GdR Génie de la Programmation et du Logiciel, Défis 2030

Mireille Blay-Fornarino, Catherine Dubois, Pierre-Etienne Moreau

GdR Génie de la Programmation et du Logiciel
Défis 2030

Mireille Blay-Fornarino

Université Côte d'Azur, CNRS, I3S, Sophia Antipolis, France

Catherine Dubois

ENSIIE,1 square de la résistance, 91025 Evry-Courcouronnes , France

Pierre-Etienne Moreau

Université de Lorraine - LORIA, Nancy, France

5 janvier 2021

Table des matières

1 Introduction

Le manifeste [16] rédigé par une partie de la communauté du Groupement de Recherche du CNRS sur le Génie de la Programmation et du Logiciel (GdR GPL) commence par ces mots : *Le contraste est saisissant entre, d'un côté, l'omniprésence de l'informatique dans notre société et la facilité avec laquelle on peut écrire un petit programme, et d'un autre côté, la difficulté extraordinaire de garantir la correction, la fiabilité, les performances ou encore l'évolutivité d'un logiciel complexe comme on en rencontre aujourd'hui dans tous les pans de notre société : télécommunications, aérospatiale, automobile mais aussi finance, santé, administration, etc.* Dans le même temps, l'association nationale néerlandaise pour le génie logiciel publiait également un manifeste [24] qui reprenait le même argumentaire, mais le complétait à peu près en ces mots : *Bien que les logiciels aient un impact sur nous tous et partout, l'effort nécessaire pour rendre ces logiciels fiables, maintenables et utilisables sur de plus longues périodes est régulièrement sous-estimé, tant par les développeurs que par leurs responsables. En conséquence, nous voyons tous les jours des articles sur des bogues logiciels coûteux et des projets de développement de logiciels qui dépassent le budget ou qui échouent* [1].

Le Génie de la Programmation et du Logiciel est au cœur de l'activité informatique. Les concepts, méthodes et outils de conception et de validation de logiciels constituent les éléments manipulés par les informaticiens pour maîtriser et automatiser les problèmes qui leur sont soumis. Avec l'omniprésence de l'informatique dans notre vie, que ce soit en termes d'informatique embarquée, d'intelligence ambiante, d'extension du web au niveau de la planète, d'intégration dans les objets du quotidien, ou encore avec le développement de grandes infrastructures de calcul ou de traitement de grandes masses de données, de nouvelles questions de recherche sont posées. De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de test et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciels, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable, les économies d'énergie ou la maîtrise des systèmes intégrant de l'intelligence artificielle.

Au sein du GdR GPL, les équipes de recherche françaises qui travaillent sur ces thématiques ont mis en exergue, en décembre 2019, les défis auxquels elles s'intéressent. Dans cet article, nous en proposons une synthèse, non exhaustive, à destination de lecteurs non nécessairement experts du domaine. Les aspects nécessitant des connaissances expertes sont décrits dans les documents joints ci-après. En 2012 et en 2014, la communauté avait également répondu à un appel à défi ; la synthèse des réponses a été publiée dans TSI [9, 12].

Nous avons choisi d'organiser cette présentation des défis selon trois axes qui sont la fiabilité (cf. 2), la maintenance (cf. 3) et l'évaluation des logiciels (cf. 4). La fiabilité est un défi majeur pour les logiciels qui sont omniprésents dans nos vies, la maintenance pose des problèmes économiques et écologiques pour notre société, tandis que l'évaluation est au cœur même de la méthode scientifique.

1. Despite the fact that software impacts everyone everywhere, the effort that is needed to make this software reliable, maintainable and usable for longer periods is routinely underestimated, both by developers and their managers. As a result, we see news items every day about expensive software bugs and over-the-budget or failed software development projects [24].

2 Fiabilité des logiciels

La fiabilité d'un logiciel est définie par la norme ISO9126 comme la faculté du logiciel de maintenir un niveau de performance spécifique quand il est utilisé sous des conditions spécifiques. Elle se décline ainsi en différentes dimensions dont la conformité aux objectifs, la sûreté de fonctionnement, la sécurité, les performances, l'efficacité énergétique... C'est un des défis majeurs mis en exergue par nombre de chercheurs dont Xavier Leroy dans son discours inaugural à l'académie des sciences [20]. Si des avancées majeures ont été obtenues dans les secteurs de la vérification, du test, de la preuve formelle, ou encore de la surveillance, il reste de nombreux défis à relever pour rendre les systèmes logiciels plus sûrs et améliorer la confiance dans ces systèmes. Les chercheurs en GPL travaillent à une continuité essentielle entre l'expression d'un problème, la mise en œuvre et la maintenance de la solution logicielle, continuité qui doit permettre de tracer, vérifier, faire évoluer les applications avec une plus grande fiabilité (cf. 2.1). Cette continuité implique de maîtriser non seulement le lien entre la spécification et sa mise en œuvre dans différents programmes, mais également l'exécution du programme, qui dépend des compilateurs utilisés et de l'environnement dans lequel il s'exécute (cf. 2.2). L'augmentation des besoins en logiciels et leur omniprésence dans notre quotidien exigent bien évidemment que nous nous assurions de la fiabilité des logiciels produits, mais également que les éléments utilisés pour attester de cette fiabilité soient efficaces au sens où ils permettent de valider de plus en plus de systèmes malgré leur complexité et que nous puissions les justifier (cf. 2.3). Ces recherches relèvent simultanément des champs théoriques et pratiques.

2.1 De l'idée à la mise en œuvre

La construction d'un système est aujourd'hui l'affaire de nombreux acteurs, des développeurs aux utilisateurs eux-mêmes, mais comporte également des exigences multiples voire contradictoires (*e.g.*, énergie, sécurité, performance). La composition de ces différentes préoccupations est alors une tâche d'une très grande complexité [5, défi] qui impose de prendre en compte différents aspects, dont la qualification des services (*e.g.*, qualité de service, d'expérience), la sécurité [13, défi] et la mise en place de modèles de coordination [2, défi].

Depuis plusieurs années, les montées en abstraction portées par l'ingénierie des modèles et des langages ont permis de renforcer la fiabilité des logiciels "par construction" en garantissant le respect des propriétés attendues définies indépendamment des langages et environnements de programmation. Néanmoins, pour appréhender la fiabilité d'une application dans sa globalité, il est nécessaire d'avoir les moyens de décrire avec suffisamment de détails les propriétés spécifiques attendues et de raisonner en intégrant de nombreux paradigmes dont l'exécution du programme par un ou plusieurs processeurs et avec des données ou dans des conditions potentiellement imprévues et ceci dès sa conception [2, 8, défi]. C'est particulièrement le cas pour les systèmes logiciels qui dépendent d'impératifs du monde réel, tels que les systèmes cyber-physiques. On constate des besoins similaires pour les systèmes intégrant de l'apprentissage automatique qui restent difficiles à tester au sens du génie logiciel et pour lesquels les notions de qualités logicielles restent à définir [3, défi]. De manière générale, la fiabilité des applications émergentes intégrant systèmes cyber-physiques et intelligence artificielle, dont par exemple les voitures autonomes, soulèvent un grand nombre de questions de recherche tant du point de vue de l'expression des propriétés recherchées qui doivent intégrer la modélisation du comportement humain en plus de celui du système ainsi que les liens entre eux, que des outils permettant de les vérifier [10, défi].

2.2 Du programme à son exécution

Dans cet objectif de continuité de l'analyse d'un problème à sa mise en œuvre et maintenance informatique, une étape est tout particulièrement étudiée, celle qui permet de passer du programme à son exécution avec deux grandes problématiques liées : la preuve et la performance.

Les preuves peuvent s'appliquer à différents niveaux de description d'un programme, de sa spécification à son exécution. Cependant la traçabilité entre ces différents niveaux est très difficile à établir. Ceci est d'autant plus complexe que les changements de niveaux induisent à la fois des choix et la prise en compte de nouvelles préoccupations. Ainsi, même si les compilateurs ne sont jamais que des programmes permettant de transformer du code en d'autres programmes, il reste très difficile de les vérifier. En effet, un des intérêts des compilateurs est d'optimiser les exécutions, en s'adaptant aux environnements d'exécution, sans exiger du programmeur de s'intéresser à ces aspects [22, défi].

Si des travaux tels que ceux qui ont produit le compilateur C optimisant *CompCert*, permettent de garantir mathématiquement la même sémantique entre la source et le programme assembleur généré, la variabilité des environnements d'exécution introduit différents verrous.

Ainsi, la variabilité des plates-formes d'exécution intégrant des processeurs séquentiels, les systèmes distribués constitués d'ordinateurs indépendants dans le nuage ou des ordinateurs quantiques posent deux problèmes majeurs : la gestion de la cohérence de la mémoire et l'ordonnancement des différentes entités de calcul. Des modèles de programmation et des bibliothèques algorithmiques spécifiques (par exemple MapReduce) ont été définis pour s'adapter à différentes configurations spécifiques, mais ne répondent que partiellement à la problématique de la preuve. La tâche est d'une telle complexité qu'elle nécessite de faire référence à l'expertise en architectures parallèles, calcul haute performance, langages de programmation, typage et analyse statique, compilation et environnements d'exécution [8, défi].

Notons que ces problématiques incluent également la production de logiciels éco-responsables qui utilisent efficacement les ressources en fonction des contextes d'exécution [15, défi] et la protection des logiciels contre les attaques matérielles [22, défi], tandis que les choix d'architecture ont un impact non seulement sur les performances mais également sur la sécurité des applications produites [13, défi].

2.3 Des outils pour élaborer la confiance

Un des points difficiles dans l'ensemble des éléments relevés précédemment est la construction des éléments de confiance eux-mêmes. Il est indispensable de disposer d'outils qui permettent d'élaborer ces preuves complexes avec plus de facilité, nous retrouvons cette dimension dans [22, 2, défi].

Sur quels artefacts est-il raisonnable de s'appuyer pour avoir confiance dans un système ? Comment justifier qu'un système est bien construit ? Ces questions vont au-delà de la preuve ou du test d'un système. Elles nous ramènent à la problématique de l'accessibilité au raisonnement suivi pour donner confiance dans un système que ce soit ou non dans un objectif de certification [5, défi]. Elles abordent sous un nouvel angle des verrous tels que la complémentarité entre preuves, tests et simulations, les relations entre les artefacts logiciels et les exigences, entre la construction des éléments de preuve et la construction du système, etc.

3 Production, maintenance et évolution efficiente des logiciels

Les méthodes et techniques introduites par la recherche en génie logiciel modifient les approches classiques du développement du logiciel [4] qui devient préventif [18], auto-adaptable [7, 23], plus proche du monde réel avec par exemple la prise en compte de l'utilisateur dans l'écosystème de développement ou la construction de lignes de produits logicielles évolutives [21]. Nous avons donc choisi de présenter ici quelques-unes des questions de recherche posées par l'explosion en besoin de production et maintenance des logiciels.

Un grand nombre des systèmes logiciels sont aujourd'hui distribués, concurrents, de très grande taille et appelés à s'adapter automatiquement aux changements de contexte d'exécution (*e.g.*, les systèmes Cloud, Fog, Edge et cyber-physiques), ce qui a conduit à accélérer les changements de méthodes et pratiques de production et maintenance des logiciels. Au-delà d'une question d'agilité, il s'agit de se donner les moyens de produire des logiciels que nous maîtrisons et qui s'adaptent à nos besoins (cf. 3.1), sans perdre la dimension éco-responsable sans laquelle le "monde numérique" devient un danger (cf. 3.2). Pour atteindre ces objectifs, il faut disposer des abstractions adaptées (cf. 3.3) et prendre en compte l'exigence de maintenance comme une fonctionnalité essentielle des outils de développements (cf. 3.4). Enfin, l'intelligence artificielle influence aujourd'hui fortement les approches de maintenance des logiciels, nous introduisons au paragraphe 3.5 quelques-uns des défis posés.

3.1 Reconfiguration dynamique et automatique des systèmes

Comme la structure des systèmes peut être utilisée en cours d'exécution pour découvrir des services ou adapter des composants, une reconfiguration dynamique correcte devient un aspect important de l'exécution du système. La conception d'outils formels intégrés de niveau d'abstraction supérieur pour aider à la conception d'une reconfiguration sûre, tolérante aux pannes et efficace des logiciels distribués est alors essentielle [2, défi]. La communauté de recherche aborde également la question d'auto-optimiser l'environnement (objets connectés, maison/industrie du futur) pour favoriser une transition énergétique et écologique plus large [15, défi].

3.2 Logiciels durables

Les logiciels qui nous entourent pour être durables doivent répondre aux besoins spécifiques de chacun, y compris dans des contextes métiers particuliers. Le problème est bien connu et les réponses apportées varient des architectures logicielles (les micro services par exemple) aux langages dédiés, en utilisant différentes techniques dont l'ingénierie des modèles [19, défi]. De manière presque contradictoire cette capacité des systèmes à s'adapter conduit parfois à des « obésiciels », qui exigent beaucoup de ressources (mémoire, énergie ou temps) par rapport aux tâches qu'ils doivent remplir. Pour pallier ces obésités, différentes pistes sont étudiées dont l'exploitation des masses de logiciels pour prédire les consommations, l'utilisation de simulateurs ou l'analyse de dépendances présentées dans le [15, défi].

3.3 Maintenir, une exigence d'abstraction et de co-évolution

L'expression des systèmes et solutions en utilisant les abstractions adaptées aux objectifs est largement exploitée en informatique [17] pour améliorer l'efficience de la production d'applications que ce soit par exemple dans les formalismes de composition des exigences, de construction de preuves, de langages dédiés, d'administration, d'environnement de production de logiciels, etc. Nous nous intéressons ici à leur exploitation tout au long du cycle de vie du logiciel.

Ces dernières années, différents outils ont été élaborés pour permettre aux utilisateurs de définir leurs propres abstractions afin de construire des langages et environnements dédiés pour augmenter leur productivité. Ces abstractions sont exploitées par composition, génération et/ou configuration de code. La question de la maintenance est alors éminemment complexe puisqu'il s'agit de s'adapter en même temps à l'évolution des concepts, aux mécanismes qui les exploitent, aux cibles, tout en garantissant la « qualité » des applications construites. On parle alors de co-évolutions. Si le problème est connu, les avancées en Génie Logiciel et les données accumulées dans ces processus devraient permettre de proposer des solutions plus automatiques, rapides et efficaces [19, défi], et de les prendre en compte dans l'étape difficile de recherche de défauts [11, défi]. Une illustration de cette problématique est posée dans le cadre du [14, défi], qui pose la question de définir des langages de spécification et de programmation adaptés aux objets de la combinatoire, impliquant la définition de théories logiques adaptées à ces représentations.

La gestion de la co-évolution doit également permettre de minimiser les coûts de maintenance, en particulier lorsque les systèmes sont soumis à des processus de certification. Ainsi, une des pistes étudiées dans le [5, défi] porte sur l'identification, la traçabilité et la préservation des justifications encore valides après une évolution du système.

3.4 Maintenir, une exigence de première classe

Face à la complexité des logiciels, malgré les grandes évolutions méthodologiques incluant le test et la preuve, il reste difficile aujourd'hui de réparer et adapter les logiciels ; quelquefois la simple reproduction d'un comportement est difficile. La construction des logiciels et les logiciels eux-mêmes utilisent des paradigmes évolués (e.g. concurrence des tâches, présence de modèles non déterministes dans les architectures, communication par des composants tiers, masse des codes générés). La recherche des défauts ne peut plus être considérée comme une tâche secondaire, elle doit être appréhendée comme une discipline à part entière, à prendre en compte comme un élément essentiel aux développements mêmes des nouveaux supports de production d'application pour une gestion efficace des logiciels [11, défi]. Par exemple, comme nous en avons discuté précédemment, les logiciels sont souvent produits par génération de code. Face aux masses de codes ainsi produites, les outils aujourd'hui ne sont pas adaptés, qu'il s'agisse de les évaluer [19, défi], de les prendre en compte dans l'étape difficile de recherche de défauts [11, défi] ou de vérifier qu'ils restent conformes à la spécification [26, défi].

3.5 Génie Logiciel et Intelligence Artificielle

Peut-on parler aujourd'hui de logiciel sans aborder la question de l'intelligence artificielle et de l'apprentissage automatique ? Le Génie Logiciel est historiquement proche de l'intelligence artificielle (IA), en particulier de l'IA symbolique (représentation des connaissances, raisonnements, modèles d'acteurs…). Les chercheurs exploitent, depuis de nombreuses années, les travaux de ce champ d'études et y contribuent largement (*e.g.*, lignes de produits logiciels et extraction de connaissances [6], évolution des systèmes utilisant des ontologies [25]). La relation spécifique à la gestion des masses de données et aux méthodes d'apprentissage statistique a été abordée dans l'ensemble des défis, qu'il s'agisse d'utiliser ces techniques ou de servir ce champ de production de logiciels pour, par exemple, assurer la fiabilité des logiciels qui intègrent de tels composants, tester ces systèmes, expliquer… Dans [3], les auteurs mettent en exergue différents défis auxquels la communauté du génie de la programmation et du logiciel s'intéresse actuellement.

4 Évaluer pour maîtriser la production de logiciels

Nous avons choisi de mettre l'accent dans cette dernière partie sur différents défis qui visent à évaluer la production de logiciels.

Évaluer les informations issues du monde réel Aujourd'hui des approches formelles permettent de vérifier la sûreté des actions entreprises par un automatisme. Mais lorsque ces actions s'appuient sur les informations issues du monde réel, la précision de ces données et la justesse des informations transmises au système sont alors essentielles. Dans [10, défi], des techniques de supervision, elles-mêmes prouvées correctes, sont donc mises en place pour évaluer la justesse des informations conduisant aux prises de décision dans les véhicules autonomes.

Évaluer les performances des logiciels Évaluer et prédire la consommation d'énergie exige de raisonner à différents niveaux d'abstraction, en intégrant dans le raisonnement les aspects logiciels, réseaux, matériels et nos propres instruments de mesure, de mettre au point de nouvelles techniques de test et d'analyse de masses de codes, et de prendre en compte la variabilité des usages. Plus largement, à la frontière du Génie Logiciel, mesurer la distance entre les gains apportés par le logiciel et sa frugalité en ressources, pour repenser la numérisation dirigée par de nouveaux référentiels est une des pistes de recherche étudiées dans [15, défi].

Méthodes statistiques appliquées à la production de logiciels La communauté s'intéresse depuis quelques années à évaluer ses propres résultats par des méthodes empiriques [27]. Cependant dans le contexte du Génie Logiciel, les méthodes statistiques classiques ne sont pas toujours adaptées, et d'autres voies devraient être investiguées pour s'adapter au contexte particulier du Génie Logiciel. En effet, de nombreuses variables influent sur le développement logiciel, dont le domaine d'application, l'équipe, les objectifs. Il en résulte la nécessité d'un grand nombre d'expérimentations impliquant beaucoup de développeurs, ce qui, dans un grand nombre d'applications, est impossible en pratique. Il est donc temps de trouver des techniques plus adaptées, notamment en intégrant dans les expérimentations menées les principes de reproductibilité indispensables à la mesure [1, défi] et en travaillant en amont avec l'industrie du logiciel pour produire les données dont nous avons besoin.

5 Conclusion

Le génie de la programmation et du logiciel est au cœur du développement logiciel et les défis à relever sont nombreux pour aller vers une révolution numérique maîtrisée tant au sens de la fiabilité des logiciels produits, de l'efficience dans leur production et maintenance et de notre capacité à comprendre pour faire des choix sensés. Ces différents champs sont abordés par la recherche en génie de la programmation et du logiciel en France, notamment par des travaux inter-disciplinaires au sein de la science informatique mais également avec d'autres domaines dont les champs applicatifs. Ces recherches portent sur un objet en mouvement, qui connaît des "révolutions" régulières. Elles exigent une appréhension profonde de la partie théorique des domaines modélisés (*e.g.*, mathématiques, biologie, automatisme), la découverte, formalisation, mise en œuvre et exploitation de nouveaux paradigmes de programmation (*e.g.*, cloud, green computing, introduction de modèles statistiques, informatique quantique) et leur transfert dans les pratiques industrielles qu'il s'agit à la fois d'analyser et d'accompagner. Ainsi les apports théoriques et méthodologiques des recherches en génie

de la programmation et du logiciel s'appliquent aujourd'hui à de nombreuses disciplines scientifiques et techniques.

Dans ce document, nous avons mis en exergue quelques-uns des défis que des équipes du GdR GPL souhaitent relever dans les prochaines années. Ce document n'a pas l'ambition d'être exhaustif, mais il nous semble représentatif de la diversité des compétences de la communauté française en génie de la programmation et du logiciel et de l'intérêt de soutenir ses recherches.

Remerciements

Nous tenons à remercier l'ensemble des porteurs de défis pour le travail réalisé en cette période si compliquée, sans autre objectif que de partager leurs travaux. Ce travail a été largement soutenu par les membres du comité de programme qui, par leurs retours et discussions, ont œuvré à une synergie plus intense entre nos différents champs de recherche. Enfin, Alain Giorgetti et Jean-Michel Bruel nous ont plus spécifiquement aidés à affiner ce texte, en apportant leur expertise et leur vision.

Références

[1] Nicolas Anquetil, Anne Etien, Jean-Rémy Falleri, and Christelle Urtado. Vers plus de fiabilité sur les résultats de recherche en génie logiciel. Défis du GDR GPL, 2020.

[2] Simon Bliudze, Clément Quinton, Hélène Coullon, Thomas Ledoux, Ludovic Henrio, Eric Rutten, Rabea Ameur-Boulifa, Françoise Baude, Adel Noureddine, Ernesto Exposito, Philippe Aniorte, Mathieu Acher, and Tewfik Ziadi. Safe and optimal component-based dynamic reconfiguration. Défis du GDR GPL, 2020.

[3] H.-L. Bouziane, D. Delahaye, C. Dony, M. Huchard, C. Nebut, P. Reitz, A.-D. Seriai, C. Tibermacine, A. Etien, N. Anquetil, C. Urtado, S. Vauttier, J.-R. Falleri, L. du Bousquet, I. Alloui, and F. Vernier. Génie logiciel et intelligence artificielle. Défis du GDR GPL, 2020.

[4] M. Broy. Yesterday, today, and tomorrow : 50 years of software engineering. *IEEE Software*, 35(5) :38–43, 2018.

[5] Jean-Michel Bruel, Rémi Delmas, Régine Laleau, Thomas Polacsek, and Florence Sedes. Quelle argumentation pour des systèmes de confiance ? Défis du GDR GPL, 2020.

[6] Jessie Carbonnel, Karell Bertet, Marianne Huchard, and Clémentine Nebut. FCA for software product line representation : Mixing configuration and feature relationships in a unique canonical representation. *Discrete Applied Mathematics*, 273 :43–64, feb 2020.

[7] Satish Chandra. Keynote : Machine Learning for Developer Productivity at Facebook. In *ICSE2020/ MSR 2020 Online Mining Software Repositories Conference*, 2020. `https://2020.msrconf.org/track/msr-2020-Keynote?track=MSRKeynote#program`.

[8] Caroline Collange, Laure Gonnord, and Ludovic Henrio. Défi en compilation et langages : du parallélisme oui, mais du parallélisme efficace et sûr ! Défis du GDR GPL, 2020.

[9] Philippe Collet, Lydie Du Bousquet, Laurence Duchien, and Pierre-Etienne Moreau. Défis 2025. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, 34(3) :311–324, 2015. `https://hal.archives-ouvertes.fr/hal-01345654v1`.

[10] Sylvain Conchon, Aurélie Hurault, Alexandre Chapoutot, Pierre-Loïc Garoche, Akram Idani, Marc Pouzet, and Matthieu Martel. Méthodes formelles pour la conception, la programmation et la vérification de systèmes critiques émergents. Défis du GDR GPL, 2020.

[11] Steven Costiou, Thomas Dupriez, and Stéphane Ducasse. New Generation Debuggers. Défis du GDR GPL.

[12] Laurence Duchien and Yves Ledru. Défis pour le Génie de la Programmation et du Logiciel GDR CNRS GPL. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, 31(3) :397–413, March 2012. https://hal.inria.fr/hal-00712942.

[13] Equipe Archware/IRISA. La sécurité dans le développement logiciel. Défis du GDR GPL, 2020.

[14] Alain Giorgetti and Nicolas Magaud. Combinatoire certifiée. Défis du GDR GPL, 2020.

[15] Olivier Le Goaër, Adel Noureddine, Franck Barbier, Romain Rouvoy, and Florence Maraninchi. Vers des Logiciels éco-responsables, Le génie logiciel au défi de la sobriété écologique. Défis du GDR GPL, 2020.

[16] GDR GPL. Manifeste GL, 2019. https://gl.frama.io/manifeste/.

[17] Denise W. Gürer. Women in computing history. *ACM SIGCSE Bull.*, 34(2) :116–120, 2002. https://doi.org/10.1145/543812.543843.

[18] M. H. Hamilton. What the errors tell us. *IEEE Software*, 35(5) :32–37, 2018.

[19] Djamel E. Khelladi, Olivier Barais, Benoît Combemale, Mathieu Acher, Arnaud Blouin, Johann Bourcier, Noel Plouzeau, and Jean-Marc Jézéquel. Gestion de la co-évolution des logiciels partiellement générés pendant la phase d'évolution et de maintenance. Défis du GDR GPL, 2020.

[20] Xavier Leroy. *Le logiciel, entre l'esprit et la matière*, volume 284 of *Leçons inaugurales du Collège de France*. Fayard, April 2019. https://hal.inria.fr/hal-02370113.

[21] Maíra Marques, Jocelyn Simmonds, Pedro O. Rossel, and María Cecilia Bastarrica. Software product line evolution : A systematic literature review, jan 2019.

[22] David Monniaux. Chaîne de compilation formellement certifiée. Défis du GDR GPL, 2020.

[23] E. Murphy-Hill, C. Jaspan, C. Sadowski, D. Shepherd, M. Phillips, C. Winter, A. Knight, E. Smith, and M. Jorde. What predicts software developers' productivity ? *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[24] VEReniging Software Engineering Nederland. Manifesto on Software Research and Education in the Netherlands, 2019. https://versen.nl/contents/manifesto.

[25] Clément Quinton, Michael Vierhauser, Rick Rabiser, Luciano Baresi, Paul Grünbacher, and Christian Schuhmayer. Evolution in dynamic software product lines. *Journal of Software : Evolution and Process*, jun 2020. https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2293.

[26] Gabriel Scherer. Permettre la programmation sans bugs. Défis du GDR GPL, 2020.

[27] Zhongxing Yu, Chenggang Bai, Lionel Seinturier, and Martin Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. *IEEE Transactions on Software Engineering*, 2019.

Appel à Défis pour le Génie de la Programmation et du Logiciel

Vers plus de fiabilité sur les résultats de recherche en génie logiciel

N. Anquetil[1], A. Etien[1], J.-R. Falleri[2], and C. Urtado[3]

[1]Équipe RMoD, Cristal, Inria, Université de Lille, CNRS, Lille
[2]Équipe Progress, LaBRI, CNRS et ENSEIRB, Bordeaux
[3]EuroMov Digital Health in Motion, Univ. Montpellier, IMT Mines Ales, Ales

D'après l'IEEE, le génie logiciel est l'"Application of a systematic, disciplined, quantifiable approach to development, operation and maintenance of a software". Malgré tout, pour mener à bien un développement logiciel, il faut bien souvent effectuer des choix cornéliens. Par exemple quel langage de programmation faut-il choisir ? Comment gérer les branches de son dépôt ? Est-ce que l'on doit écrire le code avant ou après les tests ? De notre point de vue, la communauté de recherche en génie logiciel est aux avant-postes pour répondre à ces questions que se posent fréquemment les développeurs.

A l'origine, plusieurs axiomes utilisés dans la communauté génie logiciel reposaient sur des "mythes" comme le montre les livre de Laurent Bossavit "The Leprechauns of Software Engineering" [5] et de Robert L. Glass "Facts and Fallacies of Software Engineering"[9]. Pour prendre un exemple du livre de Laurent Bossavit, l'adage selon lequel plus un défaut est découvert tard dans le procédé de développement, plus il est cher à corriger [4], semble ne pas reposer sur des données fiables, mais a pourtant été propagé par de nombreux articles de recherche en génie logiciel. (exemple récent : [14]), alors même qu'il a été potentiellement réfuté [13].

