Actes des journées Approches Formelles dans l'Assistance au Developpement du Logiciel (AFADL 2025)

Fabien Dagnat, Olga Kouchnarenko

Fabien DAGNAT, Olga KOUCHNARENKO (Éditeurs / *Editors*)

Actes des Journées / *Proceedings*

Approches formelles dans l'assistance au développement du logiciel (AFADL)

https://gdrgpl2025.sciencesconf.org/resource/page/id/1

tenues conjointement avec les Journées du
Groupement de Recherche national - Génie de la Programmation et du Logiciel
GDR-GPL 2025 à Pau (https://gdrgpl2025.sciencesconf.org)

16-18 juin 2025 / June 16-18, 2025
Pau, France

Avec la collaboration des groupes de travail /
With the collaboration of the working groups :

Langages et vérification de programmes (GT LVP)
Méthodes de test pour la validation et la vérification (GT MTV2)

Comité de programme / Programme Committee

Idir Ait Sadoune	LMF, CentraleSupélec, Université de Paris-Saclay, France
Moussa Amarani	Université de Namur, Belgique
Olivier Barais	IRISA, Université de Rennes, France
Clara Bertolissi	LIS, Université d'Aix-Marseille, France
Marius Bozga	VERIMAG, CNRS, France
Jérémy Buisson	École de l'air et de l'espace, France
David Chemouil	DTIS, ONERA et Université de Toulouse, France
Horatiu Cirstea	LORIA, Université de Lorraine, France
Frédéric Dabrowski	LIFO, Université d'Orléans, France
Fabien Dagnat	Lab-STICC, IMT Atlantique, France
Benoit Delahaye	LS2N, Nantes Université, France
Lydie Du Bousquet	LIG, Université Grenoble Alpes, France
Sophie Ebersold	IRIT, Université de Toulouse, France
Mohammed Foughali	IRIF, Université de Paris, France
Marc Frappier	Université de Sherbrooke, Canada
Laure Gonnord	LCIS, Grenoble INP - UGA, France
Pierre-Cyrille Héam	FEMTO-ST, Université Marie et Louis Pasteur, France
Ludovic Henrio	CNRS, LIP, France
Aurélie Hurault	IRIT, Toulouse INP, France
Nikolai Kosmatov	Thales Research and Technology, cortAIx Labs , France
Olga Kouchnarenko	FEMTO-ST, Université Marie et Louis Pasteur, France
Natalia Kushik	SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, France
Régine Laleau	LACL, Université Paris-Est Créteil, France
Thierry Lecomte	CLEARSY, France
Delphine Longuet	Thales Research and Technology, cortAIx Labs, France
Frédéric Mallet	I3S, Université Côte d'Azur, France
Célia Picard	ENAC, Université de Toulouse, France
Clément Quinton	CRIStAL,Université de Lille, France
Antoine Rollet	LaBRI, Bordeaux INP, France
Julien Signoles	Université Paris-Saclay, CEA, List, France
Nicolas Stouls	CITI, INSA Lyon, France
Ciprian Teodorov	Lab-STICC, ENSTA, France
Benoît Valiron	LMF, CentraleSupélec, Université de Paris-Saclay, France
Laurent Voisin	Systerel, France

Table des matières

Préface

Ces journées à Pau poursuivent les rencontres annuelles de la communauté AFADL et plus largement des groupes de recherche connexes du GDR GPL. Les journées AFADL ont pour objectif de rassembler des acteurs académiques et industriels intéressés par la mise en œuvre des techniques formelles à divers stades du développement des logiciels et/ou des systèmes. Elles visent ainsi la mise en valeur de travaux récents effectués autour de thèmes comme :

- Les techniques et outils formels contribuant à assurer un bon niveau de confiance dans la construction de logiciels et de systèmes ;
- Les méthodes et processus permettant d'exploiter efficacement les techniques et outils formels disponibles ou conçus ;
- Les méthodes et processus mettant en œuvre différentes techniques formelles et hétérogènes dans un développement ;
- Les leçons tirées de la mise en œuvre de ces outils ou principes sur des études de cas ou des applications industrielles.

Les techniques, méthodes et outils présentés assistent notamment les activités de la modélisation, la validation et la gestion d'exigences formelles applicables aux logiciels ; les spécialisations ou extensions de techniques de modélisation et d'évaluation induites par des domaines applicatifs (télécommunication, contrôle-commande, robotiques, systèmes interactifs, architectures, composition de services, applications distribuées sur le web, systèmes cyber-physiques, etc) ; des points de vue particuliers sur les systèmes (sécurité informatique, exécution temps réel, ...).

Sont abordés aussi dans le cadre de ces journées, le passage d'une étape de conception à la suivante : patrons de raffinement de spécifications, déploiement d'une architecture logicielle sur une architecture matérielle, génération automatique de code, réutilisation de composants, ... ; test et l'évaluation rigoureuse de modèles formels ou de codes ; spécification et vérification formelles d'architectures, de modèles, de programmes ou de systèmes.

Les travaux présentés s'intéressent aussi à la combinaison d'approches formelles avec des approches informelles ou semi-formelles ou à la coopération de techniques formelles de développement avec des techniques plus classiques (par exemple à la complémentarité vérification formelle / test pour les aspects V&V), aussi bien qu'à l'adaptation des approches formelles aux techniques d'apprentissage automatique et au domaine de l'informatique quantique.

Pour cette édition de 2025, le programme de AFADL a intégré des travaux spécifiquement liés aux groupes de travail : Langages et vérification de programmes (LVP) et Méthodes de test pour la validation et la vérification (MTV2) du GDR GPL lors de deux sessions communes.

Les actes sont structurés suivant les cinq sessions du programme des journées AFADL 2025 pour faciliter la consultation. Les articles présentés et listés dans les présents actes relèvent de différentes catégories, suscitées lors de l'appel à communications.

- Deux articles présentant de nouveaux travaux ou résultats de travaux académiques ou industriels non encore publiés : les articles de Vinarskii et al (page 75) et Costa et al (page 83).
- Les travaux et résultats de trois doctorantes et doctorants encadrés par les chercheuses et chercheurs de la communauté. C'est l'occasion pour les auteurs de bénéficier davantage de retours de la communauté, au delà des conseils de leur proche encadrement. Les articles suivants sont dans cette catégorie : celui de Alexandre Jean (page 16), celui d'Eléa Jacquin (page 21) et celui d'Anna Gallone (page 49).
- Des contributions présentant des projets nationaux ou internationaux : les projets VéDySec (page 91), VVaMIA (page 95) et VASSAL (page 69).
- Des résumés d'articles récemment publiés dans des conférences internationales afin de les faire connaître

à notre communauté. Ces articles sont publiés dans les conférences suivantes : Tests and Proofs (page 5), Verification, Model Checking, and Abstract Interpretation (page 10), Fundamental Approaches to Software Engineering (page 39), Symposium on Applied Computing (pages 42 et 46), Performance Evaluation Methodologies and Tools (page 55), Object-Oriented Programming Systems, Languages, and Applications (page 60), International Conference on Formal Engineering Methods (page 65).

Il y a également trois posters spécifiques à AFADL qui seront présentés. Ces actes contiennent un résumé ainsi que le poster réduit en pages 27, 30 et 33.

Ces dix-neuf contributions proviennent de quinze établissements et laboratoires de recherche publiques de toute la France, 3 universités étrangères et quatre centres de R&D d'entreprises.

Nous remercions ici nos collègues qui ont permis et assuré le bon déroulement de ces journées : Olivier Le Goaer, Adel Noureddine et tous les collègues de Pau ; Mireille Blay-Fornarino et Catherine Dubois Co-directrices du GDR Génie de la Programmation et du Logiciel ; les responsables des groupes de travail MTV2 (Nikolai Kosmatov, Natalia Kushik, Pascale Le Gall, Antoine Rollet) et LVP (Nicolas Magaud, Julien Signoles) ; le responsable de la session poster des journées du GDR GPL Thomas Degueule.

Fabien DAGNAT, Olga KOUCHNARENKO
le 10 juin 2025

Session AFADL I

Langage de spécification de contexte pour vérifier formellement des propriétés de consentement sur des modèles et du code

– Résumé long –

Myriam Clouet[1] Thibaud Antignac*[2] Mathilde Arnaud[3]
Julien Signoles[3]

[1] Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022
firstname.lastname@univ-orleans.fr

[2] CNIL (Commission nationale de l'informatique et des libertés),
3 place de Fontenoy, TSA 80715, 75334 Paris CEDEX 07, France
tantignac@cnil.fr

[3] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
firstname.lastname@cea.fr

Résumé

Cette soumission présente l'article *Context Specification Language for Formally Verifying Consent Properties on Models and Code* qui a été publié à la conférence *Tests and Proofs* en 2023 [5]. Il porte sur la définition d'un langage formel, la formalisation de propriétés liées au consentement et à l'implémentation d'un outil de vérification formelle. Cet outil peut être utilisé soit pour vérifier une modélisation d'un système, soit pour vérifier le code correspondant à son implémentation.

1 Contexte

Plusieurs domaines d'applications traitent des données personnelles, comme les sites de commerce en ligne, les assistants vocaux ou les systèmes de santé. Des lois et des régulations ont été établies autour du monde pour encadrer ces traitements, notamment le RGDP en Union Européenne. Ne pas les respecter peut mener à des amendes conséquentes, ce qui a été le cas pour Google [1] et WhatsApp [2]. Il est

*Les vues, opinions et positions exprimées dans cet article sont celles de l'auteur et non de l'institution à laquelle il appartient. Ce travail a été réalisé en grande partie pendant que l'auteur était au CEA List.

1. https://www.lemondeinformatique.fr/actualites/lire-rgpd-google-condamne-a-50-meteuro-par-la-cnil-74062.html

2. https://www.lemondeinformatique.fr/actualites/lire-rgpd-whatsapp-condamne-a-225-meteuro-d-amende-84044.html

donc crucial de vérifier qu'un système respecte des propriétés de vie privée.

Les méthodes formelles fournissent un ensemble de techniques fondées sur la logique, les mathématiques et l'informatique théorique. Elles sont utilisées pour spécifier, développer et vérifier des systèmes informatiques [7]. En particulier, elles peuvent être utiliser pour vérifier des propriétés de respect de la vie privée [8].

Une autre façon de fournir des garanties de respect de la vie privée est de suivre le principe de *protection des données dès la conception et protection des données par défaut*, qui exige que le responsable du traitement mette « en œuvre, tant au moment de la détermination des moyens du traitement qu'au moment du traitement lui-même, des mesures techniques et organisationnelles appropriées » [6]. Pour cela, le responsable du traitement doit intégrer ces mesures dès les premières étapes de développement [1]. Plus généralement, assurer qu'un système respecte la vie privée nécessite de vérifier que les propriétés de vie privée soient respectées durant tout le cycle de vie de ce système. Cela implique généralement de considérer différents niveaux d'abstraction du système (liés aux étapes du cycle de vie), ce qui complique le processus de vérification.

Les propriétés liées au consentement sont particulières car elles reposent sur un accord entre les personnes concernées par le traitement des données personnelles [4]. Ces propriétés, même si elles sont prisent en compte par les services juridiques, peuvent être parfois ignorées pendant les phases de modélisation du système et ne sont généralement pas vérifiées durant les étapes d'implémentation. Cela peut conduire à des problèmes au regard des bases légales.

2 Contributions

Notre article propose une approche pour vérifier des propriétés de consentement à deux étapes de développement différentes : la vérification de la modélisation et la vérification du code, comme illustré sur la figure 1. Le langage de spécification CSpeL permet aux ingénieurs de formellement spécifier les éléments à prendre en compte pour vérifier que le système respecte des propriétés de vie privée, tandis que l'outil CASTT permet de vérifier que certaines traces du modèle ou du programme sont correctes, ou encore de démontrer la correction exhaustive de l'implémentation du système vis-à-vis de ces propriétés. Nous avons évalué la correction et l'efficacité de notre approche sur des cas d'utilisations de deux domaines d'application différents, aussi bien au niveau modèle que programme.

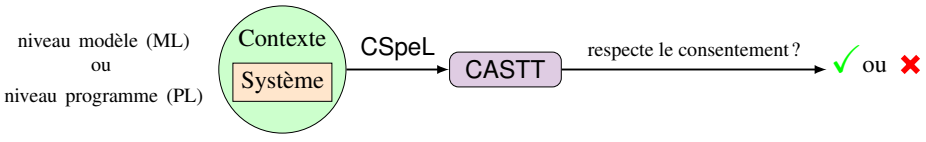


FIGURE 1 – Vue générale de l'utilisation des contributions : CSpeL et CASTT.

Plus précisément, nos contributions sont les suivantes :

— CSpeL, un nouveau langage de spécification du contexte permettant de formaliser les éléments clés nécessaires pour vérifier deux propriétés spécifiques liées au consentement : la *conformité aux finalités consenties* et la *conformité à la nécessité des données* ;

— une formalisation des deux propriétés de consentement ci-dessus, la première stipulant que les données personnelles sont seulement traitées pour des finalités de traitement consenties ; et la seconde stipulant que les données personnelles sont seulement traitées lorsque cela est nécessaire.

— CASTT, un nouvel outil de vérification qui inclut :

  - une méthode de vérification des propriétés précédentes sur des traces d'un modèle ou d'un programme ; et
  - un mécanisme de traduction de CSpeL vers le langage de spécification ACSL [3], qui permet de vérifier le respect des propriétés précédentes sur un code C à l'aide de Frama-C [2] ;

— une évalutation empirique de CASTT sur des cas d'utilisation pour deux domaines d'application qui illustre l'utilité de notre approche.

3 Résultats

Nous avons évalués les deux mécanismes de vérification de CASTT : l'analyse de trace et la traduction pour un outil au niveau programme.

Pour le premier mécanisme, nous nous sommes intéressés aux questions de recherche suivantes :

RQ1 *Est-ce que CASTT peut vérifier les propriétés sur une trace au niveau modèle/programme et est-ce qu'il peut détecter les traces invalides ?*

RQ2 *Est-ce que CASTT est utilisable sur des grandes traces ?*

Pour répondre à la question **RQ1**, nous avons testé l'outil sur plusieurs exemples, aux niveaux modèle et programme, aussi bien pour des traces devant être détectées comme valides que pour des traces devant être détectées comme invalides. Nos expérimentations montrent que CASTT fourni toujours le verdict attendu.

Pour répondre à la question **RQ2**, CASTT est utilisé pour vérifier un exemple pour lequel nous faisons varier la taille de la trace (allant de 10 à 1 000 000 d'événements). Nos expérimentations montrent que le temps de vérification est linéaire par rapport à la taille de la trace et que, pour des grandes traces (avec 1 000 000 évènements), la vérification par CASTT prend moins de 4 secondes.

Pour le mécanisme de traduction, nous nous sommes intéressés aux questions de recherche suivantes :

RQ3 *Est-ce que CASTT peut vérifier des systèmes au niveau programme en traduisant un fichier CSpeL pour un outil spécifique au niveau programme, de façon à ce que cet outil puisse toujours détecter les traces invalides ?*

RQ4 *Est-ce que CASTT peut être utilisé sur un code de grande taille (i.e. avec beaucoup de lignes de code et d'appels de fonctions) ?*

Pour répondre à la question **RQ3**, nous avons testé l'outil sur des exemples variés pour lesquels nous avons vérifié si les différents greffons de vérification de Frama-C (WP poour la preuve de programmes WP, Eva pour l'interprétation, et E-ACSL pour la vérification à l'exécution) pouvaient être utilisés pour détecter les traces invalides. Nos expérimentations montrent que CASTT, combiné avec n'importe quel greffon de vérification de Frama-C, peut vérifier le respect du consentement sur le code testé.

Pour répondre à la question **RQ4**, nous testons les différentes techniques sur un même fichier C (sauf pour le nombre d'appels de fonction) et le même fichier de spécification CSpeL. Nous utilisons un générateur d'appels de fonction pour augmenter le nombre d'appels de fonctions dans la fonction `main` du fichier source. Les résultats de nos expérimentations montrent que le temps de génération des annotations est négligeable comparé au temps de vérification des greffons. Ils montrent également que la vérification des annotations générées ne ralentit pas exagérément le temps de vérification des greffons (généralement moins de 10% pour Eva et 5% pour E-ACSL) et plus la taille du code à vérifié est grande, moins le temps de vérification est impacté.

Références

[1] Ahmadian, A. : Model-based privacy by design. Phd thesis, Universität Koblenz-Landau (2020)

[2] Baudin, P., Bobot, F., Bühler, D., Correnson, L., Kirchner, F., Kosmatov, N., Maroneze, A., Perrelle, V., Prevosto, V., Signoles, J., et al. : The dogged pursuit of bug-free c programs : the frama-c software analysis platform. Communications of the ACM (2021)

[3] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V. : ACSL : ANSI/ISO C Specification Language. Tech. rep.

[4] Clouet, M., Antignac, T., Arnaud, M., Pedroza, G., Signoles, J. : A new generic representation for modeling privacy. In : 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW. pp. 203–211 (2022). https ://-doi.org/10.1109/EuroSPW55150.2022.00027

[5] Clouet, M., Antignac, T., Arnaud, M., Signoles, J. : Context specification language for formally verifying consent properties on models and code. In : Tests and Proofs. pp. 68–93. Springer Nature Switzerland, Cham (2023)

[6] European Commission : Regulation (EU) 2016/679 (General Data Protection Regulation). Tech. rep. (2016), https://eur-lex.europa.eu/legal-content/EN/TXT/PDF/?uri=CELEX:32016R0679

[7] Huth, M., Ryan, M. : Logic in Computer Science : Modelling and Reasoning about Systems. Cambridge University Press ; 2nd edition (2004)

[8] Tschantz, M.C., Wing, J.M. : Formal methods for privacy. In : International Symposium on Formal Methods (2009)

Project & Conquer : Élimination de quantificateurs pour la vérification de problèmes d'accessibilité dans les réseaux de Petri

Résumé long

Nicolas Amat[1], Silvano Dal Zilio[2], and Didier Le Botlan[2]

[1]DTIS, ONERA, Université de Toulouse, 31000, Toulouse, France
[2]LAAS-CNRS, Université de Toulouse, CNRS, 31000, Toulouse, France

Nicolas Amat, Silvano Dal Zilio, and Didier Le Botlan. Project and Conquer : Fast Quantifier Elimination for Checking Petri Net Reachability. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 14499 of *Lecture Notes in Computer Science*. Springer, 2024. https://laas.hal.science/hal-04375443v1

Contexte général. Dans cet article nous décrivons une méthode pour accélérer la vérification de propriétés d'accessibilité dans les réseaux de Petri en tirant parti de réductions structurelles [Ber87]. La question est de savoir si un état (aussi appelé marquage) donné ou une classe d'états décrite par un prédicat F (exprimé en utilisant une combinaison booléenne de contraintes linéaires entre le marquage des places), peut être atteint par le modèle d'un système. En prenant pour exemple un réseau de Petri représentatif de la vérification d'une architecture logicielle, qui modélise un problème de partage de ressources dans un système d'exploitation, voir la Figure 1 à gauche, on peut souhaiter vérifier s'il est possible d'atteindre un état « problématique » avec plus de tâches en cours d'exécution que de segments mémoire libres. Dans ce cas $F \triangleq (\mathit{ExecutingTask} > \mathit{FreeMemSegment})$. Cette classe de formules correspond aux requêtes d'accessibilité utilisées dans le Model Checking Contest (MCC) [AAB+25], une compétition d'outils de vérification de réseaux de Petri que nous utilisons comme référence.

Réductions polyédriques. Cet article repose sur une approche que nous avons précédemment présentée, appelée *réduction polyédrique* [ABDZ21, ABDZ22, BLBDZ18], qui consiste à calculer des réductions de la forme (N, E, N') où : N est un réseau initial que nous souhaitons analyser (dans un espace de dimension n, qui correspond au nombre de places de N) ; N' est un réseau résiduel, que nous espérons beaucoup plus simple que N (donc dans un espace de dimension n', inférieur à n) ; et E est un prédicat de Presburger (théorie du premier ordre des nombres entiers naturels munis de l'addition). L'idée est de préserver suffisamment d'informations dans E pour reconstruire l'espace d'états de N, en ne connaissant que celui de N'. En résumé, nous capturons et abstrayons l'effet des réductions en utilisant des contraintes linéaires entre les places de N et N'. Par conséquent, nous pouvons représenter l'espace d'états de N comme l'image inverse, par

E, d'un sous-ensemble de vecteurs de dimension n'. Cette technique permet, dans de nombreux cas, d'obtenir une représentation très compacte de l'espace d'états, et se traduit par une nouvelle relation d'équivalence, $N \equiv_E N'$, appelée *équivalence polyédrique*, en référence aux ≪ modèles polyédriques ≫ utilisés dans l'optimisation de programmes et l'analyse statique [Fea96, BJT99]. En effet, comme dans ces travaux, nous proposons une représentation algébrique de la relation entre un modèle et son espace d'états basée sur les ensembles de solutions de contraintes linéaires.

La contribution de notre article est une procédure d'élimination de variables quantifiées existentiellement dans l'arithmétique de Presburger, afin de projeter une propriété d'accessibilité F, sur le réseau N, en une propriété d'accessibilité F' sur le réseau réduit N', tout en préservant le verdict. Nous avons implanté cette procédure dans un nouvel outil, appelé Octant [Ama24], qui peut agir comme un pré-processeur permettant à n'importe quel model-checker de bénéficier de manière transparente de notre optimisation.

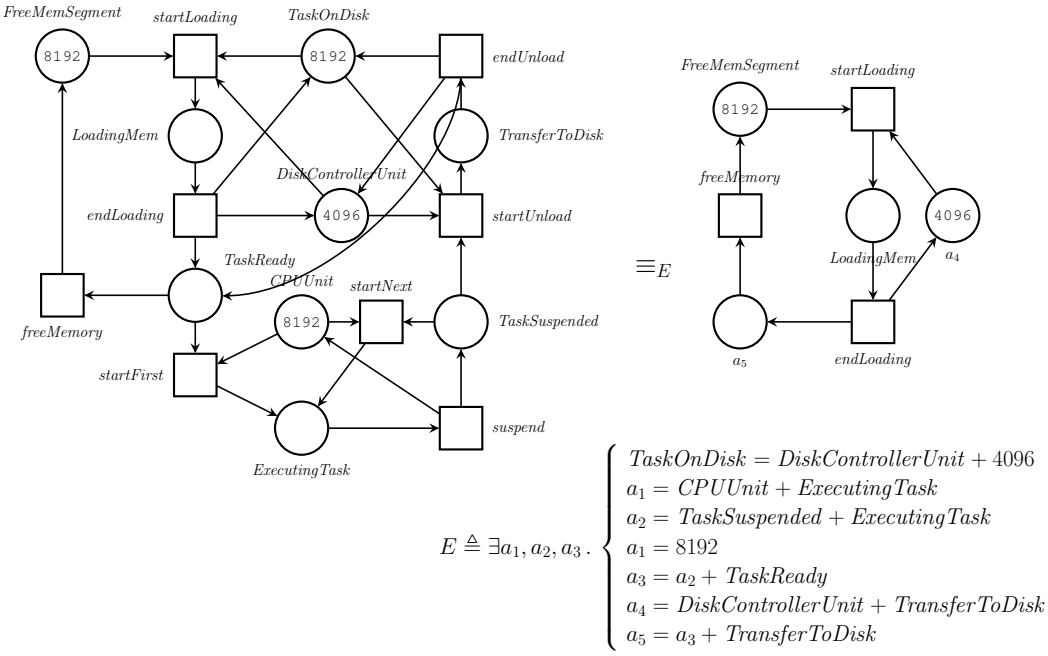


FIGURE 1 – Exemple du réseau de Petri ≪ SmallOperatingSystem ≫ (qui admet 10^{17} marquages accessibles), provenant du Model Checking Contest et sa réduction polyédrique.

Afin d'illustrer cette approche, dans la Figure 1 nous donnons, à droite, un exemple de réduction polyédrique du réseau de gauche, avec E le prédicat Presburger permettant de lier les deux espaces d'états. Une telle abstraction est généralement obtenue en appliquant un ensemble de règles de réductions structurelles de manière itérative. Il est également intéressant de noter que notre méthode n'impose pas de restrictions sur la syntaxe des réseaux, telles que des contraintes sur les poids des arcs ou des limites sur le marquage de places. Elle peut donc s'appliquer à des réseaux non bornés (dont l'espace d'états est infini).

Conservation de l'accessibilité. Une application intéressante est le *théorème de conservation de l'accessibilité* [ABDZ21] : une propriété F est accessible dans N si et seulement si $E \wedge F$ est accessible dans N'. Cette propriété est intéressante car elle signifie que nous pouvons appliquer des techniques de réduction plus agressives que, par exemple, *slicing* [Rak12,LOST17,KKG18], *cone of influence* [CGP99], ou d'autres méthodes [GRVB08, KBJ21] qui cherchent à supprimer ou à agglomérer les places qui ne sont pas pertinentes pour la propriété à vérifier (et qui ne peuvent donc pas contribuer à sa valeur de vérité). Nous ne partageons pas cette restriction dans notre approche, puisque nous réduisons les réseaux au préalable et pouvons donc réduire les places qui apparaissent dans la propriété initiale. En effet, les approches similaires au slicing ne simplifient un modèle que par rapport à une formule, alors qu'avec notre méthode, nous simplifions le modèle autant que possible et ensuite les formules si nécessaire.

Élimination de quantificateurs. Cependant, il existe une complication liée au fait que la formule $E \wedge F$ peut inclure des variables (places) qui n'apparaissent plus dans le réseau réduit N' et qui agissent donc comme des variables existentiellement quantifiées. Cela peut compliquer certaines techniques de vérification symbolique, telles que la k-induction [SSS00] (nécessitant une quantification universelle) et empêcher l'utilisation d'approches explicites et énumératives. En effet, dans ce dernier cas, cela signifie que nous devons résoudre un problème en arithmétique linéaire entière pour chaque nouvel état visité, au lieu de simplement évaluer une formule $\ll$ close $\gg$. Pour surmonter ce problème, nous proposons dans cet article une nouvelle méthode qui permet de projeter la formule $E \wedge F$ en une formule équivalente, F', qui ne fait référence qu'aux places de N'. Ensuite, F' peut être vérifiée sur N' à l'aide de n'importe quel outil ou méthode de vérification.

Nous définissons notre projection comme une semi-procédure d'élimination des quantificateurs en arithmétique de Presburger, adaptée au type spécifique de contraintes que nous manipulons dans E. Alors que l'élimination des quantificateurs a une complexité exponentielle en général pour les formules existentielles, notre construction a une complexité linéaire et ne peut que diminuer la taille d'une formule. De plus, elle se termine toujours et renvoie un résultat dont la correction est garantie. Ce qui signifie qu'elle sous-approxime l'ensemble des modèles accessibles et que, par conséquent, un témoin de F' dans N' correspond nécessairement à un témoin de F dans N. De plus, notre méthode inclut une simple condition sur F qui suffit à détecter quand notre résultat est exact, c'est-à-dire que si F' est inaccessible dans N', alors F est inaccessible dans N. Notons que si cela n'est pas le cas, notre méthode peut, éventuellement, ne pas permettre de conclure sur l'accessibilité de F (c'est-à-dire F peut être accessible sans que F' ne le soit également). Par ailleurs, nous montrons dans évaluation expérimentale que notre projection est complète pour 80% des formules utilisées dans le MCC.

Structure de données. Cette procédure d'élimination de quantificateurs repose sur une nouvelle structure de données, appelée Token Flow Graph (TFG) [ADZLB21, ADZLB23], qui permet de capturer la structure particulière des contraintes apparaissant dans le prédicat Presburger (E). Les TFGs sont des graphes orientés acycliques (DAG), avec un nœud par variable apparaissant dans le prédicat E, et deux types d'arcs deux grands types de contraintes. Le premier type d'arcs correspond aux redondances, qui sont des équations de la forme $p = q + r + \ldots$ exprimant

que la place p a été supprimée et que son marquage peut être reconstruit à partir du marquage des places $q, r, \ldots$ (voir p. ex. l'équation $TaskOnDisk = DiskControllerUnit + 4096$ dans la Figure 1). Le second type correspond aux agglomérations, qui sont des équations de la forme $a = p + q + \ldots$, générées en agglomérant plusieurs places $(p, q, \ldots)$ en une nouvelle (a), et où le marquage de $p, q, \ldots$ peut être reconstruit à partir de celui de a (voir p. ex. l'équation $a_4 = DiskControllerUnit + TransferToDisk$ dans la Figure 1). Cette structure de données conduit à des algorithmes de vérification spécifiques, de complexité linéaire, en permettant de raisonner sur l'évolution des jetons dans E.

Évaluation expérimentale. Nous avons appliqué notre approche aux meilleurs outils participant au Model Checking Contest (en excluant SMPT [ADZ23] que nous développons) : ITS-Tools [TM15], LoLA [Wol18] et TAPAAL [DJJ+12], sur l'ensemble des requêtes du Model Checking Contest 2023. Ce benchmark fournit un grand nombre de modèles (1 426) aux caractéristiques structurelles et comportementales variées, couvrant également une grande variété de cas d'utilisation et dont la taille varie, de 4 à 50 000 places, et de 7 à 200 000 transitions. La Table 1 résume les résultats obtenus sur l'ensemble des 897 formules difficiles (celles pour lesquelles au moins un des outils échoue) du Model Checking Contest 2023. Nos expérimentations montrent des gains de performance substantiels avec notre approche, allant de 20% à 60% en termes de requêtes supplémentaires que nous pouvons traiter (dans une limite de temps raisonnable).

Outils	# Propriétés vérifiées		Temps de calcul	
	Original	Réduit	Moyen	Médian
ITS-Tools	281	333	×1.63	×1.04
LoLA	188	241	×10.91	×1.40
TAPAAL	168	274	×1.43	×1.10

Table 1 – Impact des réductions polyédriques sur les outils de l'état de l'art.

