidability results and to increase the automaticity of program provers. To contribute to the finest level of analysis, this document details the logical translation of partial correction of imperative programs annotated by assertions and loop invariants. This translation is presented as a formal semantics of a simple language of annotated imperative programs.

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

Université Bourgogne Franche-Comté
32, avenue de l'Observatoire
25000 Besançon