CNRS - INP - UT3 - UT1 - UT2J
Institut de Recherche en Informatique de Toulouse

L'humain et son environnement au cœur de l'Informatique

ACTES DES 20ÈMES JOURNÉES

APPROCHES FORMELLES DANS L'ASSISTANCE AU DÉVELOPPEMENT DE LOGICIELS AFADL 2021

Édités par David Delahaye et Ileana Ober

16 - 18 Juin 2021

ISBN : 978-2-917490-31-0

AVANT PROPOS

Initié il y a plus d'une vingtaine d'années, l'atelier Approches Formelles dans l'Assistance au Développement de Logiciel arrive à sa 20ème édition et se propose de continuer à rassembler des acteurs académiques et industriels intéressés par la mise en œuvre des techniques formelles aux divers stades du développement des logiciels et/ou des systèmes. C'est pour cela que notre atelier a toujours été organisé conjointement avec d'autres ateliers, journées ou conférences, afin de permettre à notre communauté de se rassembler le plus largement possible.

Pour sa 20ème édition, l'atelier AFADL est organisé en même temps que les Journées Nationales du GDR GPL (Groupement de Recherche Génie de la Programmation et du Logiciel), qui devaient avoir lieu à Vannes en juin 2021. Comme l'année dernière, la crise sanitaire due à la pandémie de Covid-19 a bouleversé l'organisation des journées du GDR GPL, ainsi que celles de AFADL, qui ont été maintenues mais se tiendront entièrement en distanciel. Ainsi, les journées de AFADL 2021 auront lieu du 16 au 18 juin 2021 en virtuel.

Cette année, nous avons reçu 11 soumissions et 10 articles ont été retenus pour apparaître dans ces actes. Ces articles seront présentés lors des journées, ainsi que 6 des 8 articles reçus et sélectionnés l'année dernière pour AFADL 2020, mais qui n'avaient pas pu être présentés suite à l'annulation des journées en 2020.

Malgré les conditions si particulières ces deux dernières années, nous sommes contents d'avoir pu maintenir une activité au sein de la communauté AFADL. Même si les journées ont été annulées l'année dernière, le maintien du processus de relecture a permis d'assurer la continuité de cette activité. Cette année, il nous est apparu essentiel de maintenir l'organisation des journées, qui constituent un forum d'échanges inestimable pour notre communauté et pour nos jeunes chercheurs en particulier. Comme le montrent ces actes, des résultats de recherche fort intéressants ont émergé malgré le contexte compliqué et nous vous invitons à les découvrir.

Pour finir, nous profitons de cette occasion pour remercier les membres de notre comité de programme pour leur attentif travail de relecture et pour avoir accepté de travailler dans ces circonstances si particulières pendant deux années.

Bonne lecture à tous et nous espérons vous voir nombreux aux journées virtuelles de AFADL 2021,

David Delahaye

Ileana Ober

PRÉSIDENTS DU COMITÉ DE PROGRAMME

David Delahaye (LIRMM, Université de Montpellier)

Ileana Ober (IRIT, Université Toulouse 3)

COMITÉ DE PROGRAMME

Yamine Aït Ameur (IRIT, ENSEEIHT)

Béatrice Bérard (LIP6, Sorbonne Université)

Sandrine Blazy (University of Rennes 1 - IRISA)

David Deharbe (ClearSy System Engineering)

David Delahaye (LIRMM, Université de Montpellier)

Catherine Dubois (ENSIIE-Samovar)

Marc Frappier (Université de Sherbrooke)

Alain Giorgetti (FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté)

Olivier Hermant (MINES ParisTech)

Mathieu Jaume (Sorbonne Université - UPMC - CNRS LIP6 UMR)

Florent Kirchner (CEA LIST)

Régine Laleau (Paris Est Créteil University)

Thomas Lambolais (LGI2P)

Pascale Le Gall (CentraleSupelec)

Yves Ledru (Laboratoire d'Informatique de Grenoble - Université Grenoble Alpes)

Nicole Levy (Cedric, CNAM)

Delphine Longuet (Univ. Paris-Sud, LRI)

Micaela Mayero (LIPN, Université Paris 13)

Stephan Merz (Inria Nancy)

David Monniaux (CNRS / VERIMAG)

Ileana Ober (IRIT, Université Toulouse 3)

Iulian Ober (IRIT, Université Toulouse 2)

Ioannis Parissis (Univ. Grenoble Alpes - Grenoble INP)

Pascal Poizat (Université Paris Nanterre et LIP6)

Marc Pouzet (LIENS)

Jean-Baptiste Raclet (IRIT)

Vlad Rusu (INRIA)

Nicolas Stouls (CITI/INSA Lyon)

Laurent Voisin (Systerel)

Virginie Wiels (ONERA / DTIM)

TABLE DES MATIERES

Pas de Pannes, Pas d'Exploits: Vérification Automatique de Noyaux Embarqués 1
Olivier Nicole, Matthieu Lemerre, Sébastien Bardin, Xavier Rival

Vérification formelle de la sûreté d'un système contrôlé par réseaux de neurones 5
Arthur Clavière, Eric Asselin, Christophe Garion, Claire Pagetti

Méthodologie pour la validation d'exigences globales appliquée à la sécurité 8
Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, Pascale Le Gall

DEPS Studio : Un environnement intégré de modélisation et de résolution pour la synthèse de système 11
Pierre-Alain Yvars, Laurent Zimmer

Détection d'objectifs de test polluants pour les critères de flot de données 15
Thibault Martin, Nikolai Kosmatov, Virgile Prevosto, Matthieu Lemerre

Représentation de programmes Smala grâce à la théorie des bigraphes 18
Cécile Marcon, Nicolas Nalpon, Cyril Allignol, Celia Picard

Formally verified postpass scheduling with peephole optimization for AArch64 26
Léo Gourdin

Vérification d'une bibliothèque mathématique d'un autopilote avec Frama-C 31
Baptiste Pollien

La double précision suffit-elle à l'exascale ? 36
William Weens, Thibaud Vazquez-Gonzalez, Louise Ben Salem-Knapp

Taylor series revisited 44
Alexis Maffart, Xavier Thirioux

Pas de Pannes, Pas d'Exploits: Vérification Automatique de Noyaux Embarqués

Olivier Nicole[1,2], Matthieu Lemerre[1], Sébastien Bardin[1], and Xavier Rival[2,3]

[1]*Université Paris-Saclay, CEA List, Saclay, France*
[2]*Département d'informatique de l'ENS, ENS, CNRS, PSL University, Paris, France*
[3]*Inria, Paris, France*

La sûreté et la sécurité de la plupart des ordinateurs dépendent de son composant le plus critique, le noyau, qui fournit les mécanismes de protections centraux. Les pires défauts de code d'un noyau sont :

— Des *erreurs à l'exécution*, qui conduisent le système entier à tomber en panne. Dans du code machine, il n'y a pas de comportement indéfini (comme en C), mais l'exécution d'une instruction qui conduit le code du noyau à lever une exception matérielle (e.g. opcode illégal, division par zéro, erreur de protection mémoire) est assimilable à une erreur à l'exécution ;

— Des *escalades de privilèges*, où un attaquant contourne les *auto-protections du noyau* et prend le contrôle du système entier. Dans le cas d'hyperviseurs, cela correspond au cas où l'attaquant parvient à s'échapper de sa machine virtuelle.

La seule manière de garantir qu'un noyau n'est pas sujet à ces erreurs est de le vérifier entièrement à l'aide de méthodes formelles [1]. Vérifier manuellement un noyau en utilisant un assistant de preuve [1–6] ou de la vérification déductive [7–10] peut garantir qu'un noyau satisfait une spécification formelle, permettant ainsi d'atteindre un haut niveau de sûreté et de sécurité. Mais cet effort est hors d'atteinte pour la plupart des acteurs du monde de l'embarqué, à la fois à cause de l'effort titanesque que demande l'écriture de milliers de lignes de preuves (e.g. 200,000 pour seL4 [1] ou 100,000 pour CertiKOS [11]) mais également à cause de la difficulté de trouver des experts à la fois en système et en méthodes formelles. Pour ces acteurs, la méthode idéale serait une vérification entièrement automatique, où les développeurs fourniraient seulement leur code et, avec très peu ou pas du tout de configuration, l'outil vérifierait automatiquement la propriété visée. De plus, une vérification complète devrait aller jusqu'au niveau de l'exécutable binaire, car 1. une large part du code d'un noyau de système embarqué consiste en des interactions bas niveau avec le matériel, et 2. la chaîne de compilation (options de compilation, compilateur, assembleur, éditeur de lien) peut introduire des bugs.

Les méthodes de vérifications de noyau récentes, appelées méthodes "presse-bouton" [12–14] sont basées sur l'exécution symbolique, ce qui a plusieurs avan-

tages : la méthode est suffisamment précise pour analyser du code machine sans se perdre, et la méthode marche de manière native avec des formules logiques et peut donc facilement être utilisée pour démontrer des propriétés de haut niveau spécifiées à la main. D'un autre côté, l'exécution symbolique en tant que technique de vérification souffre de limitations sévères, comme l'impossibilité de gérer les boucles non bornées, le besoin de fournir à la main les invariants du noyau (pour CertiKOS^S [13] : 438 lignes de spécifications pour 1845 instructions, i.e. un ratio de 23.7%) et une difficulté à passer à l'échelle à cause de l'explosion du nombre de chemins. Ces limitations sont inhérentes à l'exécution symbolique et ne peuvent pas être adressées sans un changement radical de méthode de vérification.

Notre but est de repousser les limites de ce que peut faire une méthode de vérification automatique de noyau, afin de la rendre pratique pour des développeurs systèmes. Nous nous concentrons sur les systèmes embarqués, qui sont caractérisés par une allocation mémoire plutôt statique, et par le fait qu'ils existent en général en de multiples variantes (selon le matériel ou l'application utilisant le noyau), ce qui rend l'automatisation de l'analyse très importante. Nos contributions sont les suivantes :

— Nous fournissons une méthode pour une vérification *entièrement automatique* de l'*absence d'escalade de privilège* et *l'absence d'erreurs à l'exécution* de petits noyaux de systèmes d'exploitation, *en contexte* (c'est-à-dire pour une disposition mémoire des applications donnée). Nous éliminons le besoin d'annotations manuelles : tout d'abord en développant un *interpréteur abstrait* [15] au niveau du code machine, qui permet d'analyser toute la *boucle système* (le code du noyau + une abstraction du code des applications utilisateur) pour *inférer* automatiquement les invariants du noyau (plutôt que de seulement les vérifier) ; deuxièmement en *prouvant* que l'absence d'escalade de privilèges est une propriété *implicite*, i.e. qui ne demande pas d'écrire une spécification (tout comme l'absence d'erreur à l'exécution [16]) ;
— Nous proposons une extension de la méthode pour la vérification *paramétrée* de noyau (i.e. la vérification du noyau indépendamment des applications). Les travaux précédents [12–14] modélisent la mémoire en utilisant un *modèle à plat* (la mémoire est un gros tableau d'octets, et les adresses sont représentées numériquement) qui empêche la vérification paramétrée et limite le passage à l'échelle et la précision [17] de la vérification. Nous proposons une représentation de la mémoire basée sur les types qui résout ces problèmes. Finalement, nous différencions le traitement du code d'initialisation (dont le but est d'établir les invariants du système) du traitement du code pendant l'exécution (dont le but est de préserver les invariants du système), ce qui améliore davantage la précision. Si notre méthode requiert un très petit nombre d'annotations manuelles, celles-ci remplacent la *précondition* sur l'application dont on aurait eu besoin sinon ;
— Nous avons conduit une évaluation approfondie sur deux études de cas, où nous démontrons que 1. il est possible d'implémenter un interpréteur abstrait sur du code machine qui soit suffisamment précis pour vérifier un noyau de

système d'exploitation industriel existant, sans modification, et avec très peu d'annotations (ratio $< 1\%$) ; 2. l'interprétation abstraite est utile comme un outil d'intégration continue pour détecter des défauts pendant le développement du noyau, surtout dans le cas de systèmes embarqués qui possèdent beaucoup de variantes ; 3. l'analyse paramétrée peut passer à l'échelle pour un grand nombre de tâches utilisateurs, tandis que notre analyse "en contexte" ne le peut pas.

Au final, nous adressons d'importantes limitations dans les méthodes automatiques existantes : nous pouvons faire une vérification *paramétrée* (indépendante des applications) ; nous gérons les boucles non bornées, nécessaires notamment pour implémenter des ordonnanceurs temps-réel ; et nous *inférons* les invariants du noyau, au lieu de seulement les vérifier. Comme dans la vérification formelle de noyau, *"le raisonnement sur les invariants domine l'effort de preuve"* [2] (dans seL4, 80% de l'effort fut passé à énoncer et vérifier des invariants [1]), ce travail est une étape-clé pour des vérifications automatiques de systèmes plus complexes.

Références

[1] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4 : Formal verification of an OS kernel," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, ACM, 2009.

[2] B. J. Walker, R. A. Kemmerer, and G. J. Popek, "Specification and verification of the UCLA Unix security kernel," *Commun. ACM*, vol. 23, pp. 118–131, feb 1980.

[3] W. Bevier, "Kit : A study in operating system verification," *IEEE Transactions on Software Engineering*, vol. 15, pp. 1382–1396, 11 1989.

[4] R. J. Richards, *Modeling and Security Analysis of a Commercial Real-Time Operating System Kernel*, pp. 301–322. Boston, MA : Springer US, 2010.

[5] R. Gu, J. Koenig, T. Ramananandro, Z. Shao, X. N. Wu, S.-C. Weng, H. Zhang, and Y. Guo, "Deep specifications and certified abstraction layers," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, ACM, 2015.

[6] F. Xu, M. Fu, X. Feng, X. Zhang, H. Zhang, and Z. Li, "A practical verification framework for preemptive OS kernels," in *International Conference on Computer Aided Verification*, Springer, 2016.

[7] E. Alkassar, M. A. Hillebrand, W. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Verified Software : Theories, Tools, Experiments* (G. T. Leavens, P. O'Hearn, and S. K. Rajamani, eds.), (Berlin, Heidelberg), pp. 40–54, Springer Berlin Heidelberg, 2010.

[8] J. Yang and C. Hawblitzel, "Safe to the last instruction : automated verification of a type-safe operating system," *ACM Sigplan Notices*, vol. 45, no. 6, pp. 99–110, 2010.

[9] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überSpark : Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *25th USENIX Security Symposium (USENIX Security 16)*, USENIX Association, 2016.

[10] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, “Komodo : Using verification to disentangle secure-enclave hardware from software,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP ’17, ACM, 2017.

[11] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, “CertiKOS : An extensible architecture for building certified concurrent OS kernels,” in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, USENIX Association, 2016.

[12] M. Dam, R. Guanciale, and H. Nemati, “Machine code verification of a tiny ARM hypervisor,” in *Proceedings of the 3rd International Workshop on Trustworthy Embedded Devices*, TrustED ’13, ACM, 2013.

[13] L. Nelson, J. Bornholt, R. Gu, A. Baumann, E. Torlak, and X. Wang, “Scaling symbolic evaluation for automated verification of systems code with Serval,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP ’19, (New York, NY, USA), p. 225–242, Association for Computing Machinery, 2019.

[14] J. Nordholz, “Design of a symbolically executable embedded hypervisor,” in *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys ’20, (New York, NY, USA), Association for Computing Machinery, 2020.

[15] P. Cousot and R. Cousot, “Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints,” in *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ACM, 1977.

[16] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, “A static analyzer for large safety-critical software,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI’03)*, ACM, 2003.

[17] T. Reps, J. Lim, A. Thakur, G. Balakrishnan, and A. Lal, “There’s plenty of room at the bottom : Analyzing and verifying machine code,” in *International Conference on Computer Aided Verification*, Springer, 2010.

Vérification formelle de la sûreté d'un système contrôlé par réseaux de neurones

Arthur Clavière[1], Eric Asselin[1], Christophe Garion[2] and Claire Pagetti[3]

[1]Collins Aerospace [2]ISAE-SUPAERO [3]ONERA

Cet article est un résumé étendu de l'article "Safety Verification of Neural Network Controlled Systems", accepté à SSIV 2021 (7th International Workshop on Safety and Security of Intelligent Vehicles).

Contexte/motivation Les réseaux de neurones représentent une technologie prometteuse dans le domaine du contrôle. Parmi les applications connues, le ACAS Xu utilise des réseaux de neurones pour décider de manoeuvres d'évitement entre des drones [3]. Par rapport à un contrôleur classique basé sur des tables, le ACAS Xu a une empreinte mémoire largement réduite. Par ailleurs, afin de garantir un temps d'exécution suffisamment petit, le ACAS Xu a une architecture particulière. Au lieu d'utiliser un unique grand réseau, il utilise plusieurs petits réseaux, avec un système de switch entre ces réseaux : un seul réseau est exécuté à chaque pas de temps, ledit réseau étant choisi en fonction de la commande précédente.