Pour pallier ce problème, la mouvance "génie logiciel empirique" s'est donc mise en place dans la communauté de recherche en génie logiciel. L'idée derrière cette dénomination est de renforcer la façon dont sont validées les hypothèses et les résultats obtenus dans les différents domaines du génie logiciel. Ce renforcement passe par l'utilisation de méthodes empiriques qui ont fait leur preuves dans d'autres domaines, comme la médecine ou l'économie, par exemple les études de cas ou les expériences contrôlées, en les adaptant au contexte du génie logiciel, et pourquoi pas en les améliorant. Ce sujet intéresse déjà de nombreuses personnes dans le communauté française, avec le création d'un groupe de travail sur le sujet au sein du GDR-GPL en 2013, et plusieurs thèses soutenues sur le sujet chaque année.

Néanmoins l'obtention de résultats empiriques fiables en génie logiciel n'en est encore qu'à ses balbutiements, tant les facteurs qui peuvent très fortement influencer un résultat sont nombreux (par exemple l'expérience des développeurs ou le domaine applicatif du logiciel considéré). Un exemple particulièrement intéressant de controverse est très certainement la série d'articles et de rebuttal de Baishakhi et al. avec Berger et al [15, 2, 6, 3]. Le sujet initial concerne une étude qui analyse l'association entre les langages de programmation et le nombre

de bugs. La controverse qui s'en suit est très révélatrice car elle montre premièrement que l'obtention de résultats fiables est très complexe, et elle montre dans un second temps que notre communauté de recherche a énormément gagné en maturité sur cet aspect, car c'est la première fois qu'un article qui ne fait qu'essayer de reproduire des résultats passés fait couler autant d'encre.

L'objectif de ce défi est de s'inscrire dans cette mouvance et d'installer au cœur de la communauté française l'envie d'utiliser ces méthodes, tout en essayant de proposer des améliorations de fond sur les bonnes pratiques à suivre. C'est donc un défi très transverse dans son essence car il peut intéresser autant des chercheurs dans la communauté des méthodes formelles que des chercheurs dans le domaine la variabilité logicielle. Dans la suite du document, nous exposons les pistes qui nous semblent actuellement les plus importantes à développer.

Premièrement, utilisons nous les bonnes méthodologies pour évaluer nos résultats ? Dans son tutorial [16], Mary Shaw liste cinq méthodes permettant de valider des résultats de Génie Logiciel : l'analyse, l'évaluation, l'expérience, l'exemple et la persuasion. Concernant l'analyse, l'évaluation et les expériences, nous utilisons bien trop souvent des méthodes statistiques d'analyses très classiques reposant sur un calcul de p-value. Or, ces méthodes font l'objet de nombreuses controverses [10]. Il est donc grand temps de se poser la question de ce qu'on peut faire pour améliorer la confiance que l'on peut avoir en nos résultats, comme par exemple l'utilisation de l'inférence bayésienne [8], ou l'utilisation de méthodes croisées qui mettent en œuvre de l'évaluation qualitative et quantitative [7]. Pourquoi aussi ne pas aussi généraliser l'utilisation de l'enregistrement préalable (séparer la publication du protocole expérimental de l'analyse des résultats) pour éviter les phénomènes de type HARKing [12] ? Une autre question que l'on peut se poser : est-ce que l'exemple ou la persuasion peuvent aussi donner des validations probantes ?

Un des défis inhérent au génie logiciel est que de nombreuses variables confondantes existent au sein d'un projet de développement logiciel. Typiquement, le domaine applicatif, l'équipe de développement, le procédé de développement, le management, sont autant d'éléments qui peuvent influencer les résultats qu'on peut obtenir. Par exemple, la mise en œuvre d'une preuve de code peut avoir un effet énorme sur un logiciel de pilotage automatique d'aéronef, et un effet marginal sur un jeux-vidéo mobile. Cela rend les expériences contrôlées très complexes à monter dans le domaine du génie logiciel car idéalement elles nécessitent souvent de faire venir un nombre très important de développeurs et de faire des mesures sur un nombre très grand de systèmes logiciel, ce qui est impossible en pratique. Il est donc temps de trouver des techniques plus légères qui nous permettent de valider nos résultats, comme par exemple la technique en provenance de la recherche en économie de "Regression on Discontinuities", utilisé par Théo Zimmermann pour valider des résultats sur des changements d'outils et méthodes dans l'écosystème Coq [17]. Il ne faut pas aussi négliger les validations moins généralisables comme une étude de cas bien menée qui peut mettre rapidement en lumière les résultats d'une méthode ou d'un outil dans le contexte où il est censé être utilisé. Il faut aussi favoriser les tentatives de reproduction qui permettent de rendre plus solides nos connaissances [11].

A notre sens, en tant que communauté nous devons fournir un effort important pour mieux extraire les propriétés importantes d'un développement logiciel (domaine applicatif, profil des développeurs, etc.) pour faciliter la description du contexte d'application d'une méthode ou d'un outil dans une validation donnée, et de permettre au lecteur d'estimer lui même si les résultats peuvent avoir du sens dans son contexte.

Deuxièmement, une question nous parait primordiale : nos résultats sont ils assez repro-

ductible ? Dans l'essence même de la méthode empirique, la reproductibilité des résultats est primordiale. Il sera vain d'essayer de donner confiance à un résultat qu'on ne pourra pas reproduire. Il est donc important que les chercheurs de notre communauté visent à faciliter toujours plus cet aspect de leurs travaux. C'est un point qui peut-être très difficile quand on ne suit pas les bonnes pratiques (comme la publication du code ou des données sous licence libre). Il est important aussi de réfléchir à la pérennité des actions mises en œuvre (sera-t-il encore possible de lancer un programme dans 10 ans ?). Dans notre opinion, la plateforme Software Heritage [1] est typiquement une des meilleures solutions actuelle pour pérenniser le code. Il serait donc important de sensibiliser la communauté à son utilisation. Malheureusement il reste encore d'important problèmes. Par exemple que faire quand la machine pour laquelle le code a été écrit n'existe plus ? Dans certains cas il devient impossible de compiler et a fortiori faire tourner les prototypes de recherche. Que pouvons nous faire pour éviter ce genre de problèmes ?

En filigrane, un objectif très important dans ce défi est de continuer à sensibiliser la communauté française sur les méthodes empiriques pour le génie logiciel, de recenser et partager quelles sont les meilleures pratiques, de leur permettre d'identifier un panel de "spécialistes" qu'ils peuvent solliciter facilement quand leur vient le besoin de procéder à une validation empirique des résultats ou des hypothèses qui leurs sont importants.

Références

[1] Jean-François Abramatic, Roberto Di Cosmo, and Stefano Zacchiroli. Building the universal archive of source code. *Commun. ACM*, 61(10) :29–31, 2018.

[2] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. On the Impact of Programming Languages on Code Quality : A Reproduction Study. *ACM Trans. Program. Lang. Syst.*, 41(4) :21 :1–21 :24, October 2019.

[3] Emery D. Berger, Petr Maj, Olga Vitek, and Jan Vitek. FSE/CACM Rebuttal, 2019.

[4] Barry W Boehm and Kevin J Sullivan. Software economics. *this volume*, 1981.

[5] Laurent Bossavit. *The Leprechauns of Software Engineering.* Lulu. com, 2015.

[6] Prem Devanbu and Vladimir Filkov. On a Reproduction Study Chock-Full of Problems, 2019.

[7] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting Empirical Methods for Software Engineering Research. In Forrest Shull, Janice Singer, and Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer London, London, 2008.

[8] Carlo Alberto Furia, Robert Feldt, and Richard Torkar. Bayesian data analysis in empirical software engineering research. *IEEE Transactions on Software Engineering*, 2019.

[9] Robert L Glass. *Facts and Fallacies of Software Engineering.* Addison-Wesley Professional, 2002.

[10] John P. A. Ioannidis. What Have We (Not) Learnt from Millions of Scientific Papers with P Values ? *The American Statistician*, 73(sup1) :20–25, 2019.

[11] William G Kaelin Jr. Publish houses of brick, not mansions of straw. *Nature News*, 545(7655) :387, 2017.

[12] Norbert L. Kerr. HARKing : Hypothesizing After the Results are Known. *Personality and Social Psychology Review*, 2(3) :196–217, 1998.

[13] Tim Menzies, William Nichols, Forrest Shull, and Lucas Layman. Are Delayed Issues Harder to Resolve ? Revisiting Cost-to-fix of Defects Throughout the Lifecycle. *Empirical Softw. Engg.*, 22(4) :1903–1935, August 2017.

[14] F. Qin, Z. Zheng, Y. Qiao, and K. S. Trivedi. Studying Aging-Related Bug Prediction Using Cross-Project Models. *IEEE Transactions on Reliability*, 68(3) :1134–1153, September 2019.

[15] Baishakhi Ray, Daryl Posnett, Premkumar Devanbu, and Vladimir Filkov. A Large-scale Study of Programming Languages and Code Quality in GitHub. *Commun. ACM*, 60(10) :91–100, September 2017.

[16] Mary Shaw. Writing Good Software Engineering Research Papers : Minitutorial. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 726–736, USA, 2003. IEEE Computer Society. event-place : Portland, Oregon.

[17] Théo Zimmermann and Annalí Casanueva Artís. Impact of switching bug trackers : a case study on a medium-sized open source project. In *ICSME 2019 - International Conference on Software Maintenance and Evolution*, Cleveland, United States, September 2019.

Safe and optimal component-based dynamic reconfiguration

Simon Bliudze, Clément Quinton[*] Hélène Coullon, Thomas Ledoux[†]
Ludovic Henrio[‡] Eric Rutten[§] Rabéa Ameur-Boulifa[¶] Françoise Baude[‖]
Adel Noureddine, Ernesto Exposito, Philippe Aniorte[**] Mathieu Acher[††]
Tewfik Ziadi[‡‡]

1 Context

Modern software systems are inherently concurrent. They consist of components running simultaneously and sharing access to resources provided by the execution platform. These components interact through buses, shared memories and message buffers, leading to resource contention and potential deadlocks compromising mission- and safety-critical operations. Essentially, any software entity that goes beyond simply computing a certain function, necessarily has to interact and share resources with other such entities.

Although concurrency can arise in centralised systems, e.g. when several processes share a single processor, there is a broad spectrum of systems, such as Cloud, Fog, Edge, and cyber-physical systems (CPSs) that are distributed in nature. In addition to ensuring correctness of individual components, distributed systems must be correctly *configured*. Configuration consists in deploying the system, i.e. in placing each of its elements at a location, configuring the hosting machine so that the software element can run properly and establishing the connections among the system components. The placement of the entities is generally the solution of an optimization problem. The local configuration often involves installing and configuring packages and modules, configuring the operating system, and sometimes running containers or virtual machines.

Since the structure of components can be used at run time to discover services or adapt components in order to move and execute them at new locations, correct *dynamic reconfiguration* becomes an important aspect of the system execution. In particular, the local configuration of the software entities takes a special meaning as each component might need to be adapted, i.e. (re-)configured to run in a new environment.

[*]Inria Lille – Nord Europe
[†]IMT Atlantique, Inria, LS2N
[‡]Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP
[§]Univ. Grenoble Alpes, Inria, CNRS, LIG
[¶]Institut Polytechnique Paris, Télécom Paris
[‖]Université Côte d'Azur
[**]Université de Pau et des Pays de l'Adour - E2S
[††]University of Rennes, IRISA, Inria
[‡‡]Sorbonne University, Paris

Software components Software components have been designed to provide composition frameworks raising the level of abstraction compared to objects or modules. Components split the application programming into two phases: the writing of basic business code, and the composition phase, or assembly phase, consisting in plugging together instances of the basic component blocks. Component models provide a structured programming paradigm, and ensure re-usability of programs. Indeed, in component applications, dependencies are defined together with provided and required functionalities by the means of ports; this improves the program specification and thus its re-usability. Some component models and their implementations additionally keep a representation at run time of the components structure and their dependencies. Knowing how components are composed and being able to modify this composition at run time provides great adaptation capabilities as this allows the component system to be reconfigured. An application can be adapted to evolve in the execution environment by 1) adding or removing components, 2) changing some of the existing ones, 3) changing the connections among the components, or 4) reallocating components to different execution locations. System reconfiguration can involve any combination of these four kinds of adaptation.

The term *software components* is sometimes taken with different acceptations, we want here to accept a very flexible definition of components: we can call components very structured component systems (like CCM, Fractal, BIP, ...), classical module systems, but also software packages, etc. All these composition systems are of interest, but depending on their structure, more or less adaptation capabilities, and more or less guarantees can be provided.

Formal methods for safe components Distributed computer systems are by nature heterogeneous and large in scale. Their behavior depends on the interaction of multiple software components on varied hardware configurations, making them complex systems that are difficult to fully understand. The possibility to execute actions in parallel is also one of the main reasons to use a distributed system, but it further adds to their complexity. As a consequence, the process of configuring, deploying, and reconfiguring these systems is prone to errors that may result in the system entering an incorrect state, and ultimately to loss of service. Diagnosing the cause of these errors is often difficult and time-consuming, adding to their severity.

Testing is often inadequate for this type of system, as the nature of the components and their composition is not always known during development. Even when those elements are fixed, errors often depend on the timing of interactions between components, or on specific workload and network conditions, and are thus unlikely to be discovered by testing.

To ensure the correctness of component systems, a more promising approach relies on *formal methods* that offer strong generic guarantees about the systems. These methods are based on an abstraction of the system, which is checked against an agreed upon specification given in a formal language.

Obtaining a manually written formal model of a system is often not realistic; this can be due to the necessary effort, the required expertise in formal methods, or simply because the domain expert is not available. As a consequence one of the major challenge to disseminate the use of formal methods to ensure safety of the component systems is the design of automatic or partially automatic methods for model extraction.

Classification pattern and outline The ability to reconfigure components at run time raises different scientific challenges that could be included in a global Autonomic pattern [19]. A widely-used approach to capture the different phases of a reconfiguration mechanism is the MAPE-K loop [1]: *monitor*, *analyze*, *plan* and *execute* the reconfiguration of a software system, while sharing a common *knowledge*. We adopt this pattern in the rest of this document to classify the different challenges raised by the reconfiguration. We first discuss scientific challenges regarding the representation of knowledge about the software systems to reconfigure, including component-based models, formal models and ontologies with a goal to guarantee integrability and interoperability. Second, we introduce a few challenges that stand behind the M-A-P-E phases when managing a reconfiguration process, including control theory, machine learning as well as formal aspects of the management.

2 Knowledge representation: component-based models

The granularity of the abstraction may range from a very coarse model of the system—useful to describe matters of interoperability between components—to a much more detailed semantic model for a given programming language, used to characterize the execution of a specific component or application.

2.1 Formal models / formal methods for component models?

Several formal approaches exist for the verification of component models. They can be separated into different categories. First, the formal specification of component models themselves, these are very generic formalization that can be used to prove properties applicable to all component systems using a given component model. Second, formal tools allowing the specification of a particular component system, followed by the proof of correctness of its behaviour. Of course, this second category relies on a specification of the underlying component model but does not require this specification to be completely formalised: generic properties of a component model can be used as hypotheses for the proof of properties of component systems, especially when these proofs are (partially) automatised. Expanding the so-called *trust base*, by proving the properties of libraries used in component model implementations (e.g. execution engines) is not directly in the scope of this proposal but can serve as an opening for collaborations with other interest groups.

In this domain, we identify the following challenges:

- the design of modelling frameworks incorporating various paradigms but capable of sufficient detail to describe specific properties of a model or a given application and allowing one to reason on both the run-time component behaviour and the software structure;
- the design of tools based on formal methods for proving properties on such abstract models;
- (semi-)automatic generation of models from the component source or binary code;
- the design of integrated higher-abstraction level formal tools to help in the design of safe, fault tolerant and efficient reconfiguration of distributed software [9, 20, 29, 15, 5].

Furthermore, traditional solutions that are practiced today for modelling distributed systems impose static structure and partitioning [4]. To support dynamic system behaviours, we need a theoretical foundation for systems modelling and analysis to deal with the potential dynamic reconfiguration. The theory and the tools should also integrate multiple system aspects including robustness, safety, and security.

2.2 Reference component models

Among component models, we distinguish *flat* models (L^2C, Madeus, Aeolus, actor models etc.) and *hierarchical* models (CCM, Fractal, SCA), such that components may themselves comprise components. Reconfiguration in a hierarchical model is often challenging, but when this is possible the compositional approach makes it easier to manage reconfiguration of large complex systems. Another distinguishing feature of component-based models is the way they treat composition. Most models enforce some discipline on composition. A typical solution is to rely on port-interconnection sometimes constrained by a type and/or a cardinality system (like in GCM). Lastly, models can be characterized by their means of describing the life-cycle of components, in other words their management. At the most basic level, components have only two possible states: they exist or not. Many models also distinguish between *active* and *inactive* components. To describe the life-cycle (management behavior) of components more precisely, other models allow the definition of an arbitrary number of states, and attach different possible states to each component with predefined or user-defined transitions between these states. Aeolus [11], Madeus [8] and Concerto [7] are component models enabling rich life-cycle definitions for instance. Concerning distributed systems, component models are generally adapted for distribution but some of them have been designed specifically to address the challenges of distributed computing, like GCM [6]. We identify the following challenges:

- Some component models handle the functional behavior of each component and by composition the functional behavior of the entire distributed software system. Others handle the management behavior by enhancing the life-cycle modeling. However, this life-cycle modeling is correlated to the functional behavior and can be considered as an abstraction of the functional behavior. The combination and interactions of these behavioral approaches has not been tackled yet.

- While several different component models exist, the classification given above is a quite broad categorization that should be refined and could then be better adapted;

- When this classification is mature enough, it will be very useful to revisit each of the existing formal approaches and show that they are applicable not only on a given component models, but more generally on a set of models that have some well-defined characteristics. This could allow reusing existing formal approaches outside the component model they were originally designed for.

2.3 Domain-specific component models and architectures

One of the key challenges preventing large-scale adoption of component models and, particularly, the approaches based on formal methods is the necessity for the designers to learn a new model or language. Designed choices made by the scientific community are most of the time driven by the quest for conceptual minimalism in order to ensure that the resulting frameworks are easy to reason about (e.g. prove their expressive power, build broadly

applicable analysis tools). The resulting modelling frameworks powerful but abstract, making it difficult for developers to map business concepts specific to their application domain to modelling concepts of the component framework. This situation can be compared to programming in an assembly language, highlighting the necessity of designing appropriate high-level *domain-specific* modelling languages.

A major example of an application domain for software reconfiguration is that of Cyber-Physical Systems (CPSs). Designing and managing CPSs and Cyber-Physical Systems of Systems (CPSoS) requires the definition of new methodologies for Model-Based Systems Engineering (MBSE). In CPSs, the exponential growth of collected data adds an exponential challenge to analyze these data and produce new information and discover relevant knowledge [12]. Additionally, with the diversity of CPS and their coexistence in CPSoS, new reference architectures are needed (for example, new propositions are being made for Industry 4.0, such as RAMI 4.0 [18] or IIRA [22]).

Finally, CPS adds their share of complexity in managing non-functional requirements, such as performance, energy, security, reliability, privacy, quality of service, etc. Software systems of CPS need to autonomously guarantee certain constraints and properties. We identify three main challenges for non-functional requirements in CPS:

- Energy consumption: using software eco-design and autonomic management and reconfiguration, CPS energy consumption can be controlled and reduced. This can be achieved by leveraging the computing or memory of CPS to offload workloads from cloud environments into CPS. The goal is to have a holistic view of energy costs across CPS and its environment, and harness the unused resources of CPS.

- Security and reliability: we aim to improve the trust between the actors of CPS, the protection of the collected or generated data, and access control of individual actors in CPS and of CPS in CPSoS.

- Performance: the aim is to build a software architecture and data processing methodologies improving the performance of CPS and CPSoS (such as optimizing productivity in an industry 4.0 environment, which might require balancing between actors with various levels of performance). Also, the scaling of CPS adds an additional architectural challenge: with CPS, the scale is much larger with a huge number of actors, interactions, data collection and data processing, all of which need to be managed and reconfigured autonomously.

Although the above challenges are formulated for CPSs, many other domain-specific component models and reference architectures exist for a broad range of domains: automotive [13], avionics [27], space [17], cloud applications [24, 30] etc.

2.4 Separation of concerns in reconfiguration

When designing a distributed software system, both in its development and management aspects, multiple actors are involved: (1) *developers* who are responsible for designing and coding a set of modules or components, and responsible for the design of their composition; (2) *sysadmins* who are system administrators responsible for upkeeping, configuring, and testing multi-users computer systems such as servers or Clouds; and (3) *end-users* of the distributed software system or application. As a reconfiguration is a run-time adaptation of a software system, modifications are dynamically applied and may impact one or multiple steps of the

initial design. Thus, multiple actors are also involved in a reconfiguration process or in the design of an autonomous system. Enhancing the separation of concerns in the reconfiguration process is an important topic that raises, among others, the following challenges:

- the design of new reconfiguration-oriented programming paradigms such that reconfiguration interfaces and APIs are exposed to future other actors of the reconfiguration, and such that unpredictable reconfiguration could be handled later on;
- the design of reconfiguration models and languages at the DevOps level where sysadmins are faced to reconfiguration cases while not having much information on the internal behavior of each component to reconfigure nor their interactions [7, 11] (related to the GdR RSD);
- the design of coordination models to handle multiple and possibly concurrent reconfiguration aspects (e.g. energy, security, etc.).
- the design of high-abstraction level languages and tools for the expression of QoS (Quality of Service) and QoE (Quality of Experiment) from the end-user viewpoint, and the associated translation to a coordinated and safe reconfiguration process driven by the different QoS/QoE aspects.

3 Automatic reconfiguration: The MAPE approach

In this section, we divide the open issues related to the different phases M(onitoring), (A)nalysis-(P)lanning and (E)xecute.

3.1 Interfaces for monitoring and execution of reconfiguration

One the one hand, to handle autonomic reconfiguration the monitoring phase offers ways to introspect the software system to reconfigure, and the environment in which it evolves, including the infrastructure that host pieces of software. Execution of the reconfiguration, on the other hand, offers ways to act on the software system and its environment according to the decision taken by the analysis and the planning phases (see next section). Identified challenges are:

- the design of components to be observable and controllable w.r.t. adaptation policies, which can be related to some activities in the GdR RSD;
- the design of run-time monitoring techniques to detect deviations between components and their models at run-time;
- the efficiency of the reconfiguration execution such that disruption time of services and pending time while the new configuration is available are minimized [7, 8];
- formal guarantees of the attainability of a reconfiguration execution [9], its applicability to the current system, it robustness etc.

3.2 Analysis, planning and control of reconfiguration

The analysis and the planning phases of MAPE are the ones holding reconfiguration decisions. Indeed, the analysis decides among a set of possible new configuration the new one and the planning synthesizes from the current configuration a plan to reach the new configuration. Both these phases are very challenging from optimization and verification viewpoints. Challenges are, among others:

- the design of the decision and control, that can involve a variety of approaches, from rule-based programming to Machine Learning, or models and techniques from Control theory [23], which can be related to some activities in the GdR MACS;
- the design of a reliable reconfiguration control process, such that when model or architectural invariants are violated or when hardware crashes at run time, recovery and rollback mechanisms can be applied. A transactional approach has been studied in the past [21] and should be generalized for any components models in the context of large scale distributed system.

Specific challenges related to machine learning. Since it is not possible to entirely explore the whole configuration space and relate each configuration to the proper non-functional and functional requirements, the idea of applying machine learning techniques is appealing and more and more explored [3]. The basic principle is to train a learning model out of observations of several software configurations (a sample). Statistical machine learning can be used to predict failures/performance of certain configurations or anticipating a configuration change depending on a previously-encountered context [25, 14]. It is thus a way to *e.g.,* predict the properties of any configuration, identify influential options, ensure non-regression or select the "best" configuration.

However, learning a model of a configurable system may be very costly (see, *e.g.,* for the case of Linux [2]). The learning process is trained on a configuration sample whose metrics are measured (*e.g.,* a subset of the whole configuration space) and is then generalized to other configurations [14, 26, 16]. The fact is that each configuration measure may require a significant amount of time and resources while the measure itself is somehow uncertain. Overall, learning a configuration space is a trade-off between cost and accuracy. Hopefully, the investments realized to learn a software configuration space pay off (*e.g.,* at runtime) and generalize to all situations in which the software is deployed and executed.

Towards autonomous learning at runtime. In particular, the configuration space of the software can evolve at run time (*e.g.,* addition/removal of a device, evolving workload, connectivity loss, etc.) and may imply to update the learning model. But updating a model is costly, even more at run time with limited resources and time. In reconfiguration scenarios, there is a need for a self-configurable learning process that would learn from previous system evolution if and what *region* of the model has to be updated. One thus need mechanisms to decide (at the learning level) if the system evolution is relevant or significant enough to update the learning model (*e.g.,* based on historical data of past evolution) and if so, optimize this update by only changing parts or regions of the model impacted by the system's changes. This would result in a self-adaptive learning process in charge of properly configuring the learning process of the evolving system's configuration space.

3.3 Decentralization of MAPE

An additional aspect that should cover all phases of a reconfiguration process is its decentralization for scalability reasons and to avoid single points of failure in the autonomic process (related to the GdR RSD). At the monitoring level, the data management have to be handled in a decentralized way. The decision phases should also be decentralized which is particularly difficult as a reconfiguration process is a global and highly coordinated process [10, 28].

4 Conclusion

This document illustrates the importance and the return of an interest for component-based reconfiguration. This interest is justified by the growth of dynamic and geo-distributed software systems and infrastructures, related for instance to cyber-physical systems, IoT, fog and edge computing. Furthermore, this document also shows that many challenges are raised by reconfiguration, and are far from being solved by the research community. For this reason, we advocate reconfiguration to be one of the main focus of the software engineering community in the coming years.

References

[1] An Architectural Blueprint for Autonomic Computing. Tech. rep., IBM, 2005.

[2] M. Acher, et al. Learning Very Large Configuration Spaces: What Matters for Linux Kernel Sizes. Research report, Univ Rennes, Inria, CNRS, IRISA, 2019. URL `https://hal.inria.fr/hal-02314830`.

[3] J. Alves Pereira, et al. Learning Software Configuration Spaces: A Systematic Literature Review. Research Report 1-44, Univ Rennes, Inria, CNRS, IRISA, 2019, doi:10.1145/nnnnnnn.nnnnnnn. URL `https://hal.inria.fr/hal-02148791`.

[4] R. Ameur-Boulifa, et al. Behavioural semantics for asynchronous components. *Journal of Logical and Algebraic Methods in Programming*, 89:1 – 40, 2017. ISSN 2352-2208.

[5] T. Barros, et al. Model-checking Distributed Components: The Vercors Platform. *Electronic Notes in Theoretical Computer Science*, 182:3 – 16, 2007. ISSN 1571-0661. Proceedings of the Third International Workshop on Formal Aspects of Component Software (FACS 2006).

[6] F. Baude, et al. GCM: a grid extension to Fractal for autonomous distributed components. *Annales des Télécommunications*, 64(1-2), 2009.

[7] M. Chardet, H. Coullon, C. Perez. Predictable Efficiency for Reconfiguration of Service-Oriented Systems with Concerto. In *Proceedings 20Th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*. IEE, To appear.

[8] M. Chardet, et al. Madeus: A formal deployment model. In *4PAD 2018 - 5th Intl Symp. on Formal Approaches to Parallel and Distributed Systems (hosted at HPCS 2018)*, pp. 1–8. Orléans, France, 2018.

[9] H. Coullon, C. Jard, D. Lime. Integrated Model-Checking for the Design of Safe and Efficient Distributed Software Commissioning. In *Integrated Formal Methods*, pp. 120–137. Springer International Publishing, Cham, 2019. ISBN 978-3-030-34968-4.

[10] F. A. d. Oliveira, T. Ledoux, R. Sharrock. A Framework for the Coordination of Multiple Autonomic Managers in Cloud Environments. In *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 179–188. 2013.

[11] R. Di Cosmo, et al. Aeolus: a component model for the Cloud. *Information and Computation*, pp. 100–121, 2014.

[12] E. Exposito. Semantic-Driven Architecture for Autonomic Management of Cyber-Physical Systems (CPS) for Industry 4.0. In *International Conference on Model and Data Engineering*, pp. 5–17. Springer, 2019.

