Proceedings "Approches formelles dans l'assistance au développement du logiciel (AFADL)", Vannes, 2022

J. Christian Attiogbé, Fatiha Zaïdi

Christian ATTIOGBÉ, Fatiha ZAÏDI (Editors)

Actes des Journées/*Proceedings* ”Approches formelles dans l’assistance au développement du logiciel (AFADL)”

(afadl2022.ls2n.fr)
tenues conjointement avec les
Journées du Groupement de Recherche national - Génie de la Programmation et du Logiciel
GDR-GPL 2022 à Vannes - Morbihan
(gdr-gpl-2022.sciencesconf.org)

7-10 juin 2022 / June 7-10, 2022
Vannes, 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)

Editorial boards

Christian ATTIOGBÉ, Fatiha ZAÏDI

(Co-présidents/Co-chairs AFADL'2022, Vannes)

Comité de programme / Programme Committee

Yamine Ait Ameur	IRIT/INPT-ENSEEIHT, Toulouse
Ameur-Boulifa Rabéa	Telecom-ParisTech, Paris
Christian Attiogbé	Université de Nantes
Sandrine Blazy	IRISA, University de Rennes 1
Sylvain Conchon	Universite Paris Saclay
David Deharbe	ClearSy System Engineering
David Delahaye	LIRMM, Université de Montpellier, France
Delphine Demange	IRISA, Rennes
Catherine Dubois	ENSIIE-Samovar
Mohammed Foughali	Université de Paris/IRIF
Marc Frappier	Université de Sherbrooke, Canada
Hugo Gimbert	CNRS, LABRI, Bordeaux, France
Pierre-Cyrille Heam	FEMTO-ST - INRIA - CNRS
Ludovic Henrio	CNRS, Grenoble
Olivier Hermant	MINES ParisTech, Paris
Aurélie Hurault	IRIT/INPT-ENSEEIHT, Toulouse
Akram Idani	Laboratoire d'Informatique de Grenoble
Mathieu Jaume	Sorbonne Université - UPMC - CNRS LIP6 UMR
Thomas Lambolais	LGI2P, IMT Mines Alès
Pascale Le Gall	CentraleSupelec, Paris
Mehdi Lhommeau	ISTIA, Université d'Angers
Nicole Levy	Cedric, CNAM, Paris
Delphine Longuet	Thales R&T
Micaela Mayero	Université Paris 13, Paris
Stephan Merz	INRIA Nancy
David Monniaux	(CNRS / VERIMAG)
Ileana Ober	IRIT, Universite de Toulouse
Ioannis Parissis	Université Grenoble Alpes - Grenoble INP
Pascal Poizat	Université Paris Nanterre and LIP6
Thomas Polacsek	ONERA, Toulouse
Marc Pouzet	LIENS, Paris
Jean-Baptiste Raclet	IRIT, Touluse
Romain Rouvoy	Cristal, Université de Lille
Vlad Rusu	INRIA
Sebastien Salva	LIMOS, Université Clermont Auvergne)
Julien Signoles	CEA LIST
Benoît Valiron	LMF - CentraleSupelec, Université Paris Saclay
Laurent Voisin	Systerel
Virginie Wiels	ONERA / DTIM, Toulouse
Fatiha Zaidi	Université Paris Saclay
Tewfik Ziadi	Sorbonne Université-CNRS 7606, LIP6

Contents

Préface

Ces journées à Vannes (dans le Morbihan), poursuivent les rencontres annuelles de la communauté AFADL et plus largement des groupes de recherche connexes. 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.

Lors de cette édition de 2022, 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.

Les articles présentés à AFADL et listés dans les présents actes relèvent de différentes catégories, suscitées lors de l'appel à communications.

- Des articles présentant de nouveaux travaux ou résultats de travaux académiques ou industriels non encore publiés. L'article de Hugo Gimbert, Pierre Castéran, Claire Mathieu

and Gérald Point (page 24) est dans cette catégorie ; il présente de nouveaux resultats de preuve d'algorithme, et est sélectionné conjointement avec le GT LVP. L'article de Lydie Du Bousquet et Christophe Saint Marcel (page 8) présente des résultats de travaux sur l'enseignement des tests. L'article de Farid Arfi, Anne-Lise Courbis, Thomas Lambolais, François Bughin and Maurice Hayot (page 52) présente un retour d'expérience de travaux en cours sur la combinaison de spécifications semi-formelle et formelle.