Cependant, démontrer la sûreté de fonctionnement d'un tel système demeure problématique. Habituellement, cet objectif est réalisé via l'application de certains standards. Dans l'aéronautique par exemple, ces standards demandent de raffiner les exigences système en une spécification complète et correcte pour chaque composant du système. Des activités de vérification doivent ensuite être menées dans le but de démontrer que chaque composant satisfait sa spécification. Mais cette démarche s'applique difficilement dans le cas où le système intègre un réseau de neurones. Une première raison à cela est la difficulté à raffiner les exigences système en une spécification complète pour le comportement dudit réseau, la seule spécification disponible étant souvent un ensemble de données exemples, données à partir desquelles est entraîné le réseau. L'intéret d'un réseau de neurones réside d'ailleurs en partie dans sa capacité à résoudre un problème complexe, pour lequel on n'a pas de solution a priori. Autrement dit, on ne dispose pas d'une spécification pour le comportement attendu du réseau. De plus, les méthodes d'apprentissage actuelles ne garantissent pas la correction du réseau vis-à-vis d'une hypothétique spécification, comment alors démontrer qu'un réseau est correct s'il ne l'est pas ?

Plusieurs méthodes ont déjà été proposées pour résoudre ces problèmes. Parmi ces méthodes, certaines cherchent à enrichir la spécification du comportement attendu du réseau. Ces approches s'intéressent à définir et à vérifier des propriétés

d'intérêt sur les réseaux de neurones, telles la robustesse locale [4], mais ces propriétés n'offrent pas des garanties suffisantes quant à la sûreté du système intégrant ces réseaux. D'autres approches s'inspirent des travaux déjà menés pour la vérification de systèmes hybrides, en les adaptant au cas où la composante discrète du système est un réseau de neurones [2]. En raisonnant au niveau du système, ces méthodes permettent de s'affranchir des problèmes cités ci-dessus, essentiellement liés au raffinement des exigences au niveau des réseaux et à leur vérification. Mais ces méthodes ne permettent pas de démontrer la sûreté d'un système contrôlé par réseaux de neurones du type du ACAS Xu, notamment du fait du mécanisme de switch entre les réseaux. Nous proposons dans cet article une méthode pour vérifier la sûreté d'un tel système.

Approche En premier lieu, afin de formaliser le problème, nous introduisons un modèle pour la classe de système correspondant au ACAS Xu. Ce modèle reprend les codes d'un système en boucle fermée, avec d'une part une partie dynamique temps continu (plant) et d'autre part une partie contrôleur temps discret. La spécificité du modèle réside dans la définition du contrôleur qui consiste en un classificateur et qui d'une part intègre des réseaux de neurones et d'autre part dispose d'un état interne (commande produite au pas de temps précédent) permettant de choisir le réseau de neurones à exécuter (mécanisme de switch entre les réseaux). Comme indiqué précédemment, l'avantage d'un tel mécanisme est d'offrir un temps d'exécution réduit, aspect non négligeable dans le monde de l'embarqué. Pour la partie plant, nous faisons l'hypothèse classique d'une dynamique modélisée via une équation différentielle ordinaire avec les hypothèses de continuité habituelles, assurant l'unicité de la solution si les conditions initiales sont fixées.

La vérification de la sûreté de ce système est ensuite exprimée comme un problème de décision : décider si les états atteignables du système sur un horizon temporel fini intersectent ou pas un ensemble d'états d'erreurs (*e.g.*, l'ensemble des états correspondant à une collision dans le cas du ACAS Xu). Formulé ainsi, le problème est indécidable et ce du fait de la dynamique possiblement non-linéaire du plant. De plus le problème est rendu particulièrement complexe par l'utilisation des réseaux de neurones et le système de switch entre ces réseaux. Aussi, nous nous concentrons sur la calcul d'une sur-approximation des états atteignables du système.

Pour calculer cette sur-approximation, nous introduisons une représentation symbolique des états atteignables par le système, intégrant l'état interne du contrôleur. Il s'agit d'un liste d'états symboliques où chaque état symbolique est un doublet composé (i) d'une boîte représentant un ensemble d'états du système dynamique et (ii) de l'état interne du contrôleur c'est-à-dire la commande produite au pas de temps précédent. Cette représentation symbolique est utilisée pour approximer les états initiaux du système. A partir de cette représentation des états initiaux, les états atteignables par le système à chaque nouveau pas de temps sont cal-

culés, en utilisant la même représentation et en combinant deux méthodes : d'une part de la simulation garantie pour approximer la partie dynamique et d'autre part de l'interprétation abstraite pour approximer la partie contrôleur. La simulation garantie est réalisée à l'aide de l'outil DynIBEX [1] et l'analyse des réseaux par in-teprétation abstraite est réalisée à l'aide des outils Reluval [6] ou DeepPoly [5], qui implémentent des transformateurs abstraits spécifiques aux réseaux de neurones.

Enfin, nous illustrons la faisabilitié de notre approche via le cas d'étude ACAS Xu. Pour traiter ce cas d'étude, nous introduisons une heuristique particulière visant à partitionner l'ensemble des états initiaux possibles et ainsi paralléliser la résolution du problème de vérification.

References

[1] Julien Alexandre dit Sandretto and Alexandre Chapoutot. Validated Explicit and Implicit Runge-Kutta Methods. *Reliable Computing electronic edition*, 22, July 2016.

[2] Radoslav Ivanov, James Weimer, Rajeev Alur, George J. Pappas, and Insup Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. *CoRR*, abs/1811.01828, 2018.

[3] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. Deep neural network compression for aircraft collision avoidance systems. *CoRR*, abs/1810.04240, 2018.

[4] Guy Katz, Clark W. Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. *CoRR*, abs/1702.01135, 2017.

[5] Gagandeep Singh, Timon Gehr, Markus Püschel, and Martin Vechev. An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL), 2019.

[6] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. In *27th USENIX Security Symposium, USENIX Security 2018*, pages 1599–1614, 2018.

Méthodologie pour la validation d'exigences globales appliquée à la sécurité*

Virgile Robles[1], Nikolai Kosmatov[1,2], Virgile Prevosto[1], Louis Rilling[3] et Pascale Le Gall[4]

[1]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France, firstname.lastname@cea.fr
[2]Thales Research & Technology, Palaiseau, France, nikolaikosmatov@gmail.com
[3]DGA, France, louis.rilling@irisa.fr
[4]Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes, CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France, pascale.legall@centralesupelec.fr

Contexte Pour valider un programme, il est courant d'annoter ses fonctions avec des *contrats* : un ensemble de pré-conditions et de post-conditions formalisées au sein d'un langage de spécification tel qu'ACSL [2]. Il est alors possible de vérifier que le programme est correct vis-à-vis de sa spécification via des techniques telles que la vérification déductive.

Nos travaux précédents [3][4] montrent qu'une telle approche basée sur des contrats est limitée lorsqu'il s'agit de spécifier des exigences *haut-niveau*, qui s'appliquent à des composants entiers plutôt qu'à des fonctions individuelles. Ces travaux proposent une technique basée sur les *méta-propriétés*, ou HILARE [1] : une classe de propriétés de programmes décrivant des invariants globaux et des contraintes sur les opérations mémoires. Cette technique est implémentée dans MetAcsl [2], un greffon de la plateforme Frama-C [5], qui permet d'insérer ces propriétés au sein d'un programme C et de vérifier que ce dernier ne les viole pas.

Contributions Nous développons une méthodologie permettant d'utiliser notre approche sur un grand éventail d'exigences haut-niveau, dans le but de fournir des recommandations détaillées aux utilisateurs d'outils de vérification logicielle afin de rendre plus accessible la validation de programmes

*Cette soumission est un résumé étendu d'un article [1], accepté à FormaliSE 2021 (18-21 mai).

1. HIgh-Level Acsl REquirement
2. https://git.frama-c.com/pub/meta

de taille réelle. Cette méthodologie est illustrée sur de petits exemples et appliquée à deux cas d'études : (i) la vérification du chargeur d'amorçage (*bootloader*) du projet de clé USB sécurisée WooKey [6] et (ii) l'ébauche d'une spécification du micronoyau d'un système d'exploitation jouet.

Méthodologie L'expérience montre que le processus de vérification à l'aide d'HILARE de grandes bases de code suit un certain nombre d'étapes récurrentes. Tout d'abord, il est important de commencer par (i) s'assurer que le problème est bien dans le périmètre de MetAcsl [4] et (ii) identifier des points d'ancrages pour les propriétés dans l'état global du programme (c.à.d les variables pertinentes pour exprimer les exigences globales).

Ensuite, la formulation des exigences sous forme d'HILARE peut elle-même se décliner en plusieurs étapes distinctes. Pour chaque exigence, il est nécessaire d'identifier l'ensemble des fonctions concernées, puis d'essayer de la formuler comme un invariant sur l'état global ou bien une *contrainte* sur (i) les modifications de la mémoire, (ii) les accès mémoire ou (iii) les appels de fonction. Cette dernière étape peut être facilitée par la connaissance d'un certain nombre de *motifs* communs couvrant la plupart des exigences courantes, qui sont détaillés et illustrés dans l'article.

La validation d'exigences ainsi formulées peut s'effectuer via les greffons de Frama-C existants tels qu'Eva, E-ACSL ou Wp. Nous nous intéressons à ce dernier outil en particulier, qui permet de faire de la vérification déductive, et donnons un ensemble de recommandations pour réaliser la preuve du programme : comment analyser l'échec d'une obligation de preuve et le résoudre. Enfin, nous listons plusieurs écueils et bonnes pratiques à avoir en tête lors du travail de spécification afin que la vérification repose sur des bases solides : l'absence d'erreurs à l'exécution, la clôture des empreintes mémoires, etc.

Application La formulation d'exigences complexes en HILARE est illustrée de manière détaillée sur un micronoyau à l'architecture générique. Des contraintes telles que l'isolation des tâches ou leur ordonnancement permettent de mettre en œuvre la méthodologie présentée pas à pas.

Cette dernière est ensuite appliquée en situation réelle à un ensemble choisi d'exigences de sécurité sur le bootloader du projet WooKey [6], un prototype de clé USB chiffrante à authentification forte dont le microgiciel (*firmware*) est partiellement écrit en C. Entre autres, nous vérifions que chaque dispositif de sécurité est bien exécuté dans le bon ordre. L'implantation du bootloader comportant plus de 600 fonctions, sa vérification vis-à-vis de plusieurs propriétés complexes démontre la pertinence de l'approche sur des programmes réels.

Remerciements Nous remercions chaleureusement Ryad Benadjila, Arnauld Michelizza, Patricia Mouy, Mathieu Renard, Philippe Thierry et Philippe Trebuchet de l'ANSSI pour nos discussions à propos de WooKey, ainsi que l'équipe de Frama-C pour leur support. Recherche soutenue financièrement par le Ministère des armées - Agence de l'Innovation de Défense. Merci aux reviewers anonymes pour leurs commentaires.

Références

[1] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling et P. Le Gall. « Methodology for Specification and Verification of High-Level Requirements with MetAcsl ». In : *FormaliSE 2021 - 9th International Conference on Formal Methods in Software Engineering.* Mai 2021.

[2] P. Baudin, P. Cuoq, J.-C. Filiâtre, C. Marché, B. Monate, Y. Moy et V. Prevosto. *ACSL : ANSI/ISO C Specification Language.* `https://frama-c.com/acsl.html`. 2018.

[3] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling et P. Le Gall. « MetAcsl : Specification and Verification of High-Level Properties ». In : *Tools and Algorithms for the Construction and Analysis of Systems (TACAS).* T. 11427. LNCS. 2019.

[4] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling et P. Le Gall. « Tame Your Annotations with MetAcsl : Specifying, Testing and Proving High-Level Properties ». In : *Tests and Proofs (TAP).* T. 11823. LNCS. Springer, 2019.

[5] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles et B. Yakobowski. « Frama-C : A software analysis perspective ». In : *Formal Aspects of Computing* (2015), p. 573-609.

[6] R. Benadjila, A. Michelizza, M. Renard, P. Thierry et P. Trebuchet. « WooKey : Designing a Trusted and Efficient USB Device ». In : *Annual Computer Security Applications Conference (AC-SAC).* 2019.

DEPS Studio : Un environnement intégré de modélisation et de résolution pour la synthèse de système

Pierre-Alain Yvars[1], Laurent Zimmer[2]

[1]ISAE-Supméca, laboratoire QUARTZ EA7393, Saint Ouen, pierre-alain.yvars@supmeca.fr
[2]Dassault Aviation - Direction Générale Technique, Saint Cloud, laurent.zimmer@dassault-aviation.com

1 Introduction

Le langage DEPS (Design Problem Specification) a été développé pour modéliser à l'origine les problèmes de conception de produits et plus récemment les problèmes de conception de système [1]. DEPS Studio est l'IDE de ce langage. Il est conçu pour spécifier et résoudre des problèmes de conception, de configuration, de déploiement, d'allocation, de vérification ou de synthèse de systèmes. Les systèmes considérés peuvent être des systèmes physiques, des systèmes à forte composante logicielle ou des systèmes mixtes (embarqués, mécatroniques, cyber-physiques). Le point commun de tous ces problèmes est que leur résolution revient à compléter une représentation sous-définie (ou partielle) du système étudié afin de n'obtenir que les systèmes satisfaisant toutes les propriétés qui les caractérisent. Ces propriétés proviennent soit des exigences du cahier des charges, soit des contraintes physiques ou technologiques. Nous résumons cela par le manifeste suivant : "Résoudre un problème de conception = Compléter un modèle sous-défini". Nous cherchons à obtenir un modèle de solution avec un outil de synthèse. DEPS Studio est l'outil de synthèse que nous proposons et dont nous présentons les caractéristiques principales dans ce papier.

2 Aperçu du langage DEPS

2.1 Paradigme

DEPS [1] est un langage de modélisation dédié ou DSML (Domain Specific Modeling Language). Son objectif est de fournir un formalisme permettant de représenter des problèmes de conception de systèmes techniques. Le lecteur intéressé pourra trouver dans [1] et dans [3] un état de l'art détaillé justifiant la nécessité d'un tel formalisme. Sa grammaire non contextuelle est décrite sous la forme de Backus-Naur. DEPS est né d'une initiative de recherche académique et industrielle et est actuellement supporté par l'association DEPS Link [2]. Il combine certaines caractéristiques de modélisation structurelle propres aux langages orientés objet avec des caractéristiques de spécification de problèmes issues de la programmation par contraintes. Aux premiers ont été empruntées les caractéristiques de structuration et d'abstraction qui permettent de représenter les composants et l'architecture (éventuellement partielle) du système étudié. Aux seconds ont été empruntés les concepts logico-mathématiques nécessaires à la résolution des problèmes de l'ingénieur. Cette combinaison de paradigmes déclaratifs aboutit à un langage déclaratif qui permet de modéliser l'organisation des systèmes étudiés et les propriétés qui les régissent.

2.2 Types de données

Les types de données DEPS de base utilisés pour le calcul sont : les valeurs entières, les réels, les intervalles entiers, les intervalles réels, les domaines énumérés entiers et les domaines énumérés réels.

2.3 Type de quantité et quantité

Dans DEPS, les constantes et les variables sont associées à des quantités et des types de quantités physiques ou technologiques appelés respectivement *Quantity* et *QuantityKind* qui sont nécessaires dans l'ingénierie des systèmes. Un*QuantityKind* comporte : un type de base (entier ou réel), une valeur minimale, une valeur maximale ainsi qu'une dimension au sens de l'analyse dimensionnelle de la quantité (par exemple L pour une longueur ou U pour une quantité sans dimension). Une *Quantity* est définie à partir d'un *QuantityKind*. Le *QuantityKind* porte la dimension, la *Quantity* porte l'unité. Un *QuantityKind* et une *Quantity* peuvent avoir le même nom. Plus précisément, une *Quantity* comporte : un *Kind*, type de quantité de base (Par exemple, Real,

Integer, Length) et une borne Min (resp. Max) qui représente la valeur minimale (resp. maximale) qui peut être prise par toute constante ou variable ayant la quantité définie comme type, une unité de la quantité.

2.4 Problème et modèles

La caractéristique fondamentale du langage est le modèle. Tout modèle est défini à l'aide du mot-clé *Model* suivi de son nom et de sa liste (éventuellement vide) d'arguments. Il contient dans l'ordre : une zone de déclaration-définition des constantes du modèle (mot-clé *Constants*), une zone de déclaration des variables du modèle (mot-clé *Variables*), une zone de déclaration-définition des éléments du modèle (mot-clé *Elements*) et une zone de définition des propriétés du modèle (mot-clé *Properties*). La définition d'un modèle DEPS se termine par le mot-clé *End*. Le problème à résoudre est exprimé à l'aide du mot-clé *Problem*. Le problème est un modèle sans arguments. Le problème est la racine de l'arbre des éléments qui sont des instances de modèles. Une constante est une quantité numérique dont la valeur ne varie pas pendant la durée de vie d'une copie du modèle dans lequel elle est déclarée et définie. Une variable est une inconnue du modèle. Elle est caractérisée par sa quantité éventuellement limitée à un sous-ensemble de valeurs possibles. Les variables portent le caractère sous-défini des modèles DEPS.