[13] S. Fürst, M. Bechter. AUTOSAR for Connected and Autonomous Vehicles: The AUTOSAR Adaptive Platform. In *46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 215–217. 2016.

[14] J. Guo, et al. Variability-aware performance prediction: A statistical learning approach. In *28th International Conference on Automated Software Engineering (ASE)*, pp. 301–311. 2013.

[15] L. Henrio, et al. Integrated Environment for Verifying and Running Distributed Components. In P. Stevens, A. Wasowski (eds.), *Fundamental Approaches to Software Engineering*, vol. 9633 of *FASE*. Perdita Stevens and Andrzej Wasowski, 2016.

[16] P. Jamshidi, et al. Learning to Sample: Exploiting Similarities Across Environments to Learn Performance Models for Configurable Systems. In *26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 71–82. 2018. ISBN 978-1-4503-5573-5.

[17] A. Jung, M. Panunzio, J.-L. Terraillon. On-board software reference architecture. Tech. Rep. TEC-SWE/09-289/AJ, SAVOIR Advisory Group, 2010.

[18] H. Kagermann, et al. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0: Securing the future of German manufacturing industry; final report of the Industrie 4.0 Working Group.* Forschungsunion, 2013.

[19] J. Kephart, D. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[20] C. E. Killian, et al. Mace: Language Support for Building Distributed Systems. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07. ACM, 2007.

[21] M. Léger, T. Ledoux, T. Coupaye. Reliable dynamic reconfigurations in a reflective component model. In L. Grunske, R. Reussner, F. Plasil (eds.), *13th international conference on Component-Based Software Engineering (CBSE'10)*, vol. 6092 of *LNCS*, pp. 74–92. Springer Berlin Heidelberg, 2010.

[22] S.-W. Lin, et al. The industrial internet of things volume G1: reference architecture. *Industrial Internet Consortium*, pp. 10–46, 2017.

[23] M. Litoiu, et al. What Can Control Theory Teach Us About Assurances in Self-Adaptive Software Systems? In *Software Engineering for Self-Adaptive Systems 3: Assurances*, vol. 9640 of *LNCS*. Springer, 2017.

[24] OASIS Standard. Topology and orchestration specification for cloud applications. Version 1.0., 2013. URL `http://docs.oasis-open.org/tosca/TOSCA/v1.0/TOSCA-v1.0.html`. [accessed 07/04/2020].

[25] A. Sarkar, et al. Cost-Efficient Sampling for Performance Prediction of Configurable Systems. In *30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 342–352. 2015.

[26] P. Temple, et al. Learning Contextual-Variability Models. *IEEE Software*, 34(6):64–70, 2017.

[27] J. L. Tokar. A Comparison of Avionics Open System Architectures. *Ada Letters*, 36(2):22—-26, 2017. ISSN 1094-3641, doi:10.1145/3092893.3092897.

[28] D. Weyns, et al. On Patterns for Decentralized Control in Self-Adaptive Systems. In R. de Lemos, et al. (eds.), *Software Engineering for Self-Adaptive Systems II*, vol. 7475 of *Lecture Notes in Computer Science (LNCS)*, pp. 76–107. Springer, 2013.

[29] J. R. Wilcox, et al. Verdi: A Framework for Implementing and Formally Verifying Distributed Systems. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15. ACM, 2015.

[30] F. Zalila, S. Challita, P. Merle. Model-driven cloud resource management with OCCIware. *Future Generation Computer Systems*, 99:260–277, 2019. ISSN 0167-739X, doi:10.1016/j.future.2019.04.015.

Appel à Défis pour le Génie de la Programmation et du Logiciel

Génie Logiciel et Intelligence Artificielle

H.-L. Bouziane[1], D. Delahaye[1], C. Dony[1], M. Huchard[1], C. Nebut[1], P. Reitz[1], A.-D. Seriai[1], C. Tibermacine[1], A. Etien[2], N. Anquetil[2], C. Urtado[3], S. Vauttier[3], J.-R. Falleri[4], L. du Bousquet[5], I. Alloui[6], and F. Vernier[6]

[1]Équipe MaREL, LIRMM, Université de Montpellier, CNRS, Montpellier
[2]Équipe RMoD, Cristal, Inria, Université de Lille, CNRS, Lille
[3]EuroMov Digital Health in Motion, Univ. Montpellier, IMT Mines Ales, Ales
[4]Équipe Progress, LaBRI, CNRS et ENSEIRB, Bordeaux
[5]LIG, Université Grenoble Alpes, Grenoble
[6]LISTIC, Université Savoie Mont Blanc, Annecy

1 Introduction

Les différentes branches de l'Intelligence Artificielle (IA) offrent de nombreuses opportunités de développement dans de très nombreux secteurs de l'activité humaine, dont certains ont été identifiés dans le rapport Villani [31] comme prioritaires tels que l'énergie, la santé, les transports et la sécurité. En particulier, la science du logiciel en tire de nombreux bénéfices depuis longtemps. Dans la communauté francophone plus spécifiquement, on peut se rappeler que la conférence LMO (Langages et Modèles à Objets) a regroupé, dès son origine, des recherches en représentation des connaissances et en programmation, et a été le creuset de travaux communs aux chercheurs de ces deux domaines [6]. L'IA peut plus largement apporter des solutions pour la réduction des coûts et des temps de développement et de maintenance, ainsi que pour l'amélioration de la qualité, dont la vérification.

Ces dernières années ont été marquées par un essor de l'IA, que ce soit dans les discours grand public ou dans les travaux de recherche. Au cœur de cet essor, on trouve différentes techniques dont les plus visibles actuellement sont les techniques d'apprentissage automatique (*machine learning*), avec en particulier l'apprentissage profond (*deep learning*), qui ont tendance à occulter la largeur et la richesse de ce domaine. Cette prégnance de l'IA dans toutes les activités nous oblige à nous positionner en tant que communauté du Génie Logiciel (GL) afin d'identifier les opportunités et les interactions entre GL et IA qui s'offrent à nous dans les années qui viennent. Ce positionnement sera également l'occasion de faire le point sur les branches de l'IA qui sont les plus pertinentes pour notre communauté et de rappeler brièvement les travaux qui ont pu être entrepris en GL en interaction avec le domaine de l'IA.

Les domaines du GL et de l'IA partagent plusieurs secteurs communs. Les systèmes multi-agents et les systèmes à composants partagent de nombreuses problématiques comme l'organisation en petites entités autonomes et communicantes, la résolution distribuée de problèmes, l'auto-adaptation ou encore la fiabilité et la tolérance aux fautes, avec de la fertilisation croisée entre les domaines [2]. La définition de modèles et d'architectures de logiciels compréhensibles s'appuie sur des techniques

de représentation des connaissances, telles que les graphes de connaissances ou les ontologies [7]. L'interopérabilité entre services web ou l'automatisation des transformations de modèles ou de méta-modèles s'appuient sur des techniques d'alignement de schémas ou d'ontologies [17]. La génération automatique de compositions de services utilise des techniques de planification [4]. La réparation automatique d'architectures logicielles s'appuie sur la résolution heuristique de problèmes sous contraintes [10]. Lors de processus d'évolution, la propagation automatique de changements permettant de restaurer la consistance des architectures logicielles est inférée par model-checking [18]. Les techniques de traitement de la langue naturelle sont couramment appliquées par exemple à l'ingénierie des exigences [5], à l'analyse des identificateurs [11], ou encore aux termes apparaissant dans la documentation ou les rapports de *bugs* [28]. La programmation par contraintes est utilisée dans la localisation de fautes [3] ou pour la génération de modèles conformes à un méta-modèle [12]. On peut noter aussi que de plus en plus de travaux utilisent de l'apprentissage statistique. Pour en donner quelques exemples, Tufano et al. montrent qu'il est possible, dans un contexte restreint, d'apprendre des modifications de code pour le *refactoring* ou la correction de *bugs* à partir des codes sources et des *pull requests* [29] ; Zhanh et al. [35] identifient des instructions suspectes dans du code ; Palmerino et al. [22] utilisent la régression linéaire multiple pour les systèmes auto-adaptatifs.

L'apprentissage statistique prend actuellement une grande place grâce aux nouvelles capacités des ordinateurs. Cependant, les deux courants de l'IA (symbolique et statistique) ont chacun leur rôle à jouer. Les approches symboliques et logiques sont utiles pour leur aspect déductif et leur prise en compte des aspects conceptuels des domaines étudiés (ici les données particulières du GL). Les approches statistiques sont utiles pour leur efficacité et leur aspect prédictif. Les limites des approches symboliques tiennent dans le besoin d'expliciter les connaissances et les règles, et de mener des raisonnements parfois coûteux. Les approches statistiques ont leurs limites également comme le fait de prendre peu en compte la connaissance d'un domaine et la structure des informations. On leur reproche aussi leur côté « boîte noire » et la difficulté, voire l'impossibilité, à expliquer les résultats. L'entraînement à partir d'exemples crée des biais dans l'apprentissage, qui sont potentiellement aussi graves pour les pratiques et constructions logicielles que pour les aspects sociaux [21]. Des travaux s'intéressent à certains des problèmes posés, comme le test des algorithmes à base d'apprentissage profond dans [16], les vulnérabilités [33], la prise en compte des structures comme les arbres de syntaxe abstraite et la tentative d'apporter une explication [14]. Frank van Hermelen, lors d'un exposé invité à la conférence EGC 2018 [1] développe par ailleurs une vision consistant à proposer des schémas de combinaison des deux courants (symbolique et statistique), plutôt que de les opposer. L'approche d'extraction d'architecture présentée dans [27] est un exemple d'une telle combinaison entre une approche de classification statistique (regroupement hiérarchique basé sur une métrique de similarité) et une approche symbolique (l'analyse formelle de concepts).

Du côté de la communauté IA, le développement de systèmes dans des secteurs tels que ceux de l'énergie ou la santé nécessite de plus en plus de compétences en ingénierie du logiciel. Ceci est d'autant plus vrai que les systèmes requièrent des architectures logicielles distribuées, dynamiques et évoluant dans le temps en fonction des besoins humains. D'autre part, les développeurs des systèmes ou applications IA ont besoin de méthodes, langages et outils pour le développement de ces systèmes.

Dans cet article de positionnement, nous présenterons tout d'abord ce que l'IA peut offrir au domaine du GL (section 2), puis nous verrons comment les techniques du GL peuvent servir en retour le domaine de l'IA (section 3), notamment dans le cadre de systèmes intégrant des composants produits par des techniques d'IA.

1. « Combining Learning and Reasoning : New Challenges for Knowledge Graphs », exposé disponible à l'adresse suivante : `http://www.canalc2.tv/video/15255`.

2 IA pour le GL

2.1 Sûreté de fonctionnement

L'IA symbolique a déjà beaucoup apporté au domaine de la sûreté de fonctionnement, lorsque l'on développe notamment des systèmes nécessitant une plus grande garantie de confiance, comme les systèmes critiques (systèmes dont les défaillances peuvent avoir des conséquences dramatiques en termes de pertes humaines, financières, ou encore sur l'environnement). Dans le domaine des méthodes formelles, par exemple, un certain nombre d'outils développés et utilisés reposent sur des méthodes qui visent à automatiser au maximum le processus de vérification. Parmi ces méthodes, on trouvera en particulier la déduction automatique (dans différentes logiques et différentes théories) et le *model-checking*. Ces méthodes sont souvent privilégiées par les outils utilisés en milieu industriel afin de minimiser les coûts de développement. C'est le cas, par exemple, de l'Atelier B (qui repose sur la méthode B [1]), qui intègre à la fois des techniques de preuve automatique et de *model-checking*.

Avec l'essor de l'apprentissage automatique, on peut raisonnablement se demander comment cet ensemble de nouvelles techniques devenues parfaitement effectives peuvent contribuer au domaine de la sûreté de fonctionnement en ajoutant notamment une composante prédiction à des méthodes où le calcul est prédominant. Un certain nombre de travaux ont déjà été entrepris dans ce domaine. Aujourd'hui, il semble que l'apprentissage automatique ne produit pas forcément de très bons résultats lorsqu'il est appliqué dans des contextes très formels, ce qui dès lors en fait un défi tout aussi intéressant que de taille pour l'avenir aussi bien pour la communauté GL que pour la communauté IA. Par exemple, dans le domaine de la preuve interactive (permettant de démontrer formellement la correction de programmes, par exemple), les expérimentations récentes dans des systèmes comme Isabelle [19] ou Coq [8] montrent des difficultés de l'apprentissage automatique à guider l'utilisateur dans la preuve, c'est-à-dire lorsqu'il faut lui recommander une tactique de preuve. Les difficultés sont encore plus grandes lorsque la tactique de preuve est paramétrée, car actuellement il est impossible de reconstruire ces paramètres par apprentissage automatique. On est donc loin d'une situation satisfaisante et il reste encore du chemin avant d'avoir un outil de recommandation de tactiques effectif.

De même, dans le domaine de la preuve automatique, l'apprentissage automatique ne permet pas de déduire la preuve automatiquement et il ne faut donc pas aborder le problème de front. En revanche, si on est moins ambitieux, on peut utiliser l'apprentissage automatique pour aider la recherche de preuve. Par exemple, l'apprentissage automatique peut permettre de sélectionner les axiomes d'une théorie nécessaires pour démontrer une certaine formule. Cette sélection, appelée sélection de prémisses, est importante car elle permet de réduire l'espace de recherche de preuve, et donc d'améliorer significativement la recherche de preuve dans certains cas. Les premières expérimentations remontent à il y a plus de 10 ans [30], avec la difficulté de définir des ensembles de caractéristiques pertinentes pour caractériser les énoncés mathématiques dans les théories considérées [15]. Ces caractéristiques sont principalement des symboles, mais peuvent également être des types, des sous-termes, etc. L'apprentissage profond est une piste plus récente, puisque l'une des premières expériences date d'il y a tout juste 4 ans [13], et qui permet de s'affranchir de cette difficulté car les caractéristiques seront automatiquement créées par le réseau de neurones lorsqu'il apprendra. Des expérimentations très récentes [9] montrent que cette technique donne des résultats prometteurs dans des théories réputées difficiles en preuve automatique, comme la géométrie par exemple.

Ainsi, aujourd'hui, il semble raisonnable d'utiliser l'apprentissage automatique comme aide aux méthodes formelles, mais certaines difficultés subsistent si l'on souhaite le faire apparaître comme un acteur principal du processus de développement. Ces difficultés sont liées à un certain nombre d'éléments que l'apprentissage automatique ne prend pas en compte, ou mal. Par exemple, si les

données d'entrée sont des formules de logique du premier ordre, on aimerait bien que la structure arborescente, ainsi que les liaisons des variables, soient analysées et exploitées, plutôt que de voir ces formules comme de simples chaînes de caractères ; le *GraphDL* [34] est une piste prometteuse.

2.2 Maintenance et évolution

Depuis quelques années, on voit apparaître de grandes bases de code ouvert, comme GitHub.com, GitLab.com, Maven Central ou Software Heritage. Ces bases de code ne cessent de grandir en taille. Elles fournissent de précieuses informations sur l'évolution des systèmes à code source ouvert, leurs différentes versions, voire même, dans certaines bases, les descriptions des modifications apportées à ces systèmes entre les versions (les messages des *commits*, les journaux de modifications ou *changelogs*, etc.). Toutes ces informations peuvent être utilisées dans l'apprentissage automatique pour entraîner des classificateurs permettant de résoudre des problèmes de GL comme, par exemple : (1) reproduire l'évolution de programmes utilisant une certaine bibliothèque vers une autre bibliothèque, en synthétisant des adaptateurs [20] ou en générant des opérations de *refactoring* ; (2) migrer un projet d'une version d'une bibliothèque à une autre, voire même d'un langage ou d'un paradigme à un autre, par exemple du paradigme objet vers le paradigme Workflow [26] ; (3) restructurer une grande application à objets vers un système de modules.

Un des défis à relever ici est celui de l'exploitation de ces grandes collections de données de projets à code source ouvert, qui incluent beaucoup de connaissances enfouies, pour extraire et apprendre les bonnes pratiques suivies par les développeurs afin d'assister l'activité de ré-ingénierie du code. Toute la difficulté dans ce défi réside dans la définition du modèle adéquat et de l'optimisation de ce dernier, avec les paramètres précis, pour réaliser l'apprentissage automatique mentionné ci-dessus. L'autre point difficile à traiter concerne le pré-traitement à réaliser sur ces collections de données pour, entre autres, éliminer tout le bruit qui peut exister dans ces données, comme les mauvaises pratiques de développement et d'évolution. Il est intéressant de noter que ce défi est directement connecté à d'anciens défis du GL. Par exemple, au début des années 1970, Chuck Rich, Dick Waters et Howard Shrobe, inspirés par un certain nombre d'autres personnes du laboratoire d'IA du MIT (par exemple, Terry Winograd, Carl Hewitt et Gerry Sussman), ont proposé l'idée d'un *apprenti programmeur* [23], un assistant intelligent qui aiderait un programmeur à écrire, déboguer et faire évoluer des logiciels. La présence d'un large sous-ensemble de tous les logiciels du monde disponible en ligne dans des dépôts combinée aux avancées récentes de l'apprentissage automatique, permettant de faciliter l'acquisition de connaissances par fouille de données, devrait permettre de faire un grand pas pour faire de cette vision une réalité. Un tel assistant devra également passer par le développement d'interfaces intelligentes, qui devront apprendre des habitudes de développement de l'utilisateur et non plus du code lui-même, dans la lignée de travaux comme [24], où l'on va chercher à automatiser certaines tâches répétitives.

Certaines méthodes d'IA permettent de faire de la prédiction automatique. Cependant, en évolution logicielle, si la modification à apporter est trop complexe, le développeur refusera son automatisation. En effet, certains *refactorings* ne sont pas utilisés par les développeurs par manque de confiance dans les outils. D'autre part, un certain nombre d'approches peuvent faire des recommandations et donc proposer plusieurs possibilités au développeur qui devra alors choisir. Les approches sont alors semi-automatiques ou correspondent simplement à un support outillé sans être totalement automatisé. Le défi à relever ici est donc celui de mobiliser des techniques d'IA capables de prendre en compte l'intervention humaine. Enfin, de façon plus générale, un défi consisterait à recenser les problèmes typiques de maintenance et d'évolution et à identifier, parmi les techniques d'IA, celles qui sont les plus adaptées pour leur apporter des solutions.

3 GL pour l'IA

3.1 Développement de systèmes intégrant de l'IA

Certaines approches d'apprentissage machine produisent des « composants appris » et non plus spécifiés, ce qui impose de revisiter l'ensemble des étapes du cycle de développement : récolte des exigences, architecture/conception, codage, validation, exécution et maintenance/évolution. L'intégration de composants de ce type a des impacts sur toutes les propriétés fonctionnelles et non fonctionnelles (sûreté de fonctionnement, sécurité, performance, utilisabilité, maintenabilité, etc.). D'une certaine façon, la qualité du résultat final (tous points de vue confondus) va beaucoup dépendre du travail en amont (récolte des exigences et conception), de ce que le composant « intelligent » pourra « justifier » en plus des sorties (explications, transparences, etc.), de la manière dont le système est construit, etc. La construction d'un logiciel/système qui comprend un composant appris impose un changement de point de vue. En effet, ces nouveaux composants ont des limitations :

— ils ne sont pas forcément capables de donner un résultat,
— ils ne sont pas forcément déterministes,

et peuvent évoluer et changer de comportement :

— ils peuvent continuer à apprendre,
— ils peuvent s'adapter à un contexte changeant.

De ce fait, la façon de construire le logiciel (spécification, développement, test, etc.) doit être repensée. Par exemple, on peut imaginer proposer des « patrons/styles architecturaux » ou des points de questionnement :

— Comment prendre en compte l'utilisateur dans la boucle ?
— Comment exploiter les explications des algorithmes d'IA ?
— Comment s'assurer à la volée de la pertinence des entrées du composant ?
— Comment gérer un modèle qui apprend à la volée ?

Des stratégies de *monitoring* doivent être mises en œuvre pour suivre le comportement du système logiciel pendant son exécution.

Revisiter l'ingénierie dirigée par les modèles et la production de *Domain Specific Languages* (DSLs) semblent être des pistes intéressantes pour le développement de ces systèmes. Un des défis réside en la composition et la coordination de différents DSLs. Des approches sémantiques issues de l'IA telles que les ontologies sont justement complémentaires des DSLs, puisque dans les deux cas, il s'agit d'expliciter les notions d'un domaine.

3.2 Validation et vérification

Ces dernières années, l'apprentissage automatique a concentré l'attention du monde de la recherche académique, mais aussi celle de l'industrie. Tous les secteurs sont concernés aussi bien critiques comme l'automobile, l'industrie, la finance ou la santé, que les secteurs a priori non critiques. Si l'on dispose aujourd'hui d'une large palette de méthodes et d'outils pour valider ou vérifier les programmes que l'être humain écrit en se basant notamment sur des spécifications, on manque d'approches pour établir la correction de programmes/composants appris. Un certain nombre de métriques sont utilisées afin de déterminer la qualité d'un composant appris, cependant ces dernières restent souvent d'ordre fonctionnel, comme le taux de bonnes ou mauvaises réponses sur un panel d'informations. D'autres méthodes permettent d'attester la qualité des données d'apprentissage [25], mais cela ne permet pas d'extrapoler la qualité du composant final.

La situation est particulièrement problématique dans le cas des systèmes critiques. Dans l'automobile par exemple, l'apprentissage automatique combiné à la puissance de calcul informatique qui ne cesse de croître permet d'identifier en temps réel les éventuels obstacles qui se présentent sur une route, une compétence indispensable pour développer des véhicules autonomes. Mais ce type de détection n'est pas infaillible et des accidents dramatiques sont déjà à déplorer (voir par exemple l'accident mortel en 2018 impliquant un véhicule autonome Uber et un piéton [32]).

Aujourd'hui, on ne sait pas spécifier formellement les résultats produits par les algorithmes d'apprentissage automatique, dont on ne peut pas toujours expliquer le comportement et dont les résultats ne sont pas toujours reproductibles (deux apprentissages sur le même jeu de données peuvent donner deux systèmes différents). Faute de pouvoir proposer approche certifiée, il faudra probablement opter pour une approche certifiante pour les systèmes critiques. Elle consiste à valider/vérifier le résultat au coup par coup, par exemple avec des approches de *monitoring*. D'une façon générale, du point de vue de la validation/vérification, il est indispensable que la méthode d'apprentissage automatique puisse fournir un certificat, qui permettra de vérifier le résultat et qui est nécessaire lorsqu'il est impossible de vérifier le résultat seul. Ce certificat peut être vu comme une explication et c'est justement ce que les algorithmes d'apprentissage sont incapables de bien produire aujourd'hui.

À noter que même pour des systèmes a priori non critiques, l'intégration de l'IA peut s'avérer problématique en termes de validation. Ainsi, l'utilisation de modèles d'apprentissage à la volée dans un logiciel permet par exemple d'adapter les comportements du systèmes aux comportements et attentes des utilisateurs et de son environnement. En d'autres termes, le logiciel doit être capable d'affiner ses services en fonction de besoins non exprimés (donc non spécifiés), mais déduits de son utilisation. Dans cette situation :

- Comment valider un système dont l'une des caractéristiques est d'évoluer naturellement ?
- Comment garantir, au cours de sa vie et son évolution, qu'il rende le service attendu par l'utilisateur, avec une qualité au moins identique ?
- Comment maintenir et faire évoluer un système depuis sa conception initiale ?
- Comment intégrer et garantir que ce qu'il a appris ne soit pas dégradé par une maintenance ou une évolution du système ?

4 Conclusion

Dans cette courte et partielle synthèse et selon notre point de vue, nous avons cherché à montrer comment l'IA peut soutenir nos travaux en GL dans ses efforts pour proposer des méthodes, des pratiques et des outils pour maîtriser le développement et la maintenance (voir la section 2). Nous avons également chercher à montrer que le GL pouvait en retour apporter ses méthodes à l'IA (voir la section 3), notamment sur la confiance, et que le GL devait se réinventer dans le développement de systèmes intégrant des composants produits par apprentissage automatique. Il nous a semblé aussi important de ne pas restreindre l'IA aux techniques d'apprentissage (même si ces dernières sont clairement à l'origine de l'engouement actuel pour le domaine de l'IA), et aussi de ne pas opposer IA symbolique et statistique en montrant que ces deux courants pouvaient se compléter de manière générale et dans le cadre de travaux de la communauté GL en particulier. Pour toutes ces raisons, il nous semble opportun de proposer la création d'un groupe de travail du GDR GPL, qui nous permettrait de réfléchir aux différents défis et questions de recherche s'offrant à nous à l'intersection du GL et de l'IA. Du fait de l'interaction forte avec le domaine de l'IA, il nous semble également important de favoriser les collaborations avec certains groupes de travail du GDR IA (des discussions sont déjà en cours sur le sujet et seront concrétisées ultérieurement si ce groupe de travail est créé).

Références

[1] J.-R. Abrial. *The B-Book, Assigning Programs to Meanings*. Cambridge University Press, Cambridge (UK), 1996. ISBN 0521496195.

[2] J.-P. Arcangeli, V. Noel, and F. Migeon. Software Architectures and Multi-Agent Systems. In *Software Architectures*, volume 2, chapter 5, pages 171–208. Wiley, mai 2014.

[3] N. Aribi, M. Maamar, N. Lazaar, Y. Lebbah, and S. Loudni. Multiple Fault Localization Using Constraint Programming and Pattern Mining. In *ICTAI 2017*, pages 860–867, 2017.

[4] A. Bekkouche, S. M. Benslimane, M. Huchard, C. Tibermacine, H. Fethallah, and M. Mohammed. QoS-Aware Optimal and Automated Semantic Web Service Composition With User's Constraints. *Service Oriented Computing and Applications*, 11(2) :183–201, 2017.

[5] E. E. Bella, M. Gervais, R. Bendraou, L. Wouters, and A. Koudri. Semi-Supervised Approach for Recovering Traceability Links in Complex Systems. In *ICECCS 2018,*, pages 193–196, 2018.

[6] B. Carré, R. Ducournau, J. Euzenat, A. Napoli, and F. Rechenmann. Classification et objets : programmation ou représentation ? In *5e journ. nat. PRC-GDR IA*, pages 213–237, Feb. 1995.

[7] C. Coral, R. Francisco, and P. Mario. *Ontologies for Software Engineering and Software Technology*. Springer-Verlag, Berlin, Heidelberg, 2006.

[8] D. Delahaye and A. Izard. Utilisation de techniques d'apprentissage automatique pour l'aide à la preuve dans le système Coq. Technical report, Université de Montpellier, 2019.

[9] D. Delahaye and B. Lemoine. Déduction automatique en géométrie en utilisant l'apprentissage profond. Technical report, Université de Montpellier, 2019.

[10] N. Desnos, M. Huchard, G. Tremblay, C. Urtado, and S. Vauttier. Search-Based Many-To-One Component Substitution. *Journal of Software Maintenance and Evolution : Research and Practice.*, 20(5) :321–344, September/October 2008.

[11] J. Falleri, M. Huchard, M. Lafourcade, C. Nebut, V. Prince, and M. Dao. Automatic Extraction of a WordNet-Like Identifier Network from Software. In *ICPC'10*, pages 4–13, 2010.

[12] A. Ferdjoukh, A. Baert, E. Bourreau, A. Chateau, R. Coletta, and C. Nebut. Instantiation of Meta-models Constrained with OCL - A CSP Approach. In *MODELSWARD 2015*, pages 213–222, 2015.

[13] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. DeepMath – Deep Sequence Models for Premise Selection. In *Advances in Neural Information Processing Systems (NIPS)*, volume 29, pages 2235–2243, Barcelona (Spain), Dec. 2016.

[14] L. Jiang, H. Liu, and H. Jiang. Machine Learning Based Automated Method Name Recommendation : How Far Are We. In *ASE 2019*, 2019.

[15] C. Kaliszyk, J. Urban, and J. Vyskočil. Efficient Semantic Features for Automated Reasoning over Large Theories. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 3084–3090, Buenos Aires (Argentina), July 2015. AAAI Press.

[16] J. Kim, R. Feldt, and S. Yoo. Guiding Deep Learning System Testing using Surprise Adequacy. In *ICSE 2019*, pages 1039–1049, 2019.