- Des travaux et résultats de 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 : les articles de Mickaël Laurent (page 33), de Theo De Castro Pinto (page 19), de Thaïs Baudon, Gabriel Radanne et Laure Gonnord (page 38) et de Perla Tannoury, Samir Chouali et Ahmed Hammad (page 48).

- Dans la catégorie des articles relevant de la présentation d'outils démontrant la pertinence et les performances d'outils de recherche académique ou industrielle, nous avons les articles de Alain Giorgetti et Jérome Ricciardi (page 4, résumé d'un article publié dans le journal *Software Quality Journal*, 2022) et de Akram Idani, German Vega (page 44).

- Des contributions présentant quelques résultats obtenus dans le cadre de projet nationaux ou internationaux ; l'article de Bruno Legeard, Roland Groz et Abbas Ahmad (page 12) est dans cette catégorie et présente des résultats obtenus dans le cadre d'un projet ANR.

- Des résumés d'articles récemment publiés dans des conférences ou journaux internationaux, afin de communiquer aussi au niveau national sur les chercheurs de notre communauté. Ce sont les articles suivants : Dylan Marinho (page 65) publié dans le journal *Transactions on Software Engineering and Methodology* (TOSEM, 2022), Étienne André, Masaki Waga, Natsuki Urabe and Ichiro Hasuo (page 72), publié à la conférence *Nasa Formal Methods* (NFM, 2022), Arthur Clavière, Laura Altieri Sambartolomé, Eric Asselin, Christophe Garion and Claire Pagetti (page 61) publié dans la conférence *ACM International Conference on Hybrid Systems: Computation and Control* (HSCC, 2022), Adel Djoudi, Martin Hána and Nikolai Kosmatov (page 69), publié dans la conférence *Formal Methods* (FM, 2021).

Nous remercions ici nos collègues qui ont permis et assuré le bon déroulement de ces journées : Isabelle Borne, Salah Sadou et tous les collègues de Vannes ; 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 (Alain Giorgetti, Julien Signoles).

Christian ATTIOGBÉ, Fatiha ZAÏDI
Avril 2024

AFADL & MTV2

Test aléatoire et énumératif pour OCaml et Why3*

Alain Giorgetti[1] Jérome Ricciardi[2]
Clotilde Erard
[1] Institut FEMTO-ST, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France
alain.giorgetti@femto-st.fr
[2] Université Paris-Saclay, LMF/QuaCS, CEA/LIST/LSL, Palaiseau, France
jerome.ricciardi@cea.fr

Résumé

Nous présentons AutoCheck, un prototype d'outil de test aléatoire et énumératif de propriétés définies dans le langage fonctionnel OCaml ou dans le langage WhyML de la plateforme de vérification déductive Why3. Une originalité est que les tests énumératifs utilisent des générateurs de données eux-mêmes écrits dans le langage WhyML, et dont la correction et la complétude sont formellement prouvées avec Why3. Une autre spécificité est que l'effort de développement est réduit en exploitant le mécanisme d'extraction de Why3 vers OCaml et un outil de test aléatoire existant pour OCaml.

1 Introduction

Nous présentons la version 0.1.2 du prototype AutoCheck [EGLR21], pour tester des propriétés exécutables définies dans le langage fonctionnel OCaml [1] ou dans le langage WhyML de spécification et de programmation de la plateforme de vérification déductive Why3 [BFM+18]. AutoCheck vise l'intégration des techniques complémentaires de génération aléatoire et énumérative de données de test. Dans ce but, il exploite le mécanisme d'extraction vers OCaml de Why3, la bibliothèque ENUM de programmes d'énumération spécifiés et implémentés en WhyML et vérifiés avec Why3 [EG19], et l'outil QCheck de test aléatoire pour OCaml [CGDM20]. AutoCheck complète ce dernier avec des tests aléatoires pour les propriétés WhyML et des tests énumératifs pour les propriétés WhyML et OCaml. En s'interfaçant avec les programmes d'énumération certifiés d'ENUM, AutoCheck propose des tests énumératifs certifiés.

Ce résumé est organisé de la manière suivante. La partie 2 présente quelques aspects du test de propriétés. La partie 3 présente les principes de conception et de fonctionnement d'AutoCheck.

2 Méthodes et outils de test de propriétés

Le test de propriétés (PBT, pour *Property-Based Testing*) de programmes consiste à identifier et à tester un ensemble de propriétés que certaines fonctions doivent satisfaire. Au-delà du cas de base des contrats de fonction, qui sont des propriétés qui portent sur un seul appel d'une unique fonction, les

*Cet article est un résumé étendu d'un article [EGR22] publié le 22/02/2022 dans le *Software Quality Journal*.