2.5 Fonctionnalités orientées objet

Les modèles DEPS peuvent hériter les uns des autres (mot-clé *extends*). Il s'agit d'un héritage public et simple : les constantes, les variables, les éléments et les propriétés sont ainsi hérités des modèles ancêtres. Un élément est une instance de modèle. Il peut être passé en argument à un Modèle pour représenter une agrégation et doit alors être déclaré dans la zone de champ *Elements* du Modèle. Il peut également être créé dans un modèle pour modéliser une composition et doit alors être déclaré et créé dans la zone des éléments du modèle. Tous les éléments d'un problème sont organisés en utilisant des relations d'agrégation et de composition formant une structure arborescente. L'accès aux éléments de cette structure est autorisé par l'utilisation d'une notation pointée. Une constante, une variable ou un élément à différents niveaux de cette structure arborescente peut être désigné et manipulé en utilisant un chemin. Chaque Modèle possède une signature. Ce mécanisme lève toute ambiguïté lorsque des modèles portent le même nom mais ont des arguments différents.

2.6 Les propriétés

Une propriété est une relation nécessairement respectée par toute instance du modèle qui la contient. Dans la version actuelle de DEPS, les propriétés sont des relations algébriques (ou contraintes) :égalité et/ou inégalité entre expressions, définition de la valeur d'une expression déclarée relative à des constantes et variables du modèle ou d'un ou plusieurs éléments du modèle. Tous les opérateurs de la norme IEEE754 pour l'arithmétique à virgule flottante sont disponibles. Une expression déclarée ou nommée (mot-clé *expr*) pointe vers une expression algébrique et permet de la référencer et de l'utiliser dans de nombreuses propriétés. Les équations et les inéquations linéaires ou non linéaires, sont naturellement des propriétés. Elles peuvent aussi être des relations dédiées au domaine de la conception. Ce sera par exemple le cas pour la contrainte catalogue qui permet à l'utilisateur de créer des relations définies en extension par une table de n-uplets.

3 DEPS Studio IDE

3.1 Généralités

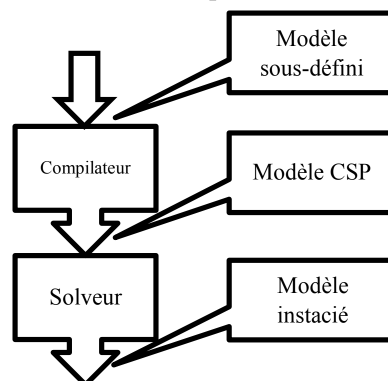


Figure 1: DEPS Studio - chaîne de synthèse

L'environnement intégré de modélisation et de résolution DEPS Studio [3] associé au langage DEPS comprend un éditeur de modèle, un gestionnaire de projet, un compilateur et un solveur (cf. Figure 1). L'expérience montre que la spécification d'un problème de conception de système n'est jamais correcte du premier coup et que de nombreuses erreurs de modélisation ne sont détectables que par le calcul. Nous avons donc décidé de développer et d'intégrer notre propre solveur dans l'environnement de développement afin que la recherche de solutions contribue efficacement à l'ajustement du processus de modélisation du problème. Il s'agit d'une approche de développement rapide de modèles (analogue à une approche RAD) qui, contrairement à une approche de transformation de modèles, réduit le temps d'exécution de la boucle de développement de modèles. Elle permet également de remonter les erreurs au bon niveau d'abstraction. DEPS Studio a été développé en Delphi. Les couches de base du calcul d'intervalle ont été écrites en C++ et utilisent la bibliothèque open source de calcul par intervalle gaol [4]. L'ensemble des développements représente environ 100 000 lignes de code.

3.2 Edition du modèle et gestion du projet

Un problème à résoudre est organisé en un projet. Un projet est constitué de plusieurs paquets (*package*). Chaque paquet est enregistré dans un fichier ".deps". Les paquets contiennent des modèles, des types de quantités, des quantités et des tableaux. L'un des paquets doit contenir un modèle particulier sans arguments et déclaré comme étant le problème. Ce modèle représente le problème global à résoudre exprimé sous forme de constantes, variables, éléments et propriétés.
L'environnement dispose :
- d'un éditeur multi-modèles pour charger, modifier et sauvegarder les paquets,
- d'un gestionnaire de projet pour charger, modifier et sauvegarder le projet de modélisation d'un problème composé de tous ses paquets.

Un projet est défini comme un ensemble de *packages*. Chaque *package* suit la structure suivante :

Package <packageName> ;
Uses <ListOfPackageName> ;
<List of DEPSFeature>
With
<DEPS Feature> : <QuantityKind>
| *<Quantity>*
| *<Table>*
| *<Model>*
| *<Problem>*

3.3 Le compilateur

Le compilateur que nous avons développé transforme directement le modèle "source" DEPS d'un problème de conception en un réseau de propriétés "objet" associées au modèle <V, D, C> d'un problème de satisfaction de contraintes (CSP) [5]. Il s'agit donc d'un compilateur "natif" qui n'est pas une surcouche d'un langage de programmation par contraintes. La compilation est anticipée ; l'ensemble du réseau est donc généré avant la résolution. Le typage statique du langage DEPS est exploité par le compilateur pour détecter les erreurs de type sur les constantes, variables, éléments et propriétés avant la résolution. La compilation se fait en deux passes :
- La première passe de compilation vérifie les paquets utilisés par le projet, analyse lexicalement et syntaxiquement leur contenu et crée la hiérarchie des modèles du projet ;
- La deuxième passe crée l'ensemble des éléments qui définissent le problème à partir de la création de l'élément d'instance unique du modèle déclaré comme problème. Les erreurs sont traitées et signalées à l'utilisateur à toutes les étapes de la compilation : vérification du paquetage, analyse lexicale, analyse syntaxique, création de la hiérarchie du modèle, création des éléments sous-définis.

Si l'étape de compilation réussit, l'étape de résolution aura pour tâche d'attribuer des valeurs aux variables satisfaisant toutes les propriétés/contraintes du problème. L'intérêt d'une approche intégrée et maitrisée noous permet de faire évoluer en parallèle les traits du langage DEPS, son compilateur et le sover.

3.4 Le solveur

La résolution d'un problème de conception nécessite la capacité de prendre en compte les problèmes sous-contraints, les équations et inégalités algébriques non linéaires sur des domaines mixtes ainsi que d'autres types de relations telles que des tables de valeurs. Les problèmes de conception que nous adressons nécessitent à la fois de pouvoir travailler sur des inconnues à domaine de valeurs de type intervalles continus, intervalles d'entiers ou bien des énumérés (entiers ou réels) et de poser des propriétés logique et/ou algébriques sur ces inconnues. Un solveur SAT ou SMT ou bien un solveur purement discret ou purement continu ne seront donc pas suffisants. Nous avons ainsi développé un solveur mixte à base de contraintes dédié au calcul sur des modèles DEPS structurés. Les méthodes de calcul que nous utilisons sont issues des techniques de résolution de CSP [5]. La structure des modèles DEPS est préservée tout au long de la chaîne de compilation jusqu'aux modèles de calcul. Le solveur implémente une méthode de propagation HC4 révisée [6] sur les équations et les inégalités. Initialement conçue pour les domaines continus, nous avons étendu la méthode à quatre types de domaines : intervalles réels ouverts, intervalles entiers, ensembles énumérés de valeurs flottantes et ensembles énumérés de valeurs entières signées. Pour des raisons de performance, les réductions sont effectuées directement sur les domaines typés sans revenir aux intervalles réels. L'algorithme de recherche de la racine est une méthode de branchement et d'élagage. Pour l'instant, seules les stratégies classiques round-robin et first-fail sont implémentées. Dans le cas d'un problème sur-contraint, un échec peut survenir lors de la première propagation ou après avoir exploré l'ensemble de l'arbre de recherche. Suivant le paradigme CSP, l'échec est interprété comme la preuve qu'il n'existe pas de solution au problème et non comme un échec de l'algorithme de résolution. L'architecture orientée objet du solveur a été conçue de manière à pouvoir être étendue à d'autres méthodes de propagation et/ou de résolution existantes (box-consistance, méthodes locales, ...).

4 Conclusion

DEPS Studio a été utilisé sur plusieurs applications industrielles. Il a été notamment appliqué à des problèmes de dimensionnement de système électriques [7] ainsi qu'à des problèmes de déploiement de fonctions sur des architectures informatiques embarquées [8-10]. Les résultats obtenus sont positifs en modélisation comme en résolution. Les futures versions de DEPS Studio tireront parti des prochaines évolutions du langage DEPS.

Réferences

[1] P.A. Yvars, L. Zimmer, 2014. *DEPS Un langage pour la spécification de problèmes de conception de Systèmes*, proc of the 10th International Conference on Modeling, Optimization & SIMulation (MOSIM 2014), France.

[2] www.depslink.com

[3] P.A. Yvars, L. Zimmer, *Integration of Constraint Programming and Model-Based Approach for System Synthesis*, IEEE International Systems Conference, SYSCON 2021, Vancouver, Canada

[4] Goualard F., 2015. *Gaol 4.2.0 Not Just Another Interval Arithmetic Library,* https://frederic.goualard.net/software/gaol-4.2.pdf

[5] E. Tsang, 1993. *Foundations of Constraint Satisfaction*. London and San Diego:Academic Press.

[6] Benhamou F., Goualard F., Granvilliers L., Puget J.F., 1993. Revising Hull and Box consistency,*16th International Conference on Logic Programming.*.

[7] S. Diampovesa, A. Hubert, P.A. Yvars, *Designing Physical Systems through a Model-Based Synthesis Approach. Example of a Li-ion Battery for Electrical Vehicles*, Computers In Industry Journal, vol 129, 2021

[8] L.Zimmer, M. Lafaye, P.A. Yvars, *Modélisation d'exigences pour la synthèse d'architecture avionique : Application à la sûreté de fonctionnement,* 16ème journées AFADL, Approche Formelle dans l'Assistance au Développement de Logiciel, Montpellier, 2017.

[9] L. Zimmer, P.A. Yvars, M. Lafaye, *Models of requirements for avionics architecture synthesis: safety, capacity and security,* Complex System Design and Management conference – CSD&M 2020, December 2020, Paris, France.

[10] L. Zimmer, P.A. Yvars, *Synthesis of software architecture for the control of embedded electrical generation and distribution system for aircraft under safety constraints: The case of simple failures*, 14th International Conference of Industrial Engineering, CIGI-QUALITA 2021, Grenoble, France.

Détection d'objectifs de test polluants pour les critères de flot de données *

Thibault Martin[1] Nikolai Kosmatov[1,2] Virgile Prevosto[1]
Matthieu Lemerre[1]

[1]Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
firstname.lastname@cea.fr
[2]Thales Research & Technology, Palaiseau, France
nikolaikosmatov@gmail.com

Contexte Pour évaluer la qualité d'une suite de tests logiciels, c'est-à-dire s'assurer qu'elle est représentative d'un ensemble de cas d'usage aussi large que possible, de nombreux critères de couverture ont été définis [2]. Parmi ceux-ci, les critères de flot de données, comme all-defs et all-uses, appartiennent aux plus avancés. Ces critères sont définis à partir de la notion de *paire def-use*, composée d'une position où une variable `x` est définie et d'une position où `x` est utilisée, et telle qu'il existe au moins un chemin reliant la définition à l'utilisation dans le graphe de contrôle du programme sous test. Si un chemin reliant les deux nœuds d'une paire def-use de x ne contient pas de nœud redéfinissant `x`, on parle d'un chemin sans redéfinition (*def-clear path*).

Comme pour beaucoup d'autres critères, certains objectifs individuels de test sont *polluants* : ils doivent être retirés pour mesurer correctement la couverture de test [3]. On trouve plusieurs types d'objectifs de test polluants pour les critères de flot de données :

les objectifs *non applicables*, où une paire def-use ne peut pas être reliée par un chemin sans redéfinition (i.e. il n'existe aucun chemin permettant d'éventuellement couvrir cet objectif) ;

les objectifs *infaisables*, où une paire def-use peut être reliée par au moins un chemin sans redéfinition, mais aucun de ces chemins n'est activable par un cas de test ;

les objectifs *équivalents* (ou *dupliqués*), qui sont toujours couverts simultanément : il suffit de garder un objectif par classe d'équivalence [1].

---

*Cette soumission est un résumé étendu d'un article [1], publié à IFM 2020.

1. L'équivalence d'objectifs de test est en effet une relation d'équivalence.

Problème Bien que la création d'une liste d'objectifs (candidats) de test pour les critères de flot de données puisse sembler facile, la définition complexe de ces critères, mêlant atteignabilité et chemin sans redéfinition, rend la détection des objectifs polluants difficile. Il est cependant crucial d'éviter de perdre du temps pendant la génération de tests (en essayant de couvrir un objectif polluant) et de mesurer correctement la couverture (en ignorant les objectifs polluants dans le nombre total d'objectifs).

La détection d'objectifs de test polluants pour des critères plus simples a été étudiée précédemment (voir [3,4] pour quelques résultats récents). Cependant l'évaluation et la combinaison de plusieurs techniques d'analyse pour leur détection sur les critères de flot de données —l'objectif de ce travail— n'ont pas été examinées.

Contributions Dans ce travail, nous évaluons trois approches pour détecter les objectifs de test polluants pour les critères de flot de données : une analyse de flot de données simple, une analyse de valeurs basée sur l'interprétation abstraite et une analyse de plus faible précondition. Nous avons implanté ces approches dans LTEST [2], une boîte à outils open-source pour le test de programme C [5]. Nous avons ensuite évalué et comparé ces techniques sur différentes études de cas et avons analysé leurs capacités à détecter les objectifs polluants et leurs limites. Nous nous sommes concentrés sur la partie clef des critères de flot de données : les paires def-use. Les capacités de détection que nous avons observées sont différentes de celles des expériences similaires faites précédemment pour d'autres critères.

Enfin, nous proposons une méthode pour une combinaison efficace de plusieurs techniques. Sur la base de nos études de cas, combiner l'analyse statique simple pour détecter à la fois les objectifs non applicables et les objectifs équivalents, avec l'analyse de valeurs pour identifier les objectifs infaisables semble être le meilleur compromis pour une détection rapide et efficace.

Travaux futurs Ce travail fournit une comparaison de la capacité de détection des techniques d'analyse. Lors de travaux futurs, il sera nécessaire d'évaluer leurs résultats par rapport au *nombre réel* d'objectifs polluants (ou une sur-approximation calculée en rejouant une suite de tests riche). D'autres améliorations possibles incluent une meilleure prise en charge du langage C (pointeurs et tableaux), l'amélioration des analyses, notamment celle basée sur la plus faible précondition, en ajoutant automatiquement des annotations de code, ainsi que l'extension de notre étude aux objectifs subsumés (c.a.d. impliqués) et à d'autres critères.

Remerciements. Ce travail a été partiellement financé par le projet ANR SATOCROSS (grant ANR-18-CE25-0015-01). Nous remercions Sébastien Bardin et les rapporteurs anonymes pour leurs commentaires.

2. disponible sur `https://github.com/ltest-dev/LTest`

Références

[1] Thibault Martin, Nikolai Kosmatov, Virgile Prevosto, and Matthieu Lemerre. Detection of Polluting Test Objectives for Dataflow Criteria. In *IFM*, 2020.

[2] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2017.

[3] Sébastien Bardin, Mickaël Delahaye, Robin David, Nikolai Kosmatov, Mike Papadakis, Yves Le Traon, and Jean-Yves Marion. Sound and quasi-complete detection of infeasible test requirements. In *ICST*, pages 1–10, 2015.

[4] Michaël Marcozzi, Sébastien Bardin, Nikolai Kosmatov, Mike Papadakis, Virgile Prevosto, and Loïc Correnson. Time to clean your test objectives. In *ICSE*, page 456–467, 2018.

[5] M. Marcozzi, S. Bardin, M. Delahaye, N. Kosmatov, and V. Prevosto. Taming coverage criteria heterogeneity with LTest. In *ICST*, pages 500–507, 2017.

Représentation de programmes SMALA grâce à la théorie des bigraphes

Cécile Marcon, Nicolas Nalpon, Cyril Allignol, Célia Picard
ENAC, Toulouse, France

Résumé

Dans le but de certifier le compilateur du langage réactif-interactif SMALA, nous cherchons à représenter sa sémantique en instanciant la théorie des bigraphes. En nous limitant dans un premier temps à un sous-ensemble de ce langage, nous avons pu représenter ses différents acteurs comme des entités distribuées dans l'espace. Des règles de réaction sur les bigraphes permettent de mettre ces entités en mouvement et en relation les unes avec les autres afin de représenter l'exécution du programme en plus de sa structure.

Mots-clés : bigraphes, sémantique, compilation certifiée, langage réactif

1 Introduction

Dans le secteur aéronautique comme dans de nombreux autres, les humains interagissent avec des systèmes toujours plus nombreux et critiques, y compris pour leur sécurité. Un système *interactif* est manipulé par des humains et doit en réponse réagir de façon correcte, fiable voire certifiée. Un système interactif-reactif réagit aux entrées utilisateurs mais aussi à d'autres (issues de capteurs par exemple).