[17] L. Lafi, J. Feki, and S. Hammoudi. Metamodel Matching Techniques : Review, Comparison and Evaluation. *IJISMD*, 5(2) :70–94, 2014.

[18] A. Mokni, M. Huchard, C. Urtado, S. Vauttier, and H. Y. Zhang. A Formal Approach for Managing Component-Based Architecture Evolution. *Science of Computer Programming*, (127) :24–49, 2016.

[19] Y. Nagashima and Y. He. PaMpeR : Proof Method Recommendation System for Isabelle/HOL. In *ASE 2018*, pages 362–372, 2018.

[20] M. Nita and D. Notkin. Using Twinning to Adapt Programs to Alternative APIs. In *ICSE'10 - Volume 1*, ICSE '10, pages 205–214. ACM, 2010.

[21] C. O'Neil. *Algorithmes : la bombe à retardement* . Les Arènes, Paris, 2018.

[22] J. Palmerino, Q. Yu, T. Desell, and D. Krutz. Improving the Decision-Making Process of Self-Adaptive Systems by Accounting for Tactic Volatility. In *ICSE 2019*, 2019.

[23] C. Rich and R. C. Waters. The Programmer's Apprentice : A Research Overview. *Computer*, 21(11) :10–25, Nov. 1988.

[24] J. Ruvini and C. Dony. Learning Users' Habits to Automate Repetitive Tasks. In *Your Wish is My Command*, The Morgan Kaufmann series in interactive technologies, pages 271–296. Morgan Kaufmann / Elsevier, 2001.

[25] S. Sadiq, N. K. Yeganeh, and M. Indulska. 20 Years of Data Quality Research : Themes, Trends and Synergies. In *Australasian Database Conference (ADC)*, volume 115, pages 153–162, Darlinghurst, Australia, 2011. Australian Computer Society, Inc.

[26] A. Selmadji, A. Seriai, H. Bouziane, and C. Dony. From Object-Oriented to Workflow : Refactoring of OO Applications into Workflows for an Efficient Resources Management in the Cloud. In *ENASE 2018, Revised Selected Papers*, pages 186–214, 2018.

[27] A. Shatnawi, A.-D. Seriai, and H. Sahraoui. Recovering Software Product Line Architecture of a Family of Object-Oriented Product Variants. *Journal of Systems and Software*, 131 :325–346, Sept. 2017.

[28] Y. Tian and D. Lo. A Comparative Study on the Effectiveness of Part-of-Speech Tagging Techniques on Bug Reports. In *SANER 2015*, pages 570–574, 2015.

[29] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On Learning Meaningful Code Changes via Neural Machine Translation. In *ICSE 2019*, pages 25–36, 2019.

[30] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskočil. MaLARea SG1 – Machine Learner for Automated Reasoning with Semantic Guidance. In *International Joint Conference on Automated Reasoning (IJCAR)*, volume 5195 of *LNCS*, pages 441–456, Sydney (Australia), Aug. 2008. Springer.

[31] C. Villani, M. Schoenauer, Y. Bonnet, C. Berthet, A.-C. Cornut, F. Levin, and B. Rondepierre. *Donner un sens à l'intelligence artificielle : Pour une stratégie nationale et européenne*. 03 2018.

[32] D. Wakabayashi. Self-Driving Uber Car Kills Pedestrian in Arizona, Where Robots Roam. *New York Times*, 19th March 2018. https ://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html [Retrieved April, 2020].

[33] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang. Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing. In *ICSE 2019*, pages 1245–1256, 2019.

[34] Z. Zhang, P. Cui, and W. Zhu. Deep Learning on Graphs : A Survey. *CoRR*, abs/1812.04202, 2018.

[35] Z. Zhang, Y. Lei, X. Mao, and P. Li. CNN-FL : An Effective Approach for Localizing Faults using Convolutional Neural Networks. In *SANER 2019*, pages 445–455, 2019.

Quelle argumentation pour des systèmes de confiance ?

Jean-Michel Bruel, IRIT, Université de Toulouse, Toulouse
Rémi Delmas, Uber Advanced Technology Center, Paris
Régine Laleau, Université Paris-Est Créteil, LACL, Créteil
Thomas Polacsek, ONERA, Toulouse
Florence Sedes, IRIT, Université de Toulouse, Toulouse

1 Contexte

La nécessité d'avoir confiance dans le fonctionnement d'un logiciel est quelque chose de connu, étudié et pratiqué depuis de nombreuses années dans, par exemple, le monde de l'aéronautique, du ferroviaire ou du médical. Dans le cadre d'applications critiques, au sens où des vies humaines sont en jeu, il est nécessaire de s'assurer qu'un logiciel fonctionne correctement, c'est-à-dire conformément à ce que l'on attend de lui. Cette exigence de confiance n'est cependant pas circonscrite aux seuls domaines impactant des vies humaines, nous la retrouvons aussi dans des domaines sensibles, comme le domaine bancaire ou la distribution d'énergie. Nous parlerons donc ici d'informatique de confiance. Notons que la confiance n'est pas un concept absolu, on a confiance en quelque chose pour quelque chose. Dès lors, si nous considérons le besoin de confiance dans une application pour quelque chose de précis, nous pouvons étendre le domaine de l'informatique de confiance à toute application qui doit absolument garantir un certain fonctionnement. En plus des applications critiques, nous pouvons inclure les applications qui gèrent des données privées dans lesquelles nous voulons avoir confiance en ce qui concerne la protection des données ou les applications de type blockchain et contrats intelligents dans lesquelles nous voulons avoir confiance en ce qui concerne l'impossibilité de manipulations frauduleuses.

Dans une optique d'établir un haut niveau de confiance, nous avons vu, depuis de nombreuses années, le développement et l'usage des méthodes formelles pour garantir des propriétés sur des systèmes. Cependant, l'utilisation de méthodes formelles ne nous ôte pas de certains doutes. Considérons que nous disposons de la preuve mathématique de la correction d'un artefact, sommes-nous sûrs que cette preuve ne contienne pas elle-même des erreurs ? Si elle a été établie par une machine, sommes-nous sûrs que le programme utilisé est lui aussi prouvé ? Nous pouvons ainsi remettre en question tous les éléments, chercher des preuves aux preuves, sans jamais trouver de fin à nos questionnements. Nous sommes typiquement face au problème épistémologique de la régression infinie. Loin d'être un problème purement philosophique, le problème de la confiance dans les moyens utilisés pour établir la preuve de correction se pose cruellement dans le monde de l'ingénierie en général et, plus particulièrement, dans le cadre de la certification. En effet, toute la démarche visant à certifier un artefact n'a qu'un seul but : prévenir les erreurs. Il est donc crucial que les moyens utilisés ne soient pas eux-mêmes entachés d'erreurs ou, tout au moins, que nous ayons confiance en eux.

Nous pouvons simplifier la certification en considérant qu'il s'agit de « savoir si un artefact est correct ». Dans ce contexte, la preuve formelle n'est qu'un élément parmi d'autres permettant d'établir cette connaissance. Comme le souligne Tony Hoare [4], ce n'est pas grâce à l'utilisation de méthodes formelles que les logiciels sont devenus plus fiables, mais par l'usage de techniques déjà employées

dans les autres branches de l'ingénierie comme : des procédures rigoureuses de relecture des spécifications de conceptions, l'assurance qualité fondée sur de larges éventails de tests ou de l'amélioration continue. Dès lors, comment, à partir de ces éléments informels, être sûr que l'artefact final est correct ?

Le problème qui nous préoccupe ici est en fait un problème d'inférence. Nous cherchons à déterminer s'il est acceptable ou pas de passer d'un ensemble de justifications à une conclusion. Pour être plus précis, nous visons l'étude des documentations techniques qui permettent la certification d'artefacts. Nous trouvons ce type de documents, par exemple, dans le domaine de l'évaluation des risques et de la fiabilité des systèmes critiques sous le nom de safety case ou d'assurance case. Le safety case est un document structuré qui fournit une justification et des arguments valables sur le fait qu'un système satisfait des propriétés relatives à sa sécurité.

Parallèlement, hors de la sphère de la certification, depuis quelques années, nous voyons émerger le besoin de démontrer la conformité d'un système par rapport à une norme. En effet, qu'il s'agisse de sécurité ou de protection des données et de la vie privée, les systèmes doivent de plus en plus se conformer à un nombre croissant de réglementations. Ce besoin de conformité à de nouvelles règles (comme par exemple le nouveau règlement général sur la protection des données de l'Union Européenne) implique des changements techniques profonds. Ces règles doivent non seulement être prises en compte par les systèmes, mais il est également nécessaire de démontrer qu'un système s'y conforme. Outre la protection de la vie privée, nous pouvons légitimement penser que, dans le futur, les systèmes, qu'ils soient déjà existants ou à construire, devront de plus en plus démontrer leur conformité à des attentes plus sociétales telles que le développement durable, l'éco-énergétique (energy-aware programming) ou des considérations d'explicabilité ou de traitements éthiques.

Par conséquent, nous devons prendre en compte deux aspects : premièrement, veiller à ce qu'un système soit conforme à un règlement et, deuxièmement, faire en sorte que les moyens de mise en conformité soient accessibles à tous.

Si nous nous basons sur les pratiques existantes dans l'univers de la certification, une grande partie de la démonstration de conformité, ce qui permet d'établir la confiance, est basée sur une approche processus et sur les aspects organisationnels. Ainsi, une organisation doit démontrer que sa structure, ses rôles et processus sont conformes à ce qui est demandé : le fonctionnement effectif de l'organisation est conforme à la structure déclarée, les rôles sont pourvus et les différents acteurs sont conscients des missions exigées par les rôles.

Concernant les artefacts techniques, la réponse est bien évidemment moins organisationnelle, mais repose sur une approche de confiance par conception, design, au sens où la composante critique est prise en compte à l'origine. Cette approche de garantie par construction ne tient malheureusement plus une fois sorti du domaine critique. Pour des problèmes de coûts, ou tout simplement par ce que l'on cherche à montrer a posteriori, il serait illusoire de penser pouvoir appliquer une approche garantie par construction pour la conformité à des normes futures, surtout si celles-ci concernent des aspects non cruciaux pour le fonctionnement du système (comme l'éco-énergétique). De plus, cette approche extrêmement coûteuse de garantie par construction semble atteindre ses limites avec, par exemple, les problèmes soulevés par l'*embarquabilité* de l'apprentissage automatique.

2 Problématique

Ce défi se positionne à la convergence de ces deux problématiques que sont la certification et la conformité à un règlement ou des attentes sociétales. Dans les deux cas, il est nécessaire d'identifier les exigences propres à ce besoin de convaincre une autorité ou des usagers. En ce qui concerne le res-

pect d'une réglementation, pour faire valoir qu'un système s'y conforme il faut tout d'abord identifier les exigences résultant de cette réglementation. De plus, dans le contexte d'un système préexistant, il peut être utile d'effectuer une étude d'impact réglementaire sur un système. Il est à noter que cette articulation entre le texte d'un règlement (une norme, une loi, etc.), qui tend à définir des objectifs de haut niveau, et un système est commun dans le monde de la certification, qui est également basé sur des normes. Par ailleurs, il est aussi important de s'intéresser à la structuration de l'argumentation qu'elle relève de la certification ou de la conformité à un règlement. Dans les deux cas, en portant notre attention sur les aspects justification, nous opérons un glissement du raisonnement déductif, de la preuve logique, vers une forme de raisonnement plus informel. Cet aspect informel ne doit pas nous arrêter dans notre démarche. En effet, il doit être possible de dégager des structures permettant de capturer la rationalité de telles argumentations et de définir des approches pour prévenir les raisonnements fallacieux.

3 Challenges identifiés

Parmi les travaux qui cherchent à organiser une argumentation dans le but de convaincre de l'efficience d'un système, nous pouvons citer tout ce qui se rapproche de près ou de loin de la thématique des *assurance cases*. Un assurance case est "une argumentation structurée selon laquelle un système est acceptable pour l'usage auquel il est destiné en ce qui concerne des préoccupations spécifiques" [1] [10]. Si, en pratique, il n'y a pas d'exigence particulière sur le format et la structuration d'un assurance case, de nombreux travaux proposent de structurer l'ensemble des justifications sous une forme inspirée du schéma de Toulmin [12]. Parmi ces approches, nous pouvons citer par exemple *Goal Structuring Notation* (GSN) [5, 6], *Claim-Argument-Evidence* [2], une approche textuelle de John Rushby [11] *Justification Diagram* [9] ou *Structured Assurance Case Meta-model* [8]. Dans cette idée de structuration, tout reste à faire. D'ailleurs des travaux récents suggèrent qu'une approche par patrons de conception pourrait-être une solution possible [1, 3, 7, 13].

Par ailleurs, puisqu'il est question d'argumentation et de justification, des liens peuvent être créés avec des groupes de travail du GDR GPL (MFDL, IE, GLACE, MTV2, IDM, AFSEC, ...) et des GDR qui s'intéressent à la question : Sécurité Informatique, Réseaux et Systèmes Distribués (RSD), Masses de Données, Informations et Connaissances en Sciences (MaDICS), Aspects Formels et Algorithmiques de l'Intelligence Artificielle (IA), Traitement Automatique des Langues (TAL)

Comme nous le voyons, beaucoup de travaux restent à mener et bien des questions méritent d'être creusées. Notre défi s'intéressera en particulier aux challenges suivants :

- Comment exprimer les exigences relatives à une norme, un standard, une réglementation ?
- Comment relier ces exigences aux exigences fonctionnelles et non fonctionnelles (comme la sécurité et la sûreté) d'un système ?
- Comment s'assurer qu'un système est conforme à une réglementation ?
- Comment un système d'information peut aider un processus d'argumentation ?
- Comment renforcer, d'un point de vue argumentatif, la complémentarité entre preuves, tests et simulations ?
- Quels formalismes d'argumentation et de méthodes pour une approche incrémentale de la mise au point et de la certification des systèmes ?
- Modularité, réutilisation, analyses d'impact ?

1. Traduction de "*an organized argument that a system is acceptable for its intended use with respect to specified concerns*".

— La plupart des outils existants d'argumentation sont graphiques et difficilement utilisables pour des gros projets industriels, comment développer des interfaces utilisateurs pour la saisie et la navigation d'argumentations complexes ?

Références

[1] Clément Duffau, Thomas Polacsek, and Mireille Blay-Fornarino. Support of justification elicitation : Two industrial reports. In *Proceedings of International Conference Advanced Information Systems Engineering, CAiSE 2018*, 2018.

[2] Luke Emmet and George Cleland. Graphical notations, narratives and persuasion : a pliant systems approach to hypertext tool design. In *Proceedings of Hypertext and Hypermedia, HYPERTEXT 2002*, 2002.

[3] Richard Hawkins, Tim Kelly, John Knight, and Patrick Graydon. A new approach to creating clear safety arguments. In *Advances in systems safety*. Springer, 2011.

[4] C. Hoare. How did software get so reliable without proof ?, 1996. `https://www.gwern.net/docs/math/1996-hoare.pdf`. Last Accessed : 1-4-2020.

[5] Tim Kelly and Rob Weaver. The goal structuring notation - a safety argument notation. In *DNS 2004 Workshop on Assurance Cases*, 2004.

[6] John A McDermid. Support for safety cases and safety arguments using SAM. *Reliability Engineering & System Safety*, 43(2), 1994.

[7] Dominique Méry, Bernhard Schätz, and Alan Wassyng. The pacemaker challenge : Developing certifiable medical devices (Dagstuhl seminar 14062). In *Dagstuhl Reports*, volume 4 :2, 2014.

[8] OMG. Structured assurance case meta-model (SACM). Technical report, Object Management Group, 2013.

[9] Thomas Polacsek. Validation, Accreditation or Certification : a New Kind of Diagram to Provide Confidence. In *Proceedings of International Conference on Research Challenges in Information Science, RCIS*, 2016.

[10] David J Rinehart, John C Knight, and Jonathan Rowanhill. Current practices in constructing and evaluating assurance cases with applications to aviation. Technical report, NASA, 2015.

[11] John Rushby, Xidong Xu, Murali Rangarajan, and Thomas L Weaver. Understanding and evaluating assurance cases. Technical Report NASA/CR-2015-218802, NASA Langley Research Center, 2015.

[12] Stephen E. Toulmin. *The Uses of Argument*. Cambridge University Press, Cambridge, UK, 2003. Updated Edition, first edition 1958.

[13] Alan Wassyng, Paul Joannou, Mark Lawford, Maibaum Thomas, and Neeraj Kumar Singh. New standards for trustworthy cyber-physical systems. In *Trustworthy Cyber-Physical Systems Engineering*, chapter 13, pages 337–368. Addison-Wesley Longman Publishing, 2016.

Défi en compilation et langages: du parallélisme oui, mais du parallélisme efficace et sûr!

Caroline Collange [*], Laure Gonnord [‡], Ludovic Henrio [†]

Signataires Gabriel Radanne (Inria Paris), Emmanuel Chailloux (LIP6, Sorbonne Université), Julien Tesson and Mathias Bourgoin (Nomadic Labs), Christophe Alias and Matthieu Moy (Univ Lyon, ENS de Lyon, UCBL, CNRS, Inria, LIP), Sid Touati (Université Côte d'Azur), Erven Rohou (Inria Rennes), Emmanuelle Saillard (Inria Bordeaux), Kevin Martin (Université Bretagne Sud).

1 Context

The advent of parallelism in supercomputers and in more classical end-user computers increases the need for high-level code optimization, advanced programming languages, and improved compilers. New architectures such as multi-core processors, Graphics Processing Units (GPUs), many-core and FPGA accelerators, and soon, quantum computers, are introduced, resulting into complex heterogeneous platforms. In particular, FPGAs are now a credible solution for energy-efficient HPC. The multiplicity of hardwares and languages for parallelism raises constant challenges for compilers. We review below existing solutions and raise the necessity to gather the results brought by different communities that could contribute to the correct compilation and execution of parallel programs.

Parallelism and concurrency There might be several reasons to introduce parallelism when executing a program. First, the problem to solve can be by nature parallel, for example because it needs to be performed by several distinct computing units. A typical example of such a scenario is the gathering of information originating different geographical locations.

However, in many cases, and especially in high-performance computing, parallelism is not a desired feature but the only way to achieve the desired performance by spreading a computational task over multiple cores/machines/servers ...At a smaller scale, to be responsive, a program often needs to implement local parallelism, in order for example to take into account incoming information while performing a long computation.

The tasks that run in parallel might have conflicting effects, this is why some parallel languages are rather called "concurrent languages" : the concurrency between the different tasks running in parallel might have an effect or not.

Research statement We propose to study and enhance the handling of parallelism in programs, from languages to runtimes. We believe that the two following challenges are equally important : 1 - understanding the different forms of parallelism and 2 - benefiting from the different ways to execute programs while ensuring strong properties about them.

Programming without errors is a difficult task because of the complexity of the algorithms, and because of the complexity due to the interaction between program entities ; this problem is at the

[*] Inria, Univ Rennes, CNRS, IRISA
[†] Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP

center of many research directions inside the GdR GPL group. However in case of parallel programs the task is even more difficult because the interaction between two program entities can occur at any moment. The programmer has to take into account additionally the concurrency between the different tasks of the program which makes the verification of the program correctness much more difficult. *Race condition* is the notion related to such concurrency : it describes a situation where two tasks perform a conflicting action, and depending on the task that acts first the result can be different.

In particular, parallel programs can raise two categories of bugs that are difficult to find and very annoying : deadlocks (several tasks blocked because each of them needs an action of another task to progress) and data-races (two tasks trying to access the same data in a conflicting manner). Data-races are a special case of race condition where the race concerns the access and modification to some data, this is the lower level race condition and the most undesirable. The design of programming languages for parallelism should find ways to reduce the risk of having such bugs while allowing the programmer to write complex and efficient programs.

When a program is deterministic, it has no race condition at all, and it either systematically deadlocks or never deadlocks but limiting parallel programming to deterministic languages is a bit too restrictive in terms of expressiveness in general.

Finally, the latest kind of hardware accelerators, quantum computers, offers a completely different and much less mature programming model. Designing compilers for these targets demands a radical departure from the traditional ways of classical computer architecture and compilers.

2 Research agenda : parallelism everywhere

In this section we propose to make a tour of different approaches to achieve the goal of safe yet efficient parallel programs. In particular we identify challenging questions for the GDR GPL groups.

2.1 Language-based approaches for parallelism

The goal of language-based approach is two-fold : enable the use of modern software development techniques such as modular programming, and improve the safety of programs by preventing some errors by providing programming abstractions that forbid undesired behaviours. As stated above, in many cases non-determinism is an undesired behaviour, and consequently several languages provide abstractions restricting or forbidding non-deterministic behaviours. We distinguish three approaches :

— Introduce new programming constructs which, by construction, will never exhibit bad properties regardless how they are used. Such constructs must be supported by appropriate runtime ensuring that the semantics never exhibit any non-determinism, regardless of how programs are composed. This is the case, for instance, of Kahn process networks [Kah74], or of Actors [Agh86] that by nature ensure determinism and consequently the absence of any kind of race-condition. But Kahn process networks are a bit restrictive in terms of allowed parallelism ; many programming models allow more parallelism while allowing the programmer to easily control race-conditions, and often preventing data-races [BSH+17].

— Augment an existing language which rich type systems (and type check or type inference) to statically prevent bad behavior such as data races. This is the case of Rust, which does not require any runtime support and support very low level access to the system, but uses a rich type system to rule out several forms of data races. More generally linear and ownership types is one solution frequently used to prevent data-races from the type system point of view [BCC+15], while keeping a rich, efficient and expressive programming language. More

generally, it is crucial to design static analysis and type systems that complement adequately the languages designed above, guaranteeing the correct behaviour of programs that adopt the programming model specified in the previous approach.

— Design analyses on top of existing languages in order to a posteriori prove the absence of deadlock or race conditions. These analyses can be performed on any type of language, from general purpose languages with explicit threads and mutexes to languages that already enable to express coarse grain construction like premises. It is thus necessary to design scalable analysers that can analyse parallel code, in the spirit of AstreeA [Min15].

Naturally, in practice these approaches form a continuum, however, in the current structuration of the GDR they are addressed by different subgroups : Compilation, LAMHA, and LTP.

In the "language approach" we also classify all the approaches that are more attached to the study or a programming model, i.e. a way to program parallelism, without being tied to a programming language. Here are the challenges we identify in this approach :

1. Design programming models that are convenient to program, can lead to efficient execution of programs, and help the programmer to write programs without bugs, deadlocks, or data-races.
2. Design static abstractions, e.g. type systems, that increase the reliability of the programs while keeping the programming language expressive enough.
3. Design proper abstractions to get precise analysis ; and analyses that benefit from high level constructions (or guarantee by construction) of the language.

2.2 Compilation approaches

The goal of compilation-based approach is two-fold : let the user declare the *optimisation potentiality* for her program while still remaining readable and independant from the final usage (architecture, level of parallelism, ...). We can distinguish two approaches :

— Many variants of C have been proposed, the most famous is OpenMP (`https://www.openmp.org/` that enables the programmer to declare independant tasks through pragmas. It is up to the compiler to parallelise these tasks effectively (and generate communications).The expected behavior parallelisation pragmas, such as the ones of OpenMP, is generally unformally described in the norm, and some of them are only user declarations.

— Express the computation itself and let the compiler generate and optimise code. For instance in the polyhedral model framework [FL11], DSLs such as Alpha [RGK11] are used to express and then are aggressively compiled into equivalent tiled/pipelined sequential code or parallel code (with explicit communications).

These compilation approaches are currently studied in the Compilation subgroup of the GDR. Scientific research questions :

1. How to formally specify the behaviour of parallelisation pragmas ? How to ensure that the compiler does not introduce bugs ? How to ensure statically or at runtime that the asumptions made by the developer are made explicit ?
2. Polyhedral aggressive compilation and loop rescheduling often produce complex code with a non trivial structure : how to be sure that these transformations are safe ? This is a non trivial application for *certified compilation*.

2.3 Runtime approaches

As for runtime the objective is to guarantee that the execution of the parallel programs will follow the properties proven. This is generally ensured by checking that the runtime allows exactly the executions specified by the language semantics. This part can be ensured more or less formally :

— Ideally, there would be a complete formal proof of the runtime platforms according to the parallel language semantics. However the the complexity of the runtime platform and of the parallel languages make the complete formal proof of correctness often not realisable. We believe that static properties could still be verified by allowing coarse grain abstractions.

— By runtime verification [BFFR18, AHO19, EHF18] : instrumenting the generated code to at least warn during executions if one of the semantics assumptions or properties are not required or statically proven.

It is necessary that the developers of the runtime environment, the person that specifies the behaviour of the language, and the developers of the static analyses for the parallel languages agree on the semantics of the program, at least in an informal way, and make sure that all the developments respect this semantics. Scientific questions [1] :

1. How to design runtimes and prove that they execute the semantics of the code they are given ? Even in the simpliest case where runtimes make no decision choices for scheduling it is not trivial to guarantee a correct instruction scheduling at runtime (and thus propagate the properties proven on the program, like e.g. absence of race conditions).
2. Correct proofs of complex runtimes, such as for example StarPU [ATNW11], seems to be untractable for formal proofs with provers, however some of its parts such as the task sheduler is by itself interesting in order to understand relationships between operational semantics and its execution support. This emphasizes the need to collaborate with other groups that currently do not belong to the GDR GPL.

2.4 Complementary approaches

Of course, parallel programs have also other characteristics that deserve to be studied :

— These programs need to be designed in productive ecosystems : from specification to runtime, verification and experimental evaluation, all software engineering techniques in general have to be rethought, notably in terms of user-friendliness, scale, and heterogeneity of code. These activities are also part of the GDR topics (MFDL, RIMEL, GLACE).

— The intrinsic sequential part of parallel programs should also be studied. Code experts are able to invoke clever sequence of compiler optimisations, none of them being used by default. We also advocate in favor of designing expert benchmarks that demonstrate state-of-the art optimisation potential.

— Parallel programs execute themselves on physical machines, and runtimes are usually designed to fit these particular machines, that also contain increasing hardware-based solutions : branch prediction, vectorisation. A proper study of safe concurrency should also not forget to take these architecture into account, the main challenge being to properly describe their (complex) behavior. This is part of the activity of the Compilation Group in the SOC^2 GDR.

1. In the GDR GPL, some of these questions are studied by the LAMHA group

3 Compilers for quantum computing

While multi-core CPUs, GPUs and even FPGAs fall under the same category of parallel processors and may benefit from the aforementioned approaches, quantum computers are currently a category on their own that comes with its own new set of challenges.

— As the first quantum computers will remain too limited to accommodate error correction for a while, quantum program compilers will have to deal with noise. Effective compiler optimization are thus critical, not just for the execution time, but more importantly for noise sensitivity. Noise and its characteristics should be modeled and taken into account at compile-time.

— Quantum compilers need the equivalent to instruction scheduling and register allocation, that is quantum gate scheduling and qubit allocation. The quantum context gives new constraints : the no-cloning theorem forbids the simple duplication of information, and the connectivity across qubits is typically limited. The proper level(s) of abstraction for quantum gate selection and scheduling in particular is still an open research question.

— Quantum compilation currently lacks an equivalent of the successful SSA form of classical compilers. An intermediate representation that enables both powerful analyses and efficient code generation is still to be found.

— Compilers and runtimes also need to manage interactions between the quantum and classical world. Integration issues for a quantum co-processor within a classical system and interplay between the quantum and classical parts will need taking into account. Compilers will have to to support dynamic quantum-driven classical control-flow.

4 Concluding remarks

Different possible classifications

The challenges raised in this document can be additionally classified according to two dimensions : the dimension of the target running hardware and the dimension of the expressed parallelism.

On one side side platforms for running programs range from sequential processors to distributed systems made of independent computers spread over the intermediate. Between these two extremes, we find multi-cores, many-cores, GPUs, and FPGAs. Depending on the hardware, two problems arise : memory consistency and scheduling of different computing entities. Without aiming at a full review here, it is obvious that the most efficient way to coordinate two cores with cache consistency issues cannot be the same as the coordination of two computers not sharing their memory.

On the other side, the same kind of classification could be made from programming models, ranging from simple threads that can be efficient but difficult to coordinate safely, to actors considering each computing unit as independent entity communicating by message passing. Intermediate solutions include bulk-synchronous parallel model adapted to data-parallelism ; algorithmic skeleton library (like MapReduce) providing more abstraction and more automation, but less expressive ; standard lower-level parallel libraries like MPI and OpenMP ; etc.