1. https://ocaml.org.

propriétés relationnelles sont des propriétés qui peuvent concerner plusieurs fonctions et/ou plusieurs appels de la même fonction [BKLG+18].

Les deux approches les plus populaires pour automatiser le test de propriétés sont le test aléatoire et le test énumératif. Le test aléatoire (RT, pour *Random Testing*) utilise des générateurs pseudo-aléatoires pour produire automatiquement des cas de test. L'ancêtre des outils de test aléatoire de propriétés est QuickCheck [CH00]. Initialement écrit pour le langage Haskell, il a été adapté à plus de trente langages de programmation [2] Parmi ces outils, QCheck [CGDM20] est un outil de test aléatoire de propriétés de fonctions OCaml [MM17]. Il fournit de nombreux combinateurs utiles pour générer différents types de données, et permet également aux utilisateurs d'écrire leurs propres générateurs, en particulier pour les types inductifs. Comme la plupart des variantes de QuickCheck, QCheck fournit également une fonction de réduction (*shrinking* en anglais), qui réduit la taille du contre-exemple fourni en cas d'échec du test. Par exemple, si la propriété testée est l'absence d'un nombre donné dans une liste, elle doit fournir comme contre-exemple une liste de longueur 1 contenant uniquement ce nombre. Nous avons choisi d'intégrer QCheck dans AutoCheck, pour servir de base à un outil similaire pour WhyML. Le test énumératif (ET, pour *Enumerative Testing*) a pour objectif de générer toutes les entrées possibles des fonctions testées. Lorsque le domaine de ces entrées est trop vaste pour être exploré exhaustivement, le test exhaustif borné (BET, pour *Bounded Enumerative Testing*) n'énumère que les données de taille inférieure à une limite prédéfinie. Le test énumératif a d'abord été utilisé pour vérifier des propriétés de langages fonctionnels, un pionnier étant l'outil SmallCheck pour Haskell [RNL08]. Ensuite, il a été adapté à plusieurs assistants de preuve, par exemple à Isabelle [Bul12] et à Coq [DGG16]. Dans un travail précédent, deux des présents auteurs ont initié un outil de BET pour le langage WhyML de Why3, mais ce prototype était limité aux tableaux d'entiers [EG19].

Le test aléatoire et le test énumératif sont complémentaires. En effet, le test énumératif est utile pour les données de petite taille [DJW12], souvent peu nombreuses, mais il ne convient plus au-delà d'une certaine taille, car le nombre de données croît généralement exponentiellement avec leur taille. Le test aléatoire peut générer des données de grande taille, mais il est peu efficace pour invalider une propriété rarement fausse, tandis qu'un test énumératif borné peut soit en trouver un plus petit contre-exemple, soit prouver qu'il n'en existe aucun en dessous d'une certaine taille.

3 Architecture de l'outil

Cette partie présente l'architecture et les principes de fonctionnement du prototype AutoCheck.

La structure interne d'AutoCheck est représentée dans la figure 1. L'outil lui-même est représenté par le plus grand rectangle aux coins arrondis. Les fichiers générés automatiquement sont distingués par des rectangles en pointillés. Le code externe est indiqué en italique.

Chaque entrée d'AutoCheck est représentée par un rectangle avec des coins carrés. Il s'agit d'un fichier WhyML ou OCaml (respectivement nommé Tests.mlw ou Tests.ml dans la figure) contenant l'implémentation testée et une description des tests. Les propriétés à tester sont des fonctions OCaml ou des fonctions WhyML exécutables par extraction en OCaml, et leurs tests sont des appels de fonctions. Par conséquent, les implémentations à tester, leurs propriétés et les tests de ces propriétés peuvent être répartis dans plusieurs fichiers, comme dans n'importe quelle application OCaml ou WhyML.

2. Voir, par exemple, fast-check (`https://github.com/dubzzz/fast-check`) pour JavaScript, jetCheck (`https://github.com/JetBrains/jetCheck`) pour Java, PropEr (`https://github.com/proper-testing/proper`) pour Erlang, QuickChick (`https://github.com/QuickChick/QuickChick`) pour Coq ou theft (`https://github.com/silentbicycle/theft`) pour C.