La plupart des langages de programmation ne facilitent pas le passage de la phase de design à celles de conception, architecture et développement, compliquant la création de logiciels interactifs. Ce constat motive le développement de SMALA, un langage réactif [1] mettant l'accent sur l'aspect interactif des programmes. Il permet de décrire un programme à la manière des systèmes basés composant [2]. On y retrouve des entités qu'on peut composer et connecter entre elles afin de les coordonner. SMALA repose sur de larges bibliothèques de composants, notamment graphiques, développés en C++. Il est surtout utilisé pour concevoir des IHM pour l'aviation, comme l'interface du tableau de bord de l'hélicoptère électrique Volta [3].

Pour pouvoir utiliser un langage dans la conception de systèmes critiques il faut fournir des garanties sur l'exécution de ses programmes. Ainsi, nous envisageons de vérifier le compilateur de SMALA en commençant par s'assurer de la préservation des liens de causalité entre les entités. Pour cela, nous nous inspirons de l'approche des compilateurs Vélus [4] et CompCert [5] : nous implémentons et vérifions un

compilateur SMALA grâce à l'assistant de preuve Coq. Pour l'instant, nous ignorons la partie graphique du langage. De plus, pour faciliter la vérification, nous travaillons avec SMALIGHT, un sous-ensemble de SMALA, couvrant ses principes de base et permettant de redéfinir les bibliothèques actuelles en C++.

Notre première définition formelle de ce sous-ensemble [6] a soulevé plusieurs problèmes. En effet, l'ensemble choisi était trop petit, ne permettant pas de couvrir tous les concepts de SMALA. De plus, l'expression de l'ordre d'exécution des entités du programme nécessitait l'introduction de trop nombreux paramètres, rendant les règles peu intuitives et difficiles à faire évoluer. De même que le lambda calcul est approprié pour définir les sémantiques opérationnelles de langages fonctionnels, nous voulons exprimer notre sémantique avec une théorie mathématique adaptée. La sémantique de SMALA est basée sur une double structure d'arbre et de graphe définissant les règles d'activation des entités du programme. Mais dans notre première approche, nous avons exprimé la priorité d'exécution avec une liste d'ensembles, compliquant la compréhension. Étant donnée la structure de SMALA, nous avons décidé d'utiliser la théorie des bigraphes pour formaliser la sémantique d'une version étendue de SMALIGHT. La capacité des bigraphes à modéliser et implémenter des langages de programmation a déjà été démontrée [7]. Ils ont ainsi été utilisés pour exprimer la sémantique du langage Kappa basé sur des règles d'interaction entre protéines [8]. Ici, nous utilisons la théorie des bigraphes pour modéliser la structure d'un programme SMALIGHT et les règles d'activation de ses éléments.

La section 2 présente SMALA et SMALIGHT. La section 3 définit les éléments de la théorie des bigraphes utiles pour la suite. La section 4 détaille la formalisation de la sémantique avec les bigraphes. La section 5 présente la suite des travaux.

2 Réduction de SMALA en SMALIGHT

SMALA repose sur deux concepts : les processus et une notion de dynamicité.

2.1 Processus

Un processus est une entité qui a une sémantique et un état (Activé ou Désactivé). Si un processus est Activé, il s'exécute selon sa sémantique. S'il est Désactivé, il est inactif. De plus, un processus peut être *persistant* (son activation perdure jusqu'à la fin de l'exécution ou sa désactivation par un autre processus) ou *transitoire* (il s'exécute selon sa sémantique à l'activation puis se désactive immédiatement).

Les processus de SMALA couvrent tous les aspects du langage : graphiques, interactifs, structures de contrôle, création de composants, gestion de l'activation et de la mémoire. Néanmoins, la plupart peuvent être vus comme la composition de processus plus élémentaires. Dans SMALIGHT, nous visons à identifier l'ensemble minimal de ces processus permettant la redéfinition de tous les autres. Ainsi, actuellement, les processus de SMALIGHT (en omettant la partie graphique) sont :

— binding : processus persistant permettant à un processus p_1 (le notifiant) de

communiquer son état d'activation à un processus p_2 (l'abonné). Ainsi, le binding $p_1 \rightarrow p_2$ active p_2 quand p_1 s'active. Il en existe trois autres types :
  - Le binding $p_1 \rightarrow ! \, p_2$ désactive p_2 quand p_1 s'active
  - Le binding $p_1 \, ! \rightarrow p_2$ active p_2 quand p_1 se désactive
  - Le binding $p_1 \, ! \rightarrow ! \, p_2$ désactive p_2 quand p_1 se désactive
- component : processus persistant correspondant à un contenant nommé. On appelle enfants les processus qu'il contient. Ils sont activés dans l'ordre de leur définition dans le code (figure 1a) lorsque le component est activé.
- spike : processus transitoire dont la sémantique est de ne rien faire. Il est généralement utilisé pour notifier ses abonnés d'un changement d'état.
- property : processus persistant encapsulant une valeur stockée en mémoire et notifiant ses abonnés lorsque cette valeur change. Il en existe de différents types mais SMALIGHT contient seulement les IntProperty pour l'instant, sur lesquelles on peut effectuer des opérations arithmétiques.
- assignment : processus transitoire noté =:. Ainsi, $e_1 \; =: \; p_2$ copie la valeur de l'expression e_1 dans la property p_2 lorsqu'il est activé.

La figure 1a présente un exemple d'un programme SMALIGHT très simple.

2.2 La dynamicité

La notion de dynamicité dans SMALA couvre trois aspects : la propagation (seule traité pour l'instant), la mémoire et la création dynamique de processus.

La propagation concerne toutes les modifications dues à des changements d'état de processus dans un programme SMALA. Ces propagations sont de deux types différents : celles issues des components et celles issues des bindings.

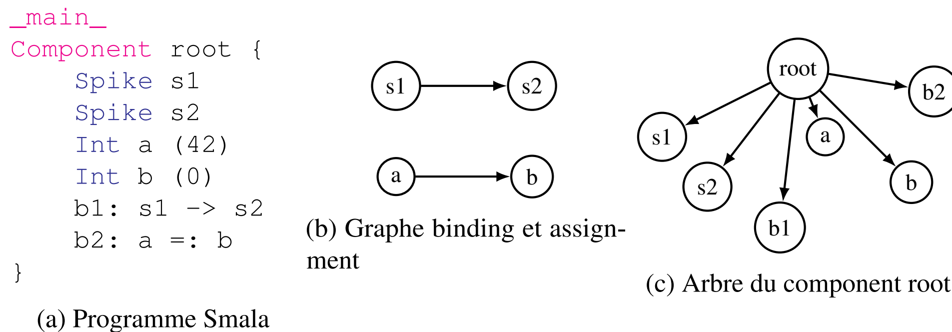
La définition d'un component induit la création d'une relation hiérarchique avec ses enfants, représentée par un arbre (figure 1c). Si la racine change d'état, on propage ce changement aux enfants en parcourant les dans l'ordre de leur déclaration et récursivement. De plus, un enfant est Activé seulement si son parent l'est aussi.

Un binding ou un assignment crée une relation entre les processus qui peut être représentée par un graphe orienté acyclique (figure 1b). Ainsi, si un processus source (i. e., n'ayant pas de prédécesseur) change d'état, ce changement est propagé aux autres processus du graphe en suivant un ordre total issu d'un tri topologique.

3 La théorie des bigraphes

La théorie des bigraphes, formellement définie par Milner [9], représente les relations et les interactions entre des entités grâce à un système réactif bigraphique consistant en une paire de graphes (graphe de places et graphe de liens, représentant respectivement l'imbrication des entités les unes dans les autres et leurs relations) appelée bigraphe et un ensemble de règles de réaction sur ce bigraphe.

Le graphe de places (figure 2b) est une forêt d'arbres qui permet de décrire la localisation des nœuds et en particulier leur imbrication les uns dans les autres. La



(a) Programme Smala

(b) Graphe binding et assignment

(c) Arbre du component root

FIGURE 1 – Structure d'un programme SMALA

racine de ses arbres est une région (une entité qui peut en contenir d'autres mais ne peut être contenue ; dans la figure 2a ce sont les rectangles blancs, dans la figure 2b, les racines) et leurs feuilles sont des nœuds ou des sites (une entité qui peut être contenue par une région ou des nœuds et pouvant être substituée par un graphe de places ; dans la figure 2a, c'est le rectangle grisé et dans la figure 2b, la feuille '0'). Les régions et les sites permettent de composer des graphes de places entre eux.

Le graphe de liens (figure 2c) est un hypergraphe décrivant des relations entre entités par des hyperarêtes (arêtes connectant plusieurs nœuds). Ses nœuds ont des ports permettant aux arêtes de se fixer. Les arêtes brisées ont une extrémité non connectée et permettent la composition de graphes de liens ('e4' en figure 2).

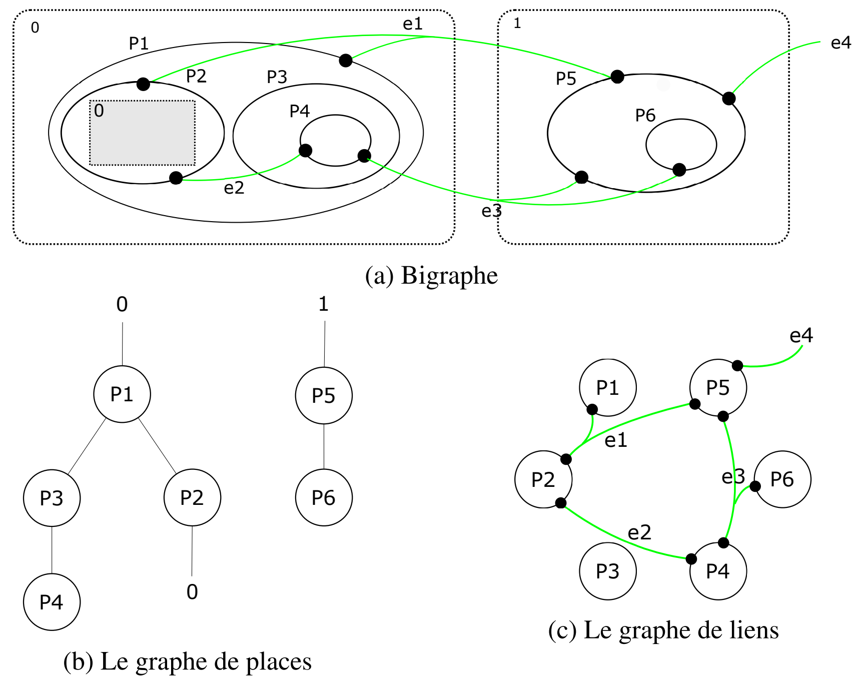
Une règle de réaction R est une paire de bigraphes (G, D) ayant autant de sites, de régions et d'arêtes brisées. On la note $R : G \longrightarrow D$. Appliquer R à un bigraphe B consiste à remplacer une occurrence de G se trouvant dans B par D.

4 De SMALIGHT vers les bigraphes

Nous avons choisi de représenter les règles d'activation d'un programme SMALIGHT en l'assimilant à un système réactif bigraphique. L'assignment et la property ne sont pas présentés ici. En effet, bien qu'ils possèdent un mécanisme de propagation d'activation, une part important de leur fonctionnement est liée à la mémoire qui n'est pas traitée dans cet article.

4.1 Représentation générique des processus

Nous proposons tout d'abord une représentation générique des processus de SMALIGHT qui nous permet d'appliquer les règles de réaction de façon efficace. Ainsi, un processus est représenté par un nœud Process. Il contient un ensemble de nœuds, possédant chacun un site, décrivant ses attributs (voir figure 3a). A ce stade, nous avons identifié 4 attributs qui suffisent pour décrire totalement n'importe quel processus de SMALIGHT. Ils sont décrits ici avec les conventions graphiques utilisées le cas échéant : un identifiant, (représenté si nécessaire par une police

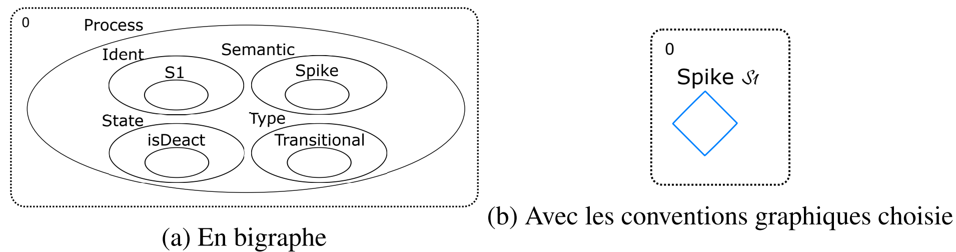


(a) Bigraphe

(b) Le graphe de places

(c) Le graphe de liens

FIGURE 2 – Un bigraphe et les graphes de places et de liens correspondants



(a) En bigraphe

(b) Avec les conventions graphiques choisie

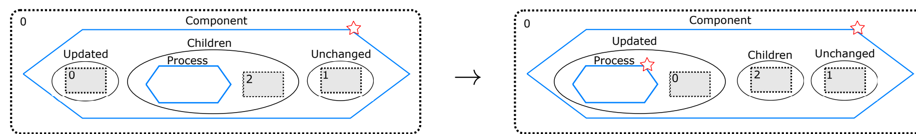
FIGURE 3 – Le modèle générique présenté avec le spike de SMALIGHT

cursive), une sémantique (component, spike, etc. représentant un processus), un état (un nœud Activé est violet ; Désactivé, il est bleu) et un type (transitoire est représenté par un losange, persistant par un hexagone). Nous ne représentons les ports que si la règle de réaction concerne le graphe de liens.

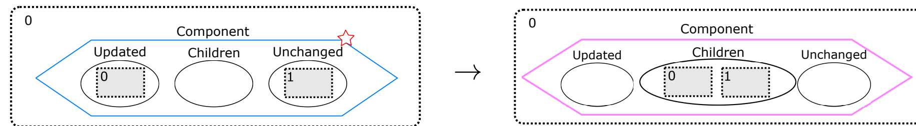
Nous décrivons l'activation et la désactivation des processus par un ensemble de règles de réaction, afin de représenter l'exécution d'un programme SMALIGHT.

4.2 Logique de base des règles de réaction

La section 2.2 présente l'aspect dynamique de SMALA notamment les deux règles de propagation d'activation. L'une concerne le component et dit que son activation provoque celle de tous ses enfants. On le traduit par des règles de réaction sur le graphe de places. L'autre concerne l'activation des bindings et se traduit par



(a) Règle récursive appliquée tant que *Children* n'est pas vide



(b) Règle d'arrêt quand *Children* est vide

FIGURE 4 – Activation du *Component*

des règles de réaction sur le graphe de liens, détaillées en section 4.3.2.

Pour activer un component on applique plusieurs règles de réaction sur le bigraphe. On doit savoir que le component est en train de s'activer et que ses enfants doivent l'être aussi. Pour cela, on place le component dans un nœud *Activate* servant de *flag*, symbolisé par une étoile rouge en haut à droite du processus. Finalement, par souci de généralisation et pour ne pas avoir à écrire trop de règles de réaction, nous appliquons le flag *Activate* à tous les processus dès leur activation avant de passer leur état à l'état Activé. La désactivation suit un principe similaire.

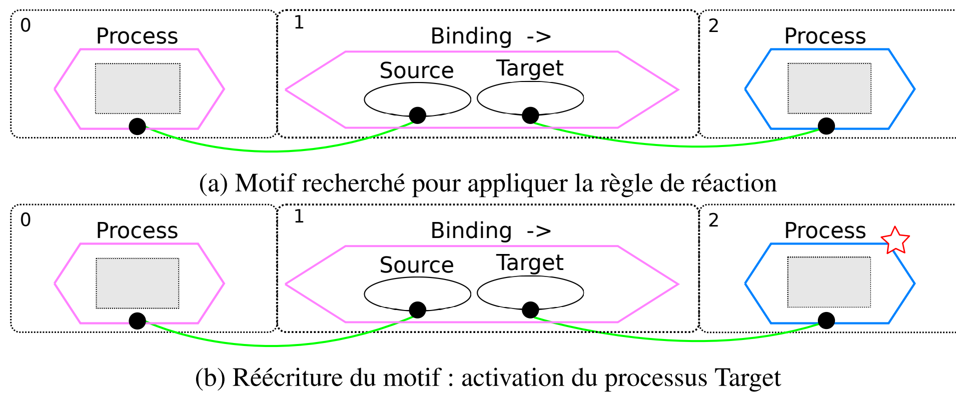
4.3 Instanciation sur les processus de SMALIGHT

Ces modèles généraux sont instanciés ici pour le component et le binding. Le spike est représenté comme un nœud *Spike* unique (voir figure 3). La property et l'assignment, utilisant la mémoire pas encore traitée, ne sont pas présentés.

4.3.1 Le component

Le component est un processus qui en contient d'autres. On le représente par un nœud *Component* possédant un enfant *Children* contenant tous ses enfants.