One could think that one programming model is better adapted to one kind of hardware or running platform, but, experience has shown that there is no obvious correspondence in practice. For example the actor model enforces a strong separation of data where each memory location can be manipulated by a single thread. This seems better adapted to distributed systems and actors indeed shine naturally in such setting. However in the last 20 years huge efforts have been made to make actors efficient on single machines, perhaps because language designers were convinced that they provide a good programming abstraction. First efforts have been made to implement actor runtimes that can instantiate

thousands of logical threads on the same machine, then different approaches have been proposed to avoid systematic copy of data upon actor communication, or to make actors multi-threaded.

The different efforts to make a programming paradigm work in different settings rely on a combination of language, compilation and runtime solutions, according to the organisation proposed above.

A working group on a language and compilation approach for safe parallelism

As mentioned in the presentation of the challenge, several combination of approaches already exist and several distinct communities address the challenge of safely and efficiently compiling parallelism. Our challenging objective is to gather the ideas and results originating from these different communities into a single working group, providing coherent solutions that combine modern state of the art results in expressive programming languages, powerful analysis and compilation techniques, and efficient runtime environments. We additionally want to stress that we are aiming at an approach that is well specified and provides guarantees of correct behaviour and efficiency to the programmer.

We believe that the complexity of the task highlights the need for a research group that would gather researchers that are experts in parallel architectures, programming languages, high-performance computing, typing and static analysis, compilation, and runtimes.

This workgroup could take the following forms :

1. Slightly extend the scientific outlines of the Compilation group toward parallel languages, quantum computing and semantics, and organise regular joint meetings with other groups like LAHMA and LTP.
2. A more ambitious solution is to merge the Compilation group and the LAMHA group, including also quantum computing in the research agenda. As the group would get bigger, subgroups meetings would be organised in addition to the meetings of the research group.

Références

[Agh86] Gul Agha. *Actors : a model of concurrent computation in distributed systems*. MIT Press, 1986.

[AHO19] Wolfgang Ahrendt, Ludovic Henrio, and Wytse Oortwijn. Who is to blame ? runtime verification of distributed objects with active monitors. *Electronic Proceedings in Theoretical Computer Science*, 302 :32–46, Aug 2019. Post-proceedings of VORTEX 2018.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience*, 23(2) :187–198, 2011.

[BCC+15] Stephan Brandauer, Elias Castegren, Dave Clarke, Kiko Fernandez-Reyes, Einar Broch Johnsen, Ka I. Pun, S. Lizeth Tapia Tarifa, Tobias Wrigstad, and Albert Mingkun Yang. Parallel objects for multicores : A glimpse at the parallel language Encore. In Marco Bernardo and Einar Broch Johnsen, editors, *Formal Methods for Multicore Programming*, volume 9104 of *Lecture Notes in Computer Science*, pages 1–56. 2015.

[BFFR18] Ezio Bartocci, Yliès Falcone, Adrian Francalanza, and Giles Reger. Introduction to Runtime Verification. In *Lectures on Runtime Verification. Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*, pages 1–33. Springer, February 2018.

[BSH+17] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. A survey of active object languages. *ACM Comput. Surv.*, 50(5) :76 :1–76 :39, October 2017.

[EHF18] Antoine El-Hokayem and Yliès Falcone. Can We Monitor All Multithreaded Programs ? In *RV 2018 - 18th International Conference on Runtime Verification*, pages 1–24, Limassol, Cyprus, November 2018.

[FL11] Paul Feautrier and Christian Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. 2011.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*. North-Holland, 1974.

[Min15] Antoine Miné. AstréeA : A Static Analyzer for Large Embedded Multi-Task Software. In *16th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'15)*, volume 8931 of *Lecture Notes in Computer Science*, page 3, Mumbai, India, January 2015. Springer.

[RGK11] Sanjay Rajopadhye, Samik Gupta, and Dae-Gon Kim. Alphabets : An extended polyhedral equational language. *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*, 0 :656–664, 2011.

Méthodes formelles pour la conception, la programmation et la vérification de systèmes critiques émergents

Sylvain Conchon, Aurélie Hurault *
Alexandre Chapoutot, Pierre-Loïc Garoche*, Akram Idani*, Marc Pouzet, Matthieu Martel

Les méthodes formelles ont été élaborées depuis de nombreuses années afin d'assurer un niveau aussi élevé que possible en matière de précision et de fiabilité et ont ainsi montré que l'objectif du zéro-faute est réalisable pour des systèmes dits "fermés" (qui fonctionnent dans un environnement complètement maîtrisé). Cependant, les systèmes informatiques qui émergent aujourd'hui sont de plus en plus ouverts sur des environnements plus incertains, à temps continus ou basés sur de l'apprentissage de jeux de données difficilement prédictible. En outre, grâce aux progrès réalisés dans la miniaturisation des circuits intégrés et leur autonomie, de plus en plus de systèmes ouverts sont implémentés par des programmes complexes s'exécutant sur des circuits programmables qui s'apparentent à de véritables mini-ordinateurs (microcontrôleurs, FPGA, System on Chip). On retrouve ainsi ces programmes embarqués un peu partout dans les objets de notre quotidien, dans les *smartphones* ou l'électroménager, mais également dans les appareils médicaux, les voitures, les avions, etc.

La complexité de ces systèmes est évidente et la maîtrise des risques inhérents à leur utilisation est ainsi de plus en plus pressante. Il est clair que pour garantir leur sûreté et leur sécurité, l'adoption de moyens d'investigation indubitables et des techniques sûres et fiables reposant sur des fondements mathématiques s'impose. Aussi, les méthodes formelles ont-elles vocation à jouer un rôle d'envergure dans ce cadre avec des bénéfices indéniables. Ce faisant, pour être efficaces, les méthodes formelles devront inévitablement s'adapter au caractère ouvert, imprécis et intelligent de ces systèmes. Ce défi vise à faire progresser la science informatique, en particulier les méthodes formelles, pour la conception, la programmation et la vérification de ces systèmes critiques émergents.

1 Évolution des systèmes embarqués critiques

Les systèmes informatiques qui émergent aujourd'hui sont à la fois plus ouverts sur des environnements incertains, à temps continus ou basés sur de l'apprentissage de jeux de données, mais également conçus pour être exécutés sur des circuits électroniques programmables autonomes (microcontrôleurs, FPGA, System on Chip).

Systèmes hybrides ou cyber-physiques. Un système cyber-physique est un système dont le comportement ou la sémantique combine un aspect physique décrit typiquement par des équations différentielles (ordinaires ou algébriques) avec un calculateur contrôlé par un programme informatique. La physique Newtonienne décrit l'évolution du système physique par des équations qui dépendent du temps. Certains paramètres du modèle, comme justement le temps, évoluent dans des domaines continus ou denses. Le composant informatique, bien que réalisé par un mécanisme physique à base de transistors, a un comportement qui peut être décrit dans un monde plus discret. Le temps n'est plus

*Pour le groupe MFDL

dense mais cadencé par l'horloge de l'ordinateur. Les autres variables ou paramètres manipulés sont codés par des séquences de bits.

Systèmes avec apprentissage sur jeux de données massifs. Il est difficile aujourd'hui d'ignorer l'engouement pour l'intelligence artificielle et sa diffusion dans tous les domaines applicatifs, y compris les domaines critiques. De nombreux systèmes informatiques critiques reposent maintenant sur des réseaux de neurones entraînés à partir de grands jeux de données. Directement exécutées par des composants dédiés ou par des instructions machines directement implémentées dans les microprocesseurs, ces modules de *Machine Learning* fournissent des informations cruciales au système pour fonctionner, comme la reconnaissance de panneaux de signalisation dans une voiture, la reconnaissance faciale dans les smartphones, etc.

Systèmes électroniques programmables. Les progrès réalisés dans la miniaturisation des circuits intégrés sont incroyables. En quelques années, le nombre de transistors par puce est ainsi passé de quelques milliers à quelques dizaines de milliards. Cette révolution (qui suit la loi de Moore) a complètement bouleversé la manière de construire des systèmes électroniques. Là où on passait du temps hier à concevoir un circuit électronique complet dédié à une certaine tâche, on programme désormais un microcontrôleur, un FPGA voire un des ces "mini ordinateurs", appelés *System on Chip*, qui sont composés de CPU multi-coeurs cadencés avec des horloges de plusieurs gigahertz, disposant de capacités mémoire de plusieurs gigaoctets et accompagnés d'un ensemble de co-processeurs (graphique, etc.) et de périphériques (modems, wifi, capteurs CCD, etc.). Ces circuits électroniques programmables d'une centaine de millimètres carrés se retrouvent partout dans notre quotidien (pour preuve, il se vend près de 30 milliards de microcontrôleurs par an) et le développement des logiciels embarqués dans de tels composants représente un coût de plus en plus important dans l'industrie et leur maintenance est parfois difficile voire impossible.

2 Applications

Véhicules autonomes. La composante logicielle domine le véhicule de demain, c'est pourquoi l'usage de méthodes formelles est un enjeu majeur pour leur sûreté. Les défis relevés par ces méthodes durant les années à venir porteront sur plusieurs axes de recherche : la conception de systèmes temps réels embarqués, la correction des algorithmes décisionnels, la tolérance aux fautes dans un environnement incertain, la sécurité, etc. Dans la suite nous focalisons notre réflexion principalement sur deux verrous scientifiques majeurs.

Systèmes électroniques programmés de contrôle-commande. Aujourd'hui de nombreux systèmes électroniques sont pilotés et synchronisés par du logiciel. Un exemple d'architecture consiste à implanter des boucles de commande dans des circuits logiques programmables et des modules chargés d'assurer la coopération entre ces boucles de contrôle. La méthode de développement de ces systèmes suit habituellement une approche *model-based design* se basant sur des modèles Matlab/Simulink ou Scade dont du code C (ou C++) est extrait (automatiquement) par un compilateur. Ce code est ensuite lié à des programmes écrits à la main avant d'être traduit vers un langage de description de matériel tel que VHDL ou Verilog pour être enfin exécuté par des microcontrôleurs ou FPGA.

Il est important de développer des techniques et des outils pour gagner en confiance sur ces modules de contrôle-commande ainsi que sur les modules de coopération. Bien que les approches existantes reposent sur une bonne méthodologie basée modèles et sur de la génération de code, il nous

semble que l'écriture de ces systèmes dans des langages synchrones hybrides avec une sémantique bien définie, ainsi que des outils de méthodes formelle pour vérifier certaines propriétés fonctionnelles, permettrait de trouver des bugs en phase amont et de mieux appréhender la complexité de ces architectures.

3 Verrous scientifiques et techniques à lever

Le cadre scientifique général de notre défi est résumé dans le schéma de la figure 1. À droite du schéma se trouve l'implémentation d'un système embarqué critique. On y trouve des plateformes matérielles qui peuvent inclure un système d'exploitation temps-réel ou des circuits intégrés programmables. Les programmes considérés à ce niveau sont principalement écrits en assembleur ou dans des langages de description de matériel comme VHDL ou Verilog. Ces programmes interagissent avec leur environnement, en entrée à travers des capteurs, et en sortie à l'aide d'actionneurs sur lesquels ils envoient des ordres de contrôle. Les nouveaux systèmes embarqués peuvent également reposer sur l'utilisation de processeurs spécialisés qui exécutent par exemple des réseaux de neurones pour de la reconnaissance d'image. Le cadre de gauche contient quelques langages, modèles et outils utilisés pour concevoir ces systèmes critiques. Certaines de ces méthodes de conception sont exécutables, c'est-à-dire qu'elles permettent de produire des programmes qui peuvent être donnés en entrée à des compilateurs. Ces derniers peuvent alors produire du code plus ou moins efficace pour les plateformes matérielles, avec plus ou moins de confiance dans leur correction et efficacité. D'autres méthodes de description ne permettent pas cette génération de code mais elles peuvent être plus adaptées pour mener certaines analyses de fiabilité du système. Qu'elles soient exécutables ou non, les méthodes utilisées pour décrire le comportement d'un système critique doivent inclure une description formelle et suffisamment détaillée de son environnement, qu'il soit à temps continu ou discret, ou qu'il implique une partie basée sur de l'apprentissage à partir de jeux de données massifs. Le cadre du haut de la figure rassemble les techniques en méthode formelle couramment utilisées dans la vérification formelle de systèmes critiques. On y trouve des techniques de preuve déductive, de *model checking*, de test ou d'interprétation abstraite. Pour garantir la plus grande fiabilité des systèmes critiques, il faut que ces techniques s'appliquent partout, aux modèles formels, aux codes cibles sur les plateformes matérielles et aux compilateurs.

Ce cadre général soulève de nombreux verrous scientifiques qu'il est nécessaire d'attaquer afin d'améliorer les méthodes formelles pour la conception, la programmation et la vérification de systèmes critiques émergents. Les domaines de recherche liés à ce défi sont multiples. Une liste non exhaustive de thématiques est donnée ci-dessous.

Systèmes hybrides. Les outils mathématiques pour la modélisation et la vérification des systèmes discrets sont très différents de ceux à temps continu. Les premiers s'appuient plutôt sur l'algèbre générale et la logique, avec des raisonnements basés sur le principe d'induction ; les seconds sur la topologie et l'analyse. Même lorsque les systèmes à temps continu sont discrétisés, les modèles obtenus sont souvent trop complexes et peu intuitifs pour être traités efficacement par les méthodes des systèmes discrets. Par exemple, la représentation des réels par des flottants ou des encodages à base d'entier, rend les analyses difficiles. La complexité entraîne des problèmes de passage à l'échelle des outils de vérification automatique et le côté peu intuitif de certains modèles ne permet pas aux utilisateurs des méthodes automatiques ou semi-automatiques de construire des preuves rapidement. L'étude des systèmes hybrides désigne parfois l'étude de systèmes physiques dont le comportement est décrit par une combinaison discrète (typiquement finie) de dynamiques à temps continu. Même

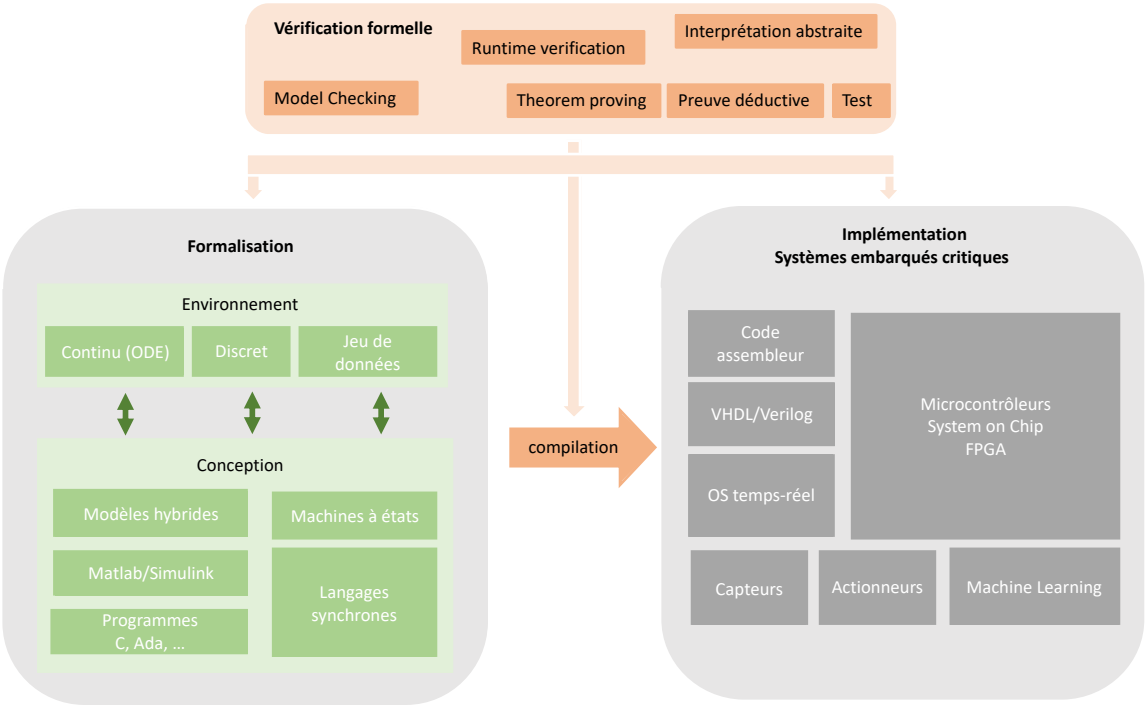


FIGURE 1 – Approche pour la formalisation et vérification de systèmes critiques embarqués vers de l'électronique programmée

si plusieurs travaux ont montré la faisabilité de développement formel de systèmes hybrides, les méthodes et outils actuellement disponibles manquent encore de maturité par rapport aux méthodes et outils dédiés aux systèmes exclusivement discrets ou continus. Ainsi il est difficile de plonger un système cyber-physique même basique, comme par exemple une boucle fermée entre un balancier inversé et un contrôleur linéaire avec saturations, dans ces formalismes.

Langages synchrones et leurs extensions aux systèmes hybrides Les langages synchrones comme Scade et leurs extensions aux systèmes hybrides comme Zélus pourraient être utilisés pour écrire des programmes qui ne sont pas facilement modélisables en Matlab/Simulink, comme des contraintes de synchronisation ou certaines machines à états. Leur expressivité permet d'exprimer des spécifications mathématiques exécutables très précises. Leur sémantique et leur compilation précisément définies assure que le code (pour l'exécution ou la simulation) reproduit fidèlement ce qui est écrit dans le modèle pour afin d'assurer que les propriétés vérifiées sur le modèles sont préservées sur l'implémentation.

Test de systèmes hybrides. Les techniques de test basées sur des propriétés (PBT), appelées également QuickCheck, semblent pertinentes dans ce contexte. Elles procèdent par génération aléatoire d'entrées filtrées par la propriété à tester. Pour les systèmes envisagés, cette génération devrait faire appel à un solveur de contraintes (par ex. SMT) qui saura traiter des équations différentielles et générer (et faire varier) des modèles. Une autre approche à développer consiste à utiliser les domaines abstraits de l'analyse statique pour construire des trajectoires abstraites. Ces trajectoires ensemblistes correspondent à des tests ou simulations à horizon borné mais capturent toutes les trajectoires faisables. Ces méthodes pourraient être

développées et appliquées dans ce contexte de systèmes hybrides complexes combinant des sémantiques à temps continu décrites par des EDO avec des programmes ; ces derniers pouvant décrire autant des composants logiciels que matériels.

Model Checking de systèmes hybrides. Un des enjeux important de ce défi est la vérification exhaustive de propriétés sur les modèles hybrides. Des approches comme dReach proposent de combiner satisfiabilité et résolution d'ODE pour raisonner sur l'atteignabilité de propriétés sur des systèmes hybrides. Pour résoudre les problèmes de précision et de passage à l'échelle, il pourrait être pertinent d'intégrer plus finement ces deux méthodes.

Analyse numérique. Les calculs décrits dans des outils comme Simulink sont exprimés dans le corps des réels, sans considérer les détails d'implémentation. Or, lors de la production du code, ces calculs doivent être implémentés en utilisant des nombres à virgule fixe ou flottante. Les difficultés concernent la précision des calculs du code objet et de son efficacité en termes d'espace utilisé sur les FPGA. Il nous semble important de travailler à des outils de synthèse de code numérique de bas niveau, en virgule fixe et flottante, qui pourront prendre en compte la précision et la contrainte d'espace en s'appuyant sur des techniques d'analyse statique et de transformation de programmes.

Systèmes avec apprentissage sur jeux de données massifs. Il est indispensable d'avoir des garanties sur les algorithmes d'intelligence artificielle (IA) pour pouvoir bénéficer des avancées de ce domaine en évitant ses dangers. Pour y arriver, plusieurs disciplines doivent cohabiter : économie, éthique, juridique, sécurité, méthodes formelles [2]. Il existe de nombreux verrous pour la vérification des algorithmes / systèmes d'IA liés aux caractéristiques de ceux-ci :

Spécification : les algorithmes / systèmes d'IA sont souvent peu ou mal spécifiés. Ils consistent à reconnaître des patterns dans les données d'entrée. L'algorithme d'IA est performant justement car on ne sait pas caractériser algorithmiquement / formellement les données d'entrée.

Explicabilité : bien souvent les spécialistes de l'IA eux-mêmes ont du mal à expliquer (en langage humain ou mathématiques) pourquoi les algorithmes donnent une réponse.

Reproductibilité : un même apprentissage répété plusieurs fois peut construire des IA ayant des comportements différents (part d'aléatoire dans l'apprentissage).

Prédictabilité et robustesse : face à une même (ou presque) configuration, il n'y a pas (toujours) de garanties d'avoir la même réponse en sortie.

La preuve de correction d'un algorithme est une preuve par rapport à une spécification. Sans spécification, pas de preuve. Il faudra donc trouver une façon / un langage / un formalisme /... pour décrire la spécification d'un algorithme d'IA. Il est déjà difficile d'assurer la correction de programmes que l'être humain écrit en se basant notamment sur des algorithmes. Comment garantir la correction de programmes qui ne sont plus écrits mais appris ? Pour contourner ce problème d'explicabilité, ainsi que ceux liés à la reproductibilité, la prédictabilité et la robustesse, une certification à postériori peut être plus adaptée : au lieu de certifier l'algorithme lui-même, chaque sortie est accompagnée d'un certificat de validité qui peut être rejoué.

Véhicules autonomes. Bien que les techniques de preuve (theorem proving et model-checking) et de vérification à l'exécution (runtime verification) - largement adressées par la communauté des méthodes formelles - soient incontournables, elles s'avèrent insuffisantes pour ces systèmes étant donné le côté imprévisible du comportement du véhicule. Ce dernier, ayant la responsabilité de reconnaître son environnement, est amené à engager en conséquence des actions qui se doivent d'être sûres. Ceci n'est pas sans failles car la reconnaissance de l'environnement se base sur des techniques qui

manquent parfois de précision tels que l'apprentissage, la prédiction ou encore l'approximation. À titre d'exemple, un rapport public dressé par le département des transports américain [1] désigne des failles de sûreté observées sur des véhicules Tesla et qui indiquent des comportements non conformes à ceux normalement attendus : "*The Automatic Emergency Braking or Autopilot systems may not function as designed, increasing the risk of a crash*". Il est important de distinguer la précision des informations issues du monde réel, et la justesse des actions engagées suite à ces informations. L'application d'une approche formelle donne aujourd'hui des solutions quant à la sûreté des actions entreprises par un automatisme ; en revanche, cela est souvent fondé sur l'hypothèse que les données en entrée sont bien précises. Ceci n'étant pas le cas du véhicule autonome, des techniques de supervision, elles-mêmes prouvées correctes, permettant d'évaluer la précision et la justesse des informations environnantes au système, sont devenues de mise.

La conduite est un processus qui se veut social car il implique des interactions parfois intenses et complexes avec des humains : autres conducteurs, cyclistes, piétons, etc. De nombreux travaux montrent qu'il est très difficile de remplacer les décisions de nature humaine par des décisions issues d'algorithmes vu que les humains s'appuient sur une intelligence généralisée et sur le bon sens. Dans un monde où le véhicule interagit très peu (voir pas du tout) avec des humains lors de la prise de décision, comme dans le cadre d'un train sans conducteur ou d'un pilote automatique d'avion, les techniques formelles ont montré leur efficacité. Néanmoins, dès lors que la prise de décision est collective et est le fruit d'interactions sociales, il devient nécessaire d'envisager une multitude de scénarios. Google par exemple, a mené des travaux pour que le véhicule puisse reconnaître un cycliste, interpréter ses signaux gestuels et prédire son intention (*e.g.* dans quel sens il envisage de tourner). Pour garantir la sûreté d'un tel système, les approches formelles se doivent de proposer des mécanismes qui couvrent la modélisation du comportement humain en plus de celui du système ainsi que les liens entre eux. Cela aura un impact direct sur les techniques de vérification actuelles (tel que le model-checking borné et/ou symbolique, la simulation, etc) qui devront être étendues non seulement pour exhiber les failles possibles, mais aussi pour élaborer des niveaux de confiance et/ou de tolérance, et identifier les responsabilités qui pourraient découler de ces failles. En effet, l'incertitude et l'hostilité de l'environnement dans lequel ces véhicules seront plongés, donnera une nouvelle vision à la cyber-sécurité et à la surveillance des usagers, et impactera de fait le cadre juridique et éthique de la société d'aujourd'hui.

4 Projet nationaux et internationaux sur le thème

Projets sur le thème des systèmes cyber-physiques

- Projet ANR DISCONT - Correct Integration of Discrete and Continuous Models
- Projet ANR EBRP PLus
- Projet ANR JCJC FEANICSES - Formal and Exhaustive Analysis of Numerical Intensive Control Software for Embedded Systems
- Projet IPL INRIA ModeliScale
- Projet ARTEMIS UnCoVerCPS - Unifying Control and Verification of Cyber-Physical Systems
- Projet ANR Cafein - Combinaison d'approches formelles pour l'étude d'invariants numériques

Projets sur le thème de l'IA et des méthodes formelles

- Au niveau national

  - Une recherche sur le site de l'ANR permet de se rendre compte de la difficulté du problème : aucun projet n'est financé sur ce sujet, probablement car le domaine n'est pas assez mûr.
  - Dans le cadre du Labex DigiCosme, une école d'été (ForMaL) a été organisée sur le thème des méthodes formelles et du machine learning pour aider à la création de synergies entre ces deux mondes : `https://formal-paris-saclay.fr/`
- Au niveau international
  - "Summit on Machine Learning Meets Formal Methods", 13 Juillet 2018, `http://www.floc2018.org/summit-on-machine-learning/`, en marge de la conférence FLoC.
  - En 2017, l'école d'été de Dagstuhl était sur le thème des méthodes formelles et du machine learning : `https://www.dagstuhl.de/en/program/calendar/semhp/?semnr=17351`

Projets sur le thème des véhicule autonome Il existe une foultitude de projets sur le thème du véhicule autonome avec un intérêt manifeste de la part des secteurs publics et privés. Nous donnons quelques pointeurs vers des projets qui font usage de méthodes formelles dans le développement de ces systèmes :

- TrustMeIA (Toulouse) `https://www.laas.fr/projects/trustmeia/`
- Programmes "Train Autonome" de l'IRT Railenium et Tech4Rail de la SNCF
- SynC Contest : Automatic Driving Challenge using Model-based Design and Synchronous Programming
- TraCE-IT : Train Control Enhancement via Information Technology (`http://traceit.isti.cnr.it`)

Références

[1] National Highway Traffic Safety Administration. Automatic vehicle control systems - ODI Resume. Technical report, U.S. Department of Transportation, 2017. `https://static.nhtsa.gov/odi/inv/2016/INCLA-PE16007-7876.PDF`.

[2] Stuart J. Russell, Daniel Dewey, and Max Tegmark. Research priorities for robust and beneficial artificial intelligence. *AI Magazine*, 36(4), 2015.

New Generation Debuggers

Steven Costiou, Thomas Dupriez, Stéphane Ducasse
RMoD, Inria Lille - Nord Europe

Résumé

Debugging is a painful and costly practice, due to the nature of bugs, of the debugged programs, or to tools limitations. We describe several difficulties of debugging that present scientific challenges (*i.e.*, we don't know how to do it) or technological challenges (*i.e.*, we can't do it). We believe that addressing these challenges will lead to new generation debuggers that will significantly ease and lower the cost of debugging.

1 Bugs are ineluctable

Software systems are getting more and more complex. Changes happen more and more dynamically. Many domains are complex by essence : *e.g.*, a Siemens robot machine to sculpt metal has 30 degrees of freedom and the program to drive it has to handle such intrinsic complexity. Today, we see new programs based on AI : in such systems the behavior may change based on provided sample and learning database. Bugs will always appear because building reliable software systems is an extremely complex task. While formal proofs of programs help to catch bugs ahead of time, the cost of writing specifications, and the complexity of modeling the domain of execution and its environment means that bugs might continue to occur. In this context, we need better and new debugging approaches.

2 Definition of *bug* and *debugging*

The literature describes very accurately what practitioners observe when debugging programs [15]. The source of a bug is a *defect*, *i.e.*, a piece of code that produces a run-time infection. An infection is an inconsistent state or behavior of the program that differs from what is expected, we may call that *errors*. This infection may spread and produce other errors, in a cascade effect which ultimately ends in an user-observable error : this is a *failure*. What we commonly call *bug* is actually the whole chain from the defect to the failure.

Debugging is the activity of tracking and fixing bugs. It follows a set of rigorous and systematic methods [14, 15, 13].

Reproducing the Failure. The first step is to make sure the failure can be reproduced. This is critical : without this step, gathering information on the bug is very difficult, and checking if a fix actually solves it is impossible.

Simplifying the Reproduction. After making the failure reproducible, the goal is to simplify and shrink the conditions needed to reproduce it, to have a simpler execution to inspect, and reduce the size of the suspect codebase.

Finding the Defect by using the Scientific Method. In order to reason backward from the failure to the defect, the developer observes the flawed execution, formulates a hypothesis as to what the defect is, tests this hypothesis through experiments, and refines her hypothesis based on the result. This loop continues until the developer finds the defect and why it caused the failure.

Fixing the Defect. The final step is fixing the defect, and checking that the failure no longer happens.

3 Challenges of debugging

In this section, we list problems that slow down, limit or prevent debugging, and formulate research questions to capture the inherent challenges they pose.

The symptom−source distance. This is one of the most common difficulty reported by practitioners when debugging. It refers to the fact that an observed failure (the symptom) does not occur at the same point in the code as the defect that provokes it (the source). For example, a defect in a program may produce an erratic value at some point, but the related failure is only observed when this value is used in a distant part of the code, for example when displayed in a GUI. This problem was first reported as a major difficulty in 1997 [7], but recent studies report that it is still a problem 20 years after [11].

— *How to design tools that help the developers overcome large symptom−source distances, and what are the requirements to build such tools ?*

Concurrency and parallelism. They represent the second most frequent cause of today's hard bugs [11]. Reproducing bugs in concurrent/parallel programs requires more than running the same code and inputs. In such systems, processes and threads perform concurrent access to shared memory, and interact with each others in ways that may depend on external factors.

Ways by which single-threaded programs can be stepped and analyzed with existing debuggers cannot directly be applied to concurrent programs [10, 9]. Building adequate tools is hard, because we lack abstractions to express new debugging operations targeting concurrency bugs (*e.g.*, race conditions).

— *Can we find ways to systematically reproduce bugs in concurrent programs ? If such ways do exist, what tools do we need to implement them and what abstractions do we need to support these implementations ?*

— *We believe the best moment to observe a bug is the moment it actually appears and not in a post-mortem analysis. Could we debug a concurrent program on-the-fly [4], to catch and observe bugs at the moment they appear ? How would this capability improve debugging practices ?*

Bug Reproduction. Reproducing bugs is a vital step in understanding faults and finding the defect from which they originate [15, 13]. Developers can rerun the buggy program multiple times to better understand the bug, and test that the bug is actually gone after they make changes to the source code.

However, some bugs are hard (or even impossible) to reproduce in a controlled manner [12, 1, 15]. Bugs with non-deterministic aspects (*e.g.* concurrency) are notably hard to reproduce. Some bugs only happen after long executions and in specific conditions : those are materially inconvenient to reproduce. A company reported a bug occurring on its servers only after 15 days of heavy loads. Other

bugs produce infections but then mask the faulty behavior or the inconsistent state of the program. Symptoms are no longer observable, which hinders bug reproduction and understanding.

— *We believe that to ease bug reproduction, we need to capture and leverage contextual run-time information. But how can we identify information relevant with the investigated bug ? In addition, the evolution of the program state and execution path throughout an entire execution represents a lot of data. How can we overcome this limitation ?*
— *However, if there are bugs that truly cannot be reproduced in a controlled manner, what alternative methodology can we use, and if it does not exist can we define new ways of debugging these bugs ?*

Designing usable debugging tools. In 1997, the debugging scandal[8] was the observation that debugging tools had progressed very little over 30 years. Recent studies [11, 2] show that new technologies have not significantly improved the debugging experience. Tools implementing these technologies are hard to understand and require considerable learning time, sometimes developers do not know about their existence [11].

Another pitfall for tools is to be either too specific, in such a way that they are almost never applicable in practice, or too generic, in such a way that they are always usable but not helpful enough for the specific bug being encountered. The challenge is to design tools that actually help the developers in the field, by striking the right balance between genericity and specialization, and the right balance between ease-of-use and feature-wealth [3, 5].

— *Can we pinpoint what characteristics a tool needs to have to be easy to pick-up by developers and powerful enough to help them ?*
— *What is a debugger API that is powerful and versatile enough to allow developers to perform many debugging tasks that would normally require specific debugging tools or tedious manual operations ?*

4 New Generation Debuggers

We believe that tackling the challenges laid out in this paper is the key to unlock new debugging capabilities, allowing the design and implementation of better debugging strategies and tools : **New Generation Debuggers ! [6]**

We should give developers means to build ad-hoc and domain-specific debugging tools for the exploration of their problems. To do that, we aim to provide control over debugging architectures through generic and versatile scripting APIs [5]. We believe such API will help exploring program executions and simplify the implementation of debuggers.

Hot debugging, or unanticipated debugging [4], is a promising solution to obtain information about bugs during run time, and lower the risk of losing information by restarting the program. In addition of better programs comprehension, it provides ways of fixing bugs without restarting programs.

From the reverse-engineering domain, other techniques could be beneficial to debugging and should be explored. Among others, profiling and benchmarking are interesting prospects. By providing a vision on the execution of a failing code, these techniques could be complementary to standard debuggers in the understanding of a failure and also help close the *symptom—source* gap.

We believe these are good technical research directions. But to support the implementation of new techniques, fundamental basis are required and do not always exist. We need to identify and study the properties that programming languages and their infrastructure (*i.e.*, virtual machines) must exhibit to support debugging features that effectively help debugging.

In addition, we need to better evaluate new approaches and tools by performing controlled and industrial experiments as well as field studies. This is a difficult but necessary step to understand what techniques and tools really benefit to developers. New empirical results will improve the validation of tools, and highlight the most promising research directions.

Finally, few individuals and research teams do work on debugging. We know of RMoD from Inria Lille, DiverSE from Inria Rennes and MOCS from the University of Brest. Despite these numbers, we think that debugging is actually a concern to the whole community, that has real impact and cost even on research (*e.g.*, when implementing prototypes). We need to gather forces and create discussion groups to enforce research around the challenges of debugging.

Références

[1] AGANS, D. J. *Debugging : The 9 indispensable rules for finding even the most elusive software and hardware problems*. Amacom, 2002.

[2] BELLER, M., SPRUIT, N., SPINELLIS, D., AND ZAIDMAN, A. On the dichotomy of debugging behavior among programmers. In *Proceedings of ICSE 18 : 40th International Conference on Software Engineering* (2018).

[3] CHIŞ, A., GÎRBA, T., AND NIERSTRASZ, O. The Moldable Debugger : A framework for developing domain-specific debuggers. In *Software Language Engineering* (2014), Springer, pp. 102–121.

[4] COSTIOU, S. *Unanticipated behavior adaptation : application to the debugging of running programs*. Theses, Université de Bretagne occidentale - Brest, Nov. 2018.

[5] DUPRIEZ, T., POLITO, G., COSTIOU, S., ARANEGA, V., AND DUCASSE, S. Sindarin : A versatile scripting api for the pharo debugger. In *DLS'19, Dynamic Language Symposium* (2019).

[6] DUPRIEZ, T., POLITO, G., AND DUCASSE, S. Analysis and exploration for new generation debuggers. In *Proceedings of the 12th Edition of the International Workshop on Smalltalk Technologies* (New York, NY, USA, 2017), IWST '17, ACM, pp. 5 :1–5 :6.

[7] EISENSTADT, M. My hairiest bug war stories. *Commun. ACM 40*, 4 (1997), 30–37.

[8] LIEBERMAN, H. Introduction. *Commun. ACM 40*, 4 (Apr. 1997), 26–29.

[9] LOPEZ, C. T., SINGH, R. G., MARR, S., BOIX, E. G., AND SCHOLLIERS, C. Multiverse debugging : Non-deterministic debugging for non-deterministic programs. In *33rd European Conference on Object-Oriented Programming* (2019).

[10] MARR, S., LOPEZ, C., AUMAYR, D., GONZALEZ BOIX, E., AND MOSSENBOCK, H. Kompos : A platform for debugging complex concurrent applications. In *Programming'17* (apr 2017), pp. 1–2.

[11] PERSCHEID, M., SIEGMUND, B., TAEUMEL, M., AND HIRSCHFELD, R. Studying the advancement in debugging practice of professional software developers. *Software Quality Journal 25*, 1 (2017), 83–110.

[12] RAYMOND, E. S., AND STEELE, G. L. *The new hacker's dictionary*. Mit Press, 1996.

[13] SPINELLIS, D. Modern debugging : The art of finding a needle in a haystack. *Commun. ACM 61*, 11 (Oct. 2018), 124–134.

[14] TELLES, M., AND HSIEH, Y. *The science of debugging*. Coriolis Group Books, 2001.

[15] ZELLER, A. *Why programs fail : a guide to systematic debugging*. Elsevier, 2009.

Défi : La sécurité dans le développement logiciel

Equipe Archware/IRISA

1 Introduction

Sécuriser un logiciel doit être un principe de conception qui concerne chaque étape du développement, de l'ingénierie des exigences à la conception, implémentation, test et déploiement. En effet, de plus en plus le principe *secure by design* en génie logiciel devient la principale approche de développement pour assurer la sécurité et la confidentialité des logiciels. Elle s'attache à sécuriser le logiciel dès les fondations et démarre par la conception d'une architecture robuste. Le défi que nous proposons consiste à passer du cycle de vie du logiciel au cycle de vie du logiciel *sécurisé*, mettant en œuvre le principe *secure by design* du point de vue du génie de la programmation et du logiciel.

2 Transversalité du défi

Ce défi se place à l'intersection de la sécurité et du génie de la programmation et du logiciel. Le GDR Sécurité [1] structure la communauté de recherche en sécurité en France. Il se concentre, au travers de ses groupes de travail, essentiellement sur des problématiques liées aux réseaux et à la protection des données. À l'exception du groupe de travail « Méthodes formelles pour la sécurité [2] » et en partie du groupe « Sécurité des systèmes, des logiciels et des réseaux [3] », la question du logiciel nous semble peu traitée par ce GDR Sécurité. Or ces deux groupes de travail ne traitent pas des questions de sécurité au niveau des exigences ou de l'architecture. Le défi que nous proposons adopte un point de vue différent, qui n'est pas couvert par les groupes de travail existants.

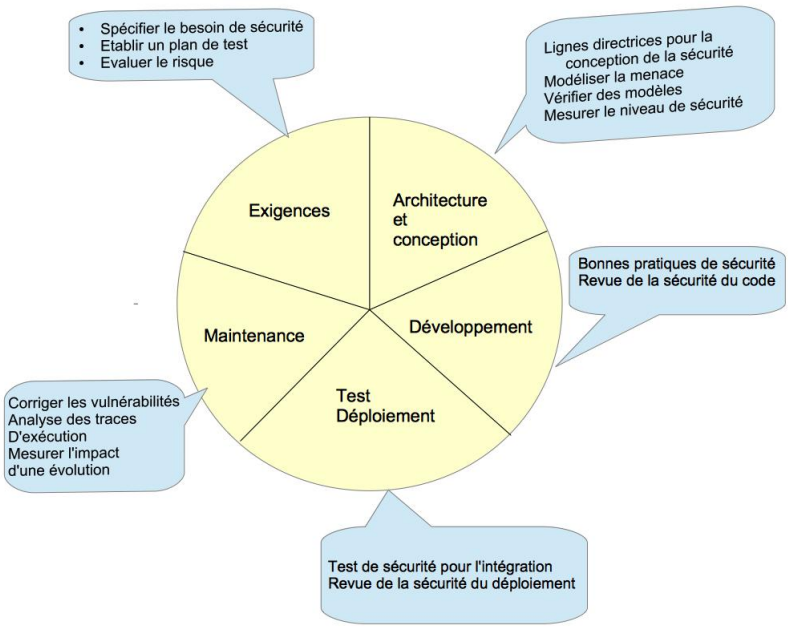
La majorité des groupes de travail du GDR/GPL pourraient contribuer à ce défi : IE (Ingénierie des exigences), RIMEL (Rétro-Ingénierie, Maintenance et Evolution des Logiciels), SDS (Systèmes de systèmes), IDM (Ingénierie dirigée par les modèles), MTM2 (Méthodes de test pour la validation et la vérification) pour ne citer que ceux-là.

Nous pensons qu'une initiative est utile pour rassembler et animer une communauté autour de ce défi.

3 La sécurité dans le cycle de vie du logiciel

Nous avons organisé cette proposition de défi en examinant comment la sécurité est prise en compte [3] ou pourrait être prise en compte dans les différentes étapes de développement d'un logiciel ainsi que dans l'étape de maintenance et évolution. La figure ci-dessous montre des activités d'ingénierie de sécurité et où elles pourraient être intégrées dans le cycle de vie d'une application.

1. https://gdr-securite.irisa.fr/
2. http://gtmfsec.irisa.fr/
3. https://gdr-securite.irisa.fr/securite-des-systemes-des-logiciels-et-des-reseaux/



Nous allons parcourir le cycle de vie du logiciel afin de pointer les aspects de sécurité dans chaque étape et formuler les questions à résoudre pour les problématiques associées.

3.1 Sécuriser les exigences

La prise en compte des aspects sécurité dans le développement logiciel (et système) nécessite de spécifier en amont quels sont les objectifs de sécurité et ce que l'on souhaite sécuriser. Cela pose plusieurs problèmes. D'une part, les exigences de sécurité sont généralement issues des analyses de sécurité. Ces analyses, menées via des méthodes telles que EBIOS [4] ou encore NIST-800-30 [5], ne sont pas conçues pour éliciter des exigences mais produire une politique de sécurité. Si ces rapports identifient les biens (*assets*) à prendre en compte, ils ne définissent pas clairement les exigences. Passer de l'un à l'autre n'est pas trivial et nécessite la recherche de moyens guidant et/ou automatisant ce passage. Donc une première étape critique est d'obtenir les exigences de sécurité dans le cycle de développement. Pour faciliter cela, il est possible d'imaginer l'élaboration d'un catalogue d'exigences de sécurité ou de catégories d'exigences. Il s'agit d'une autre direction à explorer. D'autre part, si l'utilisation de cette démarche peut s'appliquer dans des projets de grande taille, ce n'est pas le cas dans des projets plus modestes ou dans des processus plus agiles dans lesquels le coût nécessaire à les conduire serait prohibitif. Cela nécessite donc des efforts afin de simplifier et/ou d'automatiser les analyses de risques, et de capturer leur résultat de manière simple et rapide. Enfin, si les spécialistes des domaines sont généralement aptes à éliciter des exigences fonctionnelles, ils le sont moins pour les exigences non fonctionnelles. L'élicitation de telles exigences passe donc par une collaboration entre

4. https://www.ssi.gouv.fr/guide/ebios-2010-expression-des-besoins-et-identification-des-objectifs-de-securite/
5. https://csrc.nist.gov/publications/detail/sp/800-30/rev-1/final

experts des domaines et experts de la sécurité. Il est donc besoin de proposer des moyens collaboratifs afin de tisser un lien entre ces deux mondes.

3.2 Sécuriser l'architecture logicielle et la conception

Pour éviter de nombreuses vulnérabilités introduites par de mauvais choix de conception, l'activité de conception doit utiliser des bonnes pratiques, des modèles et des principes de conception éprouvés.

Connaître les menaces Il est important de connaître les menaces auxquelles l'architecture et la conception sont exposées. L'activité de modélisation des menaces [6] permet d'identifier et de comprendre les menaces, de comprendre les risques que chaque menace pose et de découvrir les vulnérabilités que nous pouvons utiliser pour façonner les décisions de conception et de mise en œuvre de la sécurité. Comme nous l'avons écrit précédemment, les analyses de sécurité comme EBIOS prévoient d'identifier les menaces et les objectifs susceptibles d'être visés, les objectifs recherchés par ces menaces. Une piste pourrait être de considérer qu'il existe, en plus des cas d'utilisation, des cas malicieux correspondant à ces objectifs visés.

Modéliser Depuis plus d'une dizaine d'années des propositions de langages de modélisation intégrant des aspects de sécurité ont vu le jour [7]. Les plus connus sont SecureUML [1] et UMLSec [5]. Certains langages se concentrent sur un ou plusieurs attributs déclaratifs de sécurité, tels que la confidentialité, l'intégrité ou la disponibilité. D'autres langages se concentrent sur un ou plusieurs aspects opérationnels de sécurité, tels que le contrôle d'accès ou l'authentification. Finalement il existe des langages se concentrant sur un ou plusieurs domaines, tels que les architectures orientées services ou les bases de données, ou au contraire restant génériques, indépendants du domaine.

Ces langages concernent le plus souvent la modélisation des solutions, des contre-mesures, des contrôles de sécurité. Néanmoins, nous sommes en manque d'approches utilisant des méthodes et outils théoriques et pratiques pour modéliser, définir ou identifier des failles (ou vulnérabilités) architecturales. Alors que de nombreuses bases de vulnérabilités existent au niveau du code (par exemple CVE, CWE), il est difficile de caractériser les vulnérabilités architecturales : sont-elles des *bad smells*, des *anti-patterns*, etc ? Comment les identifier : avec des approches de *model matching* ou d'apprentissage automatique, etc ? Une piste à explorer serait de reprendre tous ces concepts, de trouver une façon uniforme de les modéliser, de les positionner et de les associer éventuellement à des outils pour constituer un catalogue exploitable au moment de l'identification des failles de l'architecture.

Vérifier les mesures de sécurité En majorité les langages de modélisation, intégrant des aspects de sécurité, proposent une syntaxe concrète pour la représentation de la sécurité, basée sur UML, ils ciblent les aspect structurels de la sécurité dans la phase de conception détaillée. Très peu de propositions offrent l'analyse de la représentation. Or c'est bien un des objectifs de l'ingénierie d'un systèmes sécurisé : vérifier que les mesures adéquates ont été prises pour contrer les cas malicieux identifiés. Peu nombreuses sont les approches qui assurent l'analyse de l'architecture par rapport aux exigences de sécurité, et la plupart de ces approches se concentrent sur le contrôle d'accès. Il est nécessaire de proposer des analyses pour les autres types de mesures de sécurité, afin de vérifier la prise en compte des cas malicieux.

Pour répondre à ce défi de l'analyse, une piste pourrait être de proposer comment évaluer, mesurer le niveau de sécurité d'une architecture, dans sa globalité. Cela pose plusieurs questions. Comment identifier les éléments d'une architecture en lien avec la sécurité ? Comment exprimer la sémantique

d'un élément d'architecture ? Quelles métriques définir pour caractériser le niveau de sécurité d'une architecture ? Serait-il possible de définir des métriques indépendantes du domaine auquel l'architecture s'applique ? Comment tester et comparer ces métriques ?

Dans un système complexe ou dans un système de systèmes, des comportements émergent de l'interaction entre les constituants. Comme l'a montré par exemple la notion d'attaque en cascade, certains de ces comportements émergeants peuvent introduire des vulnérabilités, exploitables par un acteur malveillant. Analyser ces systèmes pour détecter de telles vulnérabilités émergeantes est un réel défi.

Proposer des améliorations En supposant qu'il soit possible d'identifier les vulnérabilités ou, par analyse, de détecter des exigences de sécurité non satisfaites, resterait le défi de proposer à l'architecte des solutions potentielles, pour améliorer la sécurité au niveau de l'architecture.

Un nombre important de patrons de conception de sécurité a été proposé. Ils sont recensés dans des catalogues tels que [4]. Dans un objectif de *refactoring* d'une architecture pour la sécuriser, la question de l'intégration semi-automatique ou assistée par ordinateur des patrons reste néanmoins tout entière.

Dans quel format représenter les patrons de sécurité pour pouvoir les intégrer dans les langages d'architecture ? Comment identifier les situations ou le contexte dans les modèles d'architecture, dans lesquels les patrons seraient applicables ? Comment les appliquer (automatiquement) aux modèles d'architecture ? Comment vérifier qu'un patron est correctement employé, c'est-à-dire dans le bon contexte et pour répondre aux bonnes exigences ? Comment quantifier la contribution de chaque patron au niveau de sécurité de l'architecture ?

Une piste pour répondre à certaines de ces questions pourrait être, par exemple, de s'appuyer sur le catalogue d'exigences ou de catégories d'exigences de sécurité que nous envisageons comme piste à la section précédente.

3.3 Sécuriser le code et son déploiement

Les exigences et l'architecture spécifient la sécurité du système. Encore faut-il que l'implémentation et le déploiement mettent effectivement en œuvre ce qui a été exprimé lors des phases précédentes du cycle de développement.

Programmer pour la sécurité Une mauvaise implantation de propriétés de sécurité voulues et spécifiées lors des phases précédentes ruinerait les efforts concédés lors de la conception de l'architecture. Comment mesurer l'adéquation d'une implémentation aux besoins exprimés à travers certaines propriétés de sécurité ?

Par ailleurs, une écriture naïve de la partie fonctionnelle laisserait apparaître des vulnérabilités potentiellement exploitables par des attaquants. Comment s'assurer qu'un code ne contient pas de vulnérabilité ? Qu'est-ce qu'une vulnérabilité ? Nous disposons d'une certaine connaissance cumulative (CVE), et il existe des règles de bonnes pratiques de codage (*cheat sheets* OWASP [6] ou les guides de l'ANSSI comme les références [7] [8] par exemple). Mais comment exploiter ces ressources de manière efficace ? Quel est leur niveau de complétude ? La plupart des travaux dans le domaine proposent des audits ou une analyse de code pour détecter des vulnérabilités et identifier des parties

6. https://cheatsheetseries.owasp.org/
7. https://www.ssi.gouv.fr/particulier/guide/recommandations-pour-la-securisation-des-sites-web/
8. https://anssi-fr.github.io/rust-guide

du code potentiellement dangereuses. Une autre piste à explorer pour sécuriser le développement du code, une fois les problèmes identifiés, est de proposer une correction de sécurité qui passera les tests et les outils de mesures. Toutefois les outils d'analyse produisent beaucoup de faux positifs existants [2] et ne détectent que les vulnérabilités connues. Proposer des scanners de code efficaces devient une urgence au vu de la taille du code des applications d'aujourd'hui.

Tests de sécurité La phase d'implémentation inclut les tests unitaires. Quels types de tests orientés sécurité peuvent être réalisés lors des tests unitaires ? Comment exploiter la sémantique (connue) de l'application pour orienter les tests de sécurité unitaires et d'intégration ? Une piste pourrait être de s'appuyer sur les cas malicieux pour élaborer des tests ciblés.

Mesure du niveau de sécurité De même qu'au niveau de l'architecture, existe-t-il des métriques au niveau du code pour qualifier ou quantifier les propriétés de sécurité ? Des outils et des algorithmes existent pour mesurer la qualité d'un code source, il faudrait prendre une démarche similaire pour trouver des mesures du niveau de sécurité.

Déploiement Lors du déploiement il s'agit d'identifier des vulnérabilités dues à l'interaction avec l'environnement (e.g. infrastructure, OS), mais aussi avec les utilisateurs. En effet, la réussite des cyber-attaques résulte souvent d'une erreur humaine. Permettre l'évaluation des vulnérabilités humaines au même titre que des vulnérabilités système, aiderait à réduire la vulnérabilité globale. Les recherches ciblant les systèmes socio-techniques vont dans ce sens.

3.4 Sécuriser la phase de maintenance

La sécurité dans la phase de maintenance consiste à analyser l'application de manière structurée et systématique pour repérer ses menaces et ses vulnérabilité. Ses deux objectifs sont : 1) repérer et corriger des vulnérabilités rencontrées lors des dernières exécutions du système ; 2) sur la base du vécu (traces d'exécution), essayer d'identifier des vulnérabilités qui n'ont pas été encore mises à jour.

La question de recherche ici concerne le mariage des techniques d'analyse post mortem (*forensic*) et des techniques de ré-ingénierie logicielle. L'analyse post mortem est l'analyse d'un système après incident. Diverses techniques existent pour extraire la quantité maximale d'information et recueillir autant de preuves, retrouver toutes les données cachées et non tracées. Une piste peut être d'explorer ces techniques dans le cadre d'une rétro-ingénierie avec les objectifs mentionnés plus haut.

La maintenance d'une application logicielle reste l'activité la plus délicate car, en effet, suite à une évolution, il faut maintenir le niveau de sécurité tout en maîtrisant le coût de l'évolution. Finalement on peut aussi s'interroger sur l'impact de l'évolution sur la sécurité.

Il existe bien sûr d'autres aspects concernant la sécurité lors de la phase de maintenance.

4 Conclusions

Nous avons présenté notre défi sous la forme de quelques pistes pour produire des logiciels sécurisés en suivant le principe du *secure by design*. Des questions générales se posent également plus spécifiquement sur le cycle de développement : comment assurer l'intégration (continue) des aspects de sécurité tout au long du cycle de vie ? Comment se concentrer sur l'interface entre les différents phases ? Comment définir et respecter les bonnes pratiques ? Il reste également beaucoup d'aspects à prendre en compte, concernant notamment les processus agiles ou non ou d'intégration de la sécurité.

Références

[1] *SecureUML :A UML-based modeling language for model-driven security*, volume 2460 of *Lecture Notes in Computer Science*. Springer, 2002.

[2] R. Benabidallah, S. Sadou, B. Le Trionnaire, and I. Borne. Designing a code vulnerability meta-scanner. In *ISPEC2019 - The 15th International Conference on Information Security Practice and Experience, Lumpur, Malaysia, November 26-28 2019, Proceedings*, pages 194–210, 2019.

[3] Premkumar T. Devanbu and Stuart G. Stubblebine. Software engineering for security : a roadmap. In *22nd International Conference on on Software Engineering, Future of Software Engineering Track, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 227–239, 2000.

[4] Eduardo Fernandez-Buglioni. *Security Patterns in Practice : Designing Secure Architectures Using Software Patterns*. Wiley Publishing, 1st edition, 2013.

[5] Jan Jürjens. Umlsec : Extending UML for secure systems development. In *UML 2002 - The Unified Modeling Language, 5th International Conference,Dresden, Germany, September 30 - October 4, 2002, Proceedings*, pages 412–425, 2002.

[6] Adam Shostack. *Threat Modeling : Designing for Security*. Wiley Publishing, 1st edition, 2014.

[7] Alexander Van Den Berghe, Riccardo Scandariato, Koen Yskout, and Wouter Joosen. Design notations for secure software : A systematic literature review. *Softw. Syst. Model.*, 16(3) :809–831, 2017.

Combinatoire certifiée *

Alain Giorgetti[1] et Nicolas Magaud[2]
[1] Institut FEMTO-ST (UMR CNRS 6174), Université de Franche-Comté
alain.giorgetti@femto-st.fr
[2] ICube (UMR CNRS 7357), Université de Strasbourg
magaud@unistra.fr

Résumé

Ce document présente des défis liés au développement de la combinatoire certifiée, définie comme l'application des méthodes formelles du génie logiciel à la recherche en combinatoire.

Ce défi est lié aux thématiques du groupe de travail LTP (Langages, Types et Preuves) du GdR GPL. Il a été alimenté par des interactions au sein de ce groupe.

1 Contexte et problématique

La combinatoire est la branche des mathématiques qui étudie les familles de données munies d'une taille telle qu'il existe un nombre fini de données différentes de chaque taille. La combinatoire énumérative s'intéresse au dénombrement de ces structures combinatoires. La combinatoire bijective établit des bijections structurelles non triviales entre diverses familles de structures combinatoires.

Le domaine de la combinatoire est encore peu perméable aux méthodes formelles de spécification et de preuve de théorèmes et de programmes. Souvent, les combinatoriciens utilisent un système de calcul formel, comme Maple, Mathematica ou Sage. Rares sont ceux qui connaissent et pratiquent la démonstration automatique ou assistée par ordinateur. On peut néanmoins citer les exemples majeurs de démonstration formelle du théorème des quatre couleurs [19] et de la correction des fonctions principales du logiciel SCHUR de calcul des caractères des groupes de Lie et des fonctions symétriques [4].

2 Défis identifiés

En complémentarité de ces "tours de force", qui visent des théorèmes majeurs, qui ont occupé plusieurs chercheurs pendant plusieurs années et qui s'appuient sur de larges bibliothèques de formalisation des mathématiques, il reste à inventer, outiller et diffuser une "*combinatoire formelle pratique*" qui relève les défis suivants :

1. Applicabilité non pas à des théorèmes connus, mais à des conjectures en cours d'élaboration par les combinatoriciens. C'est un défi ambitieux, car le temps requis pour achever une preuve formelle est très supérieur à celui de l'élaboration et de la rédaction d'une preuve papier, mais c'est un enjeu important pour intéresser la communauté de la combinatoire.

*Le premier auteur est le porteur du défi.

2. Simplicité de mise en œuvre. Il s'agit de réduire la pente et la longueur de la courbe d'apprentissage des outils de vérification formelle.
3. Diffusion des méthodes et outils de verification formelle dans la communauté des chercheurs en combinatoire.

3 Moyens

Les moyens envisagés pour relever ces défis sont :

1. Viser la formalisation de problèmes combinatoires plus modestes que les "tours de force" précédemment cités, pour obtenir des résultats dans des délais plus courts, afin que les efforts investis en formalisation et recherche de preuve produisent des gains de productivité de nouveaux théorèmes.
2. Intégrer dans chaque outil de formalisation et de vérification la possibilité de tester des conjectures, des propriétés et les conditions de vérification produites. Ce levier important est détaillé dans la partie 4.
3. Proposer des théories logiques et des langages de spécification et de programmation adaptés aux objets de la combinatoire.
4. Intégrer dans les outils de vérification de ces théories, spécifications et programmes l'état de l'art en matière de test de propriété et d'automatisation des preuves.
5. Diffuser des bibliothèques d'exemples de théorèmes vérifiés formellement, et une méthodologie guidée pour en produire d'autres.

4 Test de propriétés

Dans un assistant de preuve, il est précieux de se convaincre qu'un lemme est correct avant d'investir du temps dans sa démonstration interactive. Tester une propriété avant d'en commencer une démonstration interactive permet de se convaincre qu'elle est correcte, ou sinon de gagner un temps précieux en détectant rapidement une faille de raisonnement ou une erreur de formulation.

Le *test de propriété* (PBT, pour *Property-Based Testing*) est la recherche d'un contre-exemple pour une propriété d'un programme en cours de vérification. Il est populaire pour les langages fonctionnels, notamment avec l'outil QuickCheck [6] dans Haskell. Le PBT a également été adapté à des assistants de preuve, comme Isabelle [2], Agda [10], PVS [22], FoCaLiZe [5] et plus récemment Coq [23], avec l'outil de test aléatoire QuickChick. Dans ce cadre des assistants de preuve, il permet de tester des conjectures.

Parmi les méthodes de test automatique, le test exhaustif borné (BET, pour *Bounded Exhaustive Testing*) d'une fonction ou propriété consiste à générer toutes ses données d'entrée jusqu'à une certaine taille. Le BET est particulièrement bien adapté aux structures combinatoires, car ses contre-exemples sont toujours de taille minimale (ce qui facilite le déboguage), sa couverture est bien identifiée (puisqu'il constitue une preuve par énumération de tous les cas jusqu'à la borne de test), et une donnée de petite taille suffit souvent pour révéler une erreur [20]. Ainsi, le BET est complémentaire du test aléatoire, plus adapté aux domaines de données de plus grande taille.

Certains outils de test généraux sont capables de générer des données ou de dériver des générateurs à partir de la définition de ces données, selon diverses techniques, détaillées dans les introductions de travaux récents sur ce sujet [21, 7]. Par exemple, Dubois, Giorgetti et Genestier ont complété

QuickChick avec une première approche de test exhaustif borné fondée sur des générateurs dérivés de définitions des données en programmation logique (Prolog) [9].

Cependant, pour certaines structures de données, ces techniques et outils peuvent être trop lents, échouer dans la dérivation d'un générateur ou dériver des générateurs peu efficaces. Il devient alors pertinent de concevoir un générateur dédié à chaque structure, voire de le certifier pour l'intégrer avec confiance dans un outil de test. Par exemple, Bowles et Caminati ont vérifié un algorithme d'énumération de structures d'événements [3]. Dubois et Giorgetti ont développé l'outil CUT, qui ajoute à Coq les commandes SmallCheck et SmallCheckWhy3 de test exhaustif borné avec des programmes d'énumération dédiés, respectivement définis dans les langages de Coq et de Why3 [8]. Afin d'accroître la confiance dans ces programmes, Dubois et Giorgetti proposent de les produire par extraction de programmes WhyML dont la correction et la complétude sont démontrées formellement avec Why3 et Coq [15]. Lorsque ces programmes énumèrent des structures combinatoires, cette preuve formelle devient elle-même une activité de combinatoire formelle, qui complète, voire remplace, une preuve informelle d'un ouvrage de combinatoire énumérative.

5 Jalons

Nous proposons de désigner par *combinatoire certifiée* l'application des méthodes formelles du génie logiciel à la recherche en combinatoire, en particulier pour prouver formellement des théorèmes de combinatoire énumérative ou bijective, ou des propriétés de programmes de comptage, d'énumération ou de transformation bijective de structures combinatoires.

Des collaborations avec des combinatoriciens sont essentielles pour le succès de cette démarche, comme ce fut le cas avec Florent Hivert pour le logiciel SCHUR [4] ou avec Alain Giorgetti sur les permutations et les cartes combinatoires [18, 16, 11, 9, 8, 15]. Ce dernier a apporté à ces travaux sa formation initiale de combinatoricien. S'étant ensuite formé aux méthodes formelles du génie logiciel, il a aussi participé à leur expérimentation pour vérifier formellement des théorèmes de combinatoire [14, 12, 13, 9, 8, 15]. Cette approche formelle a été présentée et défendue lors de recherches récentes avec d'autres combinatoriciens [17, 1, 9], puis lors de "Journées de combinatoire certifiée" organisées au LRI, du 3 au 5 juillet 2019, par Jean-Christophe Filliâtre et Alain Giorgetti. Ces journées ont été financées par la bourse mobilité 2019 du GdR GPL du CNRS d'Alain Giorgetti.

6 Conclusion

Nous pensons que la recherche en combinatoire peut être facilitée par une formalisation machine des problèmes dès le début de leur étude, et accompagnée par une utilisation systématique de méthodes formelles et d'outils de vérification. Dans cette démarche, le test de propriété joue un rôle essentiel. Il permet aux combinatoriciens de construire tranquillement des théorèmes, par exemple dans Coq, et d'être alertés rapidement que leur énoncé est faux, bien avant d'être bloqués dans leur démonstration formelle. Inversement, la combinatoire, comme d'autres domaines de recherche, peut devenir un nouveau champ d'application, d'adaptation et de perfectionnement des méthodes formelles de spécification et de vérification.

Quoique vaste, le sujet de la combinatoire certifiée n'a pas vocation à être l'unique thématique d'un futur GT. Il peut par contre s'intégrer dans un GT sur des thèmes proches ou plus généraux, tels que :

— les nouvelles applications des méthodes et outils du génie logiciel et de la théorie des types : algorithmes et programmes produits par la recherche scientifique actuelle, en particulier dans

le domaine de l'intelligence artificielle ;
— la diffusion des méthodes formelles, par leur enseignement dès la licence ;
— le lien théorique entre raisonnement et calcul, concrétisé par le développement d'interfaces générales et efficaces entre les outils de preuve formelle et les logiciels de calcul formel ;
— la fiabilité et la pérennité des logiciels produits par la recherche en génie logiciel, et la reproductibilité de ses expérimentations.

Références

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

[2] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *Software Engineering and Formal Methods (SEFM 2004)*, pages 230–239. IEEE Computer Society, 2004.

[3] J. Bowles and M. B. Caminati. A verified algorithm enumerating event structures. In *CICM'17*, volume 10383 of *LNCS*, pages 239–254. Springer, 2017.

[4] F. Butelle, F. Hivert, M. Mayero, and F. Toumazet. Formal proof of SCHUR conjugate function. In *Intelligent Computer Mathematics*, volume 6167 of *LNCS*, pages 158–171. Springer, 2010.

[5] M. Carlier, C. Dubois, and A. Gotlieb. Constraint Reasoning in FOCALTEST. In *International Conference on Software and Data Technologies (ICSOFT 2010)*, Athens, Jul. 2010.

[6] K. Claessen and J. Hughes. QuickCheck : a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279, New York, NY, USA, 2000. ACM.

[7] S. Cruanes. Satisfiability modulo bounded checking. In *Automated Deduction – CADE 26*, pages 114–129, Cham, 2017. Springer International Publishing.

[8] C. Dubois and A. Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, Jul 2018.

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

[10] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs 2003*, volume 2758 of *LNCS*, pages 188–203, Heidelberg, 2003. Springer.

[11] R. Genestier and A. Giorgetti. Spécification et vérification formelle d'opérations sur les permutations. In *AFADL'16*, pages 72–78, 2016. `http://events.femto-st.fr/sites/femto-st.fr.gdr-gpl-2016/files/content/AFADL-2016.pdf`.

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

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

[14] A. Giorgetti. Guessing a Conjecture in Enumerative Combinatorics and Proving It with a Computer Algebra System. In *SCSS'10*, pages 5–18, July 2010.

[15] A. Giorgetti, C. Dubois, and R. Lazarini. Combinatoire formelle avec Why3 et Coq. In N. Magaud and Z. Dargaye, editors, *Journées Francophones des Langages Applicatifs 2019*, pages 139–154, Les Rousses, France, Jan. 2019. publié par les auteurs. `https://hal.inria.fr/hal-01985195`.

[16] A. Giorgetti, R. Genestier, and V. Senni. Software engineering and enumerative combinatorics. Institut Henri Poincaré, Paris, France, May 2014.

[17] A. Giorgetti and V. Senni. Specification and validation of algorithms generating planar Lehman words. In *GASCom'12, 8-th Int. Conf. on random generation of combinatorial structures*, Bordeaux, France, June 2012. `https://hal.inria.fr/hal-00753008`.

[18] A. Giorgetti and V. Senni. Combining tests and proofs to check combinatorial generation algorithms. Séminaire invité dans l'équipe CPR du laboratoire CEDRIC du CNAM, Paris, may 2013.

[19] G. Gonthier. The four colour theorem : Engineering of a formal proof. In *ASCM 2007*, volume 5081 of *LNCS (LNAI)*, pages 333–333. Springer, Heidelberg, 2008.

[20] D. Jackson and C. Damon. Elements of style : Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7) :484–495, 1996.

[21] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck : a language for property-based generators. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 114–129. ACM, 2017.

[22] S. Owre. Random testing in PVS. In *Workshop on Automated Formal Methods (AFM)*, 2006.

[23] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP 2015*, volume 9236 of *LNCS*, pages 325–343, Heidelberg, 2015. Springer.

Vers des Logiciels Éco-responsables
Le génie logiciel au défi de la sobriété écologique

Olivier Le Goaër, Adel Noureddine, Franck Barbier*
Romain Rouvoy†
Florence Maraninchi‡

Depuis son origine, le génie logiciel s'est attaché à élaborer des méthodes, concepts et outils pour réduire significativement les coûts liés au développement (maintenance, réutilisation...) des logiciels, garantir leur intégrité, fiabilité... Cependant, pour relever le défi de la transition écologique, une nouvelle préoccupation doit être intégrée dans tous ces artefacts : le coût environnemental. Ainsi, le génie logiciel « éco-responsable » est la branche qui s'intéresse à l'efficience énergétique et à la durabilité des logiciels. Nous dégageons notamment 6 grandes pistes de recherche qui, combinées, peuvent apporter des réponses à la hauteur des enjeux.

Mots-clés : énergie, environnement, éco-conception, génie logiciel

1 Introduction

Selon la dernière étude Green IT [Green IT, 2019], à l'échelle planétaire, l'empreinte environnementale du numérique équivaut à un continent de 2 à 3 fois la taille de la France, et à 5 fois le poids du parc automobile français. Les fabricants matériels optimisent l'efficience de leurs équipements depuis quarante ans tandis que les éditeurs de logiciels font le chemin inverse en entassant des couches applicatives les unes sur les autres. Peut-être que l'aspect purement immatériel du logiciel rend difficilement perceptibles les émissions carbone en bout de chaîne, mais c'est un fait. « Lorsque la mémoire était comptée, les développeurs informatiques avaient l'habitude d'écrire du code synthétique et efficace. Aujourd'hui, ces préoccupations ont disparu et l'on assiste à une véritable inflation des lignes de code, ce qui signifie des calculs plus longs et plus gourmands en électricité », raconte Anne-Cécile Orgerie dans [CNRS, 2010]. Mais, devant la menace écologique planétaire, la tendance commence à se renverser et l'on se pose à nouveau la question de la frugalité, dans un contexte où les usages du numérique ont littéralement explosés. Toute la production logicielle est concernée par l'éco-conception, mais ce sont les types d'applications ayant la plus forte base d'utilisateurs qui produiront les effets les plus visibles, comme le web, le mobile et le cloud, sans même parler des chiffres vertigineux de l'IoT. C'est en ce sens que de "nouveaux" défis se posent à la communauté du GDR GPL. Toutefois, il y a lieu de bien les caractériser et de les "éclairer" en termes de démarches de recherche à amplifier ou démarrer.

L'idée d'éco-conception logicielle a fait du chemin puisque l'appel à défis pour les journées du GDR GPL à Pau en Mars 2010 comportait déjà un défi lié à « *la réification de l'énergie dans le domaine du logiciel* » [Menaud et al., 2010]. Neuf ans plus tard, la conférence d'ouverture du GDR

*Université de Pau et des Pays de l'Adour - E2S
†Université de Lille / Inria / IUF
‡Université Grenoble Alpes / Verimag

GPL 2019 est « *Quels défis pour le développement durable des logiciels ?* » [Romain Rouvoy, 2019]. Peut-être que le temps est venu de créer un groupe de travail du GDR dédié à cette thématique afin de promouvoir ces activités plus vertueuses en matière de développement logiciel éco-responsable.

2 Pistes de recherches

Nous dégageons ci-dessous 6 pistes de recherche visant à produire des logiciels plus efficients en énergie et plus durables (ou soutenables), ces deux aspects étant primordiaux dans le domaine de l'éco-conception.

2.1 Savoir mesurer, comprendre et prédire la consommation énergétique

La première chose à faire pour maitriser la problématique de la consommation d'énergie par les couches logicielles est de disposer des outils (e.g., *energy profilers*) pour la mesurer au runtime. Pas si simple lorsque l'informatique est répartie, que les couches s'entremêlent, sont masquées (que fait l'OS par exemple des services requis par une machine virtuelle ?) et que les couches matérielles entrent inévitablement en jeu. Sans compter que l'outillage de mesure engendre lui-même une consommation qu'il faut être capable de retrancher au bilan énergétique. D'autre part, si mesurer un logiciel bien défini semble aisé, mesurer à grande échelle des millions de logiciels nécessite de faire de grands progrès du côté de la génération automatique de tests afin de se baser sur des scénarios d'usage réalistes lors des mesures.

Au-delà des mesures et des observations empiriques, il faut comprendre la chaîne d'instructions amenant à cette consommation. L'automatisation de ce processus doit favoriser la récolte de données de consommation et ainsi aider à comprendre plus précisément les facteurs impactant cette consommation au niveau de l'architecture logicielle. Cette compréhension est une étape cruciale pour construire des logiciels éco-responsables dès la phase de conception. Ainsi, la prédiction de la consommation énergétique logicielle dès les phases de conception pose un défi majeur pour la communauté du génie logiciel. L'éco-conception logicielle doit notamment pouvoir se projeter sur les contextes d'exécution ciblés, et proposer des modèles de prédictions réalistes des usages finaux.

2.2 Améliorer la qualité des productions logicielles

Une autre piste intéressante est de considérer que l'efficacité énergétique est un attribut de qualité (comme la sécurité ou l'accessibilité, par exemple) des logiciels modernes. Le corollaire est que les logiciels qui contiennent des défauts de conception provoquent *de facto* des surconsommations ; il faut être capable de les trouver et idéalement de les corriger ou de les isoler. Sur les équipements fonctionnant sur batterie (e.g., smartphone), la surconsommation a même un autre effet pervers puisqu'elle peut user prématurément la batterie dont le nombre de cycles de charge est limité. Cette usure engendre donc irrémédiablement une pollution dans le monde réel.

On trouve dans ce champ d'application les travaux sur l'analyse statique ou dynamique de code, les environnements d'exécutions, ainsi que les tests logiciels en général. On peut même imaginer de nouveaux langages de programmation et de nouveaux outils formels qui garantissent, par construction, l'efficience énergétique du code écrit par les développeurs. Enfin, des outils CASE dédiés et la génération automatique de code ont le potentiel d'augmenter l'efficience énergétique globale des lignes de production logicielle.

2.3 Dégraisser les logiciels

Une autre approche consiste à s'attaquer au surplus logiciel des « obésiciels », ces logiciels en surpoids par rapport à la fonction qu'il doivent remplir. L'ingénierie des exigences est ici concernée, pour intégrer « juste ce qu'il faut » de fonctionnalités dans les productions logicielles : la « wise-tech ». La variante extrémiste de cette approche étant la « *low-tech* », une sorte de décroissance logicielle choisie. Des travaux empiriques peuvent mettre en évidence cette dérive, en observant par exemple l'évolution de *frameworks* populaires (de la version 1.0 à nos jours) ou simplement de l'évolution du poids moyen d'une application au cours de la dernière décennie. La gestion des dépendances entre modules est aussi un sujet majeur puisque l'on se retrouve rapidement à incorporer beaucoup alors même que le logiciel fait peu.

D'apparence plus anecdotique, cette piste a en réalité un potentiel énorme puisque que l'on touche ici au fléau de renouvellement prématuré des équipements électroniques qui doivent rester capables d'exécuter les obésiciels, et donc à l'épuisement des ressources naturelles (métaux, terres rares, eau) pour en fabriquer sans cesse de nouveaux toujours plus avides d'énergie.

2.4 Sensibiliser les utilisateurs, engager les développeurs

Le génie logiciel éco-responsable peut et doit aussi attaquer le problème du côté des utilisateurs finaux et de celui des développeurs. Il est en théorie possible de modifier le comportement des utilisateurs en introduisant une forme de label/score écologique dans l'industrie du logiciel, comme cela se fait réglementairement dans de nombreux autres domaines. L'utilisateur doit « consommer » du logiciel de manière éclairée.

Quant aux développeurs, ils sont à la fois une partie de la solution et une partie du problème. Les études récentes auprès des développeurs indiquent un intérêt croissant vers l'éco-conception logicielle, mais ces derniers manquent d'informations et d'outils nécessaires à cette réalisation [Manotas et al., 2016]. Dis autrement, si le GL leur fournit les bons outils, alors ils s'en empareront. Mais dans le même temps, la problématique du développement et de la maintenance logicielle dont les chaines d'intégration continue et les nombreux postes mis en jeu peuvent avoir un impact non négligeable pour la reproduction et l'éventuelle correction du moindre bug. Autre exemple de la dualité de la problématique : les *repair bots*, dont les nombreux essais pour corriger un bug logiciel peuvent engendrer des consommations déraisonnables, sous couvert d'automatisation et d'économies humaines.

Enfin et surtout ; ce travail de fond ne peut pas être mené sans mettre les étudiant.e.s de nos écoles et universités dans la boucle, car ils sont à la fois des utilisateurs et les développeurs de demain. L'éco-conception logicielle doit être intégrée dans l'offre de formation dès maintenant.

2.5 Avoir une vision holistique incluant les acteurs humains

La complexité de la dimension énergétique dans le contexte logiciel (comparée aux autres critères non fonctionnels, tels que la fiabilité) vient aussi du rôle prépondérant de l'environnement d'exécution proche et lointain du logiciel. Les différentes couches logicielles doivent ainsi intégrer la dimension énergétique : au niveau intergiciel (middleware), lors de la programmation orientée composant, au niveau des conteneurs logiciels (tels que Docker, Kubernetes), ou pour les services logiciels. La montée en popularité du logiciel en tant que service (*Software-as-a-Service*, SaaS), tend à réduire le logiciel client à une interface d'un service logiciel complexe installé sur le cloud, avec l'implication d'une multitude de couches consommatrices d'énergie.

L'explosion ces dernières années des objets connectés amène aussi son lot de complexité : réseau, puissances faibles et distribuées, sécurité et vie privée, et surtout le retour de l'humain (et du vivant

en général) comme un acteur majeur des systèmes logiciels. Le défi est ainsi de sortir de l'approche technique ou architecturale « par silo », c.-à-d. optimiser des systèmes logiciels par couche, ou par domaine d'application, et d'étudier et optimiser le système dans son ensemble, y compris la boucle d'interaction avec l'humain.

Le logiciel éco-responsable doit donc être en mesure de s'autooptimiser, mais aussi d'optimiser son environnement (objets connectés, maison/industrie du futur), et de pousser aux changements comportementaux nécessaires à une transition énergétique et écologique plus large.

2.6 Créer un nouveau paradigme du logiciel durable

Et si le numérique actuel était une voie sans issue ? Cette dernière piste de recherche, la plus disruptive il faut bien le dire, consiste à imaginer un futur en faisant table rase du passé, dans lequel les logiciels seraient durables, centrés sur des objectifs de frugalité et résilience. Il ne s'agit pas d'améliorer l'existant, mais bien de repenser totalement une infrastructure numérique utilisant au mieux les ressources matérielles existantes afin d'atteindre la frugalité et la résilience. Cela revient donc à appliquer au logiciel, et pour les propriétés de frugalité et de résilience, une approche qui a déjà été appliquée au matériel pour la propriété de prédictibilité temporelle dans le projet CompSOC [CompSOC].

Le but est aussi de repenser les solutions techniques dans une réflexion plus large sur le rôle du numérique dans nos systèmes socio-techniques et sur les usages et les besoins. Cette piste s'inscrit dans les démarches *low-tech* plus ou moins globales et de déconnexion voire de dénumérisation (*Collapse Informatics* [Tomlinson et al. 2013], *CollapseOS* [CollapseOS]). Elle peut paraître idéaliste et éloignée des contraintes économiques, mais reste cependant scientifiquement intéressante ne serait-ce que pour permettre de mesurer la distance qui existe entre la complexité de certains de nos systèmes actuels et des solutions en quelque sorte minimales, libérées du poids de la compatibilité ascendante et de l'historique.

Une piste possible dans ce thème serait d'analyser les systèmes actuels pour comprendre quelle dégradation de leurs fonctionnalités serait observée s'ils n'étaient pas alimentés en énergie et connectés 24h sur 24 et 7 jours sur 7

3 La recherche en GL éco-responsable : exemples de projets

La recherche en génie logiciel éco-responsable est une réalité, en témoigne ces quelques projets en cours :

- La société GreenSpector fournit un outil de mesure de l'énergie pour mobile (Android et iOS). Elle a collaboré avec l'Université de Lille et Inria, sur des projets et études tels que Web Energy Archive du Green Code Lab ;
- L'Université de Lille et Inria ont développé des modèles et outils de mesure de l'énergie logicielle dont notamment PowerAPI [Colmant et al., 2018]. Ces outils sont en passe d'être adoptés par la communauté industrielle, comme Orange ou Davidson consulting ;
- Le projet Creedengo, né des efforts conjoints de la société Snapp' et de l'Université de Pau et des Pays de l'Adour concerne la création d'un pool de règles orientées énergie dédiées aux applications mobiles et implantées dans un « Linter Vert » [Olivier Le Goaer, 2019] ;
- L'appel à projets PERFECTO 2018 de l'ADEME a financé 5 projets en éco-conception logicielle, ciblant essentiellement le Cloud (Data Centers) et l'IoT. Des équipes de recherche de Mines ParisTech et de l'Université de Nantes font partie des lauréats.

— Le projet ANR HELP visait à fournir aux développeurs d'applications des simulateurs de plates-formes matérielles capables de mettre en évidence l'impact du logiciel sur les états de consommation du matériel [HELP16].

4 Bibliographie

1. [CNRS, 2018] Numérique : le grand gâchis énergétique, Le Journal du CNRS, 2018, https://lejournal.cnrs.fr/articles/numerique-le-grand-gachis-energetique
2. [Menaud et al. 2010] Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé Pierre Cointe, Rémi Douence et Mario Südholt (Ecole des Mines de Nantes / INRIA / LINA) Vers une réification de l'énergie dans le domaine du logiciel : L'énergie comme ressource de première classe, Session défis, Journées du GDR GPL, Pau 2010.
3. [Romain Rouvoy, 2019] Quels défis pour le développement durable des logiciels ? Conférence invitée, Journées GDR-GPL, Juin 2019
4. [Green IT, 2019] Empreinte environnementale du numérique mondial, https://www.greenit.fr/etude-empreinte-environnementale-du-numerique-mondial/
5. [Manotas et al., 2016] An Empirical Study of Practitioners' Perspectives on Green Software Engineering, 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), Austin, TX, 2016, pp. 237-248.
6. [Olivier Le Goaer, 2019] Linter vert pour l'écoconception logicielle, Green Days, Juin 2019
7. [Colmant et al., 2018] Maxime Colmant, Romain Rouvoy, Mascha Kurpicz, Anita Sobe, Pascal Felber, Lionel Seinturier : The next 700 CPU power models. Journal of Systems and Software 144 : 382-396 (2018).
8. [Perfecto, 2018] Lauréats de l'appel à projets de recherche ADEME PERFECTO 2018. https://presse.ademe.fr/wp-content/uploads/2018/10/Laureats-AAP-Recherche-PERFECTO pdf
9. [CompSOC] http://compsoc.eu/
10. [Tomlinson et.al, 2013] Bill Tomlinson, Eli Blevis, Bonnie Nardi, Donal Patterson, M. Six Silberman, Yue Pan : Collapse Informatics and Practice : Theory, Method. ACM Transactions on Computer-Human Interaction (2013).
11. [CollapseOS] https://collapseos.org/
12. [HELP16] Modeling Power Consumption and Temperature in TLM Models – Matthieu Moy, Claude Helmstetter, Tayeb Bouhadiba, Florence Maraninchi https://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v003-i001-a003

Gestion de la co-évolution des logiciels partiellement générés pendant la phase d'évolution et de maintenance

Djamel E. Khelladi, Olivier Barais, Benoît Combemale, Mathieu Acher,
Arnaud Blouin, Johann Bourcier, Noel Plouzeau, Jean-Marc Jézéquel

Univ Rennes, INRIA, IRISA Laboratory CNRS, Univ Rennes, DiverSE Team, IRISA Laboratory
{firstname.lastname}@irisa.fr

1 Introduction et contexte

Une des visions poussées par une partie du GDR il y a maintenant 20 ans est une réalité : les utilisateurs définissent leurs propres abstractions au travers de langages dédiés qu'ils outillent (exemple, des générateurs de code) dans le but d'augmenter leur productivité. Cette vision prend notamment forme via les applications cloud native [4] construites à l'aide d'un ensemble de micro-services et mises en oeuvre à l'aide de différents langages de programmation. Les développeurs utilisent de plus en plus de générateurs de code pour initialiser de telles applications complexes à partir d'abstractions définies à haut niveau (abstraction réifiée dans des DSL, dans des wizards, ...). Le débat n'est pas pour un développeur de savoir avec quel langage ou quel framework il en utilisera plusieurs en fonction des contraintes des plate-formes d'exécution, ses compétences, etc.

Cette tendance liée à l'utilisation de générateurs ou template de code est clairement mise en avant dans les lessons apprises mis en avant par [2, 5]. Et c'est même explicitement mis en avant par Balalaie et al. [1] dans la citation suivante :

"creating service development templates is important. Polyglot persistence and the use of different programming languages are promises of microservices. Nevertheless, in practice, a radical interpretation of these promises could result in chaos in the system and even make it unmaintainable. As a solution, after architectural refactoring began, we started to create service development templates. We have different templates for creating microservices in Java using different data stores ; these templates include a simple sample of a correct implementation. We're also creating templates for Node.js. One simple rule is that a senior developer should first examine each new template to identify potential challenges."