Ces résultats montrent que les réductions polyédriques sont efficaces sur un grand nombre de propriétés, orthogonales aux optimisations existantes et que leurs avantages se combinent avec d'autres optimisations. Un résultat intéressant de ce travail est la définition d'un fragment non trivial de l'arithmétique existentielle de Presburger avec de bonnes propriétés de complexité qui, nous l'espérons, pourrait être applicable dans d'autres contextes. Ce constat est appuyé par l'évaluation expérimentale représentée dans la Figure 2, où nous avons comparé, sur les requêtes du MCC, la performance d'Octant avec Redlog [DS97] et isl [Ver10], deux outils de référence pour l'arithmétique de Presburger.

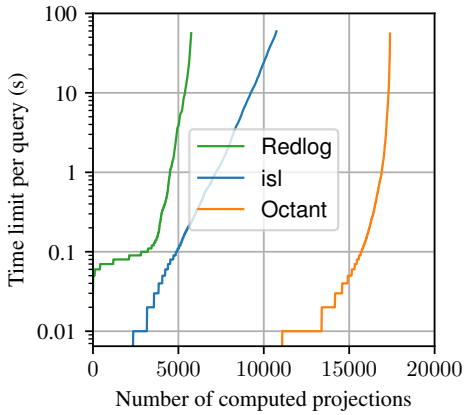


Figure 2 – Comparaison d'Octant avec Redlog [DS97] et isl [Ver10].

Références

[AAB+25] Nicolas Amat, Elvio Amparore, Bernard Berthomieu, Pierre Bouvier, Silvano Dal Zilio, Francis Hulin-Hubard, Peter G. Jensen, Loig Jezequel, Fabrice Kordon, Shuo Li, Emmanuel Paviot-Adet, Laure Petrucci, Jiří Srba, Yann Thierry-Mieg, and Karsten Wolf. Behind the scene of the model checking contest, analysis of results from 2018 to 2023. In *TOOLympics Challenge 2023*. Springer, 2025. doi:10.1007/978-3-031-67695-6_3.

[ABDZ21] Nicolas Amat, Bernard Berthomieu, and Silvano Dal Zilio. On the Combination of Polyhedral Abstraction and SMT-Based Model Checking for Petri Nets. In *Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, volume 12734 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-76983-3_9.

[ABDZ22] Nicolas Amat, Bernard Berthomieu, and Silvano Dal Zilio. A Polyhedral Abstraction for Petri Nets and its Application to SMT-Based Model Checking. *Fundamenta Informaticae*, 187(2-4) :103–138, 2022. doi:10.3233/FI-222134.

[ADZ23] Nicolas Amat and Silvano Dal Zilio. SMPT : A Testbed for Reachabilty Methods in Generalized Petri Nets. In *Formal Methods (FM)*, volume 14000 of *Lecture Notes in Computer Science*. Springer, 2023. doi:10.1007/978-3-031-27481-7_25.

[ADZLB21] Nicolas Amat, Silvano Dal Zilio, and Didier Le Botlan. Accelerating the Computation of Dead and Concurrent Places Using Reductions. In *Model Checking Software (SPIN)*, volume 12864 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-84629-9_3.

[ADZLB23] Nicolas Amat, Silvano Dal Zilio, and Didier Le Botlan. Leveraging polyhedral reductions for solving Petri net reachability problems. *International Journal on Software Tools for Technology Transfer*, 25(1) :95–114, 2023. doi:10.1007/s10009-022-00694-8.

[Ama24] Nicolas Amat. Octant : The Reachability Formula Projector. un outil pour projeter des propriétés d'accessibilité d'un réseaux de petri sur sa version réduite, 2024. URL : https://github.com/nicolasAmat/Octant.

[Ber87] G. Berthelot. Transformations and Decompositions of Nets. In *Petri Nets : Central Models and Their Properties (ACPN)*, volume 254 of *Lecture Notes in Computer Science*. Springer, 1987. doi:10.1007/978-3-540-47919-2_13.

[BJT99] Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In *Static Analysis (SAS)*, volume 1694 of *Lecture Notes in Computer Science*. Springer, 1999. doi:10.1007/3-540-48294-6_4.

[BLBDZ18] Bernard Berthomieu, Didier Le Botlan, and Silvano Dal Zilio. Petri net Reductions for Counting Markings. In *Model Checking Software (SPIN)*, volume 10869 of *Lecture Notes in Computer Science*. Springer, 2018. doi:10.1007/978-3-319-94111-0_4.

[CGP99] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[DJJ+12] Alexandre David, Lasse Jacobsen, Morten Jacobsen, Kenneth Yrke Jørgensen, Mikael H. Møller, and Jiří Srba. TAPAAL 2.0 : Integrated Development Environment for Timed-Arc Petri Nets. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-28756-5_36.

[DS97] Andreas Dolzmann and Thomas Sturm. Redlog : computer algebra meets computer logic. *ACM SIGMA Bulletin*, 31(2) :2–9, 1997. doi:10.1145/261320.261324.

[Fea96] Paul Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, volume 1132 of *Lecture Notes in Computer Science*. Springer, 1996. doi:10.1007/3-540-61736-1_44.

[GRVB08] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few : Automatic abstraction refinement for petri nets. *Fundamenta Informaticae*, 88(3) :275–305, 2008.

[KBJ21] Jiawen Kang, Yunjun Bai, and Li Jiao. Abstraction-Based Incremental Inductive Coverability for Petri Nets. In *Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, volume 12734 of *Lecture Notes in Computer Science*. Springer, 2021. doi:10.1007/978-3-030-76983-3_19.

[KKG18] Yasir Imtiaz Khan, Alexandros Konios, and Nicolas Guelfi. A survey of Petri nets slicing. *ACM Computing Surveys*, 51(5) :1–32, 2018. doi:10.1145/3241736.

[LOST17] Marisa Llorens, Javier Oliver, Josep Silva, and Salvador Tamarit. An integrated environment for Petri net slicing. In *Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, volume 10258 of *Lecture Notes in Computer Science*. Springer, 2017. doi:10.1007/978-3-319-57861-3_8.

[Rak12] Astrid Rakow. Safety slicing Petri nets. In *Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, volume 7347 of *Lecture Notes in Computer Science*. Springer, 2012. doi:10.1007/978-3-642-31131-4_15.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking Safety Properties Using Induction and a SAT-Solver. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 1954 of *Lecture Notes in Computer Science*. Springer, 2000. doi:10.1007/3-540-40922-X_8.

[TM15] Yann Thierry-Mieg. Symbolic Model-Checking Using ITS-Tools. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*. Springer, 2015. doi:10.1007/978-3-662-46681-0_20.

[Ver10] Sven Verdoolaege. isl : An integer set library for the polyhedral model. In *Mathematical Software (ICMS)*, volume 6327 of *Lecture Notes in Computer Science*. Springer, 2010. doi:10.1007/978-3-642-15582-6_49.

[Wol18] Karsten Wolf. Petri Net Model Checking with LoLA 2. In *Application and Theory of Petri Nets and Concurrency (PETRI NETS)*, volume 10877 of *Lecture Notes in Computer Science*. Springer, 2018. doi:10.1007/978-3-319-91268-4_18.

Transformations automatisées de preuves Coq

Alexandre Jean, Nicolas Magaud*

Lab. ICube UMR 7357 CNRS Université de Strasbourg, France

Résumé

Les assistants de preuves interactifs tels que Coq [10][1] permettent à leurs utilisateurs d'écrire et de vérifier des preuves sur des sujets variés. Dans cette thèse, nous explorons le cycle de vie d'une preuve Coq après son écriture, notamment la phase de maintenance mais aussi les différentes réécritures possibles de cette preuve. Dans cet objectif, nous présentons dans ce document nos travaux actuels consistant en une bibliothèque de réécriture et d'exécution d'arbre de syntaxe abstraite (AST) Coq ainsi qu'un exemple de transformation réalisable à l'aide de cette bibliothèque.

Mot clés : assistant de preuve interactif, ingénierie de la preuve, refactoring de preuve.

1 Introduction

L'objectif principal d'une preuve écrite dans un assistant de preuve interactif est de vérifier que la propriété énoncée est correcte. Dans le reste de ce document, on s'intéressera aux preuves dans le langage Coq. On s'intéresse aux transformations possibles d'une preuve après sa validation comme correcte. Transformer une preuve consiste à effectuer une suite d'opérations ajoutant, remplaçant ou supprimant des nœuds de la preuve. Chaque nœud correspond à une "phrase" dans la grammaire Coq, c'est-à-dire une commande ou tactique se terminant par un point.

Des exemples possibles de ces transformations sont la suppression des tactiques et commandes non nécessaires à la complétion de la preuve telles que des commandes *Search* ou des tactiques de simplification superflues tel que *simpl*. On peut aussi vouloir ajouter des éléments de structure à la preuve pour par exemple l'indenter à l'aide de *bullets*. Un exemple de transformation utilisant des remplacements de nœuds est une transformation remplaçant chaque appel à la tactique *intros* par *intros* $V_1\ V_2 \ldots V_n$ avec V, la liste des variables introduites par *intros*. Cette transformation permet de rendre le script de preuve plus robuste, car il n'est alors plus dépendant de l'algorithme de nommage des variables de Coq.

Il est aussi possible de combiner ces différentes méthodes de transformation, par exemple en remplaçant les commandes de recherche de preuve telles que *auto* par les étapes effectuées par *auto*. Bien qu'il soit possible de réaliser ces transformations manuellement, elles peuvent être longues et fastidieuses dans le cas de grands développements et il est possible de les automatiser.

Il existe plusieurs implémentations de transformations spécifiques dans la littérature. On trouve notamment une transformation permettant de passer d'une preuve utilisant plusieurs tactiques et commandes à une preuve n'en contenant qu'une seule, en combinant les tactiques à l'aide des *tacticals ;* et *[]* de Coq [8]. À l'inverse, une transformation visant à décomposer une telle preuve en une séquence de tactiques élémentaires a également été proposée, avec deux implémentations distinctes : l'une par Magaud *et al*. [7], et l'autre par Shi *et al*. [9].

Ces implémentations ne sont que des prototypes spécialisés dans un type de transformation. Afin de faciliter la réalisation d'un plus grand nombre de transformations, nous avons développé une bibliothèque pour l'écriture et l'exécution d'arbres de syntaxe abstraite (AST) de Coq.

*. {alexandre.jean,magaud}@unistra.fr

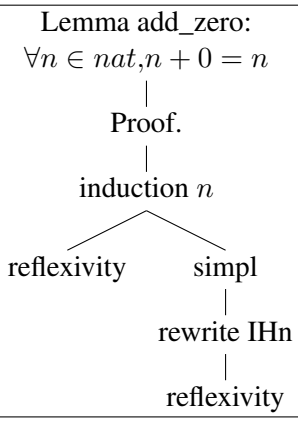
1. Désormais nommé Rocq depuis la version 9.0.0

2 Bibliothèque de réécriture d'AST

Afin de pouvoir réaliser ces transformations, le premier développement de cette thèse consiste en la réalisation d'une bibliothèque de réécriture d'AST Coq. Cette bibliothèque s'appuie sur la bibliothèque *coq-lsp* [5] pour extraire l'AST d'un fichier Coq. Une fois cet AST obtenu, on note L la liste de nœuds syntaxiques situés à la profondeur zéro de l'AST auxquels on associe leur position dans le document et un identifiant unique.

Une fois la liste L récupérée, nous avons implémenté une méthode permettant de récupérer la liste des preuves P présentes dans un document. Une preuve est ici définie comme une structure composée du nœud de la propriété à démontrer, ainsi que d'une liste de nœuds contenant les commandes et tactiques utilisées pour prouver cette propriété.

Preuve en Coq	Arbre de Preuve

`Lemma add_zero:`
`forall n : nat, n + 0 = n.`
`Proof.`
`  induction n.`
`  reflexivity.`
`  simpl.`
`  rewrite IHn.`
`  reflexivity.`
`Qed.`
	

TAB. 1 – *Preuve et arbre de preuve de* `add_zero`

Ces preuves peuvent ensuite être transformées à l'aide de notre bibliothèque en arbre de preuve.

Un arbre de preuve T est défini comme l'arbre dont les nœuds correspondent à ceux d'une preuve p, et qui satisfait les conditions suivantes : la racine de l'arbre représente la propriété démontrée par la preuve p ; chaque nœud possède un nombre d'enfants égal au nombre de sous-buts générés par la tactique qui lui est associée et chaque nœud admet pour parent le nœud contenant soit la tactique l'ayant créée, soit la tactique dont l'état final constitue l'état initial du nœud. Un exemple d'une preuve et de son arbre de preuve correspondant est présenté dans la table 1.

Nous avons en parallèle défini des méthodes de transformation d'AST qui permettent de supprimer, remplacer ou ajouter des nœuds dans une preuve, un arbre de preuve ou un document L en déplaçant automatiquement les autres nœuds dans des positions valides. En utilisant ces définitions, on définit de manière formelle une transformation de preuve comme vu au paragraphe précédent comme une fonction f prenant en argument une preuve ou un arbre de preuve et renvoyant une liste d'étapes de transformation correspondant aux étapes de la transformation parmi l'ensemble d'opérations suivant:

$$\{Remove(id), Replace(id, new\_node), Add(new\_node)\}.$$

Ici, l'opération $Remove(id)$ supprime le nœud avec l'identifiant id, l'opération $Replace(id, new\_node)$ le remplace par new_node et l'opération $Add(new\_node)$ ajoute new_node.

Les transformations renvoient une liste d'étapes de transformation à la place d'une nouvelle preuve afin de bénéficier des avantages du patron de conception commande, notamment la possibilité de traiter une transformation comme une transaction réversible ainsi que de bénéficier d'un debugging plus granulaire.

3 Exemple d'une transformation automatisée

Afin de présenter l'intérêt ainsi que les capacités actuelles de notre outil de transformation de preuve, nous présentons ici la transformation d'une preuve contenant des appels à la tactique *auto* en une preuve remplaçant les appels à cette tactique par la liste des tactiques trouvées par chaque *auto*. Nous prenons comme exemple, la preuve suivante :

```
Goal forall P Q R S : Prop,
  (P -> Q) -> (Q -> R) -> (R -> S) -> (P \/ S) -> (Q \/ R \/ S).
Proof.
  intros P Q R S HPQ QRR RSS H.
  destruct H.
  auto.
  right; right.
  assumption.
Qed.
```

Selon la documentation Coq, il est possible d'obtenir la liste des étapes utilisées par *auto* en utilisant la tactique *info_auto*. La première étape de la transformation consiste donc à exécuter la tactique *info_auto* à chaque nœud contenant la tactique *auto*.

Cette tactique renvoie une chaîne de caractères C représentant la recherche effectuée par la tactique *auto* composée d'une liste éventuelle de tactique *intro*, suivi d'un arbre de recherche de preuve. Dans notre exemple, on obtient la sortie C suivante:

Listing 1: Sortie de la commande *info_auto*

```
(* info auto: *)                                                    1
simple apply or_intror (in core).                                   2
 simple apply or_intror (in core).                                  3
  simple apply RSS.                                                 4
   simple apply QRR.                                                5
    simple apply HPQ.                                               6
 simple apply or_introl (in core).                                  7
  simple apply QRR.                                                 8
   simple apply HPQ.                                                9
    assumption.                                                     10
```

Cet arbre contient la liste des tactiques de la preuve trouvée par *auto* ainsi que les tentatives infructueuses de recherche qui devront être éliminées par l'algorithme. Dans le cas d'une recherche infructueuse, cet arbre contiendra un seul nœud contenant la tactique *idtac*.

Les nœuds de cet arbre ont une profondeur égale au nombre d'espaces précédant leur représentation sur leur ligne dans C. Le parent de chaque nœud n est le premier nœud précédent dont la profondeur est égal à la profondeur du nœud n moins un.

Une fois cette chaîne de caractères convertie en une liste de tactique *intro* (ici vide) et un arbre n-aire de recherche S, l'algorithme décrit ci-dessous est exécuté sur S pour obtenir la liste des étapes effectives permettant de remplacer la tactique *auto*.

On applique une réduction en profondeur sur l'arbre de recherche. Ici l'accumulateur de la réduction correspond à une pile P de tuples $t = (t_n, t_s, t_{depth}, t_{child\_count}, t_{goal\_count}, t_{r?})$ dont chaque élément contient le nœud précédent t_n, l'état précédent de ce nœud t_s , sa profondeur précédente t_{depth}, son nombre d'enfants t_{child_count}, son nombre de buts t_{goal_count} ainsi qu'une valeur booléenne indiquant si l'exécution du nœud a réussi à réduire le nombre de buts $t_{r?}$.

A la sortie de l'algorithme, la pile P contient une liste de tuples dont les éléments t_n représentent dans l'ordre inverse les étapes de la preuve effectuées par *auto*.

Au début de chaque itération k de la réduction, on récupère le nœud courant n_k ainsi le sommet de P qu'on nomme s. Si s a une profondeur s_{depth} supérieure a celle du nœud actuel, on dépile P jusqu'à

t_n	t_{depth}	$t_{\text{child\_count}}$	$t_{\text{goal\_count}}$	$t_{r?}$
assumption	5	0	1	true
simple apply HPQ	4	1	2	false
simple apply QRR	3	1	2	false
simple apply or_introl	2	1	2	false
simple apply or_intror	1	2	2	false

TAB. 2 – *Pile* P $t = (t_n, \_, t_{depth}, t_{child\_count}, t_{goal\_count}, t_{r?})$ après l'exécution de l'algorithme. On ignore t_s non représentable.

ce que le sommet de la pile corresponde à un nœud avec une profondeur inférieure ou à un nœud ayant réduit le nombre de buts. Cette opération élimine les branches de recherche inutiles précédentes et ce nœud est ensuite utilisé en remplacement de s pour la suite de l'itération.

On exécute ensuite n_k à partir de l'état précèdent s_s. Puis, on vérifie si le nœud courant n_k a des enfants. Si oui, alors on l'ajoute à la pile P. Sinon, on considère ce nœud comme terminal. Dans ce cas on vérifie si après l'exécution de n_k le nombre de but courant diminue. Si oui, alors on l'ajoute à P avec l'indicateur qu'il a réduit le nombre de buts. Sinon on élimine la branche de recherche de ce nœud en dépilant P jusqu'à arriver sur un nœud avec plusieurs enfants.

Une fois la réduction appliquée sur tous les nœuds de l'arbre de recherche, on récupère la liste de tactiques depuis P (Table 2) et on assigne à chaque nœud une position dans la preuve. Le résultat de l'application de cet algorithme sur la preuve servant d'exemple est le suivant:

```
simple apply or_intror.
simple apply or_introl.
simple apply QRR.
simple apply HPQ.
assumption.
```

On en déduit que la première branche (lignes 3-6) du listing 1 de la recherche initiale a été éliminée car un seul nœud a réduit le nombre de buts dans la Table 2, à savoir le nœud *assumption*.

4 Conclusion

Cet article décrit notre approche pour la transformation automatique de preuve Coq. Nous avons décrit les différentes structures actuelles que nous utilisons pour représenter une preuve ainsi qu'un exemple d'une transformation possible sur une preuve. Par la suite, nous souhaitons explorer d'autre formes de représentation de preuve telles que les *Hiproofs* [4] ainsi que d'autres transformations plus complexes telle que la constructivisation [1] de l'arithmétisation [3] de la géométrie de Tarski formellement vérifiée dans la bibliothèque GeoCoq [2]. Afin de pouvoir traiter de larges bibliothèques, nous développons un outil de traitement de tous les fichiers d'une bibliothèque en suivant l'ordre des dépendances. Cela nous permettra, à terme, d'évaluer notre outil et certaines transformations sur des bibliothèques telles que la bibliothèque standard de Coq, la bibliothèque de géométrie GeoCoq ou le compilateur certifié CompCert [6] .

Références

[1] Michael Beeson. A Constructive Version of Tarski's Geometry. *Annals of Pure and Applied Logic*, 166(11):1199–1273, 2015.

[2] Michael Beeson, Pierre Boutry, Gabriel Braun, Charly Gries, and Julien Narboux. GeoCoq, June 2018.

[3] Pierre Boutry, Gabriel Braun, and Julien Narboux. Formalization of the Arithmetization of Euclidean Plane Geometry and Applications. *Journal of Symbolic Computation*, 90:149–168, 2019.

[4] Ewen Denney, John Power, and Konstantinos Tourlas. Hiproofs: A Hierarchical Notion of Proof Tree. *Electr. Notes Theor. Comput. Sci.*, 155:341–359, 05 2006.

[5] Emilio Jesús Gallego Arias, Ali Caglayan, Shachar Itzhaky, Fréderic Blanqui, Rodolphe Lepigre, et al. rocq-lsp: a Language Server for the Rocq Prover, 2025.

[6] Xavier Leroy. Formal Verification of a Realistic Compiler. *Commun. ACM*, 52(7):107–115, July 2009.

[7] Titouan Lozac'h and Nicolas Magaud. Post-processing Coq Proof Scripts to Make Them More Robust. In *2nd Workshop on the development, maintenance, refactoring and search of large libraries of proofs , September 13-14, 2024, Tbilissi, Georgia*, 2024.

[8] Nicolas Magaud. Towards Automatic Transformations of Coq Proof Scripts. In *Automated Deduction in Geometry (ADG 2023)*, Belgrade, Serbia, November 2023. Pedro Quaresma et Kovács Zoltán.

[9] Jessica Shi, Cassia Torczon, Harrison Goldstein, Andrew Head, and Benjamin Pierce. Designing Proof Deautomation in Rocq. In *Proceedings of the 15th Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2025.

[10] The Coq Development Team. The Coq Proof Assistant.

Approche neuro-symbolique pour générer un référentiel de test *

Eléa Jacquin
Université Marie et Louis Pasteur, CNRS, institut FEMTO-ST
F-25030 Besançon, France
elea.jacquin@femto-st.fr

Résumé

Ce papier présente l'application de méthodes neuro-symboliques pour automatiser la génération de tests logiciels en combinant les capacités d'interprétation sémantique des Large Language Models (LLMs) avec la rigueur formelle des méthodes symboliques. Ce travail sera appliqué dans le cadre du projet ANR VVaMIA qui a démarré début 2025.

Mots-clés : tests logiciels, intelligence artificielle, neuro-symbolique

1 Introduction

Les tests sont une partie cruciale du cycle de développement des logiciels. Ils augmentent la confiance envers le système sous test [1]. Dans des domaines critiques tels que l'aéronautique, la santé ou les systèmes embarqués, leur importance devient encore plus cruciale, puisque le moindre défaut peut avoir des conséquences catastrophiques [2].

L'écriture manuelle des tests est un exercice long et rigoureux : il est donc courant de faire des erreurs, surtout si le système sous test est complexe. Apporter des éléments pour accompagner les ingénieurs validation dans la conception et la gestion d'un patrimoine de test serait une véritable aide.

L'équipe VESONTIO du département informatique du laboratoire FEMTO-ST, où ces travaux sont réalisés, possède une expertise de plus de 20 ans dans le Model-Based Testing (MBT), couvrant à la fois les aspects théoriques et appliqués. Une des limites qui freine l'adoption de cette technique est la conception du modèle. En effet, cela demande beaucoup de temps, nécessite une bonne compréhension des besoins du système, des langages de modélisation et une expertise métier du domaine d'application du système.

L'utilisation d'Intelligences Artificielles (IAs) génératives pourrait constituer une solution pour atténuer cette difficulté. L'arrivée récente des grands modèles

*Encadrée par Fabrice Bouquet, Frédéric Dadeau et Dorine Tabary au DISC/FEMTO-ST.

de langage (LLMs) a ouvert une nouvelle approche de génération de tests par leur réflexion plus humaine sur le langage naturel [3, 4]. Dans le domaine de l'ingénierie des exigences, les techniques de Traitement du Langage Naturel (NLP), incluant les LLMs, sont utilisées pour extraire des exigences à partir de cahiers des charges. Cependant, leur principal inconvénient réside dans leur tendance aux hallucinations et leur manque de fiabilité absolue [5].

2 Approche proposée

Le processus classique de génération de tests évolue progressivement d'un langage naturel avec le cahier des charges, les exigences et les scénarios vers un langage de programmation exécutable avec les tests symboliques (scripts de tests où les données ne sont pas encore définies) et les scripts exécutables.

Ce déroulement présente deux défis majeurs : l'interprétation des spécifications ambiguës en langage naturel et la production d'un code de test valide. En effet, bien que riche en connaissance métier, le cahier des charges présente souvent des ambiguïtés sémantiques ou des exigences métiers implicites, ce qui complexifie son traitement automatique avec les techniques traditionnelles de NLP. Au contraire, la traduction de ces exigences en tests formels exige une rigueur mathématique généralement incompatible avec les approximations tolérées en langage naturel. Ainsi, un couplage avec de l'IA symbolique, qui elle utilise des langages formels comme la logique, permettrait d'obtenir des valeurs attendues avec une plus grande confiance. Dans la littérature, cette méthode est qualifiée de neuro-symbolique [6, 7].

L'approche neuro-symbolique, illustrée dans la figure 1, propose une réponse à ces défis en combinant :

— **l'IA générative** pour l'analyse des documents initiaux (transition cahier des charges vers exigences par exemple)

— **Des méthodes symboliques** pour la génération de tests exécutables (transition tests symboliques vers tests instanciés)

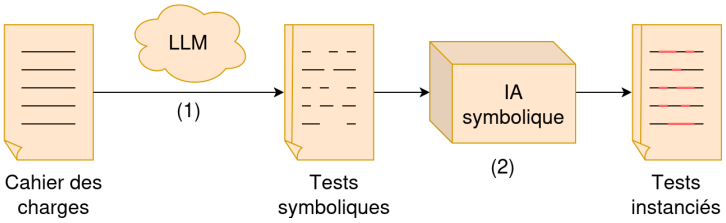


FIGURE 1 – Processus de génération de tests

Le passage direct du cahier des charges aux tests symboliques présente des risques majeurs lorsqu'il est entièrement confié à des LLMs : hallucinations, absence de traçabilité et faible confiance dans les résultats. Ces limitations nous ont conduits à développer une méthodologie plus robuste et contrôlée.

Notre approche consiste à décomposer le processus global en une série d'étapes intermédiaires vérifiables. Plutôt que de tenter une transformation directe du cahier des charges vers des tests exécutables, nous introduisons différentes phases : extraction structurée des exigences, création d'un modèle du système, génération de scénarios de test et production de tests exécutables par exemple.

Cette granularité apporte plusieurs bénéfices. D'une part, elle permet un meilleur contrôle qualité à chaque niveau d'abstraction. D'autre part, elle offre une traçabilité complète entre les différents artefacts produits. Enfin, elle autorise des interventions correctives lorsqu'une erreur est détectée, bien avant la phase finale.

Les avantages majeurs de cette approche sont qu'elle est une assistance précieuse pour l'ingénieur validation, puisqu'elle lui donne le contrôle sur ce qui est produit à chaque étape du processus et donc assure une traçabilité des résultats.

3 Premiers résultats

Une étude comparative sur les LLMs les plus utilisés sur l'extraction d'exigences depuis un cahier des charges a été réalisée. Cette étape est critique dans le processus puisqu'elle est la première donc s'il y a une mauvaise interprétation des fonctionnalités attendues dans le système, les tests produits à la fin seront erronés.

Pour effectuer cette étude, il faut constituer le meilleur prompt possible pour maximiser les résultats. Après une recherche sur les différentes techniques d'ingénierie des prompts, la stratégie adoptée est d'utiliser du zero-shot prompting (pas d'exemple donné) et la décomposition de la demande en cinq parties :

— **Rôle** : Définit le contexte et le niveau d'expertise attendu
— **Instruction** : Spécifie l'action précise à réaliser
— **Contraintes** : Fixe les règles et limitations impératives
— **Format** : Détermine l'organisation de la sortie
— **Données** : Fournit le matériel source à traiter

Aucun LLM n'a proposé d'exigences ne respectant pas le cahier des charges avec cette stratégie. Avec un prompt simple, les résultats sont plus créatifs donc plus à même de ne pas respecter strictement le cahier des charges.

L'expérience a été réalisée avec deux cahiers des charges en anglais. Une liste d'exigences de ceux-ci a été conçue manuellement par une personne experte dans le domaine du test et en comptabilise 32.

Les résultats de l'expérience montrent que, sur 10 générations, ChatGPT excellent en générant en moyenne 94,1% des exigences de la liste, suivi de DeepSeek-V3 avec 90,6%. Juste après, Claude a un score de 87,8%, Mistral 84,1%, Qwen 82,2% et en dernier DeepSeek-R1 avec 63,1%.

Ces résultats, bien qu'indicatifs, ne reflètent qu'une facette des capacités des différents LLMs évalués. Une analyse plus fine révèle des différences significatives dans leur mode de fonctionnement : par exemple, Mistral à beaucoup plus de mal à produire des exigences atomiques que Qwen.

Nous nous sommes donc interessés aux métriques existantes sur l'extraction

d'exigences et avons remarqué qu'elles avaient des failles et qu'elles n'étaient pas pertinentes pour montrer les capacités réelles des LLMs. Nous avons donc cherché à en définir de nouvelles comme par exemple évaluer l'atomicité des exigences, leur cohérence avec le cahier des charges ou leur exhaustivité.

4 Suite des travaux

Par la suite, nous envisageons de produire avec l'aide de LLMs des modèles à partir d'exigences, puis des scénarios depuis un modèle et des exigences pour pouvoir ensuite comparer leur qualité avec ceux obtenus par production directe depuis des exigences. Une autre question soulevée est la définition et l'instanciation de données concrètes pour pouvoir configurer des tests exécutables.

Les travaux de cette thèse seront appliqués dans le projet ANR RAPID VVa-MIA. Il est porté par l'entreprise Smartesting Solution & Services et inclus dans son consortium Thalès LAS ainsi que FEMTO-ST. Il s'intéresse au couplage entre l'IA générative et les modèles, dans le contexte de la vérification et la validation de systèmes embarqués en utilisant des tests logiciels. Ce projet vise à la production d'assistants IA facilitant les tâches de conception et de maintenance des tests logiciels.

Références

[1] G. Myers, *The Art of Software Testing*. Business Data Processing : A Wiley Series, Wiley, 1979.

[2] T. Huckle and T. Neckel, *Bits and Bugs : A Scientific and Historical Review of Software Failures in Computational Science*. Society for Industrial and Applied Mathematics, 2019.

[3] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models : Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, 2024.

[4] R. Feldt, S. Kang, J. Yoon, and S. Yoo, "Towards autonomous testing agents via conversational large language models," in *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2023.

[5] N. Maleki, B. Padmanabhan, and K. Dutta, "Ai hallucinations : A misnomer worth clarifying," in *IEEE Conference on Artificial Intelligence (CAI)*, 2024.

[6] B. P. Bhuyan, A. Ramdane-Cherif, R. Tomar, and T. P. Singh, "Neuro-symbolic artificial intelligence : a survey," *Neural Computing and Applications*, 2024.

[7] Z. Wan, C.-K. Liu, H. Yang, C. Li, H. You, Y. Fu, C. Wan, T. Krishna, Y. Lin, and A. Raychowdhury, "Towards cognitive ai systems : a survey and prospective on neuro-symbolic ai," 2024. arXiv :2401.01040 [cs].

Session poster AFADL

On the use of generalisation and unification for composing interactions via gate connection

Joel Nguetoum Kenne[1], Boutheina Bannour[2], and Pascale Le Gall[3]

[1] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
christian.nguetoumkenne@cea.fr
[2] Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France
boutheina.bannour@cea.fr
[3] Université Paris-Saclay, CentraleSupélec, MICS Fr-91192 Gif-sur-Yvette Cedex, France.
pascale.legall@centralesupelec.fr

A Distributed System (DS) is a collection of concurrently operating subsystems communicating via message exchange. Formal specifications of DSs are crucial for tasks such as runtime verification, model-based testing or system assembly during design. We utilise Interaction Language (IL), a formalism for the formal specification of DSs, which is based on an algebra of terms. These terms are constructed from atomic interactions, including the empty interaction, message passing between subsystems, and actions, which can be message emissions or receptions that define the communication of a subsystem with its environment. These atomic interactions are temporally structured using scheduling operators, which include weak sequencing, concurrency, alternative behaviour and repetition, adhering to algebraic laws such as associativity and commutativity, grounded in their formal semantics. The IL features a graphical representation akin to UML Sequence Diagrams or Message Sequence Charts.

To obtain an interaction specification of a given DS, one can either directly design a global interaction model of the entire system, which might not detail the inner workings of the local subsystems. Or we can independently model each subsystem with a local interaction model and derive the global model through composition. Our work aims to develop an algorithmic method for composing local interaction models.

In local models, an emission of a message to the environment paired with the reception of the same message in another local model is termed complementary actions, linked by a shared object called a gate. Our approach to the composition of local interactions involves glueing complementary actions into message passing within a composite model, employing equational unification and generalisation. Equational generalisation identifies commonalities between terms, while equational unification finds substitutions that equate terms modulo equations.

Our approach to interaction composition leverages generalisation and unification techniques, preprocessing terms to highlight shared gates and obscure irrelevant sub-interactions, thereby inherently utilising the algebraic properties of scheduling operators. Future work will focus on exploring computational aspects.

On the use of generalisation and unification for composing interactions via gate connection

Advisors: Pascale Le Gall [1], Boutheina Bannour[2]

Christian Joel Kenne Nguetoum[1,2]
christian.nguetoumkenne@cea.fr

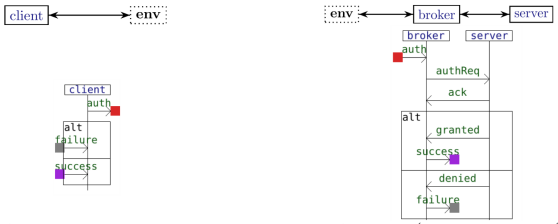
1 Université Paris-Saclay, CentraleSupélec, Laboratoire de Mathématiques et d'Informatique pour la Complexité et les Systèmes (MICS), 91192 Gif-Sur-Yvette, France

2 CEA Paris Saclay, Centre Nano-Innov, Laboratoire d'Exigences et de Conformité des Systèmes(LECS), 91120 Palaiseau, France

Interaction models with gates

We use the Interaction Language [1] to specify distributed systems. In addition to its algebraic structure, it provides an easy-to-read, diagrammatic notation similar to MSCs and UML Sequence Diagrams.
Each subsystem is represented by a vertical line called a **lifeline**, with message exchanges shown as horizontal arrows between lifelines. The language includes scheduling operators to express complex behaviours: **vp** for message passing, **loop** for repetition, **par** for concurrency, **alt** for choice, and **seq** for weak sequencing.

Actions correspond to message emissions (outgoing arrows) and receptions (incoming arrows), specifying communication with the environment (**env**). When an emission in one model and a reception in another refer to the same message, they are called **complementary actions**. Such actions are linked via a shared object known as a **gate**. In the example below, the emission of the message auth by the lifeline client (client!auth[1] in model *i*) and its reception by broker (broker?auth[1] in model *j*) are complementary and share the gate 1.



$i = \mathsf{seq}\big(\mathsf{client!auth}[1],\ \mathsf{alt}\big(\mathsf{client?failure}[3],\ \mathsf{client?success}[2]\big)\big)$ $j = \mathsf{seq}\big(\mathsf{broker?auth}[1], \mathsf{seq}\big(\dots\big)\big)$

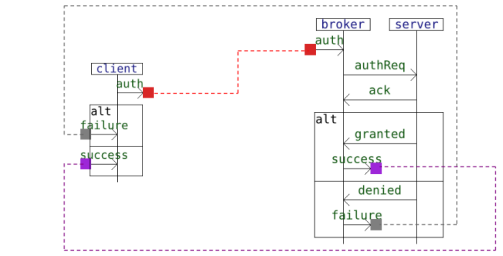
The operators obey algebraic laws grounded in the formal semantics [1] of the language.

Property	Operator	Equation
Associativity	$\mathsf{f} \in \{\mathsf{seq}, \mathsf{par}, \mathsf{alt}\}$	$\mathsf{f}(\mathsf{f}(x,y),z) \approx \mathsf{f}(x,\mathsf{f}(y,z))$
Commutativity	$\mathsf{f} \in \{\mathsf{par}, \mathsf{alt}\}$	$\mathsf{f}(x,y) \approx \mathsf{f}(y,x)$
Idempotence	alt	$\mathsf{alt}(x,x) \approx x$
Unit element $\varnothing$	$\mathsf{f} \in \{\mathsf{seq}, \mathsf{par}\}$	$\mathsf{f}(x,\varnothing) \approx x, \mathsf{f}(\varnothing,x) \approx x$
$\varnothing$-fixpoint	loop	$\mathsf{loop}(\varnothing) \approx \varnothing$

Table. Basic set *E* of algebraic equations of the Interaction Language

Interaction composition

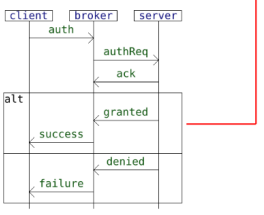
Model composition involves synthesising a new model by gluing complementary actions, those linked by a shared gate into message passings (**vp**), while preserving the scheduling information of the original models. A valid composition must **project** [2] back to the initial models from which it was constructed.
In our example, the emission and reception of auth are composed into the message passing **vp**(client!auth[1],broker?auth[1]) in the model *k*.



Projection modulo equations

Can we synthetise a global model using the algebraic equations of operators?

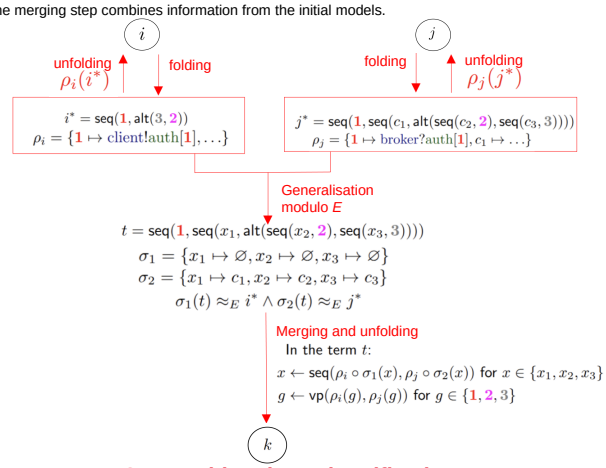
Projection modulo equations



$k = \mathsf{seq}\big(\mathsf{vp}(\mathsf{client!auth}[1], \mathsf{broker?auth}[1]), \mathsf{seq}\big(\dots\big)\big)$

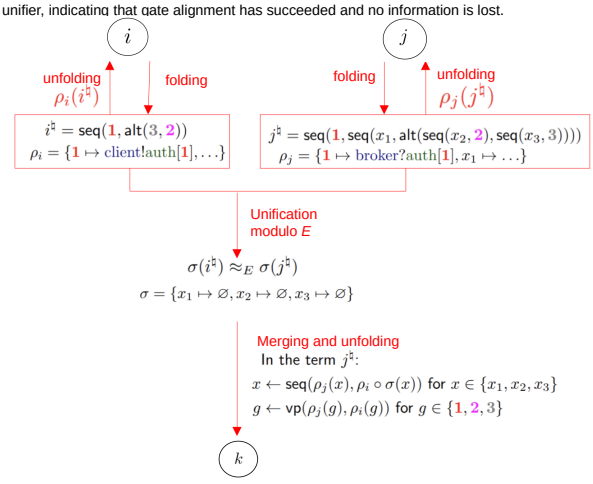
Composition through generalisation

The first step is **model folding**, which hides subterms that contain no actions relevant to the composition inside fresh constants, and replaces actions with their associated gates. The resulting folded terms are significantly simpler and are then generalised modulo *E* [3]. **The desired generaliser must include all relevant gates**, indicating successful alignment of complementary actions. Such a generaliser is said to be **well-formed**. Finally, the merging step combines information from the initial models.



Composition through unification

The first step is model folding, where actions are replaced by their associated gates. Unlike the previous approach, irrelevant subterms are hidden using fresh variables. The second step is unification modulo *E* [4]. **The desired unifier maps variables to subterms that contain no gates, and ensures that applying it to one term yields a structure containing exactly all the variables of the other**. These two conditions define what we call a **well-formed** unifier, indicating that gate alignment has succeeded and no information is lost.



Conclusion

We have explored two methods for composing interactions using algebraic equations. These approaches are not interchangeable, as there are cases where a well-formed generaliser exists while no well-formed unifier can be found. We are therefore interested in delineating the scope of each method and their respective ability to decide when composition is possible. Future work will focus on formalising correctness guarantees and investigating algorithmic aspects.

[1] Mahe, Erwan. An Operational Semantics of Interactions for Verifying Partially Observed Executions of Distributed Systems. Phdthesis, Université Paris-Saclay, 2021.

[2] Julien Lange and Emilio Tuosto. Synthesising choreographies from local session types. In 23rd international conference on Concurrency Theory (CONCUR'12), 2012.

[3] M. Alpuente, S. Escobar, J. Meseguer, and J. Sapina. Order-sorted equational generalization algorithm revisited. Annals of Mathematics and Artificial Intelligence, 2022.

[4] S. Escobar, J. Meseguer, and R. Sasse. Variant Narrowing and Equational Unification. Electronic Notes in Theoretical Computer Science, 2009.

Deductive Verification of Synchronous Reactive Programs

Frédéric Dabrowski Térence Clastres

Journées du GDR-GPL, Juin 2025

Description

Ce poster décrit un travail en cours sur le développement d'une technique de vérification de propriétés de programmes réactifs. Ces programmes s'exécutant en continu afin de réagir à leur environnement, leur spécification doit décrire leur comportement dans le temps sous forme de propriétés temporelles.

Le standard pour la vérification de ces propriétés est le model-checking: on considère des modèles qui représentent de manière abstraite le comportement des programmes. Ces modèles sont des systèmes de transitions finis. Comme la spécification caractérise l'ensemble des comportements corrects, le but est de vérifier de manière automatique que les comportements du programme sont inclus dans ceux de la spécification. Cependant, cette technique comporte deux inconvénients : Premièrement, il faut s'assurer que le modèle est bien une abstraction (sûr-approximation) du programme. Deuxièmement, les programmes complexes amènent rapidement à une explosion combinatoire ayant pour conséquence des durées de vérification déraisonnables. Après 30 ans d'existence, de nombreux travaux ont eu pour but de combattre ces faiblesses (model-checking symbolique, raffinement de modèle, parallélisme, heuristiques de recherche…). Cependant, à partir du moment où les programmes manipulent des domaines infinis, il n'est possible que de vérifier des sous-domaines finis.

Contrairement au model-checking, la vérification déductive fait le lien direct entre la syntaxe du programme, sa sémantique et sa spécification. Elle nécessite de prouver un ensemble de théorèmes afin de garantir la conformité du programme. De plus, alors que le model-checking peut être vu comme une façon de faire du test, généralement incomplet, la preuve permet de mener un raisonnement sur l'entièreté du domaine, qu'il soit fini ou infini. Il n'est néanmoins pas possible en général d'espérer trouver automatiquement une preuve. Aussi, la vérification déductive a principalement été étudiée dans le cadre de programmes qui terminent, c'est-à-dire pour lesquels on ne s'intéresse pas au comportement dans le temps.

Avec notre approche, nous désirons adapter la vérification déductive aux programmes réactifs : à partir d'un programme annoté, un ensemble de théorèmes sont générés. Ces théorèmes peuvent être prouvés automatiquement où par l'utilisateur, l'intention étant de ne faire intervenir celui-ci que pour des théorèmes non triviaux nécessitant une connaissance profonde du programme à vérifier.

Deductive Verification of Synchronous Reactive Programs

Térence Clastres, Frédéric Dabrowski

Université d'Orléans, INSA CVL, LIFO, UR 4022, Orléans, France

Deductive Verification

- Formal method to verify the functional correctness of programs
- Starts from an assumption (*precondition*) $\{P\}$ about the program initial state and tries to conclude the last state reached is within an accepted set of states $\{Q\}$ (postcondition)
- Uses deductively applied inference rules to produce mathematical statements requiring a proof
- Specification is written in a logic language (e.g. predicate logic)
- Various tools developed to ease the verification process, from SMT solvers (Alt-Ergo, Z3…), to dedicated verification platforms communicating with them (Why3, Dafny…)

Inference Rules

$$\frac{}{\{P[e/x]\}\ x := e\ \{P\}}\ assign$$

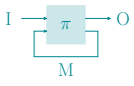
$$\frac{\{P\}\ c_1\ \{Q\} \qquad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}\ seq$$

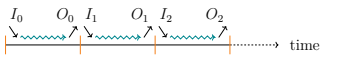
Annotated Program

$\{a = v_a \wedge b = v_b\}$
```
c := a;
a := b;
b := c
```
$\{a = v_b \wedge b = v_a\}$

Synchronous Reactive Programs

- Continuous interaction with the environment
- Infinite execution
- Logic clock, notion of *instant*





For our purposes, programs are written in a simple imperative language with arduino-like setup/loop procedures:

```
input on off : bool;
output light : bool;

setup : x := false;
loop:
  if on then
    emit true to light;
  else if off then
    emit false to light; end
  end
```

Temporal Properties

- Instead of viewing a program as a transformational machine, focus on its behaviour over time
- Decomposition Theorem: every program property is a conjunction of a safety and liveness property.

Safety: a bad thing never happens

- e.g. loop invariant
- proof tool: induction
- doing a bad thing is irremediable
  → finite counter-example

Liveness: a good thing eventually happens

- e.g. program termination
- proof tools: well-founded relations, coinduction
- a good thing can always be done later
  → infinite counter-example

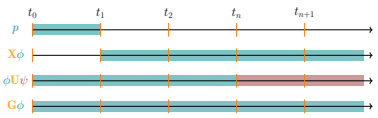
Linear Temporal Logic

- Need for a concise way to describe a program behaviour
- A behaviour is seen as an infinite word σ (e.g. abababa…)

Syntax

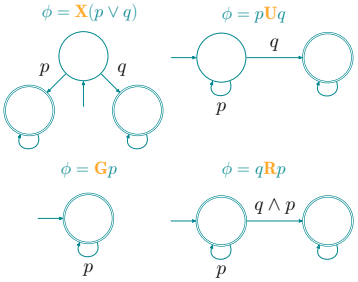
$\mathbf{LTL}(\Sigma) \ni \phi ::= p \in \Sigma \mid \neg\phi \mid \phi \vee \phi \mid \mathbf{X}\phi \mid \phi\mathbf{U}\phi \mid \mathbf{G}\phi \mid \phi\mathbf{R}\phi$

Semantics



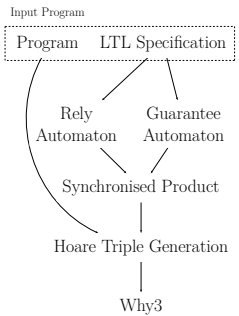
Büchi Automaton

- Nondeterministic finite-state automaton on infinite words (recognizer)
- A run is an infinite sequence of state and transitions, starting from a state marked 'initial'
- A word is read left to right. To each letter x must correspond a transition labelled x from the current state to the next
- No final state but an acceptance condition: certain states are marked 'acceptant', and at least one of them must be present infinitely many times in the run for it to be acceptant
- Effective construction from an LTL formula
- More expressive than LTL (equivalent to ω-regular expressions)



Verification Procedure for Safety Properties

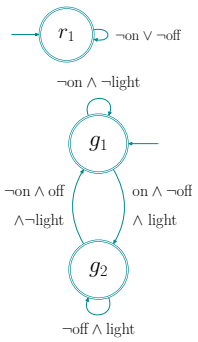
Overview



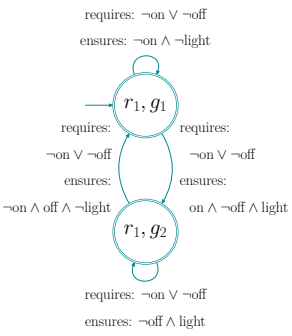
Input Program

```
input on off : bool;
output light : bool;

relies_on G (!on || !off).

guarantees
  on R (on || !light).
guarantees
  G (on -> off R (off || light)).
guarantees
  G (off -> on R (on || !light)).

setup :
  ensures { x = false }
  x := false;

loop:
  if on then
    emit true to light;
  else if off then
    emit false to light; end
  end
```

Specification Automata



Synchronized Product



Output Why3 Program

```
module Program
  (* .... *)
  let setup = begin
    ensures { outputs.light = false }
    outputs.light <- false  end

  let r1g2 = begin
    requires {(not (hd _history)._i.off /\ (outputs.light)) \/
        ((hd _history)._i.on /\ not (hd _history)._i.off /\
          ↪ outputs.light)}
    requires { not inputs.on \/ not inputs.off }
    ensures { (not inputs.off /\ outputs.light)
              \/
              (not inputs.on /\ inputs.off /\ not
                ↪ (outputs.light)) }
    if inputs.on then outputs.light <- true
    else (if inputs.off then outputs.light <- false) end

  let r1g1 = begin
    requires {(not (hd _history)._i.on /\ not outputs.light) \/
              (length _history = 0 /\ outputs.light = false)\/
      (not (hd _history)._i.on /\ (hd _history)._i.off /\ not
        ↪ outputs.light)}
    requires { not inputs.on \/ not inputs.off }
    ensures {(inputs.on /\ not inputs.off /\ (outputs.light))
              \/ (not inputs.on /\ not outputs.light) }
    if inputs.on then outputs.light <- true
    else (if inputs.off then outputs.light <- false) end
end
```

Future Work

Nondeterministic Specification

- Adds more flexibility
- Büchi automata are ∃-automata: a word is accepted if at least one path is acceptant
- Our method requires all paths to be acceptant
- This approach is correct but too naive: useless constraints are generated
  → refinement of the specification automaton

Liveness

- Safety properties enable us to mark every state acceptant
- Liveness requires showing there are infinitely many accepting states along a path
- How to deal with liveness in rely formula? → fairness assumptions

Internal Memory

- Specification needs to mention the program internal memory for successful verification
- Internal memory is an implementation detail that shouldn't be visible
- Write a public and private specification and show the private specification implies the public one

Multiple Programs

- Currently, a single program
- Goal: multiple programs in cooperation
- Parallel / sequential composition
- Controlled memory sharing
- Need for compositionality

Détection et Réparation d'Anomalies en Arithmétique à Virgule Flottante par Analyse Statique

1 Auteurs

Julien BORTOLUSSI: julien.bortolussi@enac.fr
Dorra BEN KHALIFA: dorra.ben-khalifa@enac.fr
Pierre-Loïc GAROCHE: pierre-loic.garoche@enac.fr

2 Description

Les ordinateurs manipulent les nombres réels en utilisant l'arithmétique à virgule flottante, qui est basée sur une représentation finie des nombres. Les logiciels numériques, qui utilisent l'arithmétique à virgule flottante, occupent une place prépondérante dans les systèmes de contrôle critiques de la défense nationale, des transports et des soins de santé. Bien que cette approximation soit suffisamment précise pour la plupart des applications, il existe des cas où les résultats ne sont plus pertinents et peuvent générer des exceptions et des erreurs inacceptables dans les logiciels critiques. Parmi les autres problèmes numériques existants, les calculs en virgule flottante sont sujets à l'absorption et à l'annulation. L'annulation et l'absorption catastrophiques sont connues depuis longtemps pour être d'importantes sources d'erreurs dans les calculs en virgule flottante. Dans cette thèse, nous envisageons de formaliser le problème de l'annulation catastrophique, de l'absorption et d'autres anomalies en virgule flottante à deux niveaux : une phase de détection et une phase de réparation. Formellement, nous développerons de nouvelles définitions et propriétés formelles pour la détection de ces problèmes dans les programmes en virgule flottante, basées sur une analyse statique. Cette analyse sera effectuée en générant et en résolvant un système de contraintes en raisonnant sur le calcul du bit le plus significatif des valeurs variables et leurs erreurs d'arrondi. Une fois qu'une exception a été détectée, nous pouvons utiliser le rapport d'exception pour trouver une méthode de réparation qui évite l'anomalie en déterminant la précision minimale requise pour les variables intermédiaires et en réarrangeant automatiquement les calculs.

Détection et Réparation d'Anomalies en Arithmétique à Virgule Flottante par Analyse Statique

Julien Bortolussi[1] Dorra Ben Khalifa[1] Pierre-Loïc Garoche[1]

[1]Fédération ENAC ISAE-SUPAERO ONERA, Université de Toulouse, France

Contexte

L'arithmétique à virgule flottante est l'approximation la plus courante de l'arithmétique réelle [1].

(a) Ariane 5 (2016)

(b) Supercalculateur de météo France

Utilisation des nombres flottants

Cette méthode a cependant un défaut, toutes les opérations sont susceptibles de subir une erreur d'arrondi. Bien que ces erreurs soient relativement faibles, elles peuvent être amplifiées ou conduire à des comportements contre-intuitifs. C'est ce que l'on appelle des anomalies flottantes.

L'objectif de cette thèse est de proposer une méthode basée sur l'analyse statique pour détecter les anomalies de l'arithmétique flottante et proposer des réparations.

Les Bases de L'Arithmétique Flottante

Un nombre flottant $\tilde{x}$ est représenté par :

$$\tilde{x} = b_0.b_1...b_{p-1} \times \beta^e \quad (1)$$

Où β est la base (généralement 2), $b_0.b_1...b_{p-1}$ la **mantisse**, e l'**exposant** et p la **précision**.

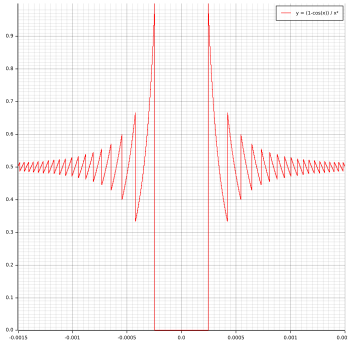
Le Standard **IEEE754** [1] définit pour les nombres flottants:

- Des **modes d'arrondi** ($\uparrow +\infty$, $\uparrow -\infty$, $\uparrow 0$, *round to nearest*).
- Des **formats** (32 bits, 64 bits, 128 bits...)
- Des opérateurs élémentaires qui doivent être exactement arrondis (évalués exactement puis arrondis)

L'Arithmétique flottante présente plusieurs anomalies [3]

Overflow	Underflow	NaN	Absorption	Cancellation
$+\infty$	~ 0	$\sqrt{-1}$	$1+\sim 0$	$1-cos(\sim 0)$

Voici un exemple de cancellation issue de [4]



Evaluation de $f(x) = \frac{1-cos(x)}{x^2}$ sur 32 bits (p=24)

Objectfs

- Générer des contraintes décrivant la propagation des erreurs dans les calculs pour détecter les anomalies de l'arithmétique flottante.
- Proposer des méthodes pour automatiquement corriger ces anomalies.

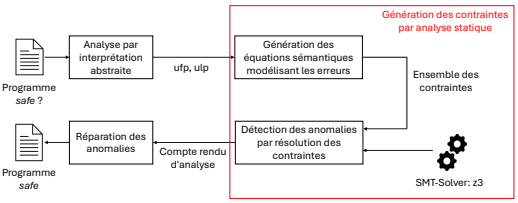
Génération de Contraintes par Analyse Statique

Notre approche se base sur l'approximation des nombres et de leur erreur par leur ordre de grandeur. Pour cela, nous utilisons les notions d'ufp, ulp et nsb [2].

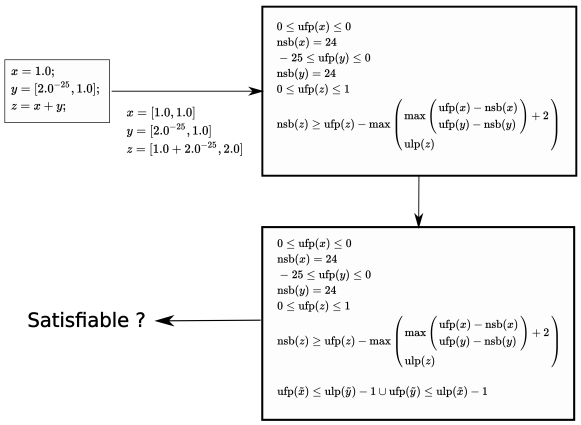
Notons $x \in \mathbb{R}$, $\tilde{x}$ sa représentation flottante et $\epsilon_x = x - \tilde{x}$ l'erreur sur $\tilde{x}$.

- **ufp** (Unit in the First Place): $\text{ufp}(\tilde{x}) = min\{i \in \mathbb{N} | 2^{i+1} > |\tilde{x}|\}$
- **ulp** (Unit in the Last Place): $\text{ulp}(\tilde{x}) = \text{ufp}(\tilde{x}) - p + 1$
- **nsb** (Number of Significant Bits): $\text{nsb}(\tilde{x}) = \text{ufp}(\tilde{x}) - \text{ufp}(\epsilon_x)$

Notre outil présente la structure suivante:



Une première phase par interprétation abstraite permet d'obtenir les ufp qui sont ensuite utilisées pour générer les contraintes portant sur les nsb.



Pour chaque anomalie possible, une contrainte est temporairement ajoutée au système. S'il devient insatisfiable alors l'anomalie ne peut pas se produire.

Perspectives

- Enrichir le langage (boucles, conditions, fonctions mathématiques...)
- Expérimenter de nouvelles méthodes de détection (interprétation abstraite)
- Proposer automatiquement des réparations pour les anomalies détectées
  - par transformation du programme [6]
  - par réglage de précision [5]
- Exercer l'outil sur un ensemble de benchmark (**FPBench**)

Références

[1] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*.

[2] D. Ben Khalifa. *Fast and efficient bit-level precision tuning*. Theses, Université de Perpignan, Nov. 2021.

[3] I. Laguna, T. Tirpankar, X. Li, and G. Gopalakrishnan. Fpchecker: Floating-point exception detection tool and benchmark for parallel and distributed hpc. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*, pages 39–50. IEEE, 2022.

[4] M. O. Lam, J. K. Hollingsworth, and G. Stewart. Dynamic floating-point cancellation detection. *Parallel Computing*, 39(3):146–155, 2013.

[5] M. Martel. Floating-point format inference in mixed-precision. In *NASA Formal Methods Symposium*, pages 230–246. Springer, 2017.

[6] P. Panchekha, A. Sanchez-Stern, J. R. Wilcox, and Z. Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015.

Session AFADL & LVP

Prouvez vos coloriages: vérification formelle du coloriage de cache de l'hyperviseur Bao*

Axel Ferréol Laurent Corbin Nikolai Kosmatov

Thales Research & Technology, cortAIx Labs, Palaiseau, France
axel.ferreol@thalesgroup.com
laurent.corbin@thalesgroup.com
nikolai.kosmatov@thalesgroup.com

Contexte et motivation. Les *hyperviseurs* permettent à un système hôte de prendre en charge plusieurs systèmes invités (*machines virtuelles* ou *VMs*) en partageant virtuellement ses ressources, telles que la mémoire et le(s) processeur(s). Déjà intensivement utilisés dans certains domaines (par exemple, les infrastructures cloud), les hyperviseurs deviennent aujourd'hui hautement pertinents pour les *systèmes embarqués critiques* en raison d'un nombre croissant de fonctions et de fonctionnalités nécessaires dans ces systèmes. De nombreuses fonctions ont déjà été ajoutées aux systèmes embarqués, telles que l'assistance à la conduite ou la gestion des capteurs, et d'autres doivent encore être intégrées aujourd'hui, comme des solutions d'intelligence artificielle (IA) pour des systèmes critiques, ou encore des fonctionnalités de divertissement et de connectivité. Dans de nombreux contextes, il n'est pas possible d'ajouter davantage de matériel en raison de contraintes de taille, de poids et de coût. Pour permettre cette intégration, il est nécessaire de partager le même matériel entre plusieurs fonctions (ou systèmes), souvent avec différents niveaux de criticité. Cela peut être réalisé grâce à la *virtualisation*, où chaque système s'exécute sur une VM distincte.

Les ressources matérielles peuvent être partagées par l'hyperviseur de deux manières : (i) le *partage temporel* : chaque machine virtuelle (VM) accède à toutes les ressources à tour de rôle, c'est-à-dire que les VMs sont planifiées (ordonnancées) ; (ii) le *partitionnement* : chaque VM n'accède qu'à la partie des ressources qui lui est dédiée. Le partage temporel nécessite un hyperviseur plus complexe et gourmand en ressources en raison de la fonction ordonnancement. C'est pourquoi les hyperviseurs basés sur le partitionnement (appelés *hyperviseurs statiques*) sont plus largement utilisés dans les systèmes embarqués. Les hyperviseurs statiques allouent toutes les ressources matérielles aux VMs au démarrage de l'hyperviseur, de sorte que chaque ressource soit attribuée à une seule VM. De plus, chaque VM a un accès direct à ses ressources, sans interception par l'hyperviseur, ce qui est particulièrement important pour les systèmes temps réel. Ainsi, un hyperviseur statique

*Cette soumission est un résumé étendu de l'article [1], accepté à FASE 2025.

semble être une solution idéale pour les systèmes à *criticité mixte*, i.e. avec des applications de différents niveaux de criticité.

Cependant, certaines ressources *doivent* être partagées, comme le cache de dernier niveau du processeur (*last-level cache*, or *LLC*), qui est par définition partagé entre plusieurs cœurs, chacun pouvant exécuter une VM différente. Pour résoudre ce problème d'isolation, certains hyperviseurs statiques mettent en œuvre la technique de *coloriage de cache*. L'idée principale est de diviser le cache — au niveau logiciel, sans matériel spécifique — en plusieurs zones, chacune associée à une couleur. Une couleur peut ensuite être assignée à une VM, de sorte que les données des pages mémoire utilisées par cette VM puissent être stockées uniquement dans la zone du cache de la même couleur. La mise en œuvre devient plus complexe et hautement critique, et sa correction est essentielle pour garantir son bon fonctionnement.

Objectifs et résultats. L'objectif de ce travail est la vérification formelle du mécanisme de coloriage de cache implémenté dans Bao [2, 3], un hyperviseur statique open-source utilisé dans les systèmes embarqués. Bien qu'il propose une implémentation élégante et optimisée, son code est également complexe à vérifier formellement en raison de la logique non triviale, des opérations au niveau des bits, des calculs arithmétiques complexes, des niveaux multiples de boucles imbriquées et des listes chaînées qu'il utilise. Au cours de cette étude de cas, nous avons identifié deux bugs subtils dans l'implémentation existante, cassant les garanties attendues, et avons proposé des correctifs [1]. Nous avons écrit une spécification formelle pour les fonctions clés du mécanisme et vérifié leur version corrigée dans la plateforme de vérification FRAMA-C [4, 5]. La preuve nécessite des prédicats soigneusement définis, du code fantôme (*ghost*), des invariants de boucle non triviaux et plusieurs lemmes. Certains objectifs de preuve ne sont pas démontrés par les solveurs automatiques : nous les avons prouvés de manière interactive (dans FRAMA-C ou avec l'assistant de preuve COQ [6, 7]). Nous présentons nos choix de spécification, notre approche de vérification et les résultats obtenus, ainsi que des optimisations potentielles supplémentaires du code actuel (voir [1] pour plus de détail). Un artéfact avec les outils utilisés et le code annoté prouvé est disponible dans [8].

Conclusion et travaux futurs. Les travaux futurs incluent la vérification des versions optimisées du coloriage de cache ainsi qu'une vérification plus large des parties critiques de l'hyperviseur Bao, avec pour objectif à long terme d'atteindre un hyperviseur statique hautement optimisé, formellement prouvé, garantissant des propriétés d'isolation et adapté aux systèmes embarqués modernes.

Remerciements. Ce travail a été soutenu par l'ANR (bourses ANR-22-CE39-0014, ANR-22-CE25-0018). Nous remercions Téo Bernier pour son aide, ainsi que Allan Blanchard, Loïc Correnson and Frédéric Loulergue pour nos discussions enrichissantes.

1. intégrés dans le code réel (commit ee73f7e dans le dépôt GIT [2] fait le 6 janvier 2025).

Références

[1] A. FERRÉOL, L. CORBIN et N. KOSMATOV. « Prove your Colorings : Formal Verification of Cache Coloring of Bao Hypervisor ». In : *Proc. of the 28th International Conference on Fundamental Approaches to Software Engineering (FASE 2025), Held as Part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2025)*. LNCS. To appear. Springer, mai 2025.

[2] *Bao project*. URL : https://github.com/bao-project/bao-hypervisor.

[3] J. MARTINS, A. TAVARES, M. SOLIERI, M. BERTOGNA et S. PINTO. « Bao : A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems ». In : *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*. T. 77. Open Access Series in Informatics (OASIcs). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020, 3 :1-3 :14. ISBN : 978-3-95977-136-8. DOI : 10.4230/OASIcs.NG-RES.2020.3.

[4] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI. « Frama-C : A software analysis perspective ». In : *Formal Asp. Comput.* 27.3 (2015), p. 573-609. DOI : 10.1007/s00165-014-0326-7.

[5] N. KOSMATOV, V. PREVOSTO et J. SIGNOLES, éd. *Guide to Software Verification with Frama-C. Core Components, Usages, and Applications*. Computer Science Foundations and Applied Logic Book Series. Springer, 2024. ISBN : 978-3-031-55607-4. DOI : 10.1007/978-3-031-55608-1.

[6] THE COQ DEVELOPMENT TEAM. *The Coq Proof Assistant*. http://coq.inria.fr,

[7] Y. BERTOT et P. CASTÉRAN, éd. *Interactive Theorem Proving and Program Development : Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004. ISBN : 978-3-540-20854-9.

[8] A. FERRÉOL, L. CORBIN et N. KOSMATOV. *Prove your Colorings : Formal Verification of Cache Coloring of Bao Hypervisor. Companion Artifact for the Paper Submitted to FASE 2025*. 2025. DOI : 10.5281/zenodo.14616331.

BICOQ : une formalisation des bigraphes dans Coq

Résumé long d'article

Cécile Marcon, Cyril Allignol, Celia Picard, Blair Archibald,
Michele Sevegnani, Xavier Thirioux

Résumé

Dans l'article "BICOQ : Bigraphs Formalisation with Coq"[1], nous présentons une librairie [2] Coq/Rocq (open-source) sur les bigraphes [3]. Nous y détaillons le type choisi pour représenter un bigraphe, les définitions d'équivalence, ainsi que des opérateurs de composition, produit tensoriel, produit parallèle et des opérateurs dérivés que nous avons implémentés. Nous donnons les théorèmes -énoncés par Milner- que nous avons prouvés qui nous permettent de nous assurer de la correction de notre implémentation. Notre objectif étant de disposer d'une implémentation complète et vérifiée de la théorie des bigraphes, cette première étape est un jalon important.

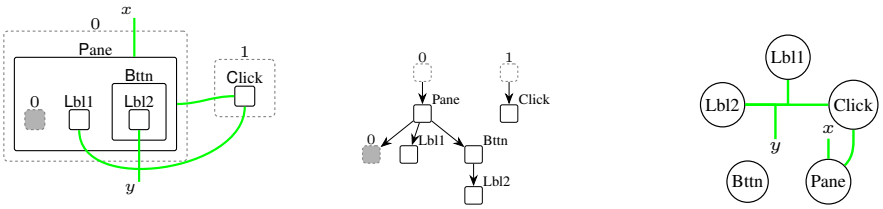
1 Introduction et État de l'art

Les bigraphes, et plus largement les systèmes réactifs bigraphiques (BRS), sont des outils de modélisation introduits par Robin Milner [3]. Ils permettent de modéliser des systèmes ubiquitaires dynamiques avec interactions spatiales et non spatiales (e.g. un bouton sur un écran à la position x, y connecté à la méthode "onClick") [4]-[6]. Dans un BRS, l'état d'un système est représenté par un bigraphe (Figure 1a). Sa dynamique est spécifiée par des règles de réaction qui remplacent un sous-bigraphe par un autre, permettant ainsi de modéliser l'évolution du système.

Les bigraphes ont été implémentés dans des outils comme BigraphER [7] ou BigMC [8], qui permettent de simuler et vérifier des systèmes bigraphiques via des techniques comme le model checking. Cependant, ces outils ne garantissent pas que la théorie sous-jacente est correctement implémentée. Ils se concentrent sur les modèles, sans assurer la conformité de leurs opérations aux définitions de Milner.

D'autres travaux antérieurs ont formalisé des théories similaires à celle des bigraphes (e.g. les catégories ou les graphes) dans Rocq ou d'autres assistants de preuves [9]-[11] (entre autres), mais aucun n'intègre complètement les spécificités des bigraphes (liens ouverts, hyperarêtes, opérateurs partiels etc.).

BiCoq est une formalisation de la théorie des bigraphes dans l'assistant de preuve Rocq [12]. Rocq permet d'exprimer rigoureusement les objets mathématiques et d'extraire du code OCaml vérifié à partir de spécifications formelles. Cela peut être étendu à la compilation vérifiée, en utilisant les bigraphes comme représentation intermédiaire, dans la lignée de projets comme CompCert [13].



(a) Bigraphe (b) Graphe de places (c) Graphe de liens

FIGURE 1 – Bigraphe modélisant une IHM, et ses graphes de place et de liens

2 Définitions

Un bigraphe est composé d'un graphe de places (Figure 1b), qui capture l'imbrication spatiale des composants, et d'un graphe de liens (Figure 1c) qui capture les connexions entre les composants, indépendamment de leur imbrication.

Les bigraphes possèdent également une interface qui permet de les composer et de les juxtaposer. Cette interface est composée, pour le graphe de places, de :

— s *sites* ($s \in \mathbb{N}$), qui sont des « trous » dans le bigraphe qui permettent d'abstraire une partie du sous-bigraphe (e.g. la case grise 0 dans Figure 1a) ;
— r *racines* ($r \in \mathbb{N}$), qui se trouvent au sommet du graphe de place et accueillent tous les nœuds (e.g. les cases en pointillés 0 et 1 dans Figure 1a).

et pour le graphe de liens de :

— *innernames*, un ensemble de noms i correspondant à des ports ouverts, et représentés par une arrête brisée vers le bas (e.g. y dans Figure 1a) ;
— *outernames*, un ensemble de noms o correspondant à des arêtes ouvertes, et représentées par une arrête brisée vers le haut(e.g. x dans Figure 1a).

Cette interface est divisible en une face interne $\langle s, i \rangle$ et une face externe $\langle r, o \rangle$. Ces faces permettent de définir les bigraphes comme des morphismes $\langle s, i \rangle \to \langle r, o \rangle$, mais également des flèches dans une catégorie monoïdale partielle symétrique (*spm*-catégorie) avec l'équivalence de support épuré $\eqslantless$ (Section 4) et les opérateurs $\circ$ et $\otimes$ (Section 5).

3 Formalisation des bigraphes dans Rocq

Nous définissons les bigraphes dans un `Record` comme un type dépendant `bigraph s i r o`, où `s` et `r` sont des `nat`, et `i` et `o` sont des listes sans doublons (`NoDupList`) représentant les inner et outer names. Ces ensembles sont représentés par des listes pour faciliter l'implémentation des algorithmes.

Le type `bigraph s i r o` représente donc un bigraphe $\langle s, i \rangle \to \langle r, o \rangle$. Il contient :

— un ensemble fini de nœuds `node` et un ensemble fini d'arêtes `edge` représentés via `finType` de MathComp [14],

— une fonction `control` associant à chaque nœud un contrôle qui détermine le nombre de ses ports
— une fonction `parent` définissant le graphe de place, avec une preuve d'acyclicité `ap` (exprimée grâce à `Acc`),
— une fonction `link` définissant le graphe de liens, reliant ports (ou port ouvert) et arêtes (ou arêtes ouvertes).

4 Notions d'équivalence

L'égalité native de Rocq est l'égalité de Leibniz qui est trop stricte pour comparer deux bigraphes ayant des structures identiques mais des identifiants internes différents. BICOQ propose donc deux notions d'équivalence :

— une équivalence de support ($\simeq$) : deux bigraphes sont équivalents si et seulement s'il existe des bijections entre leurs nœuds et leurs arêtes respectant les relations de structure du bigraphe ;
— une équivalence de support épuré ($\eqslantless$) : identique à $\simeq$ mais ignorant les arêtes inactives (non connectées).

Nous avons démontré que ces définitions sont des relations d'équivalence (i.e. elles sont réflexives, symétriques et transitives). Nous pouvons ainsi les utiliser pour raisonner sur l'égalité structurelle dans les démonstrations Rocq.

5 Opérateurs et axiomes de *spm*-catégorie

BICOQ implémente deux opérateurs principaux des *spm*-catégorie, pour lesquels nous avons prouvés qu'ils sont conformes aux lois de ces catégories :

— La composition ($\circ$) connecte deux bigraphes si leurs interfaces sont compatibles (i.e. $\forall b_1 : \langle s1, i1\rangle \to \langle r1, o1\rangle, \forall b_2 : \langle s2, i2\rangle \to \langle s1, i1\rangle, b_1 \circ b_2$ is defined). Elle est définie comme l'union disjointe des nœuds/arêtes, avec des fonctions parent et lien composées. Les lois d'associativité et d'identité sont démontrées pour nos définitions d'équivalence de support.
— Le produit tensoriel ($\otimes$) juxtapose parallèlement deux bigraphes dont les interfaces sont disjointes (d'où le terme de « partiel » dans la catégorie). Les lois d'identité, associativité et distributivité sont prouvées.

En y ajoutant quelques démonstrations de lemmes sur des bigraphes appelés *symétries*, nous prouvons ainsi que nos bigraphes respectent les axiomes d'une *spm*-catégorie et gagnons en confiance dans notre implémentation.

Nous proposons également 3 autres opérateurs que nous ne détaillons pas ici mais qui ont également été prouvés corrects : le produit parallèle ($\|$), le produit de fusion ($|$) et l'imbrication ($\cdot$).

6 Conclusion et travaux à venir

Avec BICOQ, nous proposons une formalisation rigoureuse de la théorie des bigraphes dans Rocq qui respecte les axiomes et la structure catégorique. Grâce à cette base, il devient possible de construire des outils fiables, de vérifier des systèmes interactifs complexes, et d'intégrer les bigraphes dans des chaînes de compilation ou de vérification certifiées.

Pour la suite, nous prévoyons d'implémenter un algorithme de recherche de motifs dans les bigraphes afin de pouvoir formaliser les règles de réécriture.

Références

[1] C. MARCON, C. PICARD, C. ALLIGNOL et al., « BiCoq :Bigraphs Formalisation with Coq, » in *Proceedings of the 40th ACM/SIGAPP Symposium on Applied Computing*, sér. SAC '25, Catania, Sicily : Association for Computing Machinery, 2025. adresse : https://doi.org/10.1145/3672608.3707824.

[2] C. MARCON et X. THIRIOUX, *BiCoq : Modeling bigraphs with Coq*, version 1.2, nov. 2024.

[3] R. MILNER, *The space and motion of communicating agents*. Cambridge University Press, 2009.

[4] V. DANOS, J. FERET, W. FONTANA et al., « Graphs, Rewriting and Pathway Reconstruction for Rule-Based Models, » in *FSTTCS 2012 - IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*, sér. LIPIcs, t. 18, 2012.

[5] H. SAHLI, T. LEDOUX et É. RUTTEN, « Modeling self-adaptive fog systems using bigraphs, » in *Software Engineering and Formal Methods : SEFM 2019 Collocated Workshops : CoSim-CPS, ASYDE, CIFMA, and FOCLASA, Revised Selected Papers 17*, Springer, 2020.

[6] L. BIRKEDAL, S. DEBOIS, E. ELSBORG et al., « Bigraphical models of context-aware systems, » in *Foundations of Software Science and Computation Structures : 9th International Conference, FOSSACS 2006, ETAPS 2006, Vienna, Austria. Proceedings 9*, Springer.

[7] M. SEVEGNANI et M. CALDER, « BigraphER : Rewriting and analysis engine for bigraphs, » in *Computer Aided Verification : 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, Springer, 2016, p. 494-501.

[8] G. PERRONE, S. DEBOIS et T. HILDEBRANDT, « A model checker for Bigraphs, » *Proceedings of the ACM Symposium on Applied Computing*, mars 2012.

[9] N. GASPAR, L. HENRIO et E. MADELAINE, « Bringing Coq into the World of GCM Distributed Applications, » *Int. J. Parallel Program.*, t. 42, n° 4, p. 643-662, août 2014.

[10] C. DOCZKAL et D. POUS, « Graph theory in Coq : Minors, treewidth, and isomorphisms, » *Journal of Automated Reasoning*, t. 64, p. 795-825, 2020.

[11] T. RIDGE, *Graphs and Trees in Isabelle/HOL*, 2005.

[12] Y. BERTOT, « A Short Presentation of Coq, » in *Theorem Proving in Higher Order Logics*, Berlin, Heidelberg : Springer Berlin Heidelberg, 2008, p. 12-16.

[13] X. LEROY, S. BLAZY, D. KÄSTNER et al., « CompCert- a formally verified optimizing compiler, » in *ERTS 2016 : Embedded Real Time Software and Systems*.

[14] A. M. et E. TASSI, *The Mathematical Components team : Mathematical components*. Zenodo, 2022.

Une sémantique mécanisée d'un langage FRP avec effets

Résumés longs

Jordan Ischard, Frédéric Dabrowski, Jules Chouquet et Frédéric Loulergue

Univ. Orléans, INSA CVL, LIFO EA 4022, Orléans, France

Introduction

La programmation fonctionnelle réactive (FRP) est un paradigme de programmation fonctionnelle conçu pour simplifier la programmation de systèmes interagissant continuellement avec leur environnement. YAMPA [1], une bibliothèque HASKELL, permet de construire des *fonctions de signal* qui traitent un flux d'entrée de manière synchrone afin de produire un flux de sortie. Bien que cette bibliothèque facilite la production de programmes concis et robuste, la gestion des entrées et sorties reste fastidieuse.

WORMHOLES [5, 6] propose une première solution en 2012, suivie, quelques années plus tard, par d'autres [4, 3] dont MSF. WORMHOLES propose d'interagir avec l'environnement dans le programme via deux nouveaux opérateurs : *rsf* et *wormhole* qui se basent sur un concept de ressource. Une ressource représente une interaction avec l'environnement et le langage limite l'usage à une occurrence lors d'un instant logique. Cette contrainte garantit l'hypothèse d'exécution immédiate du point de vue de l'environnement.

Il existe deux types de ressources : libre et liée. Le premier est directement en contact avec l'environnement extérieur alors que le second est interne au programme. Par conséquent, la contrainte d'utilisation unique pour les ressources liées semble pouvoir être retirée. Ceci est la motivation première des travaux présentés dans l'article [2] que nous résumons ici. En effet, avant de proposer une extension du langage, il convient de s'assurer de la solidité des bases de celui-ci. Nous avons trouvé des problèmes dans la formalisation établit par D. Winograd-Cort et P. Hudak, et par conséquent, nous avons proposé des nouvelles définitions pour la syntaxe et la sémantique du langage et nous les avons formalisées en ROCQ.

Problèmes et corrections

La formalisation du langage WORMHOLES contient trois problèmes principaux :

1. l'absence du point-fixe dans la définition de la syntaxe ;
2. la possibilité d'échappement d'une ressource liée en dehors de sa portée dans le système de types ;
3. la non-satisfaction de la propriété de progression par la sémantique dynamique.

Nous proposons d'approfondir légèrement les deux premiers points, le troisième étant déjà assez explicite.

Récursion

La syntaxe de WORMHOLES est définie informellement dans l'article original [5] comme un lambda-calcul étendu, avec une valeur de base, des paires, des projections et une récursion, en plus des fonctions de signal et des deux nouveaux opérateurs. Dans la définition formelle, il n'y a pas de terme représentant le point-fixe. Par conséquent, les lemmes et théorèmes proposés dans l'article original ne le prennent pas en compte.

L'ajout d'un point-fixe n'est pas anodin dans un langage. En effet, il rend invalide la propriété de normalisation du langage, c'est-à-dire que l'on ne peut plus établir que tout programme termine. Cependant, sa présence est bénéfique, car il fait gagner en expressivité. Par conséquent, nous avons ajouté un terme *fix* dans la syntaxe représentant le point fixe.

Échappement

Le système de type de WORMHOLES garantit l'utilisation unique des ressources en conservant, dans le type des fonctions de signal, l'ensemble des ressources utilisées. Les ressources liées sont celles associées à un opérateur *wormhole*, dans ce cas, le système de type est censé garantir qu'elles ne peuvent pas être utilisées en dehors de leur portée.

Cependant, il est possible de faire sortir une ressource liée en utilisant la définition présentée dans l'article original. Par conséquent, nous avons ajouté les contraintes minimales dans la règle qui établit le bon typage d'une expression avec l'opérateur *wormhole* afin empêcher cet échappement.

Formalisation en ROCQ

Pour nous assurer que nos corrections garantissent les propriétés établies dans l'article original, nous avons formalisé notre variante de WORMHOLES. Le projet est disponible dans le répertoire :

`https://github.com/JordanIschard/Mechanized-Wormholes`

Le travail de formalisation est d'environ 5 kLdC (mille lignes de code) : 2KLdC de spécifications (définitions, lemmes, théorèmes, ...) et 3KLdC de démonstrations.

Conclusion et travaux futurs

Nous avons suggéré de nouvelles définitions pour la syntaxe et la sémantique du langage WORMHOLES, créant une variante de celui-ci. Nous avons démontré que les théorèmes de l'article original sont satisfaits par notre variante. Finalement, nous avons utilisé l'assistant de preuve ROCQ pour s'assurer de la véracité de nos corrections.

Ce résultat nous permet de revenir à la motivation initiale de ce travail : tenter de relâcher un maximum les restrictions sur les lectures et écritures. En effet, nous pensons que les restrictions sur les ressources liées pourraient être assouplies sans invalider les propriétés du langage. Cela permettrait de rendre ce dernier plus permissif et d'attirer plus d'utilisateurs. Une autre piste possible est d'utiliser les concepts introduits par WORMHOLES pour définir une bibliothèque FRP dans OCAML. À notre connaissance, il n'existe pour le moment pas de bibliothèque OCAML qui permette d'intégrer les concepts de Yampa aux traits impératifs inhérents à OCAML.

References

[1] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon L. Peyton Jones, editors, *Advanced Functional Programming*, volume 2638 of *LNCS*, pages 159–187. Springer, 2002.

[2] Jordan Ischard, Frederic Dabrowski, Jules Chouquet, and Frédéric Loulergue. A Mechanized Formalization of an FRP Language with Effects. In ACM, editor, *ACM Symposium on Applied Computing (SAC)*, Sicily, Italy, March 2025.

[3] Ivan Perez, Manuel Bärenz, and Henrik Nilsson. Functional reactive programming, refactored. In *International Symposium on Haskell*, Haskell 2016, page 33–44, New York, NY, USA, 2016. ACM.

[4] Atze van der Ploeg and Koen Claessen. Practical principled FRP: forget the past, change the future, frpnow! In Kathleen Fisher and John H. Reppy, editors, *International Conference on Functional Programming (ICFP)*, pages 302–314. ACM, 2015.

[5] Daniel Winograd-Cort and Paul Hudak. Wormholes: Introducing effects to FRP. In *International Symposium on Haskell*, New York, NY, USA, 2012. ACM.

[6] Daniel Winograd-Cort, Hai Liu, and Paul Hudak. Virtualizing real-world objects in FRP. In *Practical Aspects of Declarative Languages (PADL)*, volume 7149 of *LNCS*, pages 227–241. Springer, 2012.

Langage Chips : Modélisation et contrôle de systèmes distribués à base de composants

Anna Gallone
*Université Marie et Louis Pasteur, CNRS UMR 6174, Institut FEMTO-ST, F-25000 Besançon, France**

Résumé

Ce document présente Chips, un langage de programmation dédié à la modélisation de systèmes à base de composants. Son objectif est de permettre la modélisation d'architectures logicielles auxquels une forme de contrôle en temps réel serait appliquée. Pour cela, Chips intègre des constructions syntaxiques facilitant l'expression des formalismes mathématiques issus de la théorie du contrôle. son niveau d'abstraction permet également l'introduction des concepts de programmation agrégée telle la généralisation de l'opération de *broadcast* ou d'accumulation pour être appliqué à des systèmes logiciels distribués. En compilant Chips vers d'autres langages comme BIP, il sera possible de vérifier par preuve ou *model-checking* des propriétés de résilience des systèmes conçus sur des automates équivalents aux programmes initiaux.

Mots-clefs : Chips †, Systèmes à Composants, Théorie du Contrôle, Modélisation, BIP ‡, Programmation Agrégée

I. Introduction

L'omniprésence des appareils connectés dans notre environnement permet d'envisager des applications qui tireraient parti de toutes les capacités de ces appareils. Que ce soit leurs capteurs (gyroscope, thermomètre, caméra...), leurs actuateurs (haut-parleur, LEDs, écrans...) ou même leur puissance de calcul associée, chacune de ces ressource peut-être mise en commun pour la réalisation de tâches. Cependant, la coordination de tous ces processus logiciels est complexe et leurs interractions fait emerger des comportements globaux qu'il est difficile de prévoir. Le terme de système complexe sera utilisé ensuite pour décrire ces ensembles de processus communiquants, chaque processus pouvant lui-même être réalisé par des sous-systèmes (potentiellement complexes eux aussi). Les appareils composant de tels systèmes sont en perpétuel mouvement, certains y entrent, d'autres en sortent. Cela donne lieu à tout une variété de configurations. De plus, les appareils sont sujets à des modifications de leur environnement (température, vent, bande passante, etc). Si un logiciel compte tirer avantage de toutes les capacités de ce système complexe, il doit pouvoir s'adapter à la fois aux changements de configurations et à l'environnement [1]. Grâce à la théorie du contrôle, il n'est pas requis de maitriser toute cette complexité pour atteindre cet objectif.

La théorie du contrôle est une discipline de l'ingénierie qui vise à asservir en temps réel un système à son environnement : le système mesure en continu un ensemble de signaux relatifs à son environnement pour réagir aux événements tout en poursuivant un objectif donné. Cela permet par exemple de réguler la vitesse d'une voiture ou d'optimiser la distribution en eau dans un réseau citadin. Cependant, même si ces concepts sont familiers à la communauté scientifique, ils demeurent difficiles à appliquer dans le domaine de l'informatique [2]. Parmi les obstacles à cette application, on peut citer la difficulté d'établir des modèles ou équations pour les systèmes complexes étudiés ici. En effet, l'espace des configurations que peuvent prendre

∗. Ces travaux sont supportés par la subvention ANR-23-CE25-0004 (ADAPT).
†. Control of Hierarchical Interconnected Programable Systems
‡. Behaviour Interaction Priority

temporairement un réseau de processus grandit de façon exponentielle avec le nombre de sous-systèmes sous contrôle. Le design d'un seul contrôleur pour toutes ces situations est inapproprié, d'où l'intérêt de développer des outils de synthèse de contrôleurs [3], et de chercher à les appliquer en temps réel [4] pour améliorer leur qualité de service.

L'application de la théorie du contôle profiterait à de nombreux domaines de l'informatique, comme en témoignent la variété des recherches intégrant des faccultés d'auto-adaptation à leurs systèmes : Services Web, IoT, Réseaux, etc [1]. Dans le cas des systèmes à composants, d'autres problématiques émergent telles que celle de synchronisation ou encore d'allocation optimale des ressources [5]. Parallèlement, le domaine de l'*Internet of Things*(IoT) a vu naître l'*Aggregate Programming*, un paradigme pour piloter des systèmes où une multitude de processus interragissent en continu [6][7]. Le langage Chips se propose comme une application de la théorie du contrôle à des systèmes à composants qui intègre des notions d'*Aggregate Programming* afin de réduire la complexité apparente des objets qu'il permet de modéliser.

La suite de ce document présente une description détaillée des challenges que le langage Chips vise à relver en section II, puis décrit en section III les moyens (qui seront) mis en place pour efficacement atteindre ces objectifs.

II. Problématique

La théorie du contrôle est un formalisme efficace pour asservir un système simple à son environnement. Dans un système complexe, il est difficile d'appliquer directement la théorie du contrôle à l'ensemble des processus contrôlés car ceux-ci peuvent avoir des effets les uns sur les autres. Modéliser formellement de tels systèmes permet d'anticiper certains de leurs dysfonctionnements dès l'étape de conception via des méthodes et outils de *model checking*. Mais si un modèle peut s'avérer efficace pour concevoir un système robuste, réaliser ce modèle demeure une tâche fastidieuse (pour la même raison qu'un système complexe est sujet à de nombreux dysfonctionnements potentiels). Le langage Chips se positionne vis-à-vis de la conception de systèmes complexes comme un langage de description de modèles. Avec un typage fort permettant d'assurer le formalisme de la théorie du contrôle, le but de Chips est d'automatiser la génération de modèles sur lesquels effectuer des simulations numériques et vérifications de propriétés. Après des transformations assurant la conservation de ces propriétés vers d'autres langages, ces modèles Chips deviendraient les premiers artefact d'une chaîne de production d'applications distribuées correctes par construction.

III. Approche Proposée

L'objectif du langage Chips est multiple, et chacune des sous-sections suivantes s'intéresse à un aspect différent de son développement :

- Formaliser la description de modèles pour des applications distribuées (sous-section III.A)
- Générer des modèles équivalents dans un autre langage, en l'occurence, sous la forme d'automates BIP (sous-section III.B)
- Vérifier la résilience de ces applications via *model-checking*, preuve, ou à défaut, les valider par simulations (sous-section III.C)

A. Formalisme du langage

Pour permettre d'adapter efficacement les formalismes de la théorie du contrôle, Chips adopte une syntaxe proche d'autres langages synchrones comme Lustre [8] ou Heptagon/BZR [9] (voir Listing 1). Chaque composant logiciel correspond à un bloc fonctionnel avec des entrées et des sorties qu'il convient de connecter pour réaliser un modèle. Pour permettre plus de souplesse dans la définition des composants, Chips se distingue des autres langages mentionnés en ce fait qu'il n'est que partiellement synchrone : La logique interne qui convertit les signaux d'entrée en signaux de sortie peut-être définie par un algorithme séquentiel et

l'usage de variables internes, un formalisme plus proche de langages de programmation générique comme le C. Pour définir un système, Chips introduit 3 types de fonctions : le type *pure*, pour factoriser des expressions sous des symboles plus expressifs, le type *virtual* pour signifier qu'il s'agit d'une suite d'opérations requérant un espace de stockage pour être implémentée, et le type *physical* pour modéliser les composants qui font intéragir les flux de données binaires avec l'environnement par le biais de capteurs ou d'actuateurs. Ce dernier type est à mettre en lien avec des specifications du support physique utilisé pour permettre de déterminer la valeur du mot réservé *dt*, le pas de temps simulé, lors de la compilation du modèle.

Listing 1 Implémentation partielle d'un modèle de ventilateur en Chips

```
pure errorf(float expected, float received)
        -> (expected - received)

virtual pid_controller(float expected,
                       float received)
init {
    float derivative = 0; float integral = 0;
    float lasterror = 0; float error = 0;
    float out = pid(error, integral, derivative);
} then {
    error = errorf(expected, received);
    derivative = (lasterror-error)/dt;
    integral = integral+error*dt;
    lasterror = error;
    out = pid(error, integral, derivative);
} -> (out)

physical fan(float power_required)
init {
    float power = 0;
    float wind_measured = 0;
} then {
    power = max(power_required, this.power_received);
    wind_measured = function_modeling_physics(power);
} -> (wind_measured)

SYSTEM {
    pid_controller virtual_pid;
    fan the_fan;
    link virtual_pid to the_fan;
    // target value 5 chosen arbitrarily
    virtual_pid.in(5, the_fan.out);
    the_fan.in(virtual_pid.out);
}
```

Le langage sera ensuite à faire évoluer avec divers éléments. Par des structures de données plus avancées et des structures de contrôle comme des boucles, on compte en améliorer l'expréssivité et permettre de modéliser des systèmes de plus grande échelle. Ces ajouts seraient ensuite à mettre en lien avec des primitives de langage plus avancées qui introduiraient des opérations issues de l'aggregate programming. On serait alors en capacité de modéliser des systèmes s'adaptant en temps réel à des changements de configuration.

B. Génération d'automates équivalents

Afin que les modèles décrits ne servent pas simplement d'outils de descriptions, il convient de les transformer sous une forme permettant des expériementations. À cette fin, le choix a été fait de compiler les modèles Chips en langage BIP [10]. BIP est un framework de modélisation dont la compilation vers C++ est correcte par construction. Il permet de décrire des systèmes à composants en représentant les processus sous la forme d'automates auxquels sont associées des variables. Pour faire interagir ces processus et faire évoluer leurs variables, BIP définit la notion de connecteurs, des composants qui définissent quelles transitions ont lieu simultanément entre des automates distincts. Par la description de composants complexes (automates connexes regroupés sous un même identifiant), BIP rend possible la conception de systèmes logiciels hierarchiques, donc parfaitement appropriés aux systèmes complexes auxquels s'intéresse Chips. An accord avec les principes de l'ingénierie des modèles, un méta-modèle de Chips est en cours de développement. Ce méta-modèle sera convertible vers le méta-modèle de BIP grâce au langage ATLAS [11]. On déterminera alors un ensemble de règles de transformations pour traduire n'importe quelle description Chips en une implémentation à base d'automates d'état finis en BIP.

C. Simulations et preuve de propriétés

Une fois des automates équivalents aux programmes initiaux générés, deux pistes sont à explorer pour valider les modèles développés :

- Le langage BIP est doté d'un moteur de simulation. Il est donc possible d'explorer l'espace des executions possible des systèmes pour les valider, soit par explorations exhaustives de l'ensemble des états possibles des systèmes, soit par une approche statistique et des méthodes de *model checking* [12].
- Des propriétés de résilience peuvent être également prouvées sur les automates directement. On peut notamment employer les preuves relatives aux automates quantitatifs sur les systèmes décrits par BIP pour vérifier la convergence de réseaux de processus vers un état désiré [13].

À plus long terme, BIP donne l'occasion de générer automatiquement des implémentation réelles des programmes modélisés pour tout types d'appareils en conservant leur aspect distribué [14].

Références

[1] Alfonso, I., Garcés, K., Castro, H., and Cabot, J., "Self-adaptive architectures in IoT systems : a systematic literature review," *Journal of Internet Services and Applications*, Vol. 12, No. 1, 2021, pp. 1–28. https://doi.org/10.1186/s13174-021-00145-8, URL https://link.springer.com/article/10.1186/s13174-021-00145-8, number : 1 Publisher : SpringerOpen.

[2] et al., A. F., "Software Engineering Meets Control Theory," *10th IEEE/ACM International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2015, Florence, Italy, May 18-19, 2015*, edited by P. Inverardi and B. R. Schmerl, IEEE Computer Society, 2015, pp. 71–82. https://doi.org/10.1109/SEAMS.2015.12, URL https://doi.org/10.1109/SEAMS.2015.12.

[3] Arioka, Y., Yamauchi, T., and Tei, K., "Pre-controller Synthesis for Runtime Controller Synthesis," *2023 IEEE 13th Int. Conf. on Control System, Computing and Engineering (ICCSCE)*, 2023, pp. 161–166. https://doi.org/10.1109/ICCSCE58721.2023.10237143, URL https://ieeexplore.ieee.org/abstract/document/10237143/authors.

[4] Shi, H., Dong, W., Li, R., and Liu, W., "Controller Resynthesis for Multirobot System When Changes Happen," *Computer*, Vol. 53, No. 12, 2020, pp. 69–79. https://doi.org/10.1109/MC.2020.3017343, URL https://ieeexplore.ieee.org/document/9269907.

[5] Ben Halima, R., Kallel, S., Gaaloul, W., Maamar, Z., and Jmaiel, M., "Toward a correct and optimal time-aware cloud resource allocation to business processes," *Future Generation Computer Systems*, Vol. 112, 2020, pp. 751–766. https://doi.org/10.1016/j.future.2020.06.018, URL https://www.sciencedirect.com/science/article/pii/S0167739X19333679.

[6] Beal, J., Pianini, D., and Viroli, M., "Aggregate Programming for the Internet of Things," *Computer*, Vol. 48, No. 9, 2015, pp. 22–30. https://doi.org/10.1109/MC.2015.261, URL https://doi.org/10.1109/MC.2015.261.

[7] Casadei, R., Viroli, M., Aguzzi, G., and Pianini, D., "ScaFi : A Scala DSL and Toolkit for Aggregate Programming," *SoftwareX*, Vol. 20, 2022, p. 101248. https://doi.org/10.1016/J.SOFTX.2022.101248, URL https://doi.org/10.1016/j.softx.2022.101248.

[8] Halbwachs, N., "A synchronous language atwork : the story of lustre," *Proceedings. Second ACM and IEEE Int. Conf. on Formal Methods and Models for Co-Design, 2005. MEMOCODE '05.*, IEEE, Verona, Italy, 2005, pp. 3–12. https://doi.org/10.1109/MEMCOD.2005.1487884, URL http://ieeexplore.ieee.org/document/1487884/.

[9] Delaval, G., Marchand, H., and Rutten, E., "Contracts for modular discrete controller synthesis," *SIGPLAN Not.*, Vol. 45, No. 4, 2010, p. 57–66. https://doi.org/10.1145/1755951.1755898, URL https://doi.org/10.1145/1755951.1755898.

[10] Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., and Sifakis, J., "Rigorous Component-Based System Design Using the BIP Framework," *IEEE Softw.*, Vol. 28, No. 3, 2011, pp. 41–48. https://doi.org/10.1109/MS.2011.27, URL https://doi.org/10.1109/MS.2011.27.

[11] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I., "ATL : A model transformation tool," *Science of Computer Programming*, Vol. 72, No. 1, 2008, pp. 31–39. https://doi.org/10.1016/j.scico.2007.08.002, URL https://www.sciencedirect.com/science/article/pii/S0167642308000439.

[12] Legay, A., Delahaye, B., and Bensalem, S., "Statistical Model Checking : An Overview," 2010. https://doi.org/10.1007/978-3-642-16612-9_11, URL https://inria.hal.science/inria-00591593.

[13] Boker, U., Henzinger, T. A., Mazzocchi, N., and Saraç, N. E., "Safety and Liveness of Quantitative Properties and Automata," , Feb. 2025. https://doi.org/10.48550/arXiv.2307.06016, URL http://arxiv.org/abs/2307.06016, arXiv :2307.06016 [cs].

[14] Jaber, M., "Centralized and Distributed Implementations of Correct-by-construction Component-based Systems by using Source-to-source Transformations in BIP," phdthesis, Université Joseph-Fourier - Grenoble I, Oct. 2010. URL https://theses.hal.science/tel-00531082.

Session AFADL II

Découverte de processus probabiliste avec des arbres de processus stochastiques

Résumés longs

Pierre Cry

Résumé

La découverte de processus vise à extraire automatiquement un modèle formel capable de reproduire les comportements observés d'un système à partir de journaux d'événements. Ces journaux décrivent des séquences d'événements qui se produisent avec une certaine fréquence, reflétant ainsi la nature stochastique du processus sous-jacent. Dans ce travail, nous adoptons une extension stochastique du formalisme des arbres de processus comme modèle cible pour la découverte de processus. Nous montrons que ces modèles offrent des avantages par rapport aux réseaux de Petri stochastiques, en particulier pour l'identification des paramètres influençant la probabilité des comportements observés. Ce document est un résumé long du papier « *Probabilistic Process Discovery with Stochastic Process Trees* », accepté à EAI VALUETOOLS 2024, co-écrit avec András Horváth et Paolo Ballarini.

1 Introduction

Le *process mining* [8] vise à découvrir des modèles formels capables de reproduire le comportement dynamique d'un processus à partir de journaux d'événements. Ces journaux consistent en des séquences temporellement ordonnées d'activités observées sur le système, appelées traces. L'ensemble de ces traces constitue un langage. Les algorithmes de découverte génèrent des modèles, souvent représentés sous forme de réseaux de Petri, afin de capturer les relations causales et les interdépendances concurrentielles entre les activités observées. La qualité de ces modèles est évaluée à l'aide de métriques spécifiques. Par exemple, l'adéquation mesure la capacité du modèle à reproduire fidèlement les comportements observés dans le journal d'événements. Parmi les approches de découverte de processus, la famille d'algorithmes *Inductive Miner* [5] se distingue par sa capacité à générer des arbres de processus avec une parfaite adéquation. Ces structures hiérarchiques combinent des opérateurs logiques (séquence, choix, concurrence, et boucle) pour modéliser explicitement les dépendances entre activités. Ces arbres offrent une représentation structurée facilitant la traduction en réseaux de Petri.

Les journaux d'événements, en plus d'enregistrer des séquences d'activités discrètes, capturent leurs fréquences d'occurrence, définissant ainsi un langage sto-

chastique. Ces informations quantitatives sont nécessaires pour l'inférence de modèles probabilistes : l'objectif est d'assurer que la distribution statistique des modèles inférés corresponde à celle observée dans le journal original. La découverte de processus stochastiques se fait en général en deux temps : un premier processus non stochastique est d'abord découvert sans tenir compte des fréquences des traces observées, et dans un second temps, le processus découvert est transformé en un processus stochastique en attachant des paramètres au modèle. Des techniques d'optimisation permettent alors de calculer les valeurs de ces paramètres. La complexité de l'étape d'optimisation dépend du nombre de ces paramètres.

A notre connaissance, la majorité des approches stochastiques s'appuient sur des réseaux de Petri [3] [2] [1] [6]. L'optimisation des paramètres est rendue complexe lorsque l'algorithme de découverte génère un réseau comportant de nombreuses transitions. Pour pallier cette difficulté, une solution consiste à exploiter une représentation plus compacte sous forme d'arbres de processus. Nous introduisons la notion d'arbre de processus stochastique comme extension des arbres de processus classiques, incorporant des paramètres probabilistes pour générer un langage stochastique.

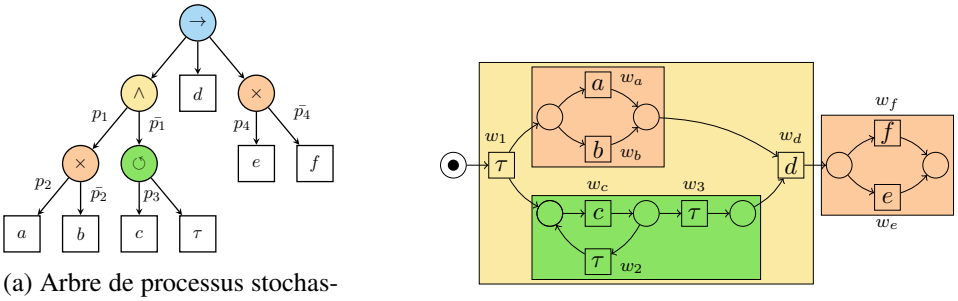
Ce papier [4] introduit les arbres de processus stochastiques comme une alternative compacte aux réseaux de Petri pour la découverte de modèles stochastiques. Cette représentation simplifie à la fois le calcul des langages stochastiques et l'optimisation des paramètres. Nous attribuons des probabilités aux nœuds d'un arbre obtenu à l'aide d'un algorithme de découverte standard. En nous affranchissant de la manipulation directe des réseaux de Petri, nous réduisons significativement la dimension de l'espace de recherche associé à l'inférence des paramètres.

2 Arbres de Processus stochastiques à travers un exemple

La Figure 1a illustre un arbre de processus stochastique pour le langage :

$$L_1 = \{\langle a,c,d,e\rangle^{65}, \langle b,c,d,e\rangle^{30}, \langle c,b,d,f\rangle^{20}, \\ \langle a,c,c,d,e\rangle^{15}, \langle c,b,d,f\rangle^{5}, \langle c,c,a,d,f\rangle^{2}\}$$

Modélisation des relations entre les activités. En analysant les traces de L_1, on observe que chaque exécution contient systématiquement l'activité d, toujours précédée d'une unique activité parmi a et b. L'activité c précède également d, et apparaît une ou deux fois, entrelacée avec a ou b. Après d, l'exécution se termine toujours par e ou f, qui sont mutuellement exclusives. Cette structure impose plusieurs contraintes de modélisation dans l'arbre de processus stochastique de la Figure 1a. Deux nœuds de conflit sont nécessaires pour capturer les relations exclusives entre a et b, ainsi qu'entre e et f. Une boucle est introduite pour modéliser la répétition éventuelle de c. Un nœud parallèle exprime la concurrence entre les occurrences de c et celle de a ou b. Enfin, un nœud de séquence structure l'ordre des activités : d'abord a ou b en concurrence avec un ou plusieurs c, puis d, puis enfin e ou f.



(a) Arbre de processus stochastique

(b) Réseaux de Petri stochastique

FIGURE 1 – Modélisation du langage $L1$

Chacune de ces relations logiques est représentée dans le réseau de Petri stochastique de la Figure 1b par une structure dédiée. Le choix exclusif entre a et b (et entre e et f) est modélisé par des places partagées : le jeton unique y est consommé par une transition, empêchant l'autre. La répétition de c est assurée par une transition muette, réinitialisant l'état du réseau pour permettre de nouvelles occurrences. Cette transition est rendue nécessaire par la contrainte d'unicité des étiquettes de transition. La transition étiquetée par c étant unique, il faut revenir à un marquage antérieur pour permettre sa réactivation. La synchronisation partielle entre la boucle de c et le conflit a/b est gérée par une transition muette de bifurcation, divisant le flux en deux sous-processus parallèles. Ces branches se rejoignent ensuite via la transition d, garantissant leur complétion avant de poursuivre l'exécution. L'ordre des activités est respecté par l'agencement séquentiel des transitions.

Modélisation des fréquences des traces. Chaque trace est associée à une fréquence qui, après normalisation, permet de représenter sa probabilité d'apparition et de l'intégrer dans le modèle stochastique. L'attribution des paramètres stochastiques dans l'arbre de processus consiste à associer des probabilités aux nœuds, en fonction de leur nature. La séquence ne nécessite aucun paramètre stochastique, car elle définit uniquement l'ordre d'exécution des sous-arbres. Dans un conflit, chaque fils reçoit une probabilité définissant sa sélection relative aux autres sous-arbres. Le parallèle attribue à chaque fils une probabilité définissant l'exécution d'un élément de l'enchevêtrement. La boucle utilise un seul paramètre pour la probabilité de répétition, son complémentaire déterminant la probabilité de sortie. Pour réduire le nombre de paramètres dans le conflit et le parallèle, la probabilité du dernier arc peut être déduite comme complémentaire de la somme des autres. Ainsi, dans un arbre binaire, chaque nœud n'a qu'un seul paramètre, sauf les séquences, qui n'en nécessitent aucun. Pour la Figure 1a, l'arbre ne nécessite que quatre paramètres pour lui permettre de traduire un langage stochastique.

En revanche, dans les réseaux de Petri, la stochasticité est introduite en attribuant un poids à chaque transition. Lorsque plusieurs transitions sont activables simultanément, leur probabilité de déclenchement est déterminée par le rapport

entre leur poids et la somme des poids des transitions activables à cet instant. Cette approche nécessite un plus grand nombre de paramètres pour un même résultat. Par exemple, le réseau de Petri de la Figure 1b comporte 9 transitions.

3 Méthode de découverte d'arbres de processus stochastique

Nous avons adapté des méthodes de découverte indirecte de réseaux de Petri stochastiques, fondées sur une optimisation paramétrique, en les appliquant à des arbres de processus. Notre approche repose sur la minimisation d'une métrique de distance stochastique entre le langage stochastique, estimé par simulation, de l'arbre de processus candidat (enrichi de probabilités associées) et le langage stochastique observé dans le journal d'événements à modéliser.

Notre algorithme prend en entrée le langage du journal, construit l'arbre de processus à l'aide de l'*Inductive Miner*, puis le transforme en arbre de processus stochastique en lui attribuant des paramètres tirés aléatoirement. Ensuite, un processus d'optimisation est lancé afin de minimiser une fonction de distance en ajustant les différentes probabilités jusqu'à convergence vers un optimum. La métrique utilisée comme fonction objectif est une adaptation de la distance de Wasserstein [7]. À chaque itération, le langage stochastique de l'arbre est estimé par simulation afin de le comparer à celui du journal.

Nous avons testé cette méthode sur 3 journaux issue du *Business Process Intelligence Challenge* fourni par le département des technologies de l'information du constructeur automobile Volvo en Belgique. En comparant nos résultats à ceux d'autres approches de découverte basées sur l'optimisation de réseaux de Petri, notre méthode permet d'obtenir une qualité équivalente tout en réduisant significativement le temps de calcul.

4 Conclusion

Les approches classiques modélisent la stochasticité en transformant d'abord un arbre de processus en réseau de Petri, ce qui complique la gestion et l'interprétation des paramètres. Pour y remédier, nous introduisons les arbres de processus stochastiques, intégrant directement la stochasticité. Nous définissons leur syntaxe, leur sémantique et proposons une méthode de découverte des paramètres probabilistes par la simulation et l'optimisation. Des expériences préliminaires montrent qu'ils offrent une précision comparable aux réseaux de Petri, tout en réduisant le nombre de paramètres et en facilitant leur optimisation. Ces travaux seront étendus à des algorithmes capables de calculer de manière exacte et efficace le langage stochastique produit par un arbre, afin de développer de nouvelles méthodes de découverte efficaces.

Références

[1] Tobias Brockhoff, Merih Seran Uysal, and Wil M.P. Van Der Aalst. Wasserstein weight estimation for stochastic petri nets. In *2024 6th International Conference on Process Mining (ICPM)*, pages 81–88, 2024.

[2] Adam Burke, Sander J. J. Leemans, and Moe Thandar Wynn. Stochastic process discovery by weight estimation. In Sander J. J. Leemans and Henrik Leopold, editors, *Process Mining Workshops - ICPM 2020 International Workshops, Padua, Italy, October 5-8, 2020, Revised Selected Papers*, volume 406 of *Lecture Notes in Business Information Processing*, pages 260–272. Springer, 2020.

[3] Pierre Cry, András Horváth, Paolo Ballarini, and Pascale Le Gall. A framework for optimisation based stochastic process discovery. In Jane Hillston, Sadegh Soudjani, and Masaki Waga, editors, *Quantitative Evaluation of Systems (QEST) and Formal Modeling and Analysis of Timed Systems (FORMATS)*, volume 14996 of *LNCS*, pages 34–51, Cham, 2024. Springer Nature Switzerland.

[4] András Horváth, Paolo Ballarini, and Pierre Cry. Probabilistic Process Discovery with Stochastic Process Trees. In *EAI VALUESTOOLS 2024*, Milan, Italy, December 2024.

[5] Sander J. J. Leemans, Dirk Fahland, and Wil M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In José Manuel Colom and Jörg Desel, editors, *Application and Theory of Petri Nets and Concurrency - 34th International Conference, PETRI NETS 2013, Milan, Italy, June 24-28, 2013. Proceedings*, volume 7927 of *Lecture Notes in Computer Science*, pages 311–329. Springer, 2013.

[6] Sander J. J. Leemans, Tian Li, Marco Montali, and Artem Polyvyanyy. Stochastic process discovery : Can it be done optimally ? In *Advanced Information Systems Engineering : 36th International Conference, CAiSE 2024, Limassol, Cyprus, June 3–7, 2024, Proceedings*, page 36–52, Berlin, Heidelberg, 2024. Springer-Verlag.

[7] Sander J. J. Leemans, Anja F. Syring, and Wil M. P. van der Aalst. Earth movers' stochastic conformance checking. In Thomas Hildebrandt, Boudewijn F. van Dongen, Maximilian Röglinger, and Jan Mendling, editors, *Business Process Management Forum*, pages 127–143, Cham, 2019. Springer International Publishing.

[8] Wil van der Aalst. *Process Mining : Data Science in Action*. Springer Publishing Company, Incorporated, 2nd edition, 2016.

Tail Modulo Async-Await (extended abstract)*

Résumé long

EMMA NARDINO, ENS Lyon, Univ Lyon, UCBL, CNRS, Inria, LIP, France
LUDOVIC HENRIO, CNRS, Univ Lyon, ENS Lyon, UCBL, Inria, LIP, France
GABRIEL RADANNE, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, France
YANNICK ZAKOWSKI, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, France

1 Introduction

The Tail Call Optimisation has been a staple of functional programming languages since its introduction in 1977 in Steele Jr. [5]'s seminal paper "LAMBDA: The Ultimate GOTO". This optimisation hinges on the observation that, when a function call is in *tail position*, i.e., a return position in the program, it can be compiled as a simple `JUMP` instruction, thus dramatically improving the efficiency of procedure calls. This transformation, which also saves space in the function stack, made its way in numerous languages and compilers, whether enabled by default in languages such as OCaml, Haskell, or Scala; or as an on-demand optimisation in compilers such as LLVM and GCC.

The notion of tail position was initially quite restrictive: only calls in exact return position were considered. Tail-Call *modulo Constructor* [1] extends the class of accepted program with single tail-positions under a data constructor. Thanks to this notion, most functions over lists are now considered tail-recursive, and thus liberated from stack constraints.

However, while lists remain a data structure of choice for functional programmers, shouldn't trees deserve the same care? More broadly, can we hope for a similar optimisation in presence of *multiple* calls in tail position? Surely, this makes little sense in a sequential context: how could we launch several tail-calls without any synchronisation? However, this makes perfect sense in an *asynchronous* context, and more specifically in the presence of futures [2],[1] which are entities representing the result of an ongoing computation. In fact, a notion similar to asynchronous tail calls already exists for OO method calls in asynchronous languages, dubbed `forward` [3].

In this article, we further extend the notion of tail-position to be "modulo Await", and define a Tail-Modulo-Await transformation exploiting these extended tail-positions: similarly to how tail-calls in sequential contexts are optimised to use constant stack space, we show how to optimise tail-calls in asynchronous context in order to use no spurious scheduler space. Furthermore, when combined with Tail-Modulo-Cons (`tmc`), we obtain a Tail-Modulo-Cons-Await optimisation (`tmca`) able to optimise functions with multiple recursive calls under a constructor. We have formalised these transformations over minimal calculi, and prove them sound via backward simulations.

Let us first demonstrate the Tail-Modulo-Cons-Await transformation over trees; then give some experimental results about its effects.

2 A tail-recursive map on trees

Fig. 1 showcases an implementation of an asynchronous `map` function on trees in an OCaml dialect with asynchronous functions.[2] Before going over details, let us state our selling point: using our

*This work has been submitted to OOPSLA25, and a draft can be found here: https://inria.hal.science/hal-05006570
[1]Coincidentally, both works were published in 1977, and both originated from the LISP community!
[2]Our dialect is implemented with an OCaml syntax extension.

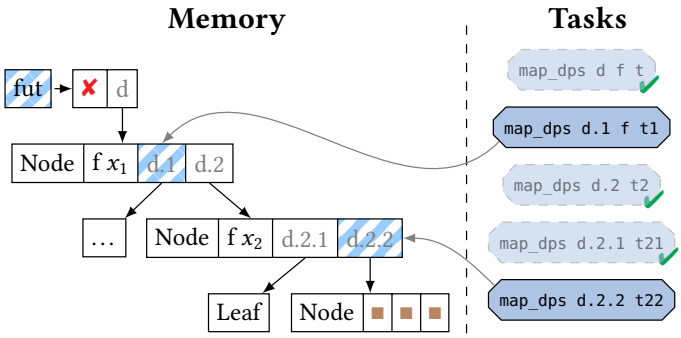
Authors' Contact Information: Emma Nardino, emma.nardino@ens-lyon.fr, ENS Lyon, Univ Lyon, UCBL, CNRS, Inria, LIP, Lyon, France; Ludovic Henrio, ludovic.henrio@cnrs.fr, CNRS, Univ Lyon, ENS Lyon, UCBL, Inria, LIP, Lyon, France; Gabriel Radanne, gabriel.radanne@inria.fr, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, Lyon, France; Yannick Zakowski, yannick.zakowski@inria.fr, Inria, Univ Lyon, ENS Lyon, UCBL, CNRS, LIP, Lyon, France.

```
1 type 'a tree = Leaf | Node of 'a * 'a tree * 'a tree
2
3 val map : ('a -> 'b) -> 'a tree -> 'b tree future
4 let%async rec map f t = match t with
5   | Leaf -> Leaf
6   | Node (x , tl, tr) -> Node (f x, await (map f tl), await (map f tr))
```

Fig. 1. A parallel and asynchronous `map` on trees.

```
1  let rec map_dps d f t = match t with
2    | Leaf -> d ◀ Leaf
3    | Node (x, tl, tr) ->
4      let v = Node (■, ■, ■) in
5      d ◀ v;
6      fork_dps map_dps v.1 f tl;
7      fork_dps map_dps v.2 f tr;
8      v.0 ◀ f x
9
10 let map f l =
11   let d, fut = create_future () in
12   fork_dps map_dps d f l;
13   fut
```

(a) Code with `tmca`



(b) Snapshot of execution of `map f t` after `tmca`

Fig. 2. Translation of `map` on trees from Fig. 1 using Tail-Modulo-Cons-Await

code transformation, this function is fully parallel (all calls to the mapped function are executed concurrently), uses a constant memory space per thread, and doesn't use any superfluous scheduler space.

We provide the signature for `map` on Line 3: it takes a function `f`, a tree `t`, and returns the tree where `f` has been applied to every element of `t`. In our code, asynchronous-related operations are indicated in **blue** and code internal to our optimisation in **peach**. Concretely, we consider a parametric `tree` type, defined on Line 1, where each `Node` is binary and contains a value. As is common for asynchronous functions, `map` returns more precisely a *future* for this tree. By tagging the function as `async` on Line 4, calls to this function create a future and launch the associated computation in a parallel sub-task. The result can then be retrieved using the `await` primitive: this is done for both recursive calls on Line 6. Finally, we assemble the value into a `Node` and return it, implicitly in a future.

In the absence of any optimisation, the behaviour of this function would be, on each `Node` constructor, to launch the task for one branch, wait for it to finish, launch the task for the other branch, wait for it to finish, then allocate the new constructor and return it. The stack would grow linearly in the height of the tree; furthermore, the execution would be overall sequential, while paying additional cost for synchronisation. The asynchronous programmer would however not write the code above and would favour a more parallel version by replacing Line 6 with:

```
let tl' = map f tl in let tr' = map f tr in Node (f x, await tl', await tr')
```

This version parallelizes the treatment of the left and right branches: both sub-tasks are launched by the calls first, and run simultaneously before being both `await`ed under the `Node`. Its main drawback is that it can have as many "active" tasks as there are nodes in the tree; indeed all the finished tasks which have at least one unfinished task below it in the tree structure are still blocked in an `await` statement.

Our approach optimises the two preceding programs by both parallelizing subtasks, and terminating all tasks that are only awaiting others. To achieve this, we translate the function to *Destiny Passing Style*, an asynchronous interpretation of Destination Passing Style [4]. An idealized version of the transformed function `map_dps` is shown on Fig. 2a; its execution is illustrated on Fig. 2b.

At the top level, a single future `fut` is tied to the filling of a memory cell, a destiny `d` (Line 11). From there, a helper function `map_dps` is called. The function `map_dps` takes as an additional argument a *destiny* `d` (Line 1) which represents the memory location where the result of the asynchronous computation should be written to. When returning a value, such as `Leaf` on Line 2, we set the destiny `d` with `d ◂ Leaf`. In the recursive case, we perform the allocation ahead of time—similarly to what happens in `tmc`— but put empty holes in lieu of its arguments. The current destiny can therefore be immediately set to `v`, the freshly allocated node. Both recursive calls are then forked in their own thread, one for each child: we must pass them their respective destiny, i.e., the corresponding hole in `v` they are responsible for filling—we address fields of constructors by position. The calls to `fork_dps` handling recursive calls each fork a new task running the function `map_dps` with the arguments passed as parameters; they create no stack for executing `map_dps`, the only stack needed is for `f`; and discard the value returned by the function so that the call uses a minimal space. Finally, we compute the value `f x` and set it in `v`. When all parallel tasks have finished, there are no more holes in the structure, the future is marked as *resolved*, and the value can be retrieved. Fig. 2b shows an intermediate state of the execution, the left side shows the status in memory of the tree being computed, the right side shows the tasks involved in the computation. Scheduler space is represented in blue. Futures and destiny both play a synchronisation role and serve as pointers to actual data, and are thus striped. The original future `fut` is yet unresolved (✘). Tasks `map_dps d f t`, `map_dps d.2 f t2`, and `map_dps d.2.1 f t21` are finished, the corresponding destinies have been filled and the tasks have been garbage-collected. Task `map_dps d.2.2 f t22` has just performed the allocation of a `Node` (with holes), and filled its destination. Task `map_dps d.1 f t1` is still ongoing. This illustrates that only the tasks actually computing are still active on the scheduler side, contrarily to the versions that do not use `tmca`.

This code transformation automatically reveals parallelism present in the source program. In fact, it doesn't fully respect the sequential semantics. Indeed on Line 6 in the code above, the transformed code can *interleave* subtasks in `map f tl` and `map f tr`, while the sequential semantics would enforce an execution order, without interleaving. Thus, this automatic parallelisation comes at the risk of introducing undesired behaviors. We mitigate this risk in two ways.

First, we only modify code *at the level of a constructor in tail position*. This means the programmer can easily enforce sequentiality via simple let-bindings. For instance, we could replace Line 6 by:

```
let x' = f x in Node (x', await (map f tl), await (map f tr))
```

In this case, it would wait for `f` to return before proceeding to the two subtrees in parallel (thus bounding the parallelism by the width of the tree). This aligns with good practice in languages like OCaml, where users who care about order of operation should state it explicitly via let-bindings.[3]

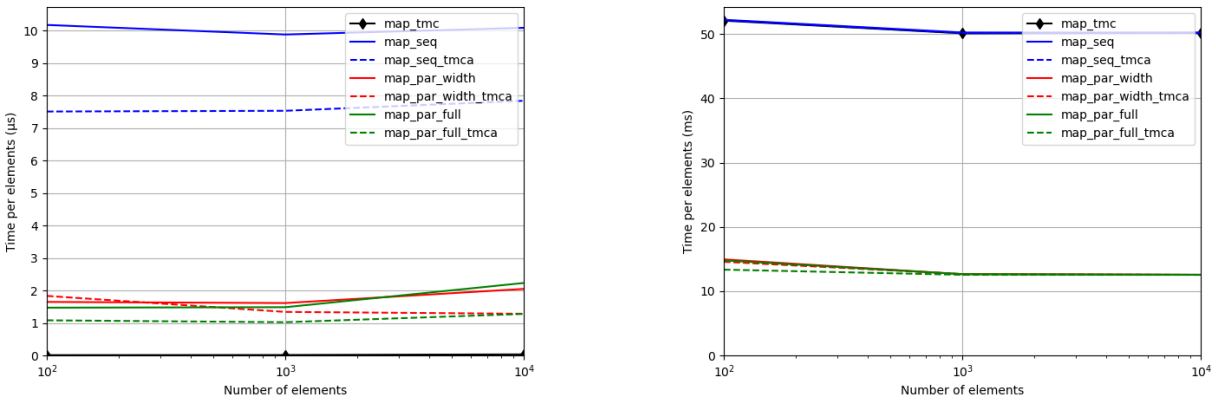
Second, we introduce two variants for each constructor: *parallel* constructors which behave as above, and *sequential* ones which don't introduce parallelism and only allow one tail position. With these variants, we show that our transformation preserves the semantics.

3 Experimental results

We now evaluate our full code transformation experimentally. Our evaluation setup is a Thinkpad T470 equipped with i5-7200U with 4 cores and 11GiB of memory. We explore the following scientific questions:

What is the effect of Tail-Modulo-Cons-Await on performances and parallelism? To evaluate the effect of Tail-Modulo-Cons-Await on time, we run several variants of `map` on randomly generated trees of growing size. Figure 3 shows the *time per elements*, i.e., the total time divided by the

[3]Naturally, this might come with some cost in scheduler- or stack- space, since the only way to optimise these aspects is through parallelism and thus interleaving of tasks.



(a) With `f = fun x -> x*.2.`. Time in μs.

(b) With `f = Unix.sleepf`. Time in ms.

Fig. 3. Time measurement for several versions of `Tree.map`. Lower is better.

number of elements, for `Tree.map f`. We consider two distinct `f` functions which are applied at each `Node`: a very fast function on the left (multiply by 2) and a slow function (sleep a random amount between 0 and 50ms). The evaluated `map` implementations are as follows: `map_tmc` is a synchronous sequential version which uses Tail-Modulo-Cons (with only one recursive tail call); all the others are asynchronous, with and without Tail-Modulo-Cons-Await: `map_seq` uses a *sequential* semantics `Node`; `map_par_full` uses a full *parallel* semantics; `map_par_width` also uses a *parallel* semantics, but applies `f` before any recursive calls, thus bounding the parallelism by *width* (see Section 2). The functions compiled with Tail-Modulo-Cons-Await are marked with `_tmca` and are represented using dashed lines. The scheduler is allowed to spawn tasks on all four available processors. All time measurements are obtained with repeated runs until a correlation of $r^2 > 0.8$ is obtained.

First, we can observe that the parallel semantics is, indeed, parallel! Regardless of the function `f` used, the parallel versions are exactly 4 times faster than the sequential versions.

Second, let us look at raw performance. With the fast `f` function, we can observe that Tail-Modulo-Cons-Await consistently improves performances, especially on large trees, although never to the point of equating the performance of the synchronous version. The cost of the slow function completely dominates any of the costs introduced by the implementation of `map`, and the cohort are separated into two groups: the sequential and the parallel versions.

What is the effect of Tail-Modulo-Cons-Await on memory consumption? We confirm that Tail-Modulo-Cons-Await behaves as advertised and avoids superfluous stack and scheduler space. As evaluating stack and scheduler space precisely is quite difficult, we settle for a lesser goal: to evaluate the total live memory during the execution of several synchronous and asynchronous variants of `List.map` on a list of 50000000 elements. The naive, non-tail-rec `List.map` runs out of stack before allocating anything, and crashes. All the other functions complete their execution while consuming the exact same amount of memory, which corresponds exactly to an allocation of the output list. This confirms that Tail-Modulo-Cons-Await behaves as advertised and avoids superfluous stack and scheduler space.

References

[1] Clément Allain, Frédéric Bour, Basile Clément, François Pottier, and Gabriel Scherer. 2025. Tail Modulo Cons, OCaml, and Relational Separation Logic. *Proceedings of the ACM on Programming Languages* 9, POPL (Jan. 2025), 2337–2363. doi:10.1145/3704915

[2] Henry. G. Baker Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proc. Symp. on Artificial Intelligence and Programming Languages*. New York, NY, USA, 55–59.

[3] Kiko Fernandez-Reyes, Dave Clarke, Elias Castegren, and Huu-Phuc Vo. 2018. Forward to a Promising Future. In *Conference proceedings COORDINATION 2018*.

[4] Amir Shaikhha, Andrew W. Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. 2017. Destination-passing style for efficient memory management. In *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing, FHPC@ICFP 2017, Oxford, UK, September 7, 2017*, Phil Trinder and Cosmin E. Oancea (Eds.). ACM, 12–23. doi:10.1145/3122948.3122949

[5] Guy L. Steele Jr. 1977. Debunking the "expensive procedure call" myth or, procedure call implementations considered harmful or, LAMBDA: The Ultimate GOTO. In *Proceedings of the 1977 annual conference, ACM '77, Seattle, Washington, USA, October 16-19, 1977*, James S. Ketchel, Harvey Z. Kriloff, H. Blair Burner, Patricia E. Crockett, Robert G. Herriot, George B. Houston, and Cathy S. Kitto (Eds.). ACM, 153–162. doi:10.1145/800179.810196

Model Checking de LTL sur Traces Finies et Infinies avec Domaines Concrets

Résumé long

David Doose, Julien Brunel

DTIS, ONERA, Université de Toulouse, France
firstname.surname@onera.fr

Abstract

Il existe différentes sémantiques pour la logique temporelle linéaire (LTL) concernant la prise en compte de traces finies. Bien que plusieurs d'entre elles puissent s'avérer utiles en fonction du contexte de vérification, aucun cadre de vérification ne permet de gérer leur diversité de manière simple. Une autre limitation des outils de vérification de LTL actuels est leur incapacité à traiter les domaines concrets (entiers bornés et infinis, nombres réels, etc.). Nous présentons une approche permettant de faire du model checking de LTL sur des traces finies et infinies avec des domaines concrets. Notre méthode repose sur un solveur SMT et sur le model checking borné (BMC). Nous présentons également quelques expérimentations et comparons notre outil avec NuSMV et nuXmv.

Ce document est le résumé long d'un article présenté à ICFEM 2024:

David Doose, Julien Brunel. *Simple LTL Model Checking on Finite and Infinite Traces over Concrete Domains*, in 25th International Conference on Formal Engineering Methods (ICFEM 2024), Volume 15394 of Lecture Notes in Computer Science, Springer, 2024

1 Introduction

L'analyse de formules de logique temporelle s'est avérée utile pour vérifier les propriétés comportementales des logiciels et des systèmes. Au cours des dernières décennies, de nombreux travaux ont proposé de nouveaux algorithmes, en particulier en utilisant un encodage SAT/SMT.

Toutefois, la plupart de ces techniques supposent que les formules temporelles sont construites à partir de propositions atomiques. Cependant, dans de nombreuses études de cas, il serait très utile d'utiliser des domaines concrets tels que les nombres entiers, les nombres flottants et les nombres réels.

Une deuxième limitation est la sémantique temporelle restreinte. Pour la logique temporelle linéaire (LTL), la sémantique habituelle ne prend en compte que les traces infinies. Cependant, il est utile de raisonner sur des exécutions finies et tronquées. Certaines exécutions peuvent se terminer en raison de pannes ou de la fin du comportement attendu. Bien que la sémantique finie soit définie dans la littérature [1, 2], il n'existe

pas de cadre unifié permettant aux utilisateurs de combiner facilement ces différents types de raisonnement.

Dans cet article, nous proposons une méthode outillée permettant de faire du model checing de LTL. Cette méthode se distingue des approches existantes par les points suivants : 1/ la capacité de traiter des variables portant sur des domaines concrets, 2/ la prise en compte de trois sémantiques pour les traces (infinie, tronquée ou finie maximale), 3/ la garantie de terminaison de l'algorithme sous hypothèses de finitude des domaines concrets. Nous avons développé un prototype et évalué son efficacité par rapport à NuSMV et nuXmv.

2 Semantique de Traces

Le problème de vérification que nous abordons suppose que le système étudié est modélisé comme un système de transition. Ainsi, les traces considérées dans notre approche proviennent de ce système de transition.

Traces Étant donné un ensemble P de propositions atomiques, un système de transition (TS) sur P est défini comme un tuple (S, I, T, V), où : S est un ensemble (éventuellement infini) d'états, I est l'ensemble des états initiaux, T est la relation de transition sur S, c'est-à-dire la partie de $S \times S$ formée par les couples d'états, et $V : S \to 2^P$ est une fonction d'évaluation associant à chaque état s l'ensemble $V(s)$ des propositions atomiques qui sont vraies dans cet état. Une trace consiste en une séquence d'états $(s_0, s_1, \ldots, s_n, \ldots)$ qui début par un état initial ($s_0 \in I$) et telle deux états successifs respectent la relation de transition T. Cette trace peut être *infinie*, finie (on dit alors que la trace est *tronquée*), ou bien *finie maximale* si elle est finie de longueur n et $\forall s \in S \quad (s_n, s) \notin T$.

Syntaxe et Sémantique LTL La grammaire de LTL est communément définie de la manière suivante: $\varphi ::= \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\,\varphi \mid \varphi\,\mathbf{U}\,\varphi$. La sémantique standard de LTL [3, 4] considère uniquement des traces infinies. Nous considérons ici également une sémantique de LTL sur des traces finies [2, 1]. Les deux sémantiques diffèrent principalement sur le "dernier" état de la trace. Étant donné une trace σ, et l'état numéro i de σ, la relation de satisfaction pour les traces infinies $\models_{\text{inf}}$ (resp. pour les traces finies $\models_{\text{fin}}$) définie comme suit pour les opérateurs temporels $\mathbf{X}$ and $\mathbf{U}$:

Traces infinies:

- $\sigma, i \models_{\text{inf}} \mathbf{X}\,\varphi$ if $\sigma, i+1 \models_{\text{inf}} \varphi$
- $\sigma, i \models_{\text{inf}} \varphi_1\,\mathbf{U}\,\varphi_2$ if $\exists j \geqslant i$ such that $\sigma, j \models_{\text{inf}} \varphi_2$ and $\forall i \leqslant k < j \quad \sigma, k \models_{\text{inf}} \varphi_1$

Traces finies:

- $\sigma, i \models_{\text{fin}} \mathbf{X}\,\varphi$ if $i < n$ and $\sigma, i+1 \models_{\text{fin}} \varphi$
- $\sigma, i \models_{\text{fin}} \varphi_1\,\mathbf{U}\,\varphi_2$ if $\exists j$ such that $i \leqslant j \leqslant n$ and $\sigma, j \models_{\text{fin}} \varphi_2$ and $\forall i \leqslant k < j \quad \sigma, k \models_{\text{fin}} \varphi_1$

À partir de ces deux sémantiques, nous pouvons définir trois relations de satisfaction entre un système de transition et une formule: 1) pour des traces infinies, 2) pour des traces tronquées et 3) pour des traces finies maximales.

3 Tatam

Nous avons développé le prototype Tatam[1] permettant de faire du model checking selon chacune des trois relations de satisfaction présentées dans la section 2. Son implémentation repose sur un algorithme de BMC avec un encodage SMT [5] et sur le solver z3.

```
var x, y: Bool
init I { x and not y }
trans T0 { x and not y and x' and not y' }
trans T1 { x and not y and x' and y' }
trans T2 { x and y and not x' and y' }
prop = F (not x)
search infinite + finite + complete solve
```

Le point d'entrée est un langage simple permettant de décrire des systèmes de transitions auxquels on ajoute une formule de LTL à analyser. L'utilisateur spécifie la ou les sémantiques (parmi les trois possibles) que nous souhaitons prendre en compte pour les traces. Il est également possible de vérifier qu'un seuil de complétude (mot clé `complete`) est atteint pour chaque itération de l'algorithme de BMC.

La procédure de décision ainsi que notre prototype reposent sur un solveur SMT et bénéficient donc de son expressivité (entiers non bornés, réels, etc.). Cependant, l'utilisation de certains de ces éléments (comme des fonctions non interprétées ayant des paramètres réels) peut avoir un impact sur l'expression du problème, sa résolution et la garantie de terminaison de celle-ci. Dans cet article, nous avons étudié l'impact de l'utilisation de ces différents éléments sur le processus de résolution.

Finalement, notre procédure de décision ainsi que notre prototype permettent de spécifier un critère à optimiser sur la trace solution, ce qui s'est avéré particulièrement utile dans des différents cas d'utilisation, comme pour la planification.

4 Comparaison

Afin d'étudier l'efficacité de notre méthode de résolution, nous l'avons comparée aux outils de référence : NuSMV et nuXmv. Cette comparaison est d'autant plus intéressante qu'ils' implémentent différents algorithmes de résolution : BDD, BMC et IC3.

Nous avons encodé un exemple protocole distribué d'élection d'un leader [6] et un exemple représentant des séries mathématiques et les avons fait grandir afin d'augmenter le temps de calcul. Les résultats montrent que les algorithmes basés sur les BDD se comportent bien pour des problèmes purement booléens ou discrets, mais qu'ils sont très inefficaces pour les problèmes mathématiques. Notre solveur basé sur BMC s'est avéré plus performant que les implémentations BMC de NuSMV et IC3 dans le cas général.

[1] Transition And Theory Analysis Machine : 0https://crates.io/crates/tatam

```
prop = F(client_boxes = produced_boxes)
search truncated + complete
    maximize (robot_fuel at last) until robot_fuel_capacity
```

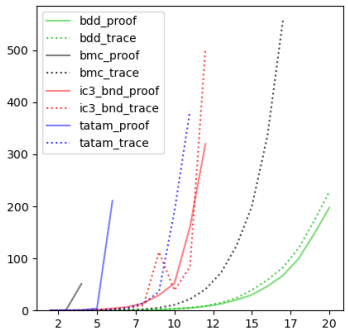


Figure 1: Leader election benchmark

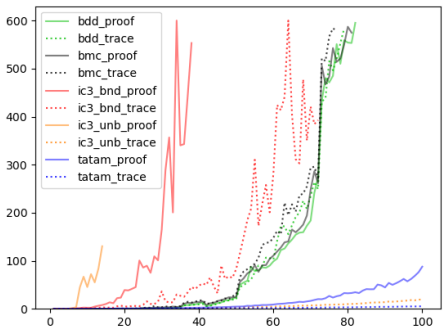


Figure 2: Arithmetic series benchmark

5 Conclusion

Nous avons proposé une méthode de model checking de LTL pour les traces finies et infinies. Notre méthode est basée sur un encodage SMT et sur le BMC classique. L'utilisation d'un solveur SMT permet d'utiliser des domaines de variables concrets, tels que les nombres entiers, réels, bornés et infinis. Nous avons également utilisé un critère simple permettant de détecter qu'un seuil de complétude a été atteint pour la longueur des traces. Cela permet d'obtenir une procédure de vérification complète, qui se termine nécessairement si les domaines de variables concrètes sont finis. Nous avons expérimenté notre outil et l'avons comparé à NuSMV et nuXmv.

References

[1] G. De Giacomo and M. Y. Vardi, "Linear temporal logic and linear dynamic logic on finite traces," in *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, AAAI Press, 2013.

[2] C. E. et all, "Reasoning with temporal logic on truncated paths," in *International Conference on Computer Aided Verification*, 2003.

[3] C. Baier and J.-P. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[4] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems. Specification*. Springer, 1991.

[5] A. Biere, K. Heljanko, T. Junttila, T. Latvala, and V. Schuppan, "Linear Encodings of Bounded LTL Model Checking," *Logical Methods in Computer Science*, 2006.

[6] E. Chang and R. Roberts, "An improved algorithm for decentralized extrema-finding in circular configurations of processes," *Communications of the ACM*, 1979.

VASSAL: Verification and Analysis for Safety and Security of Applications in Life

Project Presentation

Julien Signoles
Université Paris-Saclay, CEA, List

Milan Češka
Brno University of Technology

Florian Zuleger
Technische Universität Wien

Tomáš Kratochvíla
Honeywell International

This submission aims at presenting the EU Twinning project VASSAL (Verification and Analysis for Safety and Security of Applications in Life) that was started in June 2024 and runs for 36 months. The beneficiary partners are Brno University of Technology (BUT) in the Czech Republic, Vienna University of Technology (TUW) in Austria, and CEA List in France (CEA). Honeywell International (HON) is an associate partner. The project coordinator is Milan Češka from BUT.

Introduction

The VASSAL project is a collaborative research initiative focused on advancing verification and analysis techniques for software security and algorithmic accuracy. It aims to develop methodologies and tools to enhance the reliability and security of software systems, particularly in vulnerable fields. The project is funded by the European Union under Horizon Europe (project No. 101160022). VASSAL is under the Horizon Europe Twinning scheme, which supports partnerships between research institutions across Europe. Twinning projects specifically help institutions strengthen the research and innovation by working closely with research partners from top institutions. In the case of VASSAL, this helps to improve the institutional and administrative excellence at BUT and research collaboration in the areas of software verification and automated system design.

The research part of the project addresses the growing complexity of modern software systems, which requires formal verification techniques to meet high safety and security standards. Recent advancements in software verification have led to the development of powerful tools for control of the correctness and security in software applications. However, existing solutions often face scalability challenges and limitations in handling complex security properties. VASSAL seeks to bridge these gaps by leveraging state-of-the-art verification frameworks and innovative algorithmic techniques to provide more effective solutions.

Software safety and security has become a pressing issue, especially in domains such as finance, healthcare, and critical infrastructure, where vulnerabilities can have significant consequences. The project investigates new verification paradigms to address both theoretical and practical challenges.

Research Objectives and Focus

The key research objectives of the VASSAL project are:

1. **Enhancement of Verification Techniques**: Improving static and dynamic verification methods for detecting vulnerabilities and ensuring software correctness.

The project aims to improve both static and dynamic software verification methods. Static techniques analyze code without running it, aiming to catch bugs and vulnerabilities early in development. Dynamic techniques, on the other hand, monitor software during execution to detect issues that only appear at runtime. VASSAL will combine and refine these methods to create more accurate and reliable approaches to detecting security flaws and ensuring software correctness.

2. **Scalability and Performance**: Developing scalable algorithms and tools to handle large-scale software systems efficiently.

As modern software systems grow in size and complexity, verification tools must keep up. VASSAL will focus on developing algorithms and techniques that scale to large codebases and complex systems without compromising speed or precision. This includes optimizing performance so that the tools can be used in real-world industrial settings where efficiency is important.

3. **High-level system specification and efficient early verification**: Developing approaches for high-level specification of critical cyber-physical systems that allow for the usage of the language as natural as possible and, at the same time, for early rigorous verification

A core goal is to support developers in clearly expressing what it means for their software to be safe and secure. This includes specifying key properties such as reliability, integrity, or confidentiality. The project will create new methods for specifying these properties in a formal and precise way, as well as tools to automatically verify that software adheres to them.

4. **Practical Evaluation and Tool Development**: Implementing research findings into practical tools and validating them on real-world case studies.

The research carried out in VASSAL will not remain theoretical. A major focus is on building practical, user-friendly tools that implement the developed methods. These tools will be tested and validated on real-world software systems, ensuring their applicability and usefulness for developers.

5. **Interoperability with Existing Frameworks:** Ensuring that verification techniques integrate seamlessly with widely-used software development tools.

To maximize impact, the verification techniques and tools developed in VASSAL will be designed to integrate with popular development environments and existing verification frameworks. This ensures that the outcomes of the project can be adopted by industry and research communities without requiring major changes to existing workflows.

Project Structure

The project is structured into six key work packages:

- **WP1: Project Management and Coordination –** to ensure proper execution and smooth knowledge interchange/transfer and the synergies along the consortium, based on specific areas of excellence.
- **WP2: Raising Scientific Excellence** – to develop a scientific excellence strategy following high ethical standards based on a thorough analysis of current status and needs of the BUT.
- **WP3: Joint Research –** to collaborate and deepen specific joint research initiatives in four sub-domains. These are Logics and Automata, Model-based Design, Analysis and Synthesis, Automated analysis and verification on the source code level and Economic assessment and implications of the developed ASE (Automated Software Engineering) methods and tools.
- **WP4: Institutional and Administrative Excellence –** to develop governance and administration strategy for the R&I support framework, institutional policies and R&I services based on a thorough analysis of current status and needs of the BUT.
- **WP5: Integration and Sustainability –** to develop a strategy for integration, networking, and sustainability including a mid-term R&I roadmap, required investments, and actions to address European-wide safety/security SW-related challenges beyond the project lifetime.
- **WP6: Dissemination –** to disseminate and exploit all project outputs, especially the pilot project results, which should lead directly to larger initiatives and possibly (commercial) exploitation.

Expected Impact

VASSAL is expected to contribute significantly to the field of software verification and security analysis by:

- Providing innovative solutions to software correctness and security verification.
- Enhancing existing verification frameworks to support more complex and large-scale software systems.
- Enabling industry adoption of advanced verification tools through practical case studies and collaborations.

Publications and Dissemination:

As a result of the research, several research results have already been published at prestigious conferences, including:

- Sextl F., Rogalewicz A., Vojnar T., Zuleger F. Compositional Shape Analysis with Shared Abduction and Biabductive Loop Acceleration. In: *European Symposium on Programming (ESOP) 2025*.
  - This paper presents two improvements of abduction-based compositional analysis of list-manipulating programs – shared abduction for better handling of branching and shape extrapolation for improved analysis of loops. Those techniques are shown to improve both efficiency and precision of the analysis.
- Pontiggia F., Macák F., Andriushchenko R., Chiari M., Češka M. Decentralized Planning Using Probabilistic Hyperproperties. In: *International Conference on Autonomous Agents and Multiagent Systems (AAMAS) 2025.*
  - This paper proposes a new logic PHyperLTL and showcases its close relation to decentralized planning. Exploiting this relation, the authors implemented an algorithm that is able to solve interesting decentralized planning problems.
- Chocholatý D., Havlena V., Holík L., Hranička J., Lengál O., Síč J. Z3-Noodler 1.3: Shepherding Decision Procedures for Strings with Model Generation. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2025.*
  - The paper describes the latest improvements in the Z3-Noodler string solver, focusing on a technique for combining several decision procedures together.

Further dissemination will be carried out through workshops, conference presentations, and collaborations with academic and industry partners. VASSAL will also organize summer schools and training sessions to educate researchers and practitioners on the latest advancements in verification techniques.

Conclusion

The VASSAL project represents a significant step forward in the verification and analysis of software safety and security. By combining theoretical advancements with practical tool development, it aims to address critical challenges in software reliability and security.

The research outputs of VASSAL are expected to have a lasting impact on both academia and industry, providing methodologies that can be applied to various software engineering challenges. Through collaboration with partners, the project ensures that verification techniques remain relevant and adaptable to emerging threats in software security.

For further information, please visit the website: https://vassal.fit.vut.cz/

Session AFADL & MTV2

Apprentissage et test pour les machines à états finis temporisées avec délais de sortie

Evgenii Vinarskii, Natalia Kushik, Djamal Zeghlache
SAMOVAR, Télécom SudParis / Institut Polytechnique de Paris, Palaiseau, France

Abstract

Le travail est consacré à l'analyse et à la synthèse des machines à états finis temporisées avec délais de sortie (angl. Timed Finite State Machines ou TFSMs with output delays). Nous résumons brièvement nos résultats publiés récemment dans le domaine de l'apprentissage de TFSM [1] et la dérivation de la suite de tests à partir de la machine dérivée [2]. Pour dériver la suite de tests, nous utilisons la stratégie du «timed transition tour (TTT)». La nouvelle contribution à l'AFADL consiste en l'évaluation expérimentale de la corrélation entre les stratégies de sélection d'horodatage (angl. timestamp) et le modèle de fautes / taux de couverture du TTT. Nous réalisons des expériences pour évaluer l'efficacité des stratégies de tests, avec des implémentations mutées du logiciel Ansible [3][1].

1 Introduction

Les TFSMs ainsi que les automates temporisés (angl. Timed Automata ou TAs) sont largement utilisés dans les tests basés sur des modèles des systèmes distribués [4]. Cependant, les TFSMs [5] nécessitent que le testeur attende une sortie avant d'appliquer une nouvelle entrée, c'est pourquoi ils ne sont pas très pratiques lors des tests contre des *défauts liés au temps* (e.g., des problèmes de concurrence). Sinon, les TAs [6] pourraient être plus appropriés à cette fin. Mais les TAs manquent des restrictions explicites concernant la durée pendant laquelle le testeur peut attendre une sortie après chaque entrée appliquée; par conséquent, le processus de test devient plus difficile. Pour y parvenir, nous nous appuyons sur des «TFSMs with output delays» [1, 2] qui héritent de propriétés importantes des TAs et des TFSMs. En particulier, les entrées peuvent être appliquées consécutivement sans attendre les sorties, car chaque sortie est produite après un délai prédéfini. Cela nous permet d'utiliser la TFSM avec délais de sortie pour modéliser des systèmes distribués et les tester par rapport aux défauts liés au temps.

Nous commençons par le résumé étendu de nos résultats sur l'apprentissage de TFSM et les tests basés sur la TFSM, et présentons ensuite notre contribution principale, i.e., l'évaluation expérimentale avec des implémentations mutées d'Ansible-2.19[1] [3] pour estimer le taux de couverture de tests. Pour

[1]Un outil de gestion de configuration, utilisé dans le «cloud computing»

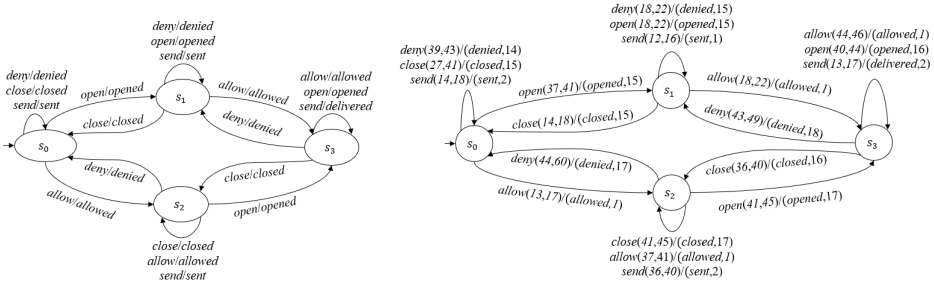
apprendre la TFSM en question, nous nous appuyons sur des conditions pré-post combinées avec des expériences avec l'implémentation «golden». En particulier, nous construisons d'abord la machine non-temporisée qui préserve les conditions pré-post données. Ensuite, la machine dérivée est complétée par les gardes temporisées et les délais de sortie. Les gardes temporisées et les délais de sortie sont obtenus via des expériences avec l'implémentation donnée (la Section 3.1). En même temps, lors de la dérivation d'une suite de tests, nous nous concentrons sur l'approche TTT (la Section 3.2). Nous considérons Ansible 2.19 pour évaluer le taux de couverture obtenu par différentes stratégies de sélection d'horodatage dans la méthode TTT. Parmi nos choix, il y a des horodatages qui sont plus proches de la borne gauche des gardes temporisées («left TTT»), des horodatages les plus proches de la borne droite des gardes temporisées («right TTT») et des horodatages qui sont égaux aux valeurs moyennes («mean TTT»). Ces stratégies sont comparées en fonction de leur capacité à détecter les implémentations mutées d'Ansible 2.19. Nous étudions deux types de mutants: i) *les mutants fonctionnels* qui sont dérivés par le framework *mutmut* [7], et ii) *les mutants liés au temps* qui sont obtenus en injectant la commande *sleep*. Nos expériences ont montré que cent pour cent des mutants fonctionnels ont été détectés par le TTT, alors que ce n'est pas le cas pour les mutants liés au temps. De plus, nous montrons que le «mean TTT» est la meilleure stratégie pour détecter les défauts liés au temps (la Section 4).

2 Machines à états finis (temporisées)

Une machine à états finis (angl. Finite State Machine ou FSM) [8] est un tuple $\mathcal{S} = (S, I, O, h_S, s_0)$, où S est *un ensemble fini d'états* avec *l'état initial* désigné par $s_0 \in S$, I (O) est *un alphabet fini d'entrée (sortie)* et $h_S \subseteq S \times I \times O \times S$ est *une relation de transition*. Une transition $(s, i, o, s') \in h_S$ signifie qu'étant à l'état s et recevant l'entrée i, $\mathcal{S}$ produit la sortie o et se déplace à l'état s'. Considérons la FSM $\mathcal{S}_\Phi$ montrée en Fig. 1a[2] et deux transitions $(s_0, send, sent, s_0)$ et $(s_3, send, delivered, s_3)$. Cela signifie que si un message est envoyé à l'état s_0, il ne sera pas livré, sinon le même message sera livré en étant envoyé à l'état s_3.

Rappelons que, selon la définition, pour les machines à états finis non-temporisées, le changement d'état et la réaction de sortie se produisent simultanément, après l'application d'une entrée. Cette fonctionnalité ne permet pas de représenter un comportement asynchrone. Afin de prendre en compte ce comportement, les «TFSMs with output delays» [1, 2] peuvent être utilisées. Le comportement de ces TFSMs dépend de l'état courant, de l'entrée et de l'horodatage de cette entrée. Formellement, *une machine à états finis temporisée avec délais de sortie* $\mathcal{T}$ est un tuple $(S, I, O, G, D, h_S, s_0)$, où G est *un ensemble fini de gardes temporisées*, $h_S \subseteq S \times I \times G \times O \times D \times S$ est *un ensemble de transitions temporisées*, D est *un ensemble fini de délais*. Une transition temporisée $(s, i, g, o, d, s') \in h_S$ signifie que si la machine reçoit l'entrée i après t unités de temps étant à l'état s ($t \in g$), alors la machine passe à l'état s' et produit la sortie o après

[2]Nous expliquons la notation de la FSM $\mathcal{S}_\Phi$ dans la Section 3.1



(a) FSM $\mathcal{S}_{\varphi_{send}}$ (b) TFSM $\mathcal{T}_{\varphi_{send}}$

Figure 1: FSM & TFSM exemple

d unités de temps. L'entrée suivante peut être appliquée avant que la machine n'ait produit la sortie à la précédente. Étant donné $t_0 = 0$, *une exécution* de la TFSM $\mathcal{T}$ pour $\alpha = (i_1, t_1), (i_2, t_2), \ldots, (i_n, t_n)$ est $s_0 \xrightarrow[(o_1,\tau_1)]{(i_1,t_1)}$ $s_1 \xrightarrow[(o_2,\tau_2)]{(i_2,t_2)} \ldots \xrightarrow[(o_n,\tau_n)]{(i_n,t_n)} s_n$ telle que pour chaque $j \in \{1, \ldots, n\}$ il existe $(s_{j-1}, i_j, g_j, o_j, d_j, s_j)$ de $\mathcal{T}$ pour lequel $t_j - t_{j-1} \in g_j$, et $\tau_j = t_j + d_j$. Si $t_j - t_{j-1} \in g_j$ pour chaque j, on dit que α est *activée* pour $\mathcal{T}$. Considérons la TFSM $\mathcal{T}_{\varphi_{send}}$ montrée en Fig. 1b qui représente (partiellement) le comportement d'Ansible-2.19, $\alpha_1 = (allow, 14), (open, 56), (send, 70)$ et $\alpha_2 = (allow, 14), (open, 56), (send, 72)$ qui ont les mêmes projections non-temporisées. La réponse de sortie temporisée de $\mathcal{T}_{\varphi_{send}}$ pour α_1, i.e., $timed\_out(\mathcal{T}_{\varphi_{send}}, \alpha_1)$, est $(allowed, 15), (opened, 73), (delivered, 72)$, tandis que la réponse de sortie $out(\mathcal{T}_{\varphi_{send}}, \alpha_1)$ est $allowed, delivered, opened$. Cela signifie que $delivered$ est généré avant $opened$, tandis que $open$ a été appliqué avant $send$. Cela peut entraîner que le message envoyé par $send$ ne soit pas livré car le port n'a pas encore été ouvert. Sinon,
$timed\_out(\mathcal{T}_{\varphi_{send}}, \alpha_2) = (allowed, 15), (opened, 73), (delivered, 74)$ et $out(\mathcal{T}_{\varphi_{send}}, \alpha_2) = allowed, opened, delivered$, i.e., le message sera livré. Ainsi, la sélection d'horodatages affecte l'ordre des sorties. Dans cet article, nous évaluons expérimentalement l'impact de la sélection des horodatages sur le taux de couverture des tests TTT (la Section 4).

3 Synthèse et analyse de TFSM

Dans cette section, nous décrivons les stratégies d'apprentissage de TFSM et de génération de tests basés sur la TFSM pour les systèmes distribués; les travaux ont été publiés dans [1] et [2], respectivement.

3.1 Apprentissage de TFSM

Nous nous concentrons sur une classe particulière de systèmes distribués conçus pour *l'exécution de tâches*. Ces systèmes fonctionnent en exécutant des séquences des tâches (angl. workflow) qui effectuent soit des *tâches*

de configuration (e.g., l'ouverture d'un port) soit des *tâches d'action* (e.g., l'envoi d'un message). Dans de tels systèmes, l'exécution réussie d'une tâche d'action dépend généralement de l'achèvement préalable des tâches de configuration spécifiques. Cette dépendance peut être formellement capturée à l'aide des conditions pré-post qui spécifient l'état de configuration requis pour qu'une tâche d'action soit terminée avec succès. Ces conditions pré-post sont la base de notre approche de pré-apprentissage pour synthétiser la «TFSM with output delays». Plus précisément, nous construisons d'abord la FSM qui préserve les conditions pré-post données, modélisant le comportement fonctionnel du système. Cette FSM sert ensuite de base pour construire la «TFSM with output delays», où les paramètres temporisés (gardes temporisées et délais de sortie) sont extraits via des expériences avec une implémentation du système.

Afin d'introduire formellement la dérivation de la TFSM, nous avons introduit les notations suivantes. Soit $C = \{c_1, \ldots, c_m\}$ un ensemble des tâches *positives* qui effectuent des modifications de configuration, et $\overline{C} = \{\overline{c_1}, ..., \overline{c_m}\}$ un ensemble des tâches *négatives* qui annulent des tâches de C; $C^+ = \{c_1^+, \ldots, c_m^+\}$ et $\overline{C^+} = \{\overline{c_1^+}, ..., \overline{c_m^+}\}$ désignent les notifications de fin des tâches de configuration correspondantes. Soit $A = \{a_1, ..., a_n\}$ un ensemble des tâches d'action, $A^+ = \{a_1^+, ..., a_n^+\}$ et $A^- = \{a_1^-, ..., a_n^-\}$ désignent les notifications d'exécutions correctes et incorrectes des tâches de A, respectivement. *Une condition pré-post positive* $\varphi := C' \Rightarrow a$, où $C' \subseteq C$ et $a \in A$, signifie que a peut être terminée correctement (a^+) si et seulement si: i) le système a été configuré de manière à permettre la réalisation de a (C'), et ii) a apparaît dans la séquence des tâches correspondantes. Par exemple, $\varphi_{send} := (open, allow) \Rightarrow send$ signifie qu'un message sera livré si et seulement si: i) le port a été ouvert, le trafic a été autorisé et ii) le message est envoyé. Soit $\varphi_a := C_a \Rightarrow a$ une condition pré-post, $I = C \cup \overline{C} \cup A$ et $O = C^+ \cup \overline{C^+} \cup A^+ \cup A^-$. La FSM $\mathcal{S}_{\varphi_a}$ avec I et O *préserve* φ_a *pour* β/γ *sur* I^*/O^* si pour toute apparition de a/a^+ dans β/γ on a les conditions suivantes: i) $c \prec_\beta a$ (c précède a dans β) pour toute $c \in C_a$, et ii) s'il existe $\overline{c} \in \overline{C_a}$ tel que $\overline{c} \prec_\beta a$, alors $\overline{c} \prec_\beta c \prec_\beta a$. La FSM $\mathcal{S}_\Phi$ *préserve* Φ si $\mathcal{S}_\Phi$ préserve chaque φ de Φ pour chaque trace d'entrée/sortie.

Soit Φ un ensemble de conditions pré-post sur C et A, nous utilisons l'approche suivante pour synthétiser la FSM $\mathcal{S}_\Phi$ qui préserve Φ. Les états sont représentés par des vecteurs booléens indiquant les tâches de configuration exécutées et non-annulées. Soit $[is\_c_1, \ldots, is\_c_m]$ un état, les transitions à partir de cet état sont définies de la manière suivante: i) l'état suivant pour toute $c \in C$ est $is\_c \leftarrow true$ et la transition est étiquetée par c/c^+, ii) l'état suivant pour toute $\overline{c} \in \overline{C}$, est $is\_c \leftarrow false$ et la transition est étiquetée par $\overline{c}/\overline{c^+}$, iii) pour toute $a \in A$ avec la condition pré-post $\varphi_a := C_a \Rightarrow a$ si à l'état courant φ_a est préservée, la transition est étiquetée par a/a^+, sinon la transition est étiquetée par a/a^-. Dans [1], nous avons montré que $\mathcal{S}_\Phi$ préserve Φ, et que $\mathcal{S}_\Phi$ possède au plus $2^{|C|}$ états. La Fig. 1a montre la FSM $\mathcal{S}_{\varphi_{send}}$ qui préserve $\varphi_{send} := (open, allow) \Rightarrow send$.

Pour augmenter la FSM $\mathcal{S}_\Phi$ avec des gardes temporisées et des délais de sortie, nous exécutons plusieurs fois une implémentation donnée. Soit $tran = (s, i, o, s')$ une transition de $\mathcal{S}_\Phi$ et une trace α amène $\mathcal{S}_\Phi$ à l'état s. Afin d'augmenter $tran$ à $timed\_tran = (s, i, g, o, d, s')$, la séquence

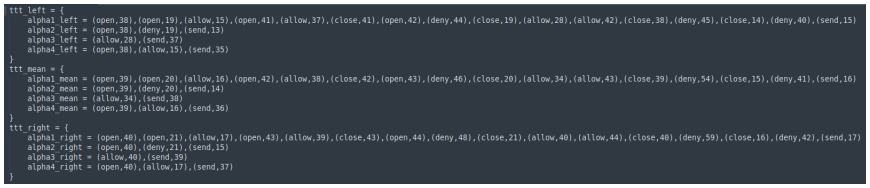


Figure 2: «left», «mean» et «right» stratégies de sélection d'horodatages pour TTT

des tâches qui correspond à α, i est exécutée $N > 0$ fois. Nous calculons les intervalles temporisés (u, v) et (p, q), où (u, v) indique la fluctuation du temps d'exécution des tâches i, tandis que (p, q) indique la fluctuation du délai des tâches i. Nous assignons $g \leftarrow (u, v)$ comme garde temporisée de $timed\_tran$, et si $i \in C$, alors $d \leftarrow q$, sinon $d \leftarrow p$. La Fig. 1b montre la TFSM $\mathcal{T}_{\varphi_{send}}$ qui augmente $\mathcal{S}_{\varphi_{send}}$ avec des paramètres temporisés.

3.2 Dérivation de tests basés sur TFSM

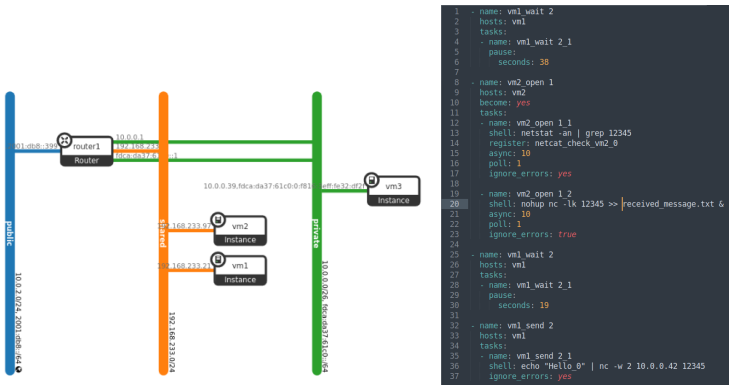
Puisque un «transition tour»[3] détecte de nombreuses fautes d'implémentation et peut être facilement dérivé, nous nous sommes concentrés sur l'étude des propriétés de la sélection de différents horodatages pour un TTT [2]. Un *TTT* pour la TFSM $\mathcal{T}$ est un ensemble fini de séquences d'entrée temporisées activées pour $\mathcal{T}$, telles que pour chaque transition e de $\mathcal{T}$ le TTT contienne une séquence d'entrée temporisée qui couvre e. Lors de la dérivation des séquences d'entrée du TTT, il est crucial de choisir soigneusement chaque entrée, mais aussi chaque instant où l'entrée correspondante est appliquée (horodatage ou angl. timestamp). Soit $e = (s, i, (u, v), o, d, s')$ une transition de $\mathcal{T}$, pour traverser cette transition e on peut choisir le t différemment. Dans [2], nous avons étudié certaines propriétés du TTT. En particulier, nous avons montré que la sélection des horodatages peut avoir un impact sur la capacité à détecter des défauts de sortie.

Dans ce travail, nous examinons trois stratégies pour choisir le t: le plus proche de la borne gauche («left») de la (u, v), le plus proche de la borne droite («right») de la (u, v) et la valeur moyenne («mean») de la (u, v). La Fig. 2 présente «left», «mean» et «right» TTTs pour $\mathcal{T}_{\varphi_{send}}$. Considérons la dérivation du ttt_{mean}. Étant donné que s_0 est un état initial et que la transition $(s_0, open, (37, 41), opened, 15, s_1)$ est définie, $t_1 = 39$. À l'état s_1, la transition $(s_1, open, (18, 22), opened, 15, s_1)$ est définie, par conséquent, $t_2 = 20$. En continuant, nous allons dériver le ttt_{mean} montré en Fig. 2. Dans la Section 4, nous comparons ces stratégies en fonction de leur capacité à détecter les implémentations mutées d'Ansible-2.19.

4 Évaluation expérimentale

Dans la Section 3.2, nous examinons les stratégies «left», «right» et «mean» pour choisir les horodatages d'un TTT. Nous évaluons l'efficacité de ces stratégies pour détecter les défauts injectés dans l'implémentation d'Ansible.

[3]Une suite de tests qui couvre toutes les transitions d'une FSM



(a) OpenStack topologie exemple

```yaml
- name: vm1_wait 2
  hosts: vm1
  tasks:
  - name: vm1_wait 2_1
    pause:
      seconds: 38

- name: vm2_open 1
  hosts: vm2
  become: yes
  tasks:
  - name: vm2_open 1_1
    shell: netstat -an | grep 12345
    register: netcat_check_vm2_0
    async: 10
    poll: 1
    ignore_errors: yes

  - name: vm2_open 1_2
    shell: nohup nc -lk 12345 >> received_message.txt &
    async: 10
    poll: 1
    ignore_errors: true

- name: vm1_wait 2
  hosts: vm1
  tasks:
  - name: vm1_wait 2_1
    pause:
      seconds: 19

- name: vm1_send 2
  hosts: vm1
  tasks:
  - name: vm1_send 2_1
    shell: echo "Hello_0" | nc -w 2 10.0.0.42 12345
    ignore_errors: yes
```

(b) Ansible-playbook exemple

Figure 3: OpenStack & Ansible

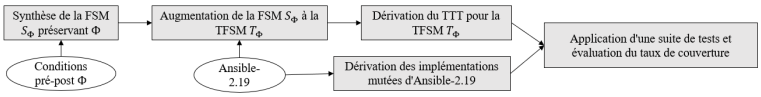


Figure 4: Workflow expérimental

4.1 Workflow expérimental

Nous utilisons la plateforme de «cloud computing» OpenStack avec Ansible-2.19 pour configurer des machines virtuelles. Plus précisément, nous exécutons trois machines virtuelles sur lesquelles Ubuntu-20 est préinstallé et dotées de 1 Go de RAM. Les machines virtuelles sont rattachées à des sous-réseaux, interconnectés via un routeur (voir la Fig. 3a). Pour simuler la communication réseau entre deux machines virtuelles, vm_1 et vm_2, nous utilisons des ansible-playbooks qui exécutent des séquences prédéfinies de tâches. Prenons le playbook montré en Fig. 3b: il commence par une attente de 38 secondes. Il configure ensuite la machine vm_2 pour écouter les messages entrants par le port 12345 à l'aide de la tâche "nc -l 12345 -k". À l'étape suivante, Ansible attend 19 secondes et exécute la tâche $send\_message$ qui envoie le message $Hello$ de la VM vm_1 à la VM vm_2. Notons que ce playbook correspond à la séquence $(open, 38)(send, 57)$.

Pour évaluer le taux de couverture de différentes stratégies de sélection d'horodatage dans le TTT, nous suivons le workflow montré en Fig. 4. En nous concentrant sur les défauts de communication réseau, nous considérons la condition pré-post $\varphi_{send} = (open, allow) \Rightarrow send$. La FSM $\mathcal{S}_{\varphi_{send}}$ qui préserve φ_{send} est montrée en Fig. 1a. À l'aide des traces d'exécution d'Ansible 2.19, nous augmentons $\mathcal{S}_{\varphi_{send}}$ pour dériver la TFSM $\mathcal{T}_{\varphi_{send}}$ montrée en Fig. 1b. Nous comparons trois stratégies de sélection d'horodatages dans le TTT: «left», «right» et «mean» TTT. Notre framework implémentant ces stratégies est disponible sur le GitHub [9]. Le taux de couverture est mesuré en comparant les sorties produites par les implémentations mutées à celles produites par l'implémentation «golden».

Figure 5: Fonction run du module **task_queue_manager** d'Ansible-2.19

	sleep=1	sleep=2	sleep=3	sleep=4	sleep=5	sleep=6	sleep=7	sleep=8	sleep=9	sleep=10	Total
ttt_left	survived	killed	killed	survived	survived	survived	killed	killed	survived	killed	5/10
ttt_mean	killed	killed	killed	killed	killed	killed	killed	killed	survived	killed	9/10
ttt_right	killed	survived	killed	killed	killed	killed	survived	survived	killed	survived	6/10

Figure 6: Mutants liés au temps & résultats

4.2 Résultats & discussion

Nous avons réalisé des expériences avec deux types de mutants: *fonctionnels* et ceux, *liés au temps*. Afin de dériver des mutants fonctionnels d'Ansible-2.19, nous avons muté la fonction run du module **task_queue_manager** à l'aide du framework *mutmut*. Les ansible-playbooks qui correspondent aux cas des «left TTT» (ttt_{left}), «right TTT» (ttt_{right}) et «mean TTT» (ttt_{mean}), ainsi que les programmes mutés, sont accessibles sur le GitHub [9]. Nous avons généré 25 mutants fonctionnels d'Ansible-2.19. De plus, nous avons exécuté les cas des tests dérivés avec des implémentations mutées d'Ansible-2.19 et avons observé que chaque TTT détecte toutes les implémentations mutées. Ainsi, les «left TTT», «right TTT» et «mean TTT» représentent des suites de tests exhaustives par rapport aux mutants fonctionnels générés par le framework *mutmut*. Afin d'expliquer cette exhaustivité, il est nécessaire de noter que les mutants générés par le *mutmut* sont relativement simples et représentent souvent des erreurs sémantiques de base, comme montré en Fig. 5. La variable ***all_vars*** est attribuée de la valeur ***None***, par conséquent, les indicateurs de contrôle tels que le registre netcat (voir la Fig. 3b) ne sont plus disponibles, ce qui entraîne un comportement incorrect du playbook, comme l'échec de l'ouverture d'un port réseau requis. Parallèlement, les séquences, dérivées de la TFSM apprise à l'aide de l'approche basée sur les conditions pré-post, contiennent de longues séquences qui couvrent de nombreuses branches du programme.

Nous avons également étudié les ttt_{left}, ttt_{right} et ttt_{mean} ainsi que leur capacité à détecter les mutants liés au temps. Ces mutants modélisent l'instabilité du réseau en injectant la commande *sleep* dans le module **playbook_executor**. Nous avons généré dix mutants d'Ansible-2.19 en injectant $sleep(1)$, $sleep(2)$, ..., $sleep(10)$. De même, nous avons exécuté les playbooks qui correspondent aux ttt_{left}, ttt_{right} et ttt_{mean}. Les résultats de l'exécution sont présentés en Fig. 6. Notons que le ttt_{left} est plus efficace pour détecter les grandes perturbations du réseau que les petites. À l'inverse, le ttt_{right} est plus efficace pour détecter les petites perturbations que les grandes. Le ttt_{mean} bénéficie d'une stratégie équilibrée, telle qu'elle évite d'être trop grande ou trop petite. Cela lui permet de détecter les petites comme les grandes perturbations du réseau.

5 Conclusion

Dans ce travail, nous avons d'abord présenté un résumé détaillé de nos articles publiés récemment sur l'apprentissage de TFSM et la dérivation de suites de tests à base de TFSM [1, 2]. Nous avons aussi présenté notre contribution principale à l'AFADL, c'est-à-dire l'évaluation expérimentale de la corrélation entre les stratégies de sélection d'horodatages et le taux de couverture des fautes, obtenu par les TTTs correspondants. Nous avons notamment démontré qu'un TTT est capable de détecter tous les mutants fonctionnels, et que le «mean TTT» est la meilleure stratégie pour détecter les mutants liés au temps.

Cet article ouvre plusieurs directions de recherche. Par exemple, des approches incrémentales de test pourraient être considérées, i.e., l'intégration de la connaissance des mutants liés au temps non détectés afin d'améliorer le taux de couverture. De plus, l'instabilité du réseau devrait être surmontée d'une manière ou d'une autre, car elle peut affecter le processus d'apprentissage de TFSM; nous travaillons actuellement sur ce problème.

References

[1] E. Vinarskii, N. Kushik, and D. Zeghlache, "Studying timed aspects for cloud configuration management tools: validation and recommendations for safe execution," in *IEEE ICWS*, pp. 1365–1367, 2024.

[2] E. Vinarskii, N. Kushik, J. López, N. Yevtushenko, and D. Zeghlache, "Timed transition tour for race detection in distributed systems," in *the 18th International ENASE Conference*, pp. 613–620, 2023.

[3] `https://github.com/ansible/ansible`, accessed 2025.

[4] E. Kim, M. Goorden, K. Larsen, and T. Nielsen, "Controlling stormwater detention ponds under partial observability," *Journal of Logical and Algebraic Methods in Programming*, vol. 141, p. 100979, 2024.

[5] D. Bresolin, K. El-Fakih, T. Villa, and N. Yevtushenko, "Equivalence checking and intersection of deterministic timed finite state machines," *Formal Methods Syst. Des.*, vol. 59, pp. 77–102, 2021.

[6] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[7] `https://github.com/boxed/mutmut`, accessed 2025.

[8] A. Gill, *Introduction to the Theory of Finite State Machines*. New-York:McGraw-Hill, 1962.

[9] E. Vinarskii, N. Kushik, and D. Zeghlache, "Apprentissage et test pour les machines à états finis temporisées avec délais de sortie." `https://github.com/vinevg1996/AFADL25`, 2025.

Experimental evaluation of LLMs for Test Generation

Tiago Costa, Ahmad Ghandour, Mariia Soltys, Nikita Volosnikov, Yves Ledru, Nicolas Hili

Univ Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000, Grenoble, France
Firstname.Lastname@univ-grenoble-alpes.fr

May 2025

Abstract

With the increased value of information, developing and testing robust software applications within shorter production cycles has become increasingly challenging. The recent outbreak in research and development of Large Language Models (LLMs) presents an opportunity to rethink software development by automating as many repetitive tasks as possible. In particular, leveraging LLMs for both black-box and white-box testing is of significant interest.

This paper provides an experience report assessing the quality of test cases generated by LLMs on the same java function, using a variety of prompts to produce black-box and white box test suites. It evaluates the quality of the resulting test suites using the following criteria : absence of build errors, correctness of the test oracle, capability to find failures and code coverage.

1 Introduction

Over the past years, a variety of Large Language Models (LLMs) have been proposed. Some of them, e.g., OpenAI/ChatGPT[1] are general-purpose, i.e., they have been trained on a variety of textual data, not necessarily related to software development. Others, e.g., Starcoder[2], Claude [3] or Deepseek [4] are specifically trained on software code. Most LLMs provide comparable features that software engineers can leverage. While all are accessible through chat-based interfaces, the majority

[1] https://openai.com/index/chatgpt/
[2] https://github.com/bigcode-project/starcoder
[3] https://claude.ai
[4] https://www.deepseek.com/

also offer APIs compatible with various programming languages, enabling seamless integration into software development toolchains. Increasingly, integrated development environments (IDEs) are embedding LLMs by default to support and enhance software engineering tasks. As a result, software engineers are often faced with the decision of selecting one model from a growing catalogue of available options.

Test case generation is an ideal application domain for LLMs. On the one hand, testing is a crucial task in software development. On the other hand, it is a repetitive task which would benefit of extensive automation provided by the combination of LLMs and IDEs such as VSCode[5], IntelliJ[6], or Eclipse[7].

Prompt engineering plays a critical role in enhancing the quality of test case generation with LLMs. By carefully crafting and structuring prompts, developers can guide models to produce more accurate, relevant, and complete test cases, addressing common issues such as missing edge cases, incorrect assertions, or improper use of testing frameworks. Effective prompt engineering involves specifying clear instructions, providing representative examples, and including context about the application under test, thereby improving both the reliability and semantic validity of the generated test cases. However, the quality of the prompt and its impact on test case generation is not sufficiently discussed in the literature.

Rather than providing an exhaustive comparison of LLMs for software testing, this paper presents a preliminary study illustrating how certain factors — such as prompt quality and the chosen testing technique (whether white-box or black-box) — can positively or negatively influence the effectiveness of LLM-generated test cases. This experimental evaluation is conducted on a single, very small program (Integer Division, see Listing 1) using one LLM-based chatbot: Le Chat by Mistral AI[8]. Both the program and the model were deliberately selected for their ability to illustrate the concepts of this experiment. Additionally, the study does not involve training or fine-tuning the model; the focus is placed solely on the impact of prompt design and the chosen testing technique.

2 Methodology and Experimental Evaluation

For the sake of this experiment, Listing 1 presents an integer division function implemented in Java. The function takes two parameters, `x` and `y`, representing the *quotient* and the *divisor*, respectively, and returns the *dividend*. Despite the apparent simplicity of this example, its implementation in Java can produce incorrect behavior when handling corner-case values. In particular, lines 7 and 11 may cause integer overflow when processing the smallest possible 32-bit integer value, potentially resulting in an infinite loop.

[5] `https://code.visualstudio.com/`
[6] `https://www.jetbrains.com/fr-fr/idea/`
[7] `https://eclipseide.org/`
[8] `https://mistral.ai/`

```java
public class IntDivision{
  public static int IntDiv(int x, int y) {
    int z = 0;
    int sign = 1;
    if (x < 0) {
      sign = -1;
      x = -x;
    }
    if (y < 0) {
      sign = -sign;
      y = -y;
    }
    if (y == 0) {
      throw new IllegalArgumentException("Argument nul:" + y);
    }
    while (x >= y) {
      x = x - y;
      z = z + 1;
    }
    z = sign * z;
    return z;
  }
}
```

Listing 1: An integer division Java program used for this test generation.

2.1 Black-Box Testing vs. White-Box Testing

A first question that arises when writing tests is whether we have knowledge about the internal structure of the software under test. If the answer is positive, we can consider that we are performing white-box tests. A complementary question is the knowledge of the software functionality, in particular, do we have a specification of the software? If the answer is positive, we are performing black-box tests.

The answer of that question influences the design of the prompt. Black-box testing means that only the specification of `IntDiv` is given to the LLM (e.g., its signature optionally accompanied with some natural language explanation of what the function is supposed to do). In contrast, white-box testing means that the full code of the function is given to the LLM. The latter case has been more widely explored in the literature, although it may introduce biases. Specifically, an LLM might interpret a buggy implementation as correct, and consequently generate tests that are valid only with respect to the flawed code rather than the intended specification.

Additionally, another source of biases we identified comes from the fact that LLMs are trained on a wide range of data sources, including materials specialized in computer science. It is very likely that these sources include multiple implementations of the integer division algorithm. As a result, one can question the reasoning capabilities of the LLM when generating test cases for critical corner cases, such as division by zero. To minimize the influence of this pre-existing knowledge, we

Table 1: Four different prompts to generate test cases for the Integer division.

Prompt ID	Prompt contents
WB1	(General, white box): <program code> + "Generate test cases for this Java program"
WB2	(Detailed, white box): <program code> + "Generate test cases for this given Java program covering as much cases as possible, so we can reach a high code coverage"
WB3	(Anonymized, white box): <anonymized program code> + "Generate test cases for this given Java program covering as much cases as possible, so we can reach a high code coverage"
BB1	(Black box): "Generate test cases in Java (JUnit version 4) for an integer division program respecting these specific signatures of the method and program: Class signature: public class IntDivision , Function signature public static int IntDiv(int x, int y) . The logic of the division is the following: Integer division without remainder. Program should consequently subtract the number y from number x (assuming x>y) and return the accumulated value of such subtractions. Also it should take into account the signs of both x and y. No usage of built-in math functions such as "abs".

designed our first experiment to provide both the original version of Listing 1 and an anonymized (obfuscated) version of the same program, i.e., a version whose identifiers are meaningless.

To investigate how LLMs behave when tasked with generating test cases, we designed four different prompts: three targeting black-box testing and one targeting white-box testing. Table 2.1 presents the four prompts we developed.

In the first two prompts, the code under test is included as part of the context, while the third prompt has the same text as WB2 but refers to a different context which only contains an anonymized version of the tested method. The fourth prompt is designed for black-box testing and is based solely on a detailed description of the method's signature and behavior.

2.2 Generated tests

Each of the four prompts was executed in a separate context, and generated around 10 test cases. In three of them, we asked the LLM to generate more tests, resulting in 5 to 9 additional test cases. Following [ACF+24], for each test we checked:

- that the test did build,
- that the test was correct,
- its level of structural test coverage,
- that failing tests revealed bugs in the original program.

2.2.1 Tests not building

All the test cases generated by the LLM compiled successfully, but in some instances, we had to manually adapt the code because the LLM produced test cases that mixed JUnit 4 and JUnit 5 syntax. This issue could be avoided by explicitly specifying which version of JUnit should be used. To address this, we incorporated this requirement into the "best practices" section of the prompts, which effectively resolved the problem. Actually it appeared that the risk of non compilable code depends on the size of the LLM and the complexity of the code. Bigger LLMs, such as the ones available on-line, tend to produce better code.

2.2.2 Erroneous Tests

We manually verified whether the test cases generated by the LLMs predicted the correct results, and found errors in many test suites. For example, one test case incorrectly expected the result of dividing 11 by 2 to be 4. Such hallucinations can be filtered out by repeating the call to the LLM and keeping the majority answer. Also, several test cases involving division by zero misspelled the error message produced by the exception (Listing 1, line 14). In one case, a black-box test suite assumed a plausible, but incorrect, error message ("Division by zero is not allowed.") and generated a test case based on this assumption, causing the test to fail.

These examples illustrate the limitations of LLMs to generate test cases, especially in the context of black-box testing: without access to the code, the LLM cannot accurately infer implementation details, such as specific exception messages. However, potential solutions could be considered to address this issue. For example, providing the LLM with a comprehensive description of the method's behavior, covering not only normal use cases but also error conditions and the exact exceptions that can be raised, could improve the quality of the generated test cases.

2.2.3 Code Coverage

The first test suites that we generated included around 10 test cases. Measuring coverage is a convenient way to evaluate the quality of the generated test suites. In this experiment, we used EclEmma[9], embedded into Eclipse. Other experiments were carried out using the coverage tool of IntelliJ.With EclEmma, all test suites, including the one resulting from WB3 (anonymized code) and BB1 (black-box testing) reached a level of 93.2%. Surprisingly, in some cases, the coverage diminishes at the second generation attempt. So, this halting criterion appears as rather weak, and must be used with care. We also note that we did not see any impact of non-anonymized code on possible biases that could lead the LLM to faulty test generation. This could come from the fact that the example is quite simple, consisting of only one method. This experiment should be extended to various examples of different sizes.

[9] `https://www.eclemma.org/`

2.2.4 Tests That Reveal Bugs

When a test fails or falls into an infinite loop, it often reveals a faulty code, unless the test case is erroneous (see above). Integer division is a well understood arithmetic operator where we do not expect to detect bugs. Yet, several test cases use Integer.MIN_VALUE and Integer.MAX_VALUE and have to take into account the consequences of overflow (e.g., adding 1 to Integer.MAX_VALUE gives -2147483648, i.e., Integer.MIN_VALUE; also -1*Integer.MIN_VALUE = Integer.MIN_VALUE). This led to failing test cases and infinite loops. For example, the following test, which is not erroneous, falls into an infinite loop:

```
1 public void testMinDivisor() {
2       assertEquals(0, IntDivision.IntDiv(1, Integer.MIN_VALUE));
3 }
```

Listing 2: Test generated by the LLM causing an infinite loop.

This observation highlights the necessity of providing additional guidance to the LLM — which we included in the prompt under the "best practices" section — such as specifying timeouts. By doing so, the LLM was able to generate timeout annotations (e.g., `@Test(timeout = 1000)`) with an arbitrary value, such as 1 second in this case. This was essential to ensure that the generated test cases could effectively reveal faults in the program, particularly those involving infinite loops or long-running operations.

We performed similar experiments with other LLMs, including Claude AI and Deepseek which have been specifically trained for code generation. Most LLMs easily reached 100% of line and branch coverage. But surprisingly, half of our requests failed to detect the bug. We suspect that this diversity of behaviours may be explained by the “temperature” of the LLM, which was not taken into account during our experiments.

3 Related work

Fan et al. [FGH+23] reviewed how LLMs are being used based on a large number of related research papers and pointed out several open challenges in the context of software engineering. Focusing on software testing only, Wang et al. [WHC+24] conduct a large-scale survey of over 100 papers using LLMs for software automation. They identify core use cases such as test generation, oracle creation, debugging, and repair, while also highlighting key challenges like flaky test detection, hallucinated code, and poor assertion quality.

[ACF+24] describes how an LLM is used at Meta in order to augment and improve regression test suites. Generated tests are filtered in three stages (build, erroneous test cases, additional coverage). This paper delved deeper into using LLMs to solve classical software testing problems such as the equivalent mutants, having some encouraging results by using an LLM-as-judge to evaluate whether two mutants are equivalent or not.

Codamosa [LILS23] demonstrates that once a *search-based software engineering* algorithm starts to generate tests that no longer increase the coverage percentage, it is beneficial to prompt an LLM with low coverage parts of code. Bhatia et al. [BGKJ23] combines ChatGPT with Pynguin, a search-based testing tool, for generating unit tests in Python. While ChatGPT produces syntactically correct and relevant tests, up to 58% of the generated assertions are incorrect. However, combining both tools improves statement coverage, showing complementary strengths. TICODER [FNS+24] is a test-driven LLM-based code generation framework that iteratively refines code using test feedback. It focuses on disambiguating implementations through user-validated test outcomes to improve code correctness.

While most of the existing literature focuses on white-box testing, often emphasizing metrics such as code coverage, black-box testing remains relatively underexplored. Yet, white-box testing carries the risk of introducing biases in presence of faulty implementation, which must be carefully analyzed. Our initial experiment serves as a preliminary study aimed at examining the impacts of both white-box and black-box testing on test case generation. We believe that combining these two approaches in a unified framework could be a promising direction to enhance the quality and effectiveness of automatically generated test suites.

4 Conclusion and Future Work

The use of LLMs opens new possibilities for automating test case generation, but it also introduces several challenges. On one hand, the anticipated productivity gains should not be undermined by the effort required to verify generated test cases, which may contain errors, and to integrate them into existing test suites, e.g. fixing incorrect assertions, removing tests that fail to build or pass, or correcting syntax errors often caused by improper use of the test framework versions. This highlights the need for more experimental studies to evaluate how LLMs perform in the task of generating test cases. In particular, attention should be given to effective prompt design (commonly known as *prompt engineering*) as a means to address issues, such as missing timeout specifications or improper use of the test framework.

On the other hand, while much of the existing literature focuses on improving code coverage through techniques like regression testing and mutation testing, there is a notable lack of feedback on how bugs and flawed implementations might bias LLMs, leading them to produce semantically invalid test cases, yet valid with respect to the code under test. Biases in the generation process of LLMs are a well-known issue in the literature and could substantially influence the quality and correctness of generated test cases. However, this issue has not been sufficiently explored in our preliminary work. For instance, we have not yet investigated how the integer overflow problem in the integer division case study might result in incorrect assertions being generated. As part of our future work, we plan to investigate whether techniques such as code obfuscation or leveraging reasoning-enhanced models could help mitigate the risk of biased test generation.

The case study presented in this paper involves a small program consisting of a single method; nevertheless, it highlights several challenges commonly encountered in test case generation, including hallucinations, incorrect test generation, omission of corner cases, and improper use of the test framework. Clearly, this preliminary work must be extended to encompass more complex and realistic examples. Moreover, this study focuses exclusively on prompt formulation as a means to optimize the generation process. However, several other factors may significantly influence the quality of generated test cases, such as the size of the language model, the composition of its training dataset, the complexity of the application under test, the programming language employed, the overall quality of the application, and the presence or absence of code smells. We aim to conduct more comprehensive experiments addressing these factors in the coming months.

Acknowledgements

This research is supported by the LLM4TestGen project of the LIG laboratory.

References

[ACF+24] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. Automated unit test improvement using large language models at meta. In Marcelo d'Amorim, editor, *Companion Proceedings of the 32nd ACM Int. Conf. on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pages 185–196. ACM, 2024.

[BGKJ23] Shreya Bhatia, Tarushi Gandhi, Dhruv Kumar, and Pankaj Jalote. Unit test generation using generative AI: A comparative performance analysis of auto-generation tools. In *2023 IEEE/ACM 45th Int. Conf. on Software Engineering (ICSE)*, 2023. `https://arxiv.org/abs/2304.11757`.

[FGH+23] Angela Fan, Beliz Gokkaya, Mark Harman, Mitya Lyubarskiy, Shubho Sengupta, Shin Yoo, and Jie M Zhang. Large language models for software engineering: Survey and open problems. In *2023 IEEE/ACM Int. Conf. on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, pages 31–53. IEEE, 2023.

[FNS+24] Sarah Fakhoury, Aaditya Naik, Georgios Sakkas, Saikat Chakraborty, and Shuvendu K. Lahiri. LLM-based test-driven interactive code generation: User study and empirical evaluation. *IEEE Trans. Software Eng.*, 50(9):2254–2268, 2024.

[LILS23] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings of the 45th Int. Conf. on Software Engineering*, ICSE '23, page 919–931. IEEE Press, 2023.

[WHC+24] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Trans. Software Eng.*, 50(4):911–936, 2024.

Projet VÉDYSEC : Vérification Dynamique de Propriétés de Sécurité

Nikolai Kosmatov[1], Frédéric Loulergue[2], and Julien Signoles[3]

[1]Thales Research & Technology, cortAIx Labs, Palaiseau, France
nikolai.kosmatov@thalesgroup.com
[2]Univ. Orléans, INSA Centre Val de Loire, LIFO EA 4022
frederic.loulergue@univ-orleans.fr
[3]Université Paris-Saclay, CEA, List, Palaiseau, France
julien.signoles@cea.fr

Cette soumission présente le projet Astrid Maturation VÉDYSEC (Vérification Dynamique de Propriétés de Sécurité) qui a été soumis et accepté en 2024, et démarre début 2025, avec une durée de 24 mois. Il porte sur les techniques de spécification et vérification à l'exécution de propriétés de sécurité et vise le passage du TRL 4 au TRL 6. Le consortium inclut Thales TRT (coordinateur : Nikolai Kosmatov), le LIFO (responsable scientifique : Frédéric Loulergue) et le CEA List (responsable scientifique : Julien Signoles). Ce projet fait suite aux thèses CEA-DGA de Dara Ly [8] et de Virgile Robles [10].

Contexte

Les progrès récents dans les techniques de vérification des logiciels ont conduit au développement de plusieurs outils de vérification. L'un des plus avancés d'entre eux, Frama-C [6, 1, 7], développé par le CEA List, est un cadre collaboratif *open source* pour l'analyse des programmes C. Il est construit autour d'un noyau qui fournit des services de base aux greffons proposant divers analyses. Les greffons communiquent via des annotations écrites en ACSL (*ANSI C Specification Language*) [2] permettant l'utilisation de plusieurs techniques pour l'analyse et la vérification de programmes C de manière collaborative [3]. L'analyse de valeurs par interprétation abstraite peut être réalisée à l'aide du greffon Eva, la vérification déductive des programmes annotés en ACSL peut être réalisée en utilisant le greffon Wp, la vérification à l'exécution d'annotations ACSL est offerte par le greffon E-ACSL [13].

Cette dernière technique a pour but de traduire une sous-classe des annotations ACSL — celles dites exécutables [14] — en instructions C intégrées au programme sous analyse. Cette transformation permet d'obtenir un nouveau programme C dont la correction vis-à-vis de sa spécification est vérifiée dynamiquement, pendant son

exécution. Les annotations sont ainsi vérifiées lors de l'exécution du programme sur des cas de test, ce qui permet de rapporter à l'utilisateur comme verdict tout échec d'annotation détecté (ou l'absence de tels échecs).

La spécification et la vérification d'exigences de haut niveau — telles que des propriétés de sécurité comme l'intégrité ou la confidentialité des données — sur du code de taille conséquente est une activité importante pour l'industrie. Ces propriétés peuvent être exprimées dans Frama-C sous la forme de propriétés globales sur le code (appelées *méta-propriétés*) grâce à une extension d'ACSL, nommée Hilare, permettant d'exprimer de telles propriétés. Les méta-propriétés peuvent ensuite être traduites en annotations ACSL classiques qui peuvent être vérifiées statiquement ou dynamiquement. Ce cadre conceptuel dans Frama-C a été réalisé sous forme d'un greffon nommé MetAcsl et a été appliqué à plusieurs exemples [11, 9, 12].

Thales a utilisé MetAcsl pour spécifier et prouver les propriétés de sécurité d'une machine virtuelle JavaCard afin de réaliser la certification de la carte à puce de Thales au plus haut niveau d'évaluation (EAL7) des Critères Communs. L'approche MetAcsl s'est avérée à la fois extrêmement pratique, pragmatique et rigoureuse. Le nombre des méta-propriétés étant réduit, il est beaucoup plus simple de les écrire et les relire pour avoir une bonne vision de ce qui est spécifié et vérifié. L'instanciation systématique des méta-propriétés en assertions assure de ne pas passer à côté de la moindre erreur, à condition de prouver ces assertions par la suite. Cette démarche a été reconnue par la communauté scientifique, un CESTI et l'ANSSI [4, 5].

Verrous, objectifs et programme de travail

L'utilisation de MetAcsl avec un objectif de vérification dynamique de propriétés de sécurité a été très peu étudiée. L'utilisation conjointe de MetAcsl avec E-ACSL n'a été que rapidement expérimentée sur des petits exemples [9] et nécessite plusieurs extensions des outils.

Le principal objectif du projet VÉDYSEC est donc le suivant :

$\mathcal{OG}$: la maturation de l'outillage et la consolidation des méthodologies d'application pour la spécification et la vérification dynamique de propriétés de sécurité.

Le programme de travail inclut une étude approfondie des besoins d'extensions, une consolidation de la vérification des annotations dans E-ACSL avec une conception rigoureuse, une extension d'E-ACSL et de MetAcsl pour les propriétés de haut niveau, ainsi qu'une réflexion méthodologique et une évaluation sur des cas d'études ouverts et industriels.

Références

[1] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs : The Frama-C software analysis platform. *Commun. ACM*, 2021.

[2] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2018. `http://frama-c.com/acsl.html`.

[3] Loïc Correnson and Julien Signoles. Combining analyses for C program verification. In *Formal Methods for Industrial Critical Systems (FMCIS)*. Springer, 2012.

[4] Adel Djoudi, Martin Hána, and Nikolai Kosmatov. Formal verification of a JavaCard virtual machine with Frama-C. In *Formal Methods (FM)*. Springer, 2021.

[5] Adel Djoudi, Martin Hána, Nikolai Kosmatov, Milan Kříženecký, Franck Ohayon, Patricia Mouy, Arnaud Fontaine, and David Féliot. A bottom-up formal verification approach for common criteria certification : Application to JavaCard virtual machine. In *European Congress on Embedded Real-Time Systems (ERTS)*, 2022. Best paper award (category "Processes, Methods and Tools").

[6] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C : A software analysis perspective. *Formal Asp. Comput.*, 2015.

[7] Nikolai Kosmatov, Virgile Prevosto, and Julien Signoles, editors. *Guide to Software Verification with Frama-C. Core Components, Usages, and Applications*. Computer Science Foundations and Applied Logic Book Series. Springer, 2024.

[8] Dara Ly. *Formalisation d'un vérificateur dynamique de propriétés mémoire pour programmes C*. PhD thesis, Univ. Orléans, 2022.

[9] V. Robles, N. Kosmatov, V. Prevosto, L. Rilling, and P. Le Gall. Tame your annotations with MetAcsl : Specifying, testing and proving high-level properties. In *Tests and Proofs (TAP)*. Springer, 2019.

[10] Virgile Robles. *Specifying and Verifying High-Level Requirements on Large Programs : Application to Security of C Programs*. PhD thesis, Univ. Paris-Saclay, 2022.

[11] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. MetAcsl : Specification and verification of high-level properties. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, 2019.

[12] Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling, and Pascale Le Gall. Methodology for specification and verification of high-level properties with MetAcsl. In *Formal Methods in Software Engineering (FormaliSE)*. IEEE, 2021.

[13] J. Signoles, N. Kosmatov, and K. Vorobyov. E-ACSL, a Runtime Verification Tool for Safety and Security of C Programs. In *Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CuBES)*. EasyChair, 2017.

[14] Julien Signoles. *E-ACSL : Executable ANSI/ISO C Specification Language*. `http://frama-c.com/download/e-acsl/e-acsl.pdf`.

ANR RAPID VVaMIA

Présentation de projet

Frédéric Dadeau[1], Fabrice Bouquet[1], Eléa Jacquin[1], Dorine Tabary[1], Louis Lefevre[1], Bruno Legeard[2], Antoine Chevrot[2], Arnaud Bouzy[2], Bruno Besace[3], Fabien Lamontre[3]

[1] Université Marie et Louis Pasteur, CNRS, institut FEMTO-ST, Besançon
[2] Smartesting Solutions et Services - Besançon
[3] Thalès LAS - Élancourt

Résumé

Ce papier présente les motivations, objectifs et travaux prévus dans le projet ANR RAPID VVaMIA, porté par l'entreprise Smartesting Solution & Services. Le consortium inclut Thalès LAS, ainsi que FEMTO-ST. Ce projet s'intéresse au couplage entre l'IA générative et les modèles, dans le contexte de la vérification et la validation de systèmes embarqués en utilisant des tests logiciels. Ce projet vise à la production d'assistants IA facilitant les tâches de conception et de maintenance des tests logiciels.

Mots-clés : IA générative, modèle, assistance, tests logiciels

1 Présentation synthétique du projet VVaMIA

Le projet VVaMIA est financé sur l'appel ANR RAPID [1] 2024. Il est prévu sur une durée de 24 mois et a officiellement démarré début février 2025. Ce projet réunit 3 partenaires : **Smartesting**, le coordinateur du projet, PME dans le domaine de l'automatisation du processus de test, historiquement spécialisé dans le test à partir de modèles, et situé à Besançon ; **Thalès LAS**, une filiale du groupe Thalès qui développe une gamme complète de solutions optroniques pour les systèmes embarqués terrestres et aériens (Land and Air Systems), située à Elancourt ; le département informatique **DISC de l'institut FEMTO-ST**, Université Marie et Louis Pasteur [2] à Besançon, dont une partie des membres est spécialisée dans le test logiciel en utilisant des méthodes formelles ou semi-formelles.

L'acronyme VVaMIA signifie **V**érification et **V**alidation des systèmes embarqués **a**ugmentés par les **M**odèles et l'**IA** générative. Le projet s'intéresse donc à l'utilisation de l'IA générative pour venir renforcer les approches de test à partir de modèles (MBT) [1]. Il s'agit d'un projet de recherche industrielle dans lequel les différents partenaires étudieront les bénéfices de l'utilisation de l'IA générative dans la chaîne de V&V logicielle.

1. Régime d'APpui à l'Innovation Duale - programme porté par l'Agence de l'Innovation Défense
2. ex-Université de Franche-Comté

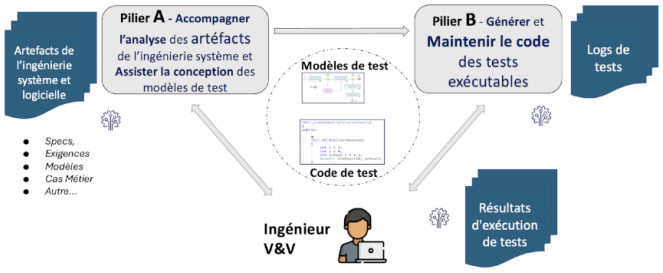


FIGURE 1 – Schéma général des travaux du projet VVaMIA

2 Motivations et objectifs du projet VVaMIA

Depuis quelques années, les Large Language Models [3] arrivent à un niveau de maturité considérable tant dans leur capacité conversationnelle, que dans leur capacité à générer des raisonnements construits, allant jusqu'à la production de code exécutable pertinent [4]. S'il n'est pas d'actualité de remplacer des équipes de développement par ce type d'outil, diverses initiatives s'interrogent sur les gains potentiels à espérer de l'usage de ce type d'outil d'IA dite génératrice [7]. Le présent projet vise à explorer et mettre en oeuvre ce type de technique dans le contexte de la validation à base de tests logiciels.

Les outils de tests à partir de modèles [6] développés depuis près d'une vingtaine d'années ont pour objectif de simplifier la tâche de conception des tests, en automatisant leur génération à partir d'un modèle décrivant le comportement du système. Néanmoins, si le gain du point de vue de l'écriture et de l'exécution des tests est conforme aux attentes, la fréquente complexité des systèmes modélisés conduit à la production de modèles eux aussi complexes, dont la conception se relève être, au final, une tâche de plus en plus ardue.

Dans ce contexte, le projet VVaMIA vise à explorer et mettre en oeuvre des techniques à base d'IA générative pour assister l'ingénieur validation. L'objectif général du projet VVaMIA consiste à apporter des assistants basés sur l'IA générative aux activités les plus critiques et consommatrices des activités de V&V, pour accélérer la phase de vérification et validation de systèmes embarqués dans les activités de conception et maintenance des tests au niveau système et logiciel.

3 Programme des travaux

Les travaux s'articulent autour de 2 piliers, nommés A et B, qui se positionnent en amont et en aval de l'outil Yest [2] de Smartesting permettant la description de cas de tests à partir d'une modélisation de type workflow, combinée avec des tables réprésentant les règles métier sur les données logiques, à partir de laquelle des tests sont produits.

Comme illustré en figure 1, le pilier A s'intéresse à assister l'ingénieur validation dans la phase amont, visant à concevoir et maintenir un modèle de test et le code exécutable des tests associés à partir des artefacts usuels d'un projet logiciel de grande ampleur. Le pilier B se focalise sur la phase aval et aura pour objectif de générer et maintenir le code des tests exécutable sur les bancs, ainsi que d'assister le dépouillement des tests exécutés.

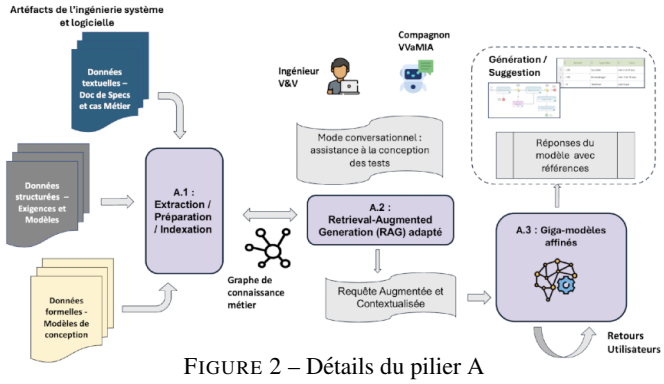


FIGURE 2 – Détails du pilier A

3.1 Pilier A – Assistance à la création/maintenance d'un modèle de test

Le processus proposé dans le pilier A est schématisé en figure 2. Il comporte plusieurs étapes.

La première étape (A.1) consistera à exploiter les artefacts classiques d'un projet logiciel qu'il s'agisse de données textuelles (spécifications métier), de données structurées (exigences), ou de données plus formelles (modèles de conception). L'analyse de ces documents produit un graphe de connaissances métier sur lequel s'appuiera l'étape suivante.

La deuxième étape (A.2) s'intéressera à utiliser des techniques de type RAG (Retrieval-Augmented Generation) [5] pour exploiter ces connaissances et ainsi extraire les différents artefacts utiles pour la création d'un modèle de test (par exemple, aiguiller vers les exigences pertinentes vis-à-vis de l'incrément considéré).

La troisième et dernière étape (A.3) aura pour objectif d'intégrer les récupérations effectuées par le RAG pour le fine-tuning d'un LLM permettant l'interaction avec l'ingénieur validation ayant pour objectif la conception d'un modèle de test pertinent.

3.2 Pilier B – Exécution des tests et retours

Comme illustré par la figure 3, le pilier B a pour objectif général l'implantation, l'exécution et la maintenance des script de tests à partir des cas de tests créés par les modèles issus du pilier A.

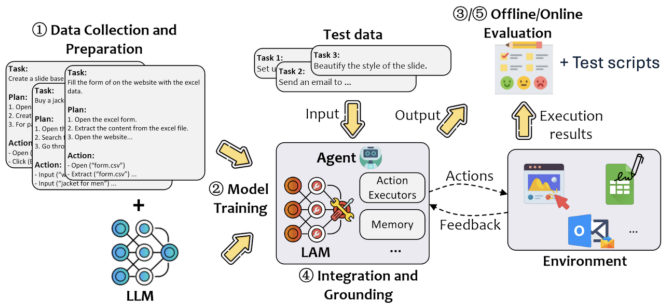


FIGURE 3 – Détails du pilier B

La solution mise en oeuvre dans cette partie s'appuiera sur l'utilisation d'agents IA (LAM) [8]. Il s'agit d'entités, appelées agents, capables de concrétiser dans le monde informatique des décisions prises par un LLM. Les agents IA ainsi définis seront en charge de la création initiale des tests ainsi que de l'évaluation de ceux-ci et de leur maintenance, à partir des évolutions du banc ou du modèle, et en interaction avec ceux-ci.

Les étapes de la création des scripts de tests sont les suivantes. Il s'agit d'abord d'associer des mots-clés issus des actions du modèle à des actions concrètes du système (suite d'appels d'API), puis de calculer les données de test visant à instancier les éventuels paramètres associés. La génération de code exécutable dans le format du banc de test employé intervient ensuite. On termine par le dépouillement des verdicts de test. Cette approche permettra d'améliorer l'efficacité et la qualité des tests fonctionnels, tout en réduisant les efforts manuels requis pour leur mise en œuvre.

4 Résultats attendus

Le projet VVaMIA se focalise sur l'utilisation pertinente de l'IA générative comme assistant. Il ne s'agit pas de l'employer comme un *marteau doré* mais plutôt de tirer bénéfice des capacités de celles-ci aux étapes-clé de la conception et de la maintenance d'un référentiel de test, qu'il s'agisse de la génération d'un modèle, ou de l'exécution et le dépouillement des tests abstraits sur le système.

Les résultats espérés de ce projet sont un gain substantiel de productivité des ingénieurs validation mesurable par une réduction des efforts et de la durée d'un cycle V&V, une aide pour assurer la définition et la maintenance de modèles et de scripts de test automatisés. Les expérimentations seront réalisés dans le cadre des applications développées par Thalès, pour qui les enjeux de sécurité des données traitées, de souveraineté des technologies employées, et de performance dans les temps de traitement sont primordiales. En ce sens, une solution opérant sur des LLMs exécutés avec une infrastructure sur site, et donc découplée du réseau, sera recherchée.

Références

[1] B. Beizer. *Black-box Testing : Techniques for Functional Testing of Software and Systems*. J. Wiley & Sons, Inc., New York, NY, USA, 1995.

[2] E. Bernard, F. Ambert, B. Legeard, and A. Bouzy. Lightweight model-based testing for enterprise it. In *IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 224–230, 2018.

[3] J. Chu-Carroll, A. Beck, G. Burnham, D. O. Melville, D. Nachman, A. E. Özcan, and D. Ferrucci. Beyond llms : Advancing the landscape of complex reasoning, 2024.

[4] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation, 2024.

[5] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel, and D. Kiela. Retrieval-augmented generation for knowledge-intensive nlp tasks. In *Proc. of the 34th Int. Conf. on Neural Information Processing Systems*, NIPS '20, Red Hook, NY, USA, 2020. Curran Associates Inc.

[6] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verif. Reliab.*, 22(5) :297–312, August 2012.

[7] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language models : Survey, landscape, and vision. *IEEE Trans. Softw. Eng.*, 50(4) :911–936, April 2024.

[8] Y. Wang, Y. Pan, Z. Su, Y. Deng, Q. Zhao, L. Du, T. H. Luan, J. Kang, and D. Niyato. Large model based agents : State-of-the-art, cooperation paradigms, security and privacy, and future trends, 2025.