Pour activer un *Component*, et donc tous ses enfants, on doit connaître les processus restant à activer à chaque application d'une règle de réaction. De plus l'activation d'un enfant dépend de son type (transitoire ou persistant). Nous avons donc ajouté deux autres sites comme enfants de *Component* pour gérer cela. Ainsi, à l'activation d'un Component, ses enfants persistants sont déplacés dans un nœud *Updated* (voir figure 4) et ses enfants transitoires dans un nœud *Unchanged*. Lorsque tous les nœuds ont été traités, le nœud *Children* est vide. On peut alors passer l'état du component à Activé et replacer l'ensemble des enfants dans le nœud *Children*.



(a) Motif recherché pour appliquer la règle de réaction

(b) Réécriture du motif : activation du processus Target

FIGURE 5 – Règle de réaction sur le *Binding* $\rightarrow$ avec un processus source persistant

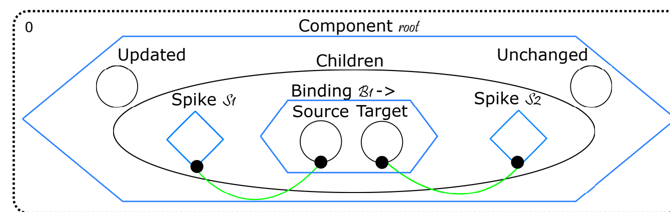


FIGURE 6 – Le bigraphe représentant le programme SMALIGHT de la figure 1a

4.3.2 Le binding

Le binding est représenté comme un nœud ayant deux enfants *Source* et *Target*. Chacun comprend un port qui le lie à un nœud Process correspondant au processus source, respectivement cible, du binding. Ainsi, les règles de réaction du binding $\rightarrow$ correspondent à la recherche de bindings dont le processus source est activé et le processus cible pas encore (figure 5). Les autres bindings suivent le même principe.

4.3.3 Exemple

On peut donc décrire un programme SMALIGHT en bigraphe grâce aux éléments précédents. Le bigraphe de la figure 6 correspond ainsi au code de la figure 1a.

5 Travaux en cours et à venir

Cet article présente une formalisation du langage SMALIGHT et de ses règles d'activation sous forme d'un système réactif bigraphique. Cette formalisation permet de représenter des interactions simples entre les différents processus de SMALIGHT. Toute la sémantique de SMALIGHT n'est pas encore couverte. Parmi les aspects importants à traiter, il y a la synchronisation de la propagation d'activation et les modifications en mémoire. La synchronisation vise à activer un processus si et seulement si tous les processus qui le précèdent ont fini leur exécution. Pour

cela, nous travaillons sur des règles de réaction qui propagent l'activation selon un tri topologique. Concernant la mémoire, nous cherchons aussi à exploiter la théorie des bigraphes pour la représenter et raisonner dessus. Nous avons testé cette formalisation en l'implémentant à l'aide de BigraphER [10], un outil développé en OCaml qui permet de simuler des systèmes réactifs bigraphiques. Par ailleurs, BigraphER a une place importante dans notre chaîne de compilation certifiée. En effet, nous travaillons à la traduction de cet outil dans Coq afin de pouvoir l'utiliser pour implémenter et vérifier notre sémantique. Enfin, voulant avoir une chaîne de compilation certifiée de bout en bout, nous cherchons à générer du code séquentiel C à partir de l'outil BigraphER afin de pouvoir s'appuyer sur CompCert.

Références

[1] E. BAINOMUGISHA, A. L. CARRETON, T. van CUTSEM et al., « A Survey on Reactive Programming », *ACM Computing Surveys*, 2013.

[2] F. ARBAB, « Abstract Behavior Types: a foundation model for components and their composition », *Sci. Comput. Program.*, t. 55, n° 1-3, p. 3-52, 2005.

[3] P. ANTOINE et S. CONVERSY, « Volta: the first all-electric conventional helicopter », *Proceedings of the More Electrical Aircraft Conference*, 2017.

[4] T. BOURKE, L. BRUN, P.-E. DAGAND et al., « A Formally Verified Compiler for Lustre », *PLDI*, 2017.

[5] X. LEROY, « Formal Certification of a Compiler Back-end or: Programming a Compiler with a Proof Assitant », *POPL*, 2006.

[6] N. NALPON, C. PICARD, C. ALLIGNOL et al., « Vers la vérification de SMALA, un langage réactif interactif », *AFADL 2020*, 2020.

[7] E. HØJSGAARD, « Bigraphical Languages and their Simulation », PhD Thesis, The IT University of Copenhagen, 2011.

[8] V. DANOS, J. FERET, W. FONTANA et al., « Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models », *FSTTCS 2012*, Hyderabad, India, 2012, p. 276-288.

[9] R. MILNER, *The Space and Motion of Communicating Agents*. New-York : Cambridge University Press, 2009.

[10] M. SEVEGNANI et M. CALDER, « BigraphER: Rewriting and Analysis Engine for Bigraphs », *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada*, 2016, p. 494-501.

PhD Student session: formally verified postpass scheduling with peephole optimization for AArch64

Léo Gourdin *

Université Grenoble Alpes — Verimag & TIMA Laboratories
leo.gourdin@grenoble-inp.org

Abstract

COMPCERT is a C compiler with a complete machine-checked proof of semantic preservation from C to assembly [Ler09]. We present here an extension of COMPCERT for AArch64 processors, that optimizes the use of the pipeline in the processor (*postpass scheduling*) and performs *instruction compaction* (e.g. replaces pairs of simple load instructions, by single double load instructions), through a technique called *peephole optimization*. Our method is founded on a *two-tier design*, introducing an *untrusted oracle* performing the translation, and a *formally-verified* checker testing whether the code produced by the oracle simulates the original code. We reuse here the generic checker, based on *symbolic execution* with an *hash-consing mechanism*, of [SBM20]. The paper presents the correctness proof of my optimizations, and experimental measurements of performance improvements. More generally, my thesis explores how to apply and generalize such mechanisms of *translation validation* in order to extend COMPCERT with target-dependent optimizations.

1 Introduction and related works

An instruction sequence may take significantly less time if executed according to a favorable schedule. Indeed, the simultaneous use of all processor units may be maximized with a smart interleaving of "parallelizable" computations. High-performance processors schedule instructions dynamically, but this complicates their design. COMPCERT is often used by industrials working with Safety-critical systems (SCS) [BFBFF+12], that must remain reliable, not too complex, and predictable. In-order cores, which do not dynamically reorder instructions, are thus a common choice to meet these needs. On such processors, performance may be improved significantly if the compiler schedules instructions intelligently. Within compilers, instruction scheduling is usually split in two passes: a "coarse-grain" one, in an Intermediate Representation (IR), before register allocation and a "fine-grain" one, after register allocation, on the emitted assembly.[1].

Such a (verified) "coarse-grain" prepass scheduling optimization has been recently proposed for COMPCERT [SGBM21]. This paper presents a (verified) "fine-grain" postpass scheduling

*This work has been partially supported by the LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01) funded by the French program Investissement d'avenir.

[1] Combining scheduling and register allocation is useful to avoid a high register pressure, but existing works such as [LCBS19, MPSR95] are challenging and does not seem to scale.

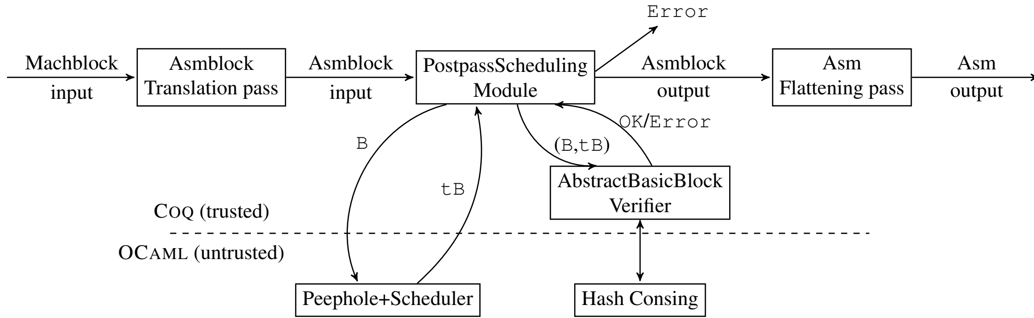


Figure 1: Architecture of our verified optimizations

and a peephole optimization (i.e. instruction compaction on load and store). Both optimizations are performed by untrusted OCAML oracles. We adapt the work of [SBM20], where they present a new IR Asmblock to define *basic blocks*[2] at the assembly level, and a generic checker, formally proved correct in COQ, defined above a dedicated IR called AbstractBasicBlock.

[Nec00] and [TGM11] have experimented that combining *symbolic execution* with *rewriting* is effective to *validate* the code produced by state-of-the-art optimizing compilers. In the meantime, [TL08] have used a *formally-verified* symbolic execution in order to *formally certify* the schedules produced by an untrusted oracle within the COMPCERT compiler. Unfortunately, their checker had exponential complexity [Tri09, §6.7.1], [TL08, §7] and their formally-verified scheduler was never released. [SBM20] tackle this issue with a dynamically verified hash-consing mechanism: an untrusted OCAML oracle memoizes symbolic terms, and its results are dynamically checked with an axiomatized pointer equality.[3]. We port their work on the AArch64 architecture, and also applies this translation validation solution to verify the correctness of peephole replacements. In contrast to the peephole optimizations of [MZTG16] (for x86-32), we do not consider register liveness nor arithmetic transformations of pointers, but they do not tackle instruction scheduling and their model of basic blocks relies on some unverified assumptions.

2 Architecture of our solution and its formal proof

Our solution, pictured in Fig. 1, reuses the generic basic blocks construction method of [SBM20] through Machblock. W.r.t. [SBM20], the whole proof effort consists in adapting to our target the Asmblock IR and the various translations from and to this IR. As pictured in Fig. 1, untrusted optimizations are together applied to each basic block `B` producing a basic block `tB`, which are then both translated to the generic AbstractBasicBlock IR for verification. Hence, the results of our combined oracles are verified in one pass, with a single checker. We apply the peephole

[2]By definition, a *basic block* is a sequence of *assembly instructions* with at most one branching instruction, which is in this case in final position, and such that the ambient program only enters this sequence at the first instruction. Hence, optimizations (e.g. instructions rewriting or reorderings) that (locally) preserve the semantics of the basic block, also (globally) preserve the semantics of the ambient program.

[3]Because representing pointer equality as a “pure” function would be unsound, OCAML pointer equality is instead axiomatized as returning a “non-deterministic” Boolean (within a dedicated monad) such that result “true” implies Coq equality. This model seems a quite reasonable, and enables an efficient symbolic simulation test.

optimization before the scheduling pass in order to leave more scheduling opportunities after load and store pairing.

Based on the existing Asm on the back-end, we build the Asmblock IR by defining a new instruction hierarchy[4]. We prove the correctness of our optimizations thanks to a simulation test on the AbstractBasicBlock IR, deducing the simulation from syntactical equalities on "symbolic states", after symbolic execution[5]. See [SBM20, §4.3]. The overall proof of the simulation of `B` by `tB` corresponds to compose the two commutative diagrams on the right-hand side of Figure 2.

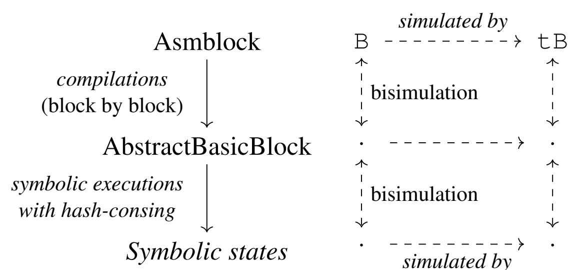


Figure 2: Simulation Test Correctness

3 The Postpass Scheduler

This oracle is declared as a COQ function taking a basic block in input, and returning a tuple containing a list of basic instructions (i.e. the basic block body) and an optional (using the option monad) control-flow instruction (i.e. the basic block exit). The main obstacle is then to retrieve the most precise information possible about latencies of instructions, to be able to correctly "tune" the oracle. The difficulty resides in the fact that measuring correctly the number of execution cycles for each instruction is hard, and the manufacturer in the case of the Cortex-A53 (i.e. ARM) does not provide such information. However, the AArch64 LLVM back-end is using a similar postpass scheduling optimization, and the source code[6] contains some latency information. Another source we used is an article by [Wig19] where some latencies are manually measured.

Concerning the set of read and written registers for each instruction, it could be deduced from the Asm semantics. When implementing our solution, we have discovered bugs in this semantics: indeed, some instructions such as `Pfmovimms`, `Pfmovimmd` and `Pbtbl` were incorrectly described. The first two are destroying a scratch register before writing the result in the destination register, and the latter is in fact preserving a scratch register contrary to what its semantics describes. These three instructions are macros expanded later on in an unverified part of COMPCERT, the TargetPrinter, into several real Asm instructions. The formalization of their behavior in COQ was not correct, and it was possible to generate incorrect code by interleaving them with other macros that are using the same scratch register, and which are expanded in COQ, at the Asm level (so before our scheduling). The bug was invisible in the sense that as instructions were never reordered at the Asm level, an incorrect code could not appear (the only way was either to reschedule instructions as we do, or to write it manually). Thus, our verifier combined with the postpass scheduling can help us find errors in the trusted base of COMPCERT[7].

[4] We do not detail the chosen hierarchy here, but the interested reader can note that a smart grouping of instructions by operands (according to the number and type of input registers) helps to produce more compact definitions in the checker and a shorter overall proof.

[5] This method simply consists in compiling each program into a big symbolic term, called a symbolic state

[6] Accessible here as a TABLEGEN code.

[7] Those bugs are now fixed in the COMPCERT mainline repository.

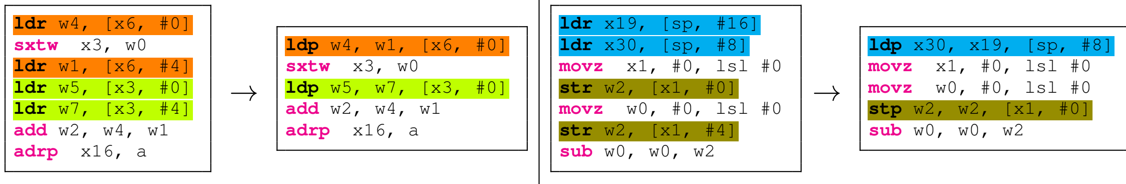


Figure 3: Four Examples of load/store Compaction on AArch64

4 The peephole optimizer

In contrast to the peephole optimizer of [SBM20], ours is able to merge non-consecutive load or store within the original basic block, as long as they respect the semantic dependencies and offset constraints on double load/store specific to AArch64 Instruction Set Architecture (ISA). Our algorithm traverses the basic block in both directions, while remembering every encountered compatible load and store as potential candidates (and forgetting them if another instruction breaks a needed dependency in-between). The first pass (forward) tries to replace the last encountered load or store by the double instruction, and the first one by a `Nop` (no operation) instruction. The second pass (backward) tries the opposite.

Figure 3 illustrates four situations found by our peephole optimizer. On the left column: 1. backward load pairing, with increasing offset (the offset of the second load is *greater* than that of the first one); 2. consecutive load pairing, with increasing offset. On the right column: 1. consecutive load pairing, with decreasing offset (the offset of the second load is *lower* than that of the first one); 2. forward store pairing, with increasing offset.

Currently, the main benefit of our peephole optimizer for AArch64 is a reduction of code size: it reduces the number of generated memory transfer instructions by about 10%, which represents approximately 3% of the total code length (on average across all our benchmarks). Like [SBM20], our formally-verified simulation test validates these rewritings by performing the reverse rewriting (i.e. from double load/store to pairs of simple load/store) in the Asmblock-to-AbstractBasicBlock pass (see Fig. 2).

5 Conclusion and future work

Using such a low level scheduling pass allows a finer tuning of instructions latencies compared to the existing prepass. On average, running on a Raspberry Pi 3 with a Cortex-A53 core, our oracle alone raises performance by 9.11% across all our benchmarks[8], and using prepass scheduling [SGBM21] on top of postpass makes us reach 22% of performance improvement. The postpass itself brings a gain of about 5.46% comparing to CompCert with all optimizations turned on[9] as they allow generating larger, more profitable, basic blocks. The overall implementation of our formally-verified optimization pipeline on AArch64 represents a bit more than three man·months of development. This constitutes for us the first step for future optimizations exploiting the same principle of a posteriori verification, which we will study in the rest of the thesis.

[8] Based on Polybench [Pou12], TACLeBench [FAH+16], and some additional benches described in [SBM20].

[9] Including among others Loop Invariant Code Motion (LICM), loop-unrolling, loop-rotate, and tail duplication.

References

[BFBFF+12] Ricardo Bedin França, Sandrine Blazy, Denis Favre-Felix, Xavier Leroy, Marc Pantel, and Jean Souyris. Formally verified optimizing compilation in ACG-based flight control software. In *Embedded Real Time Software and Systems (ERTS2)*. AAAF, SEE, February 2012.