On peut citer par exemple, les générateurs de code : définis sous la forme d'un archetype Maven ; mis en oeuvre dans la plupart des outils en ligne de commande associés à un framework technique particulier tel que AngularCLI et WordPress tool ; associés à un ensemble de langages dédiés (e.g., modèles JDL) qui permettent de générer les souches et les squelettes liés aux protocoles de communications distribués que l'on trouve entre ces services (générateur openapi, asyncAPI, etc.). Une des différences notables que l'on peut faire entre le monde des générateurs de code sur ce type d'application et le monde des compilateurs et qu'il est attendu que le développeur modifie le code généré afin de raffiner la mise en oeuvre de son microservice.

Cet état de la pratique montre clairement son intérêt afin de laisser les différents membres d'un projet travailler avec le bon niveau d'abstraction tout en leur permettant de raffiner la mise en oeuvre dans des langages de programmation efficace, bien outillé. Cependant, cet état de la pratique lève aussi un challenge pour la communauté SLE (Software Language Engineering), la communauté du versionning de code et la communauté de l'évolution et de la maintenance de code au sens large. Comment faire co-évoluer automatiquement du code généré et du code produit manuallement ? Comment co-évoluer les données produites et stockées dans les bases de données ? Qu'en est-il de la documentation ? etc.

La complexité de cette démarche est telle que dans la plupart des projets industriels, on observe soit une approche de type "generated once" visant à utiliser les générateurs une seule fois. Les évolutions futures ne sont alors plus capturées aux bons niveaux d'abstractions. Or, penser et raisonner l'évolution à un haut niveau d'abstraction permet entre autres de bien la spécifier et de garder traces des décisions et de leurs implications dans le système. Soit avec des approches du type compilateur pour lequel les utilisateurs essaient de ne jamais modifier le code généré tant qu'il y a un risque d'évolution à décrire au niveau des modèles de haut niveau génant fortement le besoin d'agilité dans un projet de développement moderne. En pratique, ce scénario est peu fréquent au vu des besoins d'évolution et de maintenance pour rester compétitif sur le marché.

Un cas d'étude intéressant pour comprendre ce phénomène est JHipster [6]. Projet opensource dynamique visant à faciliter la mise en oeuvre d'un service ou d'un micro-service en laissant un certain nombre de points de variation. Pour montrer cette pratique, nous pouvons regarder quelques chiffres liées à ce projet.

— Ils ont atteint 15 000 étoiles (de popularité) le mois dernier.
— Le nombre d'installations continue de croître au même rythme que les années précédentes.
— Ils ont maintenant plus de 120 000 installations par mois.
— Les téléchargements de nos artefacts Maven suivent la même tendance. Le mois dernier, ils ont atteint 272 000 téléchargements sur Maven Central.
— Ils ont eu des utilisateurs du monde entier. En fait, seulement trois pays n'ont pas utilisé JHipster !

JHipster est intéressant, car il cumule différents types de générateur de code. Un générateur de code pour initialiser le projet, un générateur construit par dessus un langage dédié de modélisation des entités métiers (.JDL), l'utilisation de générateur de code issu d'Angular CLI, l'utilisation de générateur de code (talon/squelette) pour mettre en oeuvre ou être client d'un service dont l'interface est définie à l'aide d'OpenAPI. Enfin JHipster génère du code vers un ensemble de plateformes, (code SQL ou liquibase pour la création et l'évolution du stockage des données), code Java pour la partie serveur, code TypeScript ou JavaScript pour la partie cliente, descripteur de projet (pom.xml, package.json), de déploiement. (yml).

Si l'on peut voir ce type de projet comme une vitrine pour vanter l'utilisation d'abstraction et d'informatique générative intégrée à un écosystème technique à l'état de l'art, il reste plusieurs points noirs associés à l'utilisation de JHipster. La principale concerne ce que l'on nomme la co-évolution abstractions mise en oeuvre en présence du code généré modifié. Ce problème n'est pas pleinement nouveau et la communauté IDM (Ingénierie Dirigée par les Modèles) [7, 3] l'a relevé depuis longtemps, pour autant ce problème n'a pas encore trouvé de réponse cohérente pleinement utilisée d'un point de vue industriel. Ceci s'explique probablement en partie, car ce problème touche de nombreuses communautés :

— Sans aucun doute, la communauté IDM et SLE doivent sans doute réfléchir à de nouveaux moyens de structurer ses générateurs de code.
— la communauté langage afin de mieux mettre en évidence dans les langages de programmation les éléments générés et les éléments manuellement décrits.
— La communauté liée au software repository afin de fournir des outils de gestion de source conscient de ses différences entre artefacts générés et artefacts modifiés.
— La construction des IDEs afin de nouveau de mieux s'adapter à ces méthodes de développement polyglots au sein desquels de nombreux générateurs de code sont utilisés.
— La communautés SPL qui permet très généralement de définir l'orchestration et la paramétrisation de ces générateurs de code.

Si ce problème de co-évolution est complexe, nous disposons maintenant d'une base de connaissances riches (repository de code, de dépendances, ...), nous disposons d'une capacité de calcul plus importante permettant d'évaluer plusieurs solutions de co-évolution automatique afin d'en proposer aux développeurs, nous disposons d'IDE plus riches. Nous proposons sur ce défi de réfléchir à l'automatisation de cette tâche de co-évolution en utilisant au mieux ses bases de connaissances disponibles, la puissance de calcul disponible, de nouvelles constructions de langages, etc. Par exemple, la construction d'un langage d'annotation pour le code qui doit être co-évolué ou même des annotations de co-évolution spécifiques. Par ailleurs, les lignes de produits pourraient aussi être utiles pour remonter des évolutions similaires faites sur des codes de produits dérivés d'un même "feature model".

Références

[1] Balalaie, A., Heydarnoori, A., Jamshidi Dermani, P. : Microservices architecture enables devops : An experience report on migration to a cloud-native architecture

[2] Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S. : Empirical assessment of mde in industry. In : Proceedings of the 33rd International Conference on Software Engineering. pp. 471–480. ACM (2011)

[3] Kleppe, A.G., Warmer, J., Warmer, J.B., Bast, W. : MDA explained : the model driven architecture : practice and promise. Addison-Wesley Professional (2003)

[4] Kratzke, N., Quint, P.C. : Understanding cloud-native applications after 10 years of cloud computing-a systematic mapping study. Journal of Systems and Software **126**, 1–16 (2017)

[5] Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J. : Assessing the state-of-practice of model-based engineering in the embedded systems domain. In : Model-Driven Engineering Languages and Systems, pp. 166–182. Springer (2014)

[6] Raible, M. : The JHipster mini-book. Lulu. com (2016)

[7] Schmidt, D.C. : Model-driven engineering. COMPUTER-IEEE COMPUTER SOCIETY- **39**(2), 25 (2006)

Chaîne de compilation formellement certifiée

David Monniaux

Résumé

Les compilateurs sont des logiciels complexes, dont les bugs peuvent introduire des bugs dans le code compilé. Il existe déjà des compilateurs formellement prouvés corrects. Le défi consiste à les améliorer (ajout d'optimisations) et à étendre le champ d'une part vers des langages de plus haut niveau, d'autre part vers la synthèse de mécanismes de sécurité ou vers les architectures parallèles.

1 État de l'art

De nos jours, on rédige rarement des logiciels en langage machine ou d'assemblage, et la plus grande partie des logiciels sont rédigés dans des langages de plus au moins haut niveau, qui sont ensuite compilés ou interprétés. Les compilateurs sont eux-mêmes des logiciels complexes, qui peuvent donc contenir des bugs. Certains de ces bugs se traduisent par un arrêt intempestif de la compilation, mais, dans certains cas, la compilation n'échoue pas et du code incorrect est émis. Comme des changements minimes du code source, d'options de compilation etc. peuvent induire des comportements différents du compilateur, ces bugs peuvent très bien ne pas se manifester dans des versions de test ou de déboguage, mais se manifester dans la version de production. Ces bugs de compilation sont également très difficiles à distinguer de bugs du code source induisant des comportements indéfinis au sens du standard du langage et produisant des comportements aberrants dans le code objet. Ils sont donc très difficiles à détecter et à isoler.

Pour ces raisons, les étapes de compilation sont difficiles à gérer dans l'optique de la qualification du code objet pour des applications critiques (conduite d'aéronefs, de véhicules, de processus nucléaires ou chimiques…). On peut utiliser des arguments tels que «ce compilateur a été utilisé pendant N années dans des applications critiques sans que cela n'ait posé de problème», ou encore procéder à des comparaisons et relectures entre code source et code objet. De telles comparaisons ne sont souvent possibles qu'en désactivant les optimisations, ce qui est restrictif voire intolérable pour de nombreuses applications. [1]

Le projet CompCert [2] a apporté une première réponse à ce problème. Il s'agit d'un compilateur pour le langage C vers différentes architectures (x86, PowerPC, ARM, Risc-V), avec la particularité unique que tant le langage source que le langage cible, mais aussi tous les langages intermédiaires, bénéficient d'une sémantique formelle, [3] et que chaque phase de compilation vient avec une preuve qu'elles préserve la sémantique du programme. Ces preuves sont vérifiées à l'aide de l'assistant Coq.

1. Dans des applications critiques, il n'est souvent pas possible de mettre un processeur plus rapide : ces processeurs peuvent être moins résistants aux conditions de température ou de rayonnement rencontrés, dissiper trop de chaleur, ou encore être trop complexes pour que l'on puisse être confiants dans leur bon fonctionnement.

2. `http://compcert.inria.fr/`

3. En réalité, le premier langage formellement spécifié n'est pas exactement C et le dernier pas exactement l'assembleur, de sorte qu'il existe de petites phases dont la correction n'est pas prouvée. Ceci est sans commune mesure avec l'absence globale de preuve dans les compilateurs classiques.

Bien que CompCert soit une *success story* de la vérification formelle, avec transfert industriel, ce n'est qu'un début vers une chaîne de compilation formellement certifiée. [4]

2 Optimisation avancée

CompCert n'optimise que modérément le code qu'il génère, qui a donc en général des performances inférieures à celles du code produit par des compilateurs tels que GCC, LLVM ou ICC dans leurs modes supérieurs d'optimisation. Pour certaines applications critiques, notamment en avionique, cette optimisation modérée est déjà un plus, car les compilateurs classiques sont utilisés en désactivant leurs optimisations. Toutefois, dans d'autres applications, par exemple dans le ferroviaire, des performances trop inférieures à celles des compilateurs de référence sont rédhibitoires.

Un premier défi est donc de rajouter dans CompCert des optimisations plus ambitieuses que celle déjà implantées, par exemple : [5]

— déplacement de code invariant en dehors des boucles
— décomposition de variables structures
— réordonnancement d'instructions, spéculation logicielle, notamment pour des cibles à exécution dans l'ordre (comme souvent pour les processeurs destinés à l'embarqué critique, notamment si le temps d'exécution doit être très prédictible)
— réorganisation de boucles, tuilage, pipeline logiciel…
— réorganisation de nids de boucles.

Si ces optimisations sont classiques, les mettre en œuvre en ayant une preuve formelle de correction de leur fonctionnement tout en préservant une complexité algorithmique acceptable est une gageure. Il faut une réflexion très fine sur les éventuelles représentations intermédiaires supplémentaires à introduire, les invariants à maintenir, les propriétés à prouver, etc. afin d'arriver à mener à bien les preuves formelles, voire, idéalement, à les rendre robustes à d'éventuels changements dans l'optimisation.

Des travaux ont été menés à l'IRISA notamment sur les représentations intermédiaires SSA, et à Verimag sur le réordonnancement d'instructions en phase finale de compilation, notamment pour processeurs VLIW. Il reste encore beaucoup à faire.

3 Langages de haut niveau

Les systèmes de contrôle critiques sont souvent spécifiés dans des langages de haut niveau adaptés, par exemple SCADE. Ces langages sont ensuite compilés vers un langage de plus bas niveau, souvent C. Les compilateurs habituels pour ces langages ne sont pas formellement certifiés et posent donc des problèmes similaires à ceux des compilateurs C : que se passe-t-il en cas de bug, comment justifier de leur absence. Des travaux ont été menés au LIENS sur le compilateur VÉLUS depuis un sous-ensemble de Scade vers une des représentations intermédiaires de CompCert, mais il ne s'agit que d'une première étape. [6]

L'exécution de modèles issus de l'apprentissage profond passe souvent par l'usage de descriptions de haut niveau (TENSORFLOW…) ensuite implantées pour exécution efficace en machine. Là

4. Collaborations possibles : Verimag, Xavier Leroy, IRISA Celtique…

5. Collaborations possibles : ICPS / INRIA Camus, INRIA Parkas, INRIA PACAP, LIP/INRIA Cash, INRIA Corse…Industriels : Kalray, ST.

6. Équipes/laboratoires potentiellement concernées : Verimag Synchrone, INRIA Parkas, ONERA. Collaborations industrielles possibles : Airbus avionics & simulation products, Safran, Hewlett Packard Enterprise, Tweag.io

encore, le processus de compilation, mettant éventuellement en œuvre des réorganisations complexes de boucles (e.g. décomposition par blocs), n'est pas certifié. Dans certains cas, un code produit incorrect peut non seulement mettre en œuvre incorrectement le modèle issu de l'apprentissage, mais aussi corrompre les données d'éventuels logiciels plus classiques dans le même espace mémoire. Là encore, il est souhaitable que les étapes de compilation de haut niveau soient formellement certifiées.

4 Intégration de mécanismes de sûreté et de sécurité à l'exécution

Un processeur peut parfois calculer incorrectement en environnement difficile (température, rayonnements...) ou en cas d'attaques matérielles (dérangement intentionnel de l'alimentation électrique, impulsions électromagnétiques ou lumineuses...). Il importe donc de s'en rendre compte voire de compenser ces dysfonctionnements. Si des mécanismes matériels (mémoire à code de correction d'erreurs) peuvent aider, ils ne suffisent pas dans bon nombre de cas (les unités d'exécution ainsi que certains niveaux de caches ne sont typiquement pas dotés de tels codes). Par ailleurs, les logiciels subissent des attaques logicielles, profitant notamment des comportements indéfinis des langages bas niveau (débordements de tampons etc.).

L'ajout naïf de calculs ou de tests redondants dans le code source peut être inefficace (un compilateur peut détecter que l'on calcule deux fois la même chose et éliminer la redondance !), et peut être source d'erreurs. Il serait souhaitable que ces ajouts soient automatiques, au moment de la compilation. Des travaux sont en cours à Verimag sur ce sujet. [7]

Dans le cadre d'une chaîne de compilation certifiée, il est bien entendu nécessaire de démontrer que ces ajouts préservent le fonctionnement normal du programme. Toutefois, on peut être plus ambitieux et vouloir démontrer qu'ils préservent le programme de certaines attaques. Ceci suppose d'avoir un *modèle d'attaquant* intégré dans une sémantique non standard de l'exécution. [8]

5 Concurrence

Actuellement, CompCert n'a pas de notion d'accès mémoire concurrents. [9] Il serait donc impossible de démontrer, dans sa sémantique, la correction d'un programme faisant des accès concurrents à une même variable depuis plusieurs *threads*. A fortiori, il serait impossible de démontrer la correction d'une phase de compilation transformant une boucle séquentielle dont tous les calculs seraient indépendants en une boucle parallèle, comme peut le faire OpenMP. Pourtant, ce type de transformation est très utile pour la parallélisation automatique, que ce soit pour le calcul scientifique ou pour l'exécution de modèles issus de l'apprentissage profond.

Une difficulté est que l'action des accès sur une mémoire partagée réelle n'est pas un entrelacement des accès des différents cœurs de calcul, comme c'est le cas dans les modèles simples de la concurrence. Les architectures réelles mettent en œuvre des modèles de cohérence faible, qui plus est différents suivants les architectures (x86 vs ARM...) et plutôt mal documentés. Des travaux ont déjà été menés pour l'intégration d'une telle sémantique faible de la mémoire dans CompCert (CompCertTSO), mais n'ont pas été poursuivis.

7. Collaborations possibles : CEA LIST, ST Microelectronics...

8. Collaboration avec le GdR sécurité.

9. Si ce n'est via des variables «volatiles», que la sémantique considère en quelque sorte comme des ports d'entrée/sortie vers l'environnement. Il peut également exister des *builtins* de lectures ou écritures atomiques en mémoire.

6 Calcul distribué

Le développement d'applications parallèles en mémoire non partagée passe communément par l'usage de bibliothèques d'envoi de messages (par exemple MPI). Cela est malaisé et il est facile d'introduire des *bugs*, parfois correspondant à des entrelacements rares d'exécutions, durs à reproduire.

Si l'on compile depuis un langage de haut niveau, par exemple un langage synchrone, il peut être possible de délimiter des tâches devant s'exécuter sur des processeurs différents et s'échangeant les données, et synthétiser le code de communication. Un tel code de communication correct par construction serait plus fiable qu'un code émis à la main.

La synthèse vérifiée de code distribué à partir d'une description de haut niveau est une perspective de plus long terme, très prometteuse, permettant d'exploiter les processeurs *many-core*, les GPU etc.

7 Implantation matérielle

Outre les CPU, le calcul haute performance cible souvent des GPU, voire des FPGA. Il se pose donc la question de la compilation vers ces architectures. [10] Ces préoccupations peuvent se prolonger à la correction des implantations matérielles, en collaboration avec des équipes hors GdR GPL : les processeurs modernes sont des systèmes très complexes, mettant eux-mêmes en œuvre des mécanismes de traduction de code.

8 Compilation juste à temps

De nombreux langages actuels ne sont pas compilés une fois pour toutes, mais sont partiellement interpretés, partiellement compilés juste à temps. Se pose alors la question de la certification de la compilation juste à temps. Il faut noter que celle-ci pose de délicats problèmes de sécurité, puisque :

— Les programmes proviennent parfois de l'extérieur potentiellement hostile (Javascript).

— Le compilateur juste à temps a le droit de créer du code exécutable, alors que les zones de donnée mémoire sont par défaut non exécutable afin de rendre plus difficile les attaques par injection de code.

La difficulté est ici de réaliser un système à la fois rapide, fortement intégré dans un environnement dynamique, et certifié. [11]

9 Défi

Parmi les items précédents, certains sont déjà implantés dans des compilateurs *mainstream*. La difficulté est de le faire d'une façon certifiée.

Les preuves papier des transformation de compilation sont souvent sur des langages jouet, avec des étapes omises ; parfois ces transformations et/ou ces preuves sont incorrectes. [12] Elles ne sont quasiment jamais formulées au niveau de détail et de rigueur nécessaires à une validation formelle, qui nécessite des sémantiques très précises, des relations de simulation, etc. Il y a là un grand travail de recherche de formalisation et de clarification, ainsi que d'ingénierie de la preuve : les preuves assistées sont aujourd'hui encore trop fastidieuses.

10. Collaborations possibles : INRIA Parkas, LIP Cash.
11. Collaborations possible : INRIA Celtique, qui a travaillé sur la formalisation de la sémantique de Javascript.
12. Par exemple, l'algorithme original de la sortie de la représentation SSA était incorrect.

Permettre la programmation sans bugs
Défi GDR GPL

Gabriel Scherer, INRIA Saclay

2 décembre 2019

Résumé

Aujourd'hui la majorité des langages utilisés en pratique pour écrire du logiciel ne permettent que de décrire l'*implémentation* du logiciel désiré. Pour réduire drastiquement le nombre de bugs logiciels, il faudrait permettre l'utilisation d'un langage dit *vérifiant*, c'est-à-dire un langage qui permet aussi d'écrire la *spécification* du logiciel et de prouver que l'implémentation la respecte – la preuve étant vérifiée par l'environnement de programmation.

Ce projet de recherche au long cours a déjà donné des succès remarquables, avec des compilateurs, des systèmes d'exploitation, des serveurs webs vérifiés ; mais ces succès restent des prototypes de recherche dont l'écriture est réservée à des super-experts.

Notre défi est de produire des langages vérifiants utilisables en pratique. Cela demande de nombreuses améliorations, en particulier de meilleurs langages vérifiants, de meilleurs outils de preuve/vérification, et des utilisateurs et utilisatrices mieux formé-e-s.

Introduction

Un *bug* logiciel, c'est un écart de comportement entre ce que fait le programme et ce qu'on aurait souhaité qu'il fasse. Ce n'est pas une erreur de l'ordinateur, c'est une erreur humaine : la machine fait toujours ce qu'on lui a demandé, mais on s'est trompé, on n'a pas demandé ce qu'on voulait vraiment.

De même qu'un système intégrant des humains et des machines contiendra toujours des failles de sécurité, il n'est pas possible de promettre l'absence totale de bugs. Le *défi* que nous proposons est de permettre aux auteurs de logiciel d'exprimer précisément *ce qu'ils veulent*, c'est-à-dire une *spécification* pour leur programme, et d'avoir des outils pour vérifier que ce qu'ils ont demandé, leur *implémentation*, respecte cette volonté exprimée.

On appellera *langage vérifiant* un langage de programmation qui permet d'écrire des spécifications en plus des implémentations, et de vérifier statiquement qu'elles sont respectées. Le défi est de produire des langages vérifiants utilisables en pratique. Même dans un monde idéal où ce défi est résolu, il restera toujours des fissures où des bugs peuvent se glisser, qu'il est important de reconnaître et de nuancer/relativiser.

— On ne connaît pas forcément la spécification du programme que l'on veut écrire. Quelle serait la spécification d'un économiseur d'écran, "afficher de jolies images" ? Mais même dans ce cas on peut donner une *spécification partielle*, qui parle de certain comportements du programme : l'économiseur d'écran ne doit pas s'arrêter avant une action de l'utilisateur ou la mise en veille de la machine. Par ailleurs, un logiciel dont nous n'avons pas de bonne spécification est la partie émergée d'un iceberg logiciel, reposant sur une grande quantité de logiciel système et de bibliothèques pour lesquelles on sait écrire des spécifications, que l'on voudrait vérifier.

— On peut encore se tromper en écrivant la spécification ! Cependant, dans la majorité des cas les spécifications sont des objets beaucoup plus simples et compacts que le code source ; elles sont plus faciles à relire et comprendre, exposent moins de surface aux bugs. (Une spécification qui se contente de répéter l'implémentation n'apporte rien, et se rapproche du cas où on ne sait pas spécifier.) On peut aussi compter sur des méthodes de tests de

spécifications, plus efficaces que les tests d'implémentation car elles opèrent sur un objet plus simple.

Avantages indirects

Écrire des spécifications ne sert pas qu'à éliminer des bugs. Nous prétendons que le fait d'encourager les programmeu-r-se-s à écrire des spécifications en même temps que leur implémentation augmente significativement la qualité logicielle.

1. Se demander systématiquement la spécification d'une nouvelle fonction ou méthode, se forcer à réfléchir plus en écrivant du code, aide à écrire le code correspondant — du Test-Driven-Development (TDD) amplifié.
2. Réfléchir à la spécification aide à concevoir l'interface (API) d'une bibliothèque. Dans tel cas particulier, quelle valeur par défaut devrait utiliser la fonction ? La valeur qui permet la spécification la plus simple.
3. Les spécifications constituent une forme précieuse de documentation. En particulier, si elles sont vérifiées automatiquement, on a la garantie qu'elles sont à jour — contrairement aux commentaires libres.
4. La présence de spécifications pourra aider grandement les outils de transformation automatique de programme (refactoring, génération de code à partir d'exemples, mise à jour automatisée, etc.) : on réduit l'espace des transformations possibles en se restreignant à celles qui valident la spécification.

Pour l'instant nous n'avons que relativement peu d'expériences concrètes sur l'impact des langages vérifiants en terme de génie logiciel, mais les trois premiers points peuvent être constatés (nous en avons fait personnellement l'expérience) en utilisant des outils de génération de tests aléatoires, en particulier le *test basé sur les propriétés*, qui encourage à écrire des spécifications partielles.

De meilleurs langages

Il existe aujourd'hui très peu de langages vérifiants, en tout cas parmi les langages de programmation généralistes. Comme outils et prototypes de recherche, on peut citer Why3 et Dafny, F* et Liquid Haskell, Coq et Idris. SPARK/Ada est peut-être le seul exemple utilisé en production dans l'industrie.

L'approche dominante est l'utilisation de systèmes de type et d'analyses statiques automatiques, qui vont dans cette direction mais donnent des spécifications très partielles ("renvoie un entier", "attend un tableau de taille N", "ne plante pas"). Les mécanismes de *programmation par contrat* méritent une mention honorable, car ils encouragent les programmeurs à écrire des spécifications. Mais ils sont vérifiés dynamiquement : ils n'apportent pas la confiance d'une vérification statique, et leur coût de calcul limite la richesse des contrats employés en pratique.

Pour relever ce défi, il faudra concevoir de meilleurs langages vérifiants, et ajouter des bons langages de spécification (et outils de vérification) aux langages existants — les propositions existantes comprennent JML pour Java, ACSL pour le C, le projet VOCAL pour OCaml. Un langage de spécification n'est pas la même chose qu'un langage de contrats (habituellement écrits dans le langage d'implémentation) ; même si on peut vouloir le tester dynamiquement, il doit pouvoir exprimer des propriétés non calculables, doit pouvoir être évalué statiquement, et demande souvent des mécanismes de réutilisation et modularité un peu différents du langage d'implémentation.

La conception d'un langage de spécification est aussi intimement liée à la conception du langage d'implémentation, ce ne sont pas deux objets totalement séparés. Par exemple, notre expérience collective en vérification de programme souligne l'importance de la notion d'*état fantôme*, qui consiste à entrelacer dans les instructions du programme des calculs "fantômes" qui ne sont pas effectués dynamiquement, servent uniquement à la vérification, en maintenant une relation précise entre l'évolution des données (réelles) du programme et la spécification de haut niveau. Par exemple, à un tableau représentant une queue FIFO on pourra associer un ensemble "fantôme" de ses

éléments, maintenue en parallèle et utilisée dans les spécifications. Cette construction est spécifique aux langages vérifiants.

En plus de questions de design et d'utilisabilité, cette question touche des sujets théoriques de *logique* : quelle est la bonne logique pour formuler des énoncés mathématiques concernants des programmes contenant des exceptions, des effets de bord, de la concurrence par passage de messages ou mémoire partagée, des systèmes distribués ? Le sujet est très ancien, mais par exemple, la *logique de séparation*, apparue au début du siècle et toujours un sujet de recherche actif, a révolutionné la façon de parler de l'état modifiable dans des spécifications.

De meilleurs outils de preuve

Un langage vérifiant demande des outils de preuve, qui peuvent vérifier que les implémentations respectent leur spécification. Le problème de construire de bons vérificateurs est intimement lié aux choix du langage de spécification et du langage d'implémentation, mais il touche aussi à des problèmes fondamentaux de recherche de preuve, démonstration automatique et assistants de preuve.

Malgré les progrès rapides et constants de outils de preuve automatique, l'expérience de notre communauté est qu'espérer une preuve totalement automatisée n'est pas une bonne approche : dans les cas délicats les outils automatiques deviennent lents et fragiles (la vérification ne passe plus après des changement mineurs du programme, réussit ou échoue selon les choix non-déterministes de l'outil de preuve).

En plus de pousser pour une meilleure vérification automatique des obligations de preuve issues de la vérification de programmes, tous les systèmes existants travaillent donc sur des façons pour l'utilisateur de guider la preuve que l'implémentation respecte la spécification. De nombreuses pistes sont explorées ; par exemple, faut-il essayer de pousser le guidage de la vérification au maximum comme une forme de programmation (fonctions-lemmes, état fantôme), ou au contraire utiliser un langage de preuve généraliste, permettant de travailler sur n'importe quel énoncé mathématique ?

Les outils de vérification posent aussi de nouvelles questions de conception logicielle. Quelle est la meilleure façon de faire évoluer une spécification en maintenant à jour les arguments de preuve ? Quels sont les approches de preuve qui donnent les arguments les plus robustes aux évolutions du programme ?

De meilleur-e-s programmeu-r-se-s

Indépendamment de tous les efforts passés et à venir de notre communauté pour rendre les langages *vérifiants* plus faciles à utiliser, l'écriture de spécification reste un art différent de l'écriture d'implémentation, qui appelle une formation spécifique.

Pour relever ce défi, tous les cursus de master d'informatique devraient contenir un cours obligatoire de vérification de programmes, enseignant l'usage d'un langage vérifiant.

Remerciements Nous remercions Arthur Charguéraud et Jean-Christophe Filliâtre pour leurs commentaires sur ce document.