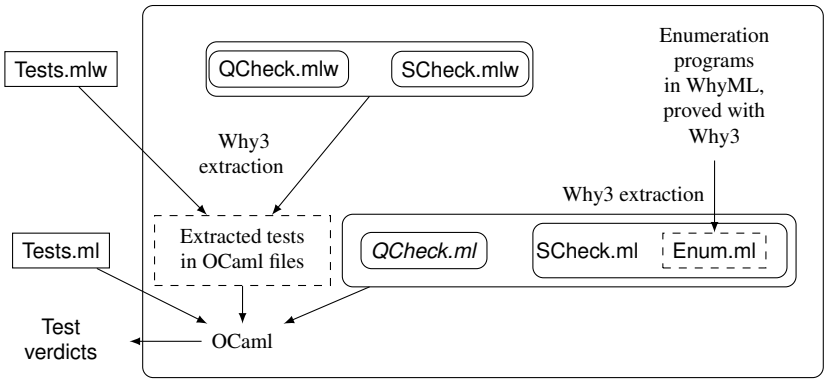


FIGURE 1 – Architecture d'AutoCheck.

Chaque test OCaml exploite un ou plusieurs générateurs de données aléatoires ou énumératifs. Les générateurs de données aléatoires sont définis dans l'outil externe de test aléatoire QCheck, dont le fichier principal est *QCheck.ml* (ce nom de fichier est écrit en italique car ce code externe est utilisé sans modifications). Les générateurs de données énumératifs sont définis dans notre outil de test énumératif pour OCaml, dont le fichier principal est SCheck.ml. Ce dernier encapsule plusieurs programmes d'énumération de la bibliothèque ENUM [EG19]. Ce code OCaml, dans le fichier Enum.ml, est automatiquement extrait par Why3 de programmes d'énumération écrits en WhyML et dont les propriétés de correction et de complétude sont prouvées avec Why3.

Les fichiers QCheck.mlw et SCheck.mlw spécifient en WhyML des fonctionnalités analogues de test aléatoire et énumératif, permettant d'écrire des tests en WhyML (dans Tests.mlw sur la figure). Leur extraction avec Why3 génère des tests en OCaml qui exploitent les outils de test aléatoires et énumératifs pour OCaml définis dans *QCheck.ml* et SCheck.ml.

4 Conclusion

AutoCheck a été conçu en privilégiant la simplicité et la facilité d'utilisation, pour ses utilisateurs et ses développeurs. D'abord, l'installation et l'utilisation sont facilitées et sécurisées par virtualisation avec Docker. Ensuite, de nombreux exemples de tests OCaml (resp. WhyML) sont fournis, dans un fichier nommé TestExamples.ml (resp. TestExamples.mlw). Ils sont ordonnés par complexité croissante et couvrent toutes les fonctionnalités de l'outil. Certains de ces exemples sont documentés dans les parties 5 et 6 de [EGR22]. Au-delà de ces exemples élémentaires, à vocation pédagogique, des exemples de tests énumératifs de fonctions et de propriétés liées aux permutations ont été présentés dans des travaux antérieurs [EG19, Gio21, EGR22].

En tant qu'outil de PBT, AutoCheck est complémentaire de l'outil Monolith [Pot21], qui implémente le test basé sur un modèle, qui doit être une implémentation de référence fournie. Côté OCaml, AutoCheck est proche de l'outil de vérification d'assertions à l'exécution **`ortac`** [FP21].

En OCaml et WhyML, les syntaxes des tests aléatoires et énumératifs ont été choisies pour être aussi similaires que possible. Une perspective est d'unifier les interfaces pour permettre d'écrire des combinaisons arbitraires de ces deux sortes de tests.

Remerciements. Ce travail est soutenu par l'EIPHI Graduate School (contrat ANR-17-EURE-0002).

Références

[BFM+18] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 Platform*, 2018. http://why3.lri.fr/manual.pdf.

[BKLG+18] L. Blatter, N. Kosmatov, P. Le Gall, V. Prevosto, and G. Petiot. Static and dynamic verification of relational properties on self-composed C code. In *Tests and Proofs. TAP 2018*, pages 44–62, Cham, 2018. Springer International Publishing.

[Bul12] L. Bulwahn. The new Quickcheck for Isabelle - random, exhaustive and symbolic testing under one roof. In *CPP 2012*, volume 7679 of *LNCS*, pages 92–108. Springer, Heidelberg, 2012.

[CGDM20] S. Cruanes, R. Grinberg, J.-P. Deplaix, and J. Midtgaard. QuickCheck inspired property-based testing for OCaml., 2020. https://github.com/c-cube/qcheck.