[FAH+16] Heiko Falk, Sebastian Altmeyer, Peter Hellinckx, Björn Lisper, Wolfgang Puffitsch, Christine Rochange, Martin Schoeberl, Rasmus Bo Sørensen, Peter Wägemann, and Simon Wegener. TACLeBench: A benchmark collection to support worst-case execution time research. In Martin Schoeberl, editor, *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, volume 55 of *OpenAccess Series in Informatics (OASIcs)*, pages 2:1–2:10, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum für Informatik.

[LCBS19] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. Combinatorial register allocation and instruction scheduling. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 41(3):1–53, 2019.

[Ler09] Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[MPSR95] Rajeev Motwani, Krishna V Palem, Vivek Sarkar, and Salem Reyen. Combining register allocation and instruction scheduling. *Courant Institute, New York University*, 1995.

[MZTG16] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for compcert. In Chandra Krintz and Emery Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 448–461. ACM, 2016.

[Nec00] George C. Necula. Translation validation for an optimizing compiler. In *Programming Language Design and Implementation (PLDI)*, pages 83–94. ACM Press, 2000.

[Pou12] Louis-Noël Pouchet. the polyhedral benchmark suite, 2012.

[SBM20] Cyril Six, Sylvain Boulmé, and David Monniaux. Certified and efficient instruction scheduling. Application to interlocked VLIW processors. *PACMPL (OOPSLA 2020)*, November 2020.

[SGBM21] Cyril Six, Léo Gourdin, Sylvain Boulmé, and David Monniaux. Verified Superblock Scheduling with Related Optimizations. preprint, April 2021.

[TGM11] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 295–305. ACM, 2011.

[TL08] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL*, pages 17–27. ACM Press, 2008.

[Tri09] Jean-Baptise Tristan. *Formal verification of translation validators*. PhD thesis, Université Paris 7 Diderot, November 2009.

[Wig19] Thom Wiggers. *Energy-Efficient ARM64 Cluster with Cryptanalytic Applications: 80 Cores That Do Not Cost You an ARM and a Leg*, pages 175–188. 07 2019.

Vérification d'une bibliothèque mathématique d'un autopilote avec Frama-C

Baptiste Pollien
baptiste.pollien@isae-supaero.fr

AFADL 2021 - Session doctorants

Résumé

Lors du développement de système critiques, comme par exemple un autopilote de drone, il est essentiel de s'assurer que le programme est sûr, en utilisant par exemple des méthodes formelles. Pour faciliter la vérification, on se restreint généralement à une abstraction du système ou un sous-ensemble. Cet article présente la vérification d'une bibliothèque mathématique de l'autopilote Paparazzi, à l'aide de l'outil Frama-C, afin de garantir l'absence d'erreur à l'exécution et certaines propriétés fonctionnelles.
Mots clés : Preuve de programme, méthodes déductives, interprétation abstraite

1 Introduction

Les méthodes formelles sont des techniques de vérification basées sur des modèles mathématiques qui permettent de vérifier et de garantir certaines propriétés. Il existe de nombreux outils de vérification formelle qui possèdent des spécificités en termes des propriétés vérifiables et qui nécessitent des efforts plus ou moins importants de spécification ou de modélisation du système afin de les mettre en œuvre.

L'objectif de ma thèse est de procéder à une revue des différentes techniques de vérification formelle afin de définir un processus d'analyse d'une architecture d'autopilote de drone tirant parti de ces techniques. Ce processus d'analyse est appliqué au cas de l'autopilote Paparazzi développé à l'ENAC en langage C.

Frama-C [3] est un outil d'analyse de code C qui nécessite l'ajout d'annotations dans le code pour spécifier les propriétés attendues : définition de contrats pour les fonctions (*préconditions*, *postconditions* et pour spécifier l'ensemble des éléments mémoire (hors pile) qui seront modifiés lors de l'exécution de la fonction), définition des *invariants* et *variants* de boucle pour vérifier la terminaison et l'ajout d'assertion. Frama-C dispose de nombreux greffons dont trois nous intéressent pour vérifier formellement les annotations : WP (*Weakest Precondition* qui utilise un calcul de plus faible préconditions), RTE (*RunTime Errors*) pour l'ajout automatique

d'assertions afin de vérifier l'absence d'erreur à l'exécution et EVA (*Evolved Value Analysis*) utilisant des techniques d'analyse statique par interprétation abstraite.

Dans le cadre de cet article, la vérification a été effectuée avec la plateforme Frama-C en n'utilisant que des démonstrateurs automatiques et s'est limitée à une bibliothèque mathématique de Paparazzi présentée en section 2. La section 3 présente la première partie de l'analyse concernant la vérification de l'absence d'erreur à l'exécution. La seconde partie, présentée en section 4, présente la vérification de propriétés fonctionnelles pour certaines fonctions mathématiques en ne considérant pas les erreurs d'arrondies potentielles liées aux calculs sur les nombres à virgule flottante.

2 Paparazzi

Paparazzi [2] est un autopilote open-source (sous licence GPL) développé à l'ENAC depuis 2003. Il supporte differents types de drones et permet le contrôle simultané de plusieurs drones. Paparazzi possède également différents modes intégrés et offre la possibilité de créer des plans de vol personnalisés. La bibliothèque mathématique étudiée correspond à la conversion de rotation de vecteurs dans différentes représentations (matrices de rotation, angles d'Euler, quaternions). Elle définit également certaines opérations élémentaires sur ces représentations. Elle est composée d'environ 4000 lignes de code C et chaque fonction dispose d'une version travaillant sur les types `int`, `float` et `double`. Cette bibliothèque est principalement utilisée pour faire le lien entre les capteurs et la partie contrôle de l'autopilote qui n'utilisent pas nécessairement les mêmes représentations. La vérification de cette bibliothèque permet donc de garantir la cohérence des données traitées malgré l'utilisation de représentations différentes.

3 Absence d'erreurs à l'exécution

La bibliothèque définit des structures C pour représenter les données manipulées (matrices de rotation, quaternions, vecteurs...). Les fonctions de la bibliothèque travaillent uniquement par référence pour les entrées et pour les sorties. Afin d'éviter des erreurs liées à un déréférencement de pointeurs non valides, des préconditions garantissant la validité des références ont été ajoutées dans les contrats des fonctions. Il a aussi été nécessaire de spécifier les variants et invariants de boucle ainsi que les variables de sorties comme seuls espaces mémoire (hors pile) qui seront modifiées.

WP dispose de différents modèles arithmétiques qui prennent en compte de façon plus ou moins précise la sémantique de C. La vérification de la bibliothèque sur les entiers a été faite en utilisant le modèle réaliste de l'arithmétique machine des entiers. Avec le greffon RTE, il est alors nécessaire de vérifier qu'il n'y a pas de débordement de valeur (ou *overflow* en anglais) pour chaque opération arithmétique. Pour vérifier l'absence de dépassement, chaque fonction a été analysée dans l'objectif de déterminer les bornes maximales possibles des différentes variables. Lorsque

ces bornes ont pu être déterminées, elles ont été ajoutées en préconditions dans les contrats des fonctions.

Malheureusement, WP associé à des prouveurs automatiques n'est pas parvenu à vérifier ces nouveaux contrats. L'utilisation de références pour l'accès aux valeurs numériques surcharge les prouveurs et ce même en spécifiant en précondition que les structures en paramètres sont stockées à des emplacements mémoire séparés.

Pour pallier ce problème, nous avons décidé d'associer EVA à WP. EVA arrive à calculer des intervalles suffisament précis des valeurs possibles pour chaque variable. Ce résultat est ensuite transmis à WP par Frama-C ce qui permet de conclure plus facilement les preuves. Cette limitation de WP avait aussi été notée par Vassil Todorov durant sa thèse [5] et il avait également utilisé un outil d'analyse statique par interprétation abstraite, Astrée, pour résoudre ce problème. En conclusion, lorsque l'on associe EVA avec WP, il est possible de vérifier l'absence d'erreur à l'exécution des fonctions de la bibliothèque sur les entiers.

WP dispose également d'un modèle arithmétique `real` qui correspond à l'arithmétique réelle au sens mathématique. Pour la vérification des versions de la bibliothèque travaillant sur des valeurs numériques à virgule flottante (aussi bien `float` ou bien `double`), nous avons décidé d'utiliser ce modèle. Il nous a permis de vérifier l'absence de division par 0 et que les variables ne prennent pas la valeur `NaN` (*Not A Number*). Pour effectuer ces vérifications, il a été seulement nécessaire d'ajouter comme préconditions que chaque valeur numérique passée en paramètre ne prend pas la valeur *NaN* et qu'elle n'est pas infinie. Là encore, l'absence de ces deux erreurs à l'exécution et la terminaison des fonctions ont été prouvées pour les versions `float` et `double` de la bibliothèque en utilisant WP et EVA. Par contre, notre vérification n'offre aucune garantie sur le risque de dépassement ou sur les erreurs liées aux arrondis. Cependant, ce modèle nous a été particulièrement utile pour la vérification des propriétés fonctionnelles présentée en section 4.

4 Vérification fonctionnelle de propriétés

La vérification fonctionnelle permet de garantir les résultats attendus d'une fonction. Dans notre cas d'étude nous avons décidé de vérifier certaines propriétés fonctionnelles de la fonction float_rmat_of_quat car elle est représentative et indépendante des autres fonctions présentes dans la bibliothèque. Cette fonction prend en paramètre un quaternion normalisé et retourne la matrice de rotation correspondante.

Afin de spécifier les propriétés fonctionnelles, des types et des prédicats ont été définis dans la logique fournie par le language ACSL. On retrouve la définition des types pour les matrices et les quaternions ainsi que des opérations élémentaires. Des lemmes ont ensuite été spécifiés et vérifiés pour garantir que ces opérations sont correctes. Une fonction logique qui convertit un quaternion en une matrice de rotation a également été définie, indépendamment du code C. Cette fonction est basée sur la formule mathématique de conversion d'un quaternion vers une matrice

de rotation [1, 4].

À partir de ces définitions, le contrat de la fonction a pu être établi. En supposant que le quaternion passé en paramètre est normalisé, nous avons voulu vérifier 2 propriétés fonctionnelles. La première est que la matrice retournée correspond bien à la conversion du quaternion passé en paramètre : notre post-condition vérifie que la matrice de rotation générée par le code C est égale à la matrice de rotation générée par notre fonction logique. Comme dans la section précédente, nous utilisons le modèle `real` de WP pour la vérification de cette fonction, ce qui permet d'ignorer les différences de résultats entre la version C et la version mathématique qui auraient pu être liées à des erreurs d'arrondis. La seconde propriété vérifie que la matrice générée est bien une matrice de rotation, i.e. que la transposée de la matrice est son inverse.

Malgré l'utilisation du modèle `real` d'arithmétique, WP n'arrivait pas à vérifier le contrat. Il a donc été nécessaire d'analyser le code. Nous avons remarqué que le code C utilisait une constante `M_SQRT2` pour représenter $\sqrt{2}$ et qu'en simplifiant les calculs, la constante `M_SQRT2` était tout le temps multipliée à elle-même. Nous avons donc suggéré une modification du code en remplaçant ces opérations par une multiplication par 2. Cette modification ne change pas le nombre de multiplications mais permet de réduire les erreurs d'arrondis propagées par la fonction. Avec ces modifications de code et avec le modèle `real` pour l'arithmétique, WP vérifie le contrat de la fonction `float_rmat_of_quat`. Ce contrat garantit bien l'absence d'erreur à l'exécution et définit les propriétés fonctionnelles attendues. Ces propriétés permettent de vérifier uniquement le comportement idéaliste de la fonction sans considérer les erreurs potentielles de calcul.

5 Conclusion

Dans cet article, nous avons pu présenter mon travail de vérification d'une bibliothèque mathématique de Paparazzi qui est disponible sur GitLab. Ce travail s'est principalement concentré sur la vérification de l'absence d'erreurs à l'exécution et a nécessité l'ajout de près de 3,5k lignes d'annotations sur le code.

Dans la continuité de ce premier travail, il est envisagé de vérifier fonctionnellement d'autres fonctions de la bibliothèque mathématique de Paparazzi et d'étudier les erreurs d'arrondis en ne se limitant plus au modèle `real` mais en utilisant un modèle représentant plus fidèlement les flottants.

Remerciements

Je remercie mon directeur de thèse Xavier Thirioux (ISAE-SUPAERO) et mes co-encadrants Christophe Garion (ISAE-SUPAERO), Gautier Hattenberger (ENAC) et Pierre Roux (ONERA). Je tiens également à remercier l'Agence pour l'Innovation de Défense (AID) du Ministère des Armées pour le financement de cette thèse (projet de recherche CONCORDE N° 2019 65 0090004707501).

Références

[1] Carl Grubin. Derivation of the quaternion scheme via the euler axis and angle. *Journal of Spacecraft and Rockets*, 7(10) :1261–1263, 1970.

[2] Gautier Hattenberger, Murat Bronz, and Michel Gorraz. Using the Paparazzi UAV System for Scientific Research. In *IMAV 2014, International Micro Air Vehicle Conference and Competition 2014*, pages pp 247–252, Delft, Netherlands, August 2014.

[3] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. *Formal aspects of computing*, 27 :573–609, 2015.

[4] Allan R. Klumpp. Singularity-free extraction of a quaternion from a direction-cosine matrix. *Journal of Spacecraft and Rockets*, 13(12) :754–755, 1976.

[5] Vassil Todorov. *Automotive embedded software design using formal methods*. Phd thesis, Université Paris-Saclay, December 2020.

La double précision suffit-elle à l'exascale ?

Louise Ben Salem-Knapp[1,2], Thibaud Vazquez-Gonzalez[1] et
William Weens[1,3]
{louise.bensalem-knapp, thibaud.vazquez-gonzalez, william.weens}
@cea.fr

[1]*CEA DAM DIF, F-91297, Arpajon Cedex, France ;* [2]*Université Paris-Saclay, CNRS, ENS Paris-Saclay, Inria, Laboratoire Méthodes Formelles, 91190, Gif-sur-Yvette, France ;* [3]*Université Paris-Saclay, CEA, Laboratoire en Informatique Haute Performance pour le Calcul et la Simulation, 91680 Bruyères-le-Châtel, France*

Résumé

La croissance des capacités de calcul des machines permet d'obtenir des résultats de simulation de plus en plus précis. Ces résultats sont souvent calculés en binary64 (double précision) avec l'idée que les erreurs d'arrondi ne sont pas significatives. Or, l'*exascale* permet d'augmenter le nombre d'opérations et des problèmes d'accumulation d'erreurs d'arrondi pourraient apparaître. Augmenter la précision des nombres flottants pour remédier à ce problème n'est pas envisageable car le surcoût en mémoire, en temps de calcul et en énergie ferait perdre une partie importante des performances des nouvelles machines. Il est donc important de mesurer la robustesse du binary64 en anticipant les ressources de calcul à venir afin d'assurer la pérennité de celle-ci dans les simulations numériques. C'est dans ce but que des expériences numériques ont été réalisées et sont présentées dans cet article. En montrant que les erreurs d'arrondi restent dominées par les erreurs du schéma, les résultats ont permis de valider le binary64 dans ces simulations.

1 Introduction

L'industrie de la simulation HPC s'applique à des domaines scientifiques toujours plus nombreux — biologie, météorologie, astrophysique, aérodynamique, etc. —, et doit donc évoluer régulièrement pour s'adapter aux changements théoriques et matériels, ainsi qu'aux nouvelles demandes réglementaires ou commerciales. Par exemple, les résultats de simulation doivent de plus en plus souvent être assortis d'une mesure des incertitudes [5].

Dans un contexte où les ressources de calcul (HPC) sont toujours renouvelées (GP-GPU), de plus en plus parallélisées (MPI, Multithreading, Vectorisation) et les performances constamment améliorées, il n'est pas surprenant de voir les attentes vis-à-vis des résultats de simulation se renforcer. Pour des résultats à la

fois plus précis et mieux contrôlés, la stratégie couramment employée pour profiter des possibilités promises par le HPC consiste à appliquer des schémas numériques éprouvés à des problèmes de taille augmentée. Cependant, cette stratégie génère en elle-même davantage d'erreurs, ce qui peut paraître contradictoire. En effet, comme les calculs en précision finie introduisent des erreurs d'arrondi, plus la taille des problèmes simulés augmente, plus le nombre d'opérations augmente, et avec lui la proportion d'erreur, dégradant ainsi les résultats. C'est pourquoi il est important de rechercher le meilleur compromis entre réduction des erreurs par une description plus fidèle du système étudié et dégradation des résultats par les erreurs d'arrondi dues à l'arithmétique flottante. D'où la question pour la prochaine génération de super-calculateurs : *la double précision suffit-elle à l'exascale ?*

L'utilisation des outils théoriques usuels ne fournit pas de réponses satisfaisantes à cette question. À cause du grand nombre d'opérations dans une simulation et de la complexité des programmes, les bornes d'erreur obtenues sont souvent trop grandes pour être utiles. Des approches d'encadrement des erreurs d'arrondi basées sur les preuves formelles ont ainsi récemment vu le jour. L'objectif de celles-ci est de déterminer des bornes théoriques fines d'encadrement qui soient davantage compatibles avec les tolérances de l'industrie. On notera en particulier les travaux de [2] pour l'étude d'un schéma en différences finies sur l'équation d'onde, ainsi que les travaux de [3] pour l'analyse de la méthode de Runge-Kutta. Néanmoins, pour des programmes de simulations plus complexes, l'approche empirique reste nécessaire afin de détecter le poids réel des erreurs d'arrondis dans les résultats numériques.

Le travail présenté dans cet article s'inscrit dans cette volonté de mieux quantifier l'erreur numérique liée à l'arithmétique flottante dans les simulations d'hydrodynamique. Elle s'appuie sur une méthode d'estimation de l'erreur d'arrondi par sous précision dont l'implémentation dans nos outils d'analyse a été nommé *weak floats*. La polyvalence des outils développés, leur facilité d'emploi et leur rapidité d'exécution ont permis d'obtenir des données suffisantes pour réaliser une preuve de concept de leur application pour une analyse empirique de nombreux modèles physiques plus complets.

La section 2 introduit le modèle d'erreur numérique et présente les *weak floats*. La section 3 décrit ensuite le modèle physique et les cas tests utilisés. Les résultats numériques obtenus avec les types flottants de la norme IEEE-754 [6] [10] et les *weak floats* sont ensuite analysés en section 4. Enfin, la section 5 conclut le travail, et propose une tentative de réponse à la question du titre en se basant sur une extrapolation des résultats obtenus.

2 Modèle de l'erreur numérique

2.1 Mesure de l'erreur due à l'arithmétique flottante

Nous nous intéressons à l'erreur numérique, notée ε^{num}, que nous définissons comme l'écart entre le résultat théorique issu du modèle mathématique, y^{model}, et

le résultat obtenu par la simulation, $y^{simulation}$. Dans ce travail, nous avons utilisé la norme L_2 et nous nous concentrons sur le cas où y^{model} et $y^{simulation}$ sont des champs spatio-temporels, si bien que :

$$\varepsilon^{num} := ||y^{model} - y^{simulation}||_{L_2} \ . \tag{1}$$

L'erreur numérique ε^{num} se caractérise par deux contributions : l'erreur due à la discrétisation des équations, ε^{scheme}, et l'erreur créée par les calculs en précision finie, ε^{float}. En effet, toute discrétisation d'équations continues introduit une erreur, mais si le schéma numérique considéré est consistant, cette erreur de discrétisation ε^{scheme} peut se voir comme l'erreur causée par le manque d'information. Il suffirait donc d'ajouter de l'information (des mailles, des particules, etc.) pour diminuer cette erreur — la diminution de l'erreur par rapport à l'information injectée définissant l'ordre du schéma. Par ailleurs, l'erreur provenant des calculs en précision finie ε^{float} résulte de l'accumulation d'erreurs due à l'arrondi commis pour chaque opération [8] [10].

Pour des raisons de stabilité numérique, plus on discrétise finement les équations (ce qui revient à ajouter de l'information), plus on augmente le nombre d'opérations. Il est ainsi clair que les deux erreurs, ε^{scheme} et ε^{float}, jouent dans des sens opposés. Pour être plus précis, c'est-à-dire pour réduire ε^{scheme}, il est nécessaire d'effectuer plus d'opérations. Mais chaque opération peut introduire une erreur d'arrondi supplémentaire, ce qui risque d'augmenter ε^{float}. On s'attend ainsi à ce que ε^{num} admette un minimum en fonction du nombre d'opérations sur lequel repose la simulation, correspondant à une certaine taille de maillage qu'il ne faudrait pas dépasser.

2.2 Modèle d'erreur sur un schéma numérique explicite

De même que pour une équation différentielle ordinaire [1], le comportement attendu de l'erreur numérique pour un problème en dimension 1 discrétisé par un schéma numérique à l'ordre 1 est le suivant :

$$\varepsilon^{num} \leq C_1 h + C_2 h^{-1} \ , \tag{2}$$

où C_1, C_2 sont des constantes propres au cas étudié et au schéma, et h un paramètre caractérisant la finesse de discrétisation des équations (par exemple égal au pas d'espace pour une discrétisation spatiale en 1D). On retrouve l'ordre de convergence du schéma numérique dans le premier membre, $C_1 h$. Le second membre, $C_2 h^{-1}$, fait référence à l'accumulation de l'erreur d'arrondi. Il dépend du nombre d'opérations et donc du pas d'espace. Lorsque le nombre de mailles devient infini, et donc que h tend vers 0, on s'attend à ce que le comportement asymptotique de ces deux contributions à l'erreur numérique soit donné par :

$$\lim_{h \to 0} \varepsilon^{scheme} = 0 \ , \quad \lim_{h \to 0} \varepsilon^{float} = +\infty \ . \tag{3}$$

Cette décomposition de l'erreur numérique est illustrée qualitativement dans la fig. 1, qui souligne la part de ε^{num} due à la discrétisation ε^{scheme} en hachures bleues et celle due aux erreurs d'arrondi ε^{float} en hachures vertes.

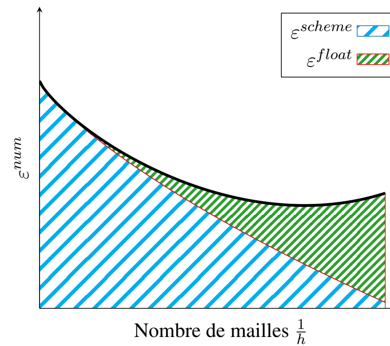


FIGURE 1 – Représentation qualitative du comportement de l'erreur numérique et de ses composantes dans un schéma convergent.

Pour certifier que le code est utilisé dans son domaine de validité, il semble ainsi nécessaire d'obtenir une estimation du point de divergence de l'erreur numérique, c'est-à-dire le nombre de mailles pour lequel la solution simulée commence à s'écarter significativement de la convergence théorique du schéma.

2.3 Les weak floats

Dans ce travail, nous proposons d'utiliser les *weak floats* pour mesurer la part de l'erreur d'arrondi dans les simulations. L'idée des *weak floats* est d'utiliser des précisions inférieures au binary64, de trouver les points de divergence pour ces sous-précisions, puis d'extrapoler les valeurs obtenues pour estimer le point de divergence du binary64. Une approche similaire exploitant une sous-précision a été utilisée dans [7]. Pour des raisons de rapidité, les opérations entre *weak floats* s'effectuent sur un type souche (*single*, *double*, *long double*), dont la taille de la mantisse est supérieure à celle du *weak float*, puis le résultat est tronqué et arrondi. Les *weak floats* conservent la plage d'exposants du flottant utilisé comme souche. Dans la suite, *weak* N désigne un flottant en binary64 dont les N premiers *bits* de la mantisse sont utilisés, les autres étant tronqués (donc une précision de $N+1$ *bits* avec le *bit* implicite).

On notera qu'en plus des opérations de troncature et d'arrondi s'ajoute l'impossibilité pour les compilateurs d'utiliser de nombreuses optimisations. Ceci se traduit par un ralentissement de l'exécution pour les *weak floats* alors même que la précision est réduite.

3 Description de l'expérience numérique

Les équations de l'hydrodynamique — appelées aussi équations d'Euler — sont un bon candidat pour évaluer l'impact des erreurs d'arrondi. Ce système

d'équations différentielles couplées est au coeur d'un grand nombre de simulations numériques réalisées dans l'industrie et le monde académique. C'est pourquoi il est important de pouvoir garantir une certaine qualité des résultats obtenus.

Le système d'équation d'Euler pour un fluide s'écrit

$$\partial_t \begin{pmatrix} \rho \\ \rho \boldsymbol{u} \\ \rho E \end{pmatrix} + \boldsymbol{\nabla} \cdot \begin{pmatrix} \rho \boldsymbol{u} \\ \rho \boldsymbol{u} \otimes \boldsymbol{u} + p \mathbb{I}_3 \\ (\rho E + p) \boldsymbol{u} \end{pmatrix} = \boldsymbol{0} \; , \qquad (4)$$

avec ρ, E, p et $\boldsymbol{u}$ respectivement la densité, l'énergie totale spécifique, la pression et le vecteur vitesse du fluide considéré.

Le système d'équation (4) est fermé en utilisant une équation d'état reliant la pression, l'énergie et la densité. Seule l'équation d'état des gaz parfaits est utilisée dans la suite. Cette équation d'état s'écrit : $p = (\gamma - 1)\rho e$, avec γ le coefficient polytropique et e l'énergie interne spécifique du fluide.

Dans les simulations présentées, nous avons calculé l'approximation numérique des solutions avec un schéma de type Godounov à l'ordre 1 avec le solveur de Riemann approchée Rusanov (méthode de Godounov [4], [11], [9]).

Nous avons retenu deux cas tests pour cette étude. Les cas utilisés possèdent une solution analytique permettant des mesures précises de l'impact des erreurs d'arrondi. Ils font aussi intervenir plusieurs caractéristiques que l'on retrouve dans les simulations numériques à grande échelle réalisées dans l'industrie ou le monde académique (choc, discontinuité de contact, détente, grand déplacement, etc.)

Le premier cas-test est une advection à vitesse constante d'un profil de densité gaussien sur un domaine avec des conditions aux bords périodiques. La solution analytique du profil de densité au temps final est identique au profil initial puisqu'elle correspond à un tour complet.

Le second cas-test, qui correspond au tube à choc de Sod, est un problème de Riemann à deux états. Ce test 1D est réalisé avec des conditions aux bords de Neumann. À l'instant initial, les deux fluides sont au repos, et soumis à des pressions différentes. Dès $t > 0$, la différence de pression entre les deux états va créer un choc, une discontinuité de contact et une détente.

Remarque. Dans les cas tests utilisés, on observe que le schéma numérique ne produit pas d'événements catastrophiques (perte sévère de nombres significatifs). L'erreur apparaît de façon inhomogène en espace, mais elle apparaît progressivement avec le nombre d'opérations sans saut brutal.

4 Résultats & discussions

Les résultats de cette section sont présentés sous forme de graphiques en échelle logarithmique avec : en abscisse le nombre de mailles et en ordonnée la différence en norme L^2 entre la densité moyenne par maille obtenue par la simulation, notée $\bar{\rho}_{sim.}$, et celle de la solution analytique, notée $\bar{\rho}_{exact}$: $\varepsilon^{num} := \|\bar{\rho}_{sim.} - \bar{\rho}_{exact}\|_{L^2}$.

Dans le cas général, lorsque la solution analytique n'est pas connue, il est possible d'utiliser la solution obtenue avec le binary64 comme solution de référence, s'il est possible de s'assurer de sa convergence en maillage.

4.1 Variations sur la taille de la mantisse

La fig. 2 est un résumé d'un ensemble de simulations concernant l'advection d'une gaussienne simulée avec différents *weak floats*. Chaque point de chaque courbe correspond à l'erreur numérique ε^{num} issue d'une simulation complète. Sur cette figure, il est d'abord intéressant de remarquer que l'on retrouve bien une évolution de l'erreur conforme à la prédiction du modèle qualitatif (équation (2) et fig. 1). On observe également que le binary64 suit l'ordre du schéma. À l'inverse, les autres précisions divergent par rapport à cette pente et on peut voir que l'abscisse du point de divergence de chaque type de nombre flottant croit avec la taille de la mantisse.

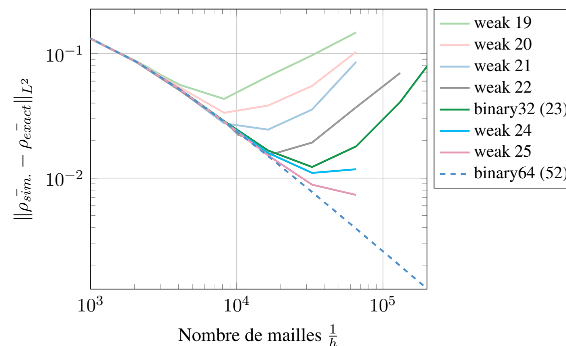


FIGURE 2 – Courbe d'erreur de l'advection d'une gaussienne simulée avec différentes précisions.

4.2 Prédiction du point de divergence du binary64

La figure 2 nous montre que le binary64 ne semble pas affectée par l'erreur d'arrondi pour un nombre de mailles allant jusqu'au million. Cela ne signifie pas que le point de divergence du binary64 n'existe pas, mais qu'il ne peut simplement pas être directement déterminé par la simulation. Il semble néanmoins possible de prédire ce point par extrapolation. En effet, en faisant varier la taille de mantisse, on observe une quasi-linéarité du point de divergence entre la taille de la mantisse et le nombre d'opérations.

Une recherche systématique du point de divergence pour un cas-test donné permet de tracer les deux graphiques de la fig. 3. Cette figure montre, pour chaque précision étudiée, le point de divergence obtenu. Par régression linéaire, on confirme expérimentalement la linéarité en échelle logarithmique. Chaque cas semble produire de l'erreur d'arrondi régulièrement au cours du calcul. Le taux de production de l'erreur croit régulièrement et sa valeur dépend de la mantisse utilisée. Concrètement, pour les cas étudiés, cela implique qu'un *bit* supplémentaire dans la mantisse

permet de repousser exponentiellement le nombre de mailles nécessaires (et donc d'opérations) pour atteindre le point de divergence. Le point de divergence n'est pas pour autant doublé pour chaque *bit* de mantisse ajouté car avec plus de mailles, les résultats deviennent plus précis et nécessitent plus de *bits* dans la mantisse.

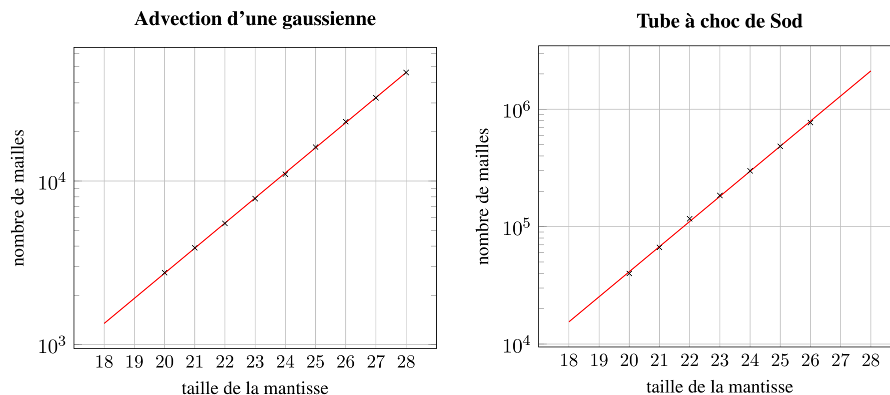


FIGURE 3 – Nombre de mailles du point de divergence en fonction de la mantisse pour deux cas. Les diagrammes sont tracés en échelle logarithmique sur l'axe des ordonnées. La droite (en rouge) est obtenue par régression linéaire.

Après extrapolation par régression linéaire, les droites obtenues sont très proches des points de divergence et autorisent donc une extrapolation. On en conclut qu'en dimension 1, le nombre d'opérations nécessaire pour faire diverger le binary64 est de 220 millions de mailles avec 9 milliards d'itérations pour l'advection et de 286 milliards de mailles avec 251 milliards d'itérations pour le tube à choc. Ces valeurs sont plusieurs ordres de grandeur au-dessus des tailles utilisées ou qui seront utilisées sur l'*exascale*.

5 Conclusion

Grâce aux *weak floats* et aux résultats obtenus sur des maillages de taille élevée du code de calcul massivement parallèle VARIANT [12], une étude de l'impact des erreurs d'arrondi a pu être menée sur des simulations d'hydrodynamique de gaz parfaits en condition HPC.

Les simulations présentées dans cette étude ont pu atteindre le point de divergence sur plusieurs tailles de mantisse. Nous avons constaté que ces points de divergence croissent linéairement avec la taille de la mantisse. Cette croissance régulière nous a autorisé à extrapoler le nombre de mailles nécessaires pour atteindre le point de divergence du binary64. Ce nombre se trouve être beaucoup trop grand pour être simulé, même sur les machines exaflopiques. Ce résultat renforce notre confiance dans le binary64 pour l'utilisation de maillages extrêmement fins en hydrodynamique et permet même d'envisager de réduire la précision sur certains calculs sans impacter le résultat.

La prochaine étape consiste à s'intéresser à des codes plus complexes que les cas-tests étudiés dans ce travail, d'un point de vue empirique (par exemple avec les

weak floats) mais également théorique, ces deux approches étant complémentaires. Pour la partie théorique, comme il semble possible de calculer les bornes de l'erreur d'arrondi sur chaque routine du code, et comme ces routines sont communes à de nombreux schémas (reconstruction, limiteurs, solveurs, Runge-Kutta), il s'agirait de chercher à assembler les bornes théoriques pour construire une borne globale de l'erreur d'arrondi. Ces bornes seront plus larges que les bornes empiriques, mais elles devraient permettre de valider n'importe quel cas test.

Références

[1] Atkinson, K.E., Han, W., Stewart, D. : Euler's method, chap. 2, pp. 15–36. John Wiley & Sons, Ltd (2011)

[2] Boldo, S., Clément, F., Filliâtre, J.C., Mayero, M., Melquiond, G., Weis, P. : Wave equation numerical resolution : a comprehensive mechanized proof of a C program. Journal of Automated Reasoning **50**(4), 423–456 (04 2013)

[3] Boldo, S., Faissole, F., Chapoutot, A. : Round-off error and exceptional behavior analysis of explicit Runge-Kutta methods. IEEE Transactions on Computers (2019)

[4] Godunov, S.K. : Eine Differenzenmethode für die Näherungsberechnung unstetiger Lösungen der hydrodynamischen Gleichungen. Mat. Sb., Nov. Ser. **47**, 271–306 (1959)

[5] Heroux, M.A., Carter, J., Thakur, R., Vetter, J.S., McInnes, L.C., Ahrens, J., Munson, T., Neely, J.R. : Ecp software technology capability assessment report. www.exascaleproject.org (02 2020)

[6] IEEE : IEEE Standard for Floating-Point Arithmetic. Institute of Electrical and Electronics Engineers IEEE Std 754-2008 pp. 1–70 (2008)

[7] Izquierdo, L.R., Polhill, J.G. : Is Your Model Susceptible to Floating-Point Errors ? Journal of Artificial Societies and Social Simulation **9**(4), 1–4 (2006)

[8] Knuth, D.E. : The Art of Computer Programming, Volume 2 : Seminumerical Algorithms. Addison-Wesley, Boston, third edn. (1997)

[9] LeVeque, R.J. : Finite Volume Methods for Hyperbolic Problems. Cambridge Texts in Applied Mathematics, Cambridge University Press (2002)

[10] Muller, J.M., Brunie, N., de Dinechin, F., Jeannerod, C.P., Joldes, M., Lefèvre, V., Melquiond, G., Revol, N., Torres, S. : Handbook of Floating-Point Arithmetic, 2nd edition. Birkhäuser Boston (2018)

[11] Toro, E.F. : Riemann Solvers and Numerical Methods for Fluid Dynamics. Springer Berlin Heidelberg (2009)

[12] Weens, W. : Toward a predictive model to monitor the balance between discretization and rounding errors in hydrodynamic simulations. SIAM Conference on Parallel Processing for Scientific Computing (2020)

Séries de Taylor revisitées

Alexis Maffart[1] and Xavier Thirioux[2]

[1]IRIT, Toulouse, France
[2]ISAE - Supaéro, Toulouse, France

June 14, 2021

Abstract

L'objectif de ce travail est de proposer une nouvelle approche pour modéliser les séries de Taylor dans le cadre d'approximations polynomiales. Nous introduisons un type coinductif qui, muni des opérations adaptées, constitue une algèbre dans laquelle nos séries de Taylor multivariées représentent des objets de premier ordre. En terme d'applications, en plus de fournir des développements classiques d'expressions algébriques ou intégro-différentielles mélangées à des fonctions élémentaires, on montre qu'il est possible de résoudre des Equations aux Dérivées Ordinaires (EDO) et Equations aux Dérivées Partielles (EDP) de manière directe, sans solveurs externes. Nous présentons également l'apport principal par rapport à [9] des erreurs certifiées dans les EDO.

1 Motivations

Nous proposons une approche où les séries de Taylor sont calculées *paresseusement*, sur demande et à un ordre arbitraire. Nous voulons également produire des erreurs certifiées (au sens de l'intégration garantie), qui regrouperaient les erreurs d'approximation et les erreurs numériques. Pour un utilisateur, un schéma classique d'utilisation serait de commencer par calculer une approximation certifiée d'une expression à un certain ordre, puis d'évaluer l'erreur maximale sur le domaine des variables et enfin éventuellement, de calculer une approximation plus fine à un ordre plus grand (sans recalculer les valeurs précédemment obtenues) si l'erreur n'était pas assez précise. On suppose que les expressions sur lesquelles on travaille sont analytiques et possèdent un développement de Taylor valide en un point donné et dans le domaine des variables. Si ce n'est pas le cas, l'erreur calculée ne diminuera pas en augmentant l'ordre et pourra même diverger.

De plus, nous visons à apporter autant de robustesse et de correction que possible, et ce grâce à une approche *correcte par construction*. Le système de type assure la correction en vérifiant à la compilation que les dimensions des différents tenseurs, fonctions, convolutions et séries entières soient conformes à leurs spécifications.

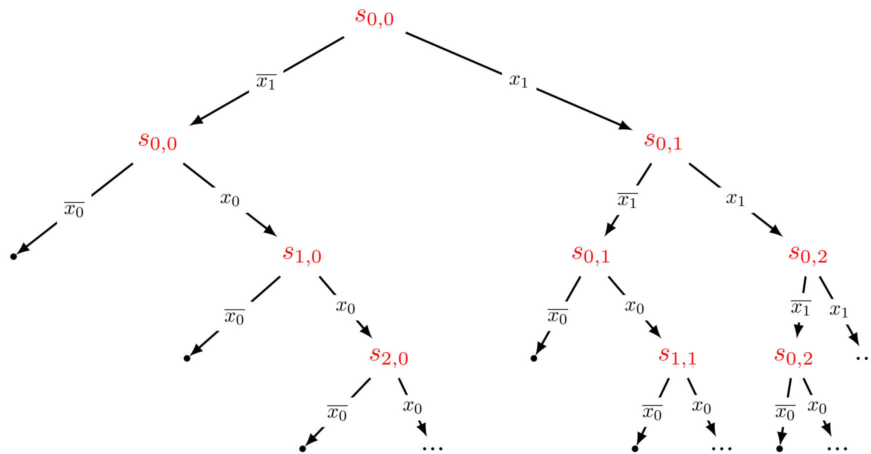
2 Formalisation

On rappelle la représentation canonique d'une série de Taylor multivariée à l'ordre R en dimension N. Cette série converge vers $f(\mathbf{x})$ quand $R \to +\infty$ pour une fonction **analytique** f seulement dans un certain voisinage autour de $\mathbf{0}$.

$$f(\mathbf{x}) = \sum_{|\boldsymbol{\alpha}|<R} \mathbf{D}_f^{\boldsymbol{\alpha}}(\mathbf{0}) \cdot \frac{\mathbf{x}^{\boldsymbol{\alpha}}}{\boldsymbol{\alpha}!} + \sum_{|\boldsymbol{\alpha}|=R} \mathbf{D}_f^{\boldsymbol{\alpha}}(\lambda * \mathbf{x}) \cdot \frac{\mathbf{x}^{\boldsymbol{\alpha}}}{\boldsymbol{\alpha}!} \tag{1}$$

Dans l'équation ci-dessus, $\mathbf{x} = (x_0, \ldots, x_{N-1}) \in \mathbb{R}^N$, $\boldsymbol{\alpha} = (\alpha_0, \ldots, \alpha_{N-1}) \in \mathbb{N}^N$ indexe l'ordre de dérivation de f dans le tenseur symétrique des dérivés partielles $\mathbf{D}_f^{\boldsymbol{\alpha}}$ et $\lambda \in [0, 1]$ est un coefficient inconnu qui caractérise le reste de Taylor exact. On doit alors calculer les dérivées au point $\mathbf{0}$ pour la partie polynomiale et au point $\lambda * \mathbf{x}$ pour l'erreur. Nous avons choisi d'utiliser une unique structure coinductive, qui est présentée ci-après, pour encoder toutes les dérivées possibles indexées par ces α. Les éléments de cette structure sont alors des paires $\langle valeur, erreur \rangle$. Notre cadre est agnostique vis-à-vis des erreurs, il est suffisamment abstrait pour être indépendant du domaine valeur/erreur. On suppose donc seulement que les éléments de notre structure forment une algèbre (addition, multiplication ainsi que des fonctions élémentaires).

Cette structure coinductive, que nous appelons "cotenseur" permet de calculer des approximations à la demande et de représenter les séries de solutions d'EDO et d'EDP quand elles sont exprimées sous forme dite "résolue". Nous avons choisi une unique structure d'arbre pour représenter un cotenseur. Des coefficients sont présents dans chaque noeud et sont notés $s_{o_0,\ldots,o_{N-1}}$ où o_i représente le nombre d'occurences de la variable x_i dans le chemin vers l'élément considéré $s_{o_0,\ldots,o_{N-1}}$.



Le principe de modélisation de la série de Taylor est le suivant : à chaque noeud qu'on parcourt en descendant dans l'arbre, on choisit, soit de garder la variable associée au noeud et la faire donc compter dans la série de Taylor finale (descente dans la branche de droite), soit de la laisser de côté et de répéter le processus sur les variables restantes de dimensions inférieures (descente à gauche). Ceci est illustré avec l'arbre de l'exemple ci-dessus où x_i est le premier cas et $\overline{x_i}$ le second. La

variable à la racine de l'arbre est $\mathbf{x}_n$ si la dimension est $n+1$. L'arbre représente donc un cotenseur symétrique s de dimension 2. L'algorithmique propre à cette structure de données est détaillée dans [9].

3 Expérimentations

On implémente directement les équations mutuellement récursives suivantes issues de l'équation d'Airy en dimension 1 : $f'' - xf = 0$

$$\begin{cases} g = g_0 + \int^x xf \\ f = f_0 + \int^x g \end{cases}$$

Le principe d'évaluation paresseuse d'OCAML permet de construire la solution selon le schéma suivant : d'après la 2ème équation, calculer le premier coefficient de f (la partie constante) revient à sommer les parties constantes de f_0 et de $\int^x g$. Or on sait que la série entière qui représente une intégrale n'a pas de partie constante, et ce, peu importe ce qu'on intègre. Le premier coefficient de g (qui est le deuxième coefficient de f) est calculé de la même façon : pas besoin de calculer l'argument de l'intégrale. Ensuite la récursion mutuelle fait effet et le 3ème coefficient de f (le 2ème de g) est le résultat de l'intégration de la partie constante de xf, calculée à la première étape.

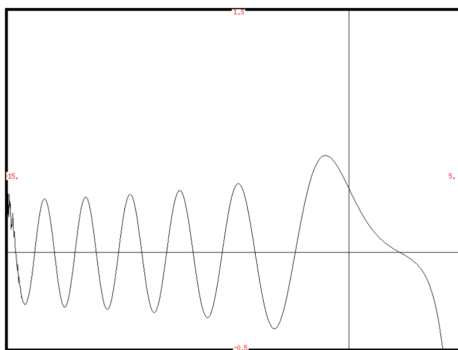


Figure 1: notre solution (ordre 150)

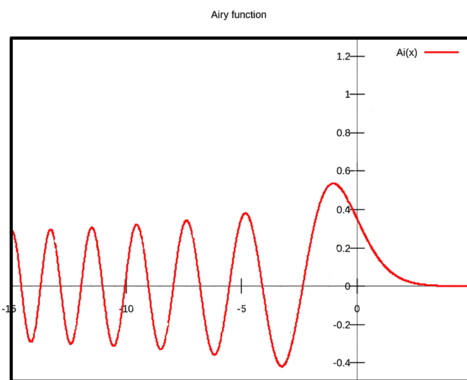
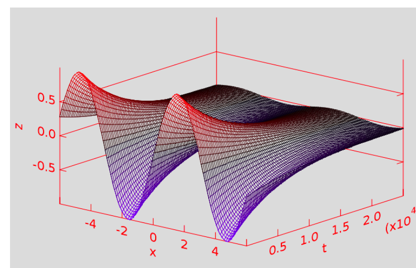
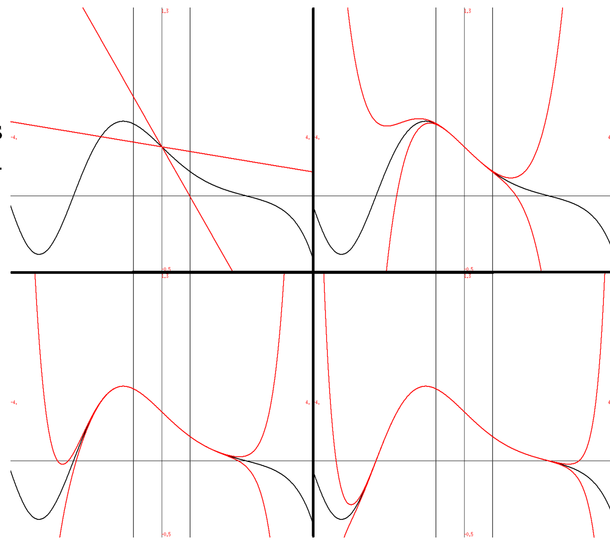


Figure 2: solution théorique

En intégrant l'équation de la chaleur : $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$ (dimension 2) par rapport au temps et en appliquant un sinus à $t = 0$ comme condition initiale, on obtient la solution ci-contre où on observe la température s'homogénéiser dans l'espace en fonction du temps. On appelle causalité la condition nécessaire pour pouvoir



résoudre une équation avec notre méth-
ode. On déterminera que cette condition pour la dimension 1 est un cas particulier de celle pour la dimension multiple et qu'il faudra la renforcer pour pouvoir calculer également les erreurs d'approximations.



Le calcul des fonctions d'erreurs fonctionne selon le même schéma que le calcul des valeurs présenté plus haut : si l'équation est causale, alors les éléments (valeurs ou erreurs) dépendent tous d'éléments d'ordre inférieur et se calculent grâce au schéma récursif. L'unique différence pour les erreurs réside dans le fait qu'un certain nombre fini de fonctions d'erreurs peuvent dépendre d'elles-mêmes. On effectue alors un point fixe sur ces fonctions et on peut ensuite calculer toutes les suivantes qui en dépendaient. La figure ci-contre présente les résultats d'erreurs certifiées (en rouge) pour l'équation d'Airy poussées à différents ordres (1, 5, 10 et 15) sur un intervalle de largeur $1,5$ (droites verticales en gris) autour de 0.

4 Conclusion

Parmi les travaux similaires, on peut citer l'outil COSY [7, 4] qui produit des intervalles certifiés en mono-dimension, [6, 5] qui présente des aspects *correct par construction*, [3] et [1] qui utilisent la paresse ou encore [2] qui porte sur l'intégration différentielle certifiée. Sans oublier le travail sur lequel ce travail se base [8, Part 2] et dont est inspirée la structure de données.
Une implémentation de séries réunissant erreurs certifiées, schéma paresseux et calcul à la demande n'avait encore jamais été réalisée, même en dimension 1. Le calcul à la demande permettant ici d'obtenir un ordre arbitraire permet également, par conséquent, d'obtenir une précision arbitraire. Ce travail se distingue également de l'existant dans le domaine par le fait qu'il permet d'obtenir des approximations sur des intervalles beaucoup plus grands (ordre de grandeur de 100) que les méthodes d'intégration classiques (type Runge-Kutta). Le calcul des erreurs dans le cas de multiples dimensions demeure néanmoins un des objectifs à la fois principaux et des plus complexes. A plus court terme, il nous faut implémenter la gestion des incertitudes sur les constantes de l'équation afin de pouvoir chaîner les approximations sur des intervalles consécutifs.

References

[1] A. Pearlmutter, B., Siskind, J.: Lazy multivariate higher-order forward-mode AD. POPL 2007 (Jan 2007)

[2] dit Sandretto, J.A., Chapoutot, A.: Validated explicit and implicit Runge–Kutta methods. Reliable Computing **22**(1), 79–103 (Jul 2016)

[3] Karczmarczuk, J.: Functional differentiation of computer programs. Higher-Order and Symbolic Computation **14**(1), 35–57 (2001). https://doi.org/10.1023/A:1011501232197

[4] Makino, K., Berz, M.: Rigorous integration of flows and ODEs using Tylor models. In: Symbolic Numeric Computation, SNC '09, Kyoto, Japan - August 03 - 05, 2009. pp. 79–84 (2009). https://doi.org/10.1145/1577190.1577206

[5] Martin-Dorel, É., Hanrot, G., Mayero, M., Théry, L.: Formally verified certificate checkers for hardest-to-round computation. J. Autom. Reasoning **54**(1), 1–29 (2015). https://doi.org/10.1007/s10817-014-9312-2

[6] Martin-Dorel, É., Rideau, L., Théry, L., Mayero, M., Pasca, I.: Certified, efficient and sharp univariate Taylor models in COQ. In: 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013. pp. 193–200 (2013). https://doi.org/10.1109/SYNASC.2013.33

[7] Revol, N., Makino, K., Berz, M.: Taylor models and floating-point arithmetic: proof that arithmetic operations are validated in COSY. J. Log. Algebr. Program. **64**(1), 135–154 (2005). https://doi.org/10.1016/j.jlap.2004.07.008

[8] Thirioux, X.: Verifying Embedded Systems. Habilitation thesis, Institut National Polytechnique de Toulouse, France (sept 2016)

[9] Thirioux, X., Maffart, A.: Taylor series revisited. In: Hierons, R.M., Mosbah, M. (eds.) Theoretical Aspects of Computing - ICTAC 2019 - 16th International Colloquium, Hammamet, Tunisia, October 31 - November 4, 2019, Proceedings. Lecture Notes in Computer Science, vol. 11884, pp. 335–352. Springer (2019). https://doi.org/10.1007/978-3-030-32505-3_19, `https://doi.org/10.1007/978-3-030-32505-3_19`