[CH00] K. Claessen and J. Hughes. QuickCheck : a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000.

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

[DJW12] J. Duregård, P. Jansson, and M. Wang. Feat : Functional enumeration of algebraic types. *ACM SIGPLAN Notices*, 12 2012.

[EG19] C. Erard and A. Giorgetti. Bounded exhaustive testing with certified and optimized data enumeration programs. In *Testing Software and Systems. ICTSS 2019*, volume 11812 of *LNCS*, pages 159–175, Cham, 2019. Springer.

[EGLR21] C. Erard, A. Giorgetti, R. Lazarini, and J. Ricciardi. Autocheck 0.1.2, 2021. https://github.com/alaingiorgetti/autocheck.

[EGR22] C. Erard, A. Giorgetti, and J. Ricciardi. Towards random and enumerative testing for OCaml and WhyML properties. *Software Quality Journal*, 30 :253–279, 2022. https://doi.org/10.1007/s11219-021-09572-z.

[FP21] J.-C. Filliâtre and C. Pascutto. Ortac : Runtime Assertion Checking for OCaml (tool paper). In *RV'21 - 21st International Conference on Runtime Verification*, Los Angeles, CA, United States, October 2021.

[Gio21] A. Giorgetti. Théories de permutations avec Why3. In *32ème Journées Francophones des Langages Applicatifs (JFLA)*, pages 202–209, France, 2021. https://hal.archives-ouvertes.fr/hal-03190426.

[MM17] J. Midtgaard and A. Møller. Quickchecking static analysis properties. *Software Testing, Verification and Reliability*, 27(6), 2017. https://doi.org/10.1002/stvr.1640.

[Pot21] F. Pottier. Strong automated testing of OCaml libraries. In *32ème Journées Francophones des Langages Applicatifs (JFLA)*, pages 3–20, France, 2021. https://hal.archives-ouvertes.fr/hal-03190426.

[RNL08] C. Runciman, M. Naylor, and F. Lindblad. SmallCheck and Lazy SmallCheck - automatic exhaustive testing for small values. In *Proceedings of the 1st ACM SIGPLAN Symposium on Haskell*, pages 37–48. ACM, 2008.

De l'adaptation de Caseine pour l'évaluation des tests des étudiants

Lydie du Bousquet, Christophe Saint-Marcel

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble

1 Introduction

Nous présentons un environnement et une approche pour l'enseignement du test.

L'enseignement du test est un exercice délicat. On peut choisir d'enseigner la philosophie (e.g., "rechercher des erreurs"), les outils (e.g., JUnit, ...), des objectifs (e.g., correction fonctionnelle, performance, non régression, ...), des phases de tests (e.g. unitaire, intégration, ...), et/ou des méthodes de sélection de données.

L'évaluation du travail des étudiants se base généralement sur l'évaluation de la qualité des tests. Pour cela, on s'appuie classiquement sur une approche d'analyse mutationnelle ou d'injection de fautes. Dans les deux cas, des programmes alternatifs au programme sous test sont construits en introduisant des fautes. Les tests sont ensuite exécutés sur ces programmes fautifs. On estime que les tests sont de qualité lorsque toutes les fautes sont découvertes. Nous avons étendu l'environnement Caseine et l'approche pour l'évaluation des tests des étudiants.

2 L'environnement Caseine

Caseine[1] est une plate-forme d'apprentissage développée par l'Université Grenoble-Alpes. Elle repose sur un *Virtual Programming Lab* de Moodle (VPL). Les enseignants peuvent définir des activités pour les étudiants, puis commenter et évaluer (automatiquement ou non) les productions. En automatisant l'évaluation des productions, on fournit un retour immédiat aux étudiants, ce qui est reconnu pour influer positivement sur la motivation et l'apprentissage [5, 6].

Pour les enseignants, Caseine offre un autre avantage : il est ouvert à toutes les universités du monde (ou presque) via la fédération d'Identités Education Recherche Edugain qui contient par exemple Renater. Par ailleurs, l'environnement est conçu pour permettre le partage des ressources. Ainsi, une large communauté (qui dépasse de loin Grenoble) contribue à l'élargissement des contenus et de la plate-forme en elle-même.

[1] https://moodle.caseine.org/

Caseine est utilisé en particulier pour l'enseignement de la programmation. Les étudiants sont invités à compléter des programmes, puis à les évaluer en exécutant des tests définis par l'équipe pédagogique. De l'évaluation des *programmes* à l'évaluation de la qualité des *tests*, il n'y a qu'un pas... mais pas si facile à franchir. Une suite de test est de qualité lorsqu'elle met en évidence des défaillances (lorsque les tests échouent). C'est donc l'inverse de ce qui est attendu lorsqu'on évalue la qualité d'un programme.

Notre *première contribution* a consisté à implémenter ce renversement de paradigme dans Caseine (contribution technique). Cela se présente sous la forme d'un modèle de "Lab" pour Java, que l'enseignant doit adapter en définissant (a) le programme à tester et (b) les programmes fautifs qui permettront d'évaluer les tests des étudiants. Ce travail peut se faire directement en ligne ou sous Eclipse, indépendamment de la façon dont sont créés les programmes fautifs.

3 Évaluer la qualité du travail de test

Évaluer la qualité des tests. On distingue deux grandes stratégies pour produire des programmes fautifs permettant d'évaluer les tests : l'analyse mutationnelle et l'injection de fautes.

L'analyse mutationnelle s'appuie sur un ensemble d'opérateurs de mutation élémentaires, qui s'appliquent *systématiquement* sur le programme pour produire un ensemble de mutants [2]. Par exemple, un $+$ est remplacé par un $-$, un $<$ est remplacé par un $>$ ou un $\geq$ (etc.). L'analyse mutationnelle présente quelques faiblesses, mais en moyenne, la communauté estime que l'approche permet d'évaluer raisonnablement la qualité des tests. C'est pourquoi, on l'utilise souvent dans des cours sur le test pour évaluer la qualité des productions des étudiants [1].

L'injection de fautes est une stratégie proche de la précédente, aussi basée sur la production de programmes erronés. Dans cette approche, les fautes choisies sont en général celles déjà identifiées dans le développement ou les promotions d'étudiants précédentes. La différence majeure entre les deux approches réside donc dans l'aspect systématisé ou non de la production de programmes erronés.

Les deux approches permettent d'évaluer la qualité des tests, mais elles ne permettent pas forcément d'évaluer la qualité de l'*application* de la méthode de tests enseignée. Quand il s'agit d'évaluer si une suite de tests couvre les instructions, les branches ou les conditions, il suffit d'utiliser des outils de couverture de code classique. Mais si l'on veut s'assurer que d'autres méthodes sont bien appliquées, il n'existe pas forcément d'outils.

Évaluer la bonne application de méthodes de tests. Notre *seconde contribution* consiste en l'adaptation du principe de l'injection de fautes. L'idée est simple. La plupart des méthodes de tests s'appuient sur un "artefact" à couvrir (élément du code ou de la spécification). Pour chaque élément à couvrir, on crée un programme erroné dont la défaillance apparaît lorsque l'exécution couvre l'artefact fixé. Par

```java
public double foo(double a, double b){
    return (a+b);
}
```
(a)

```java
public double foo(double a, double b){
    boolean inPartition;
    inPartition = (a<0) && (b<0);
    if (inPartition) return (a+b+1); else return (a+b);
}
```
(b)

Figure 1: Faute pour détecter la partie $(a < 0)$ && $(b < 0)$

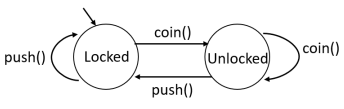
exemple, pour la méthode “catégories et partitions” [3], le testeur doit définir des partitions du domaine d'entrée et choisir un test dans chaque partie. Pour vérifier que la suite de tests couvre les N parties, on crée N programmes erronés. Pour chacun, la défaillance apparaît quand le programme est exécuté avec une entrée correspondant à la partie correspondante.

Prenons l'exemple de la méthode `foo(a,b)` ayant deux paramètres réels qui retourne un réel (Fig. 1(a)). Considérons les catégories $a < 0$; $a = 0$; $a > 0$; $b < 0$; $b = 0$; $b > 0$, sans contraintes, il y a $3 * 3$ parties. Pour vérifier si la suite de tests couvre les 9 parties, on crée 9 versions fautives du programme selon le modèle donné Fig. 1(b) : une variable booléenne capture si les données correspondent à la partie considérée; le résultat est modifié si c'est le cas.

Ce principe d'injection de fautes est assez générique pour évaluer plusieurs types de méthodes de tests, comme par exemple l'utilisation du critère def-use, la couverture des états, des transitions, des paires de transitions d'un automate. A ce jour, la production de programmes fautifs est manuelle. Mais les travaux du CEA LIST sur la production d'étiquettes pour capturer différentes notions de couverture de test montrent qu'une partie du processus pourrait s'automatiser [4].

On remarquera d'ailleurs que l'on peut introduire des fautes à différents niveaux d'abstraction. Par exemple, considérons la spécification Fig. 2. Pour évaluer la couverture de transitions de la spécification (selon une relation d'équivalence de traces ici), nous avons produit deux séries de programmes erronés. Dans la première série, les fautes changent l'état d'arrivée de la transition (l'instruction (a) est modifiée). Dans la seconde série, les fautes changent le résultat retourné par la méthode (l'instruction (b) est modifiée).

Quelques exercices proposés avec trois promotions d'étudiants de l'UGA sont disponibles pour la communauté. C'est notre *troisième contribution*. D'autres exercices sont en cours de développement et seront utilisés en classe avant d'être mis à disposition.




```java
public boolean coin() {
    if (state=="Locked") {
        state = "UnLocked";  //(a)
        return (true);       //(b)
    } else
        return (false);
}
```

Figure 2: Spécification d'un tourniquet [7] et méthode *coin* en Java

4 Conclusion et perspectives

Cet article présente 3 contributions : (a) nous avons étendu Caseine pour l'évaluation automatisée et transparente de la qualité des tests des étudiants; (b) nous avons proposé une stratégie générique, basée sur de l'injection de fautes, pour évaluer la qualité de l'application de méthodes de test; et (c) nous avons mis à disposition des exemples pour la communauté, qui ont été utilisés avec 3 promotions d'étudiants. Les premiers retours montrent qu'ils ont apprécié l'environnement. Leurs résultats ne sont pas significativement meilleurs qu'avant, mais la comparaison est biaisée du fait des conditions d'enseignement en distanciel (contre présentiel avant).

En perspective, nous continuons à développer des exemples pour enrichir ce qui est proposé, et automatiser la production de programmes erronés pour les méthodes de test les plus simples. Nous avons aussi préparé un protocole expérimental pour mesurer l'intérêt de l'approche. Nous l'appliquerons en 2022-23.

Remerciements. Sumaiya Sultana, Al-Bashir Muhammad et Mpoki Mwaisela ont contribué à élargir le contenus des exercices dans le cadre du projet région Auvergne Rhône-Alpes RAVEN (PAI 21 007381 01) et du projet ANR PHILAE (ANR-18-CE25-0013). Le projet Caseine a été partiellement financé par l'Idex de l'Université Grenoble-Alpes (ANR-15-IDEX-0002), le LabEx PERSYVAL-Lab (ANR-11-LABX-0025-01), et Flexi-TLV, trois Programmes Investissement d'Avenir.

References

[1] F. Dadeau, J.-Ph. Gros, and F. Peureux. A case-based approach for introducing testing tools and principles. In *IEEE Int. Conf. on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 429–436, 2020.

[2] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.

[3] S. K. Khalsa and Y. Labiche. Extending category partition's base choice criterion to better support constraints. *Journal of Software: Evolution and Process*, 30(3), 2018.

[4] N. Kosmatov. *Combinations of Analysis Techniques for Sound and Efficient Software Verification*. Hdr, U. de Paris-Sud 11, France, 2018.

[5] Y. Shen, Y. Wu, M. Xin, and Y. Zheng. A novel self-studying platform with its application to programming courses. In *The 9th Int. Conf. for Young Computer Scientists*, pages 2561–2566, 2008.

[6] D. Thiébaut. Automatic Evaluation of Computer Programs Using Moodle's Virtual Programming Lab (VPL) Plug-In. *J. Comput. Sci. Coll.*, 30(6):145–151, June 2015.

[7] Wikipedia. Automate fini. https://fr.wikipedia.org/wiki/Automate_fini, 2022.

Présentation des résultats du Projet ANR AAPG 2018 – PHILAE From Model-Based Testing to Cognitive Test Automation

Bruno LEGEARD (FEMTO-ST), Roland GROZ (LIG), Abbas AHMAD (FEMTO-ST)

May 2022

1 Introduction

Le projet ANR Philae - From Model-Based Testing to Cognitive Test Automation, sélectionné dans le cadre de l'appel à projet générique ANR 2018, s'est déroulé d'octobre 2018 à septembre 2022. L'équipe projet est composé de 4 partenaires français : Institut Femto-st/DISC (Coordinateur), Laboratoire d'Informatique de Grenoble, Orange Labs et Smartesting, auxquels se sont joints Mark Utting (actuellement à University of Queensland – Australie) et Arnaud Gotlieb (Simula Research Laboratory – Norvège).

Le problème auquel le projet s'intéresse est celui du coût grandissant des tests de non-régression dans les systèmes logiciels, qui constitue un goulot d'étranglement pour l'intégration continue. L'intuition de base est que la grande quantité de tests et la présence de masses importantes de traces d'exécution enregistrées dans des journaux (logs) sur les plates-formes doit permettre d'apprendre des modèles des systèmes pour réduire et améliorer les efforts de test. La thématique générale du projet est l'analyse par apprentissage automatique des traces d'exécution obtenues à partir des logs pour (1) évaluer et optimiser la couverture réalisée par les tests de l'usage réel de l'application par ses utilisateurs et (2) détecter dans les traces d'exécution de tests des anomalies sur ces tests. Contrairement à une approche de test à base de modèles (connu sous l'appellation Model-Based Testing), dans laquelle les modèles sont préexistants ou spécifiquement créés pour générer les cas de test, l'analyse dans Philae se base sur l'apprentissage machine de modèles (statistiques ou déterministes) à partir des logs.

Les résultats du projet Philae sont concrétisés et disponibles au travers d'une boîte à outils open-source, qui intègre des composants logiciels pour acquérir et formater les traces d'exécution, pour réaliser des analyses sur ces traces, et pour permettre de les utiliser à des fins de génération de tests ou d'analyse de tests. Les éléments open-source mis à disposition de la communauté intègrent aussi deux cas exemples avec les données de traces pour la reproduction des résultats, mais aussi disponibles comme jeux de données ouvertes.

Dans la suite de cet article, en section 2, nous commençons par présenter l'objet de notre recherche : la trace d'exécution. En sections 3 et 4, nous présentons les travaux et résultats obtenus respectivement sur les sujets (1) et (2). En section 5, nous présentons le répertoire de code et de données en open-source Philae mis à disposition de la communauté (sous Github).

2 Notion de trace d'exécution

La donnée brute sur laquelle se fonde le projet Philae est le recueil de logs, c'est-à-dire une suite d'événements enregistrés lors de l'exécution du logiciel. Ces logs proviennent des exécutions en production mais aussi des exécutions de tests. Les logs constituent une matière brute dont le premier traitement réalisé consiste à transformer les logs en traces.

Dans Philae, nous appelons *traces d'exécution* une séquence d'événements enregistrés présentant une session cohérente d'exécution. Dans le cas d'une exécution en production, cette session cohérente correspond généralement à une session utilisateur, et dans le cas d'une exécution de tests, cette session correspond à l'exécution d'un test. Le passage des logs aux traces est généralement réalisé par regroupement des événements à partir d'une clé de session portée par les événements enregistrés, mais parfois cela peut être plus complexe (par exemple lors d'un chaînage particulier des événements constituant une trace). Les événements de logs présentent un horodatage : les traces sont donc des séries temporelles, avec chaque événement portant un ensemble d'information.

Un domaine particulièrement ciblé par le projet Philae est celui des applications web, pour lesquels on dispose fréquemment des logs situés aux niveaux des APIs / appels de services, qu'on traitera donc préférentiellement plutôt qu'aux logs sur l'interface humain/machine de l'application.

Pour illustrer le passage des logs aux tests, nous présentons, en Figure 1, un des deux cas exemples en données ouvertes Philae : les logs issus d'une instance de boutique en ligne avec l'API Spree[1].

Nous observons sur la figure 1 une trace utilisateur. Cette trace correspond à des appels API sur la plateforme SPREE lors d'un parcours du client. La trace montre les événements qui font parties d'un scénario lorsque le client arrive sur la boutique en ligne, clique sur un produit "denim-jacket", l'ajoute à son panier et complète le processus de paiement.

Il faut noter qu'une action sur l'interface graphique se traduit généralement par plusieurs appels API. Par exemple, le fait d'ajouter un item au panier sur l'interface utilisateur se traduit par des appels API de récupération/création du panier à l'aide du token d'authentification de l'utilisateur et de l'ajout de cet item au panier récupéré/créé. De plus, les logs représentant la trace contiennent tous les paramètres nécessaires afin d'être rejoué ou utilisé dans un algorithme d'apprentissage machine pour générer de nouvelles traces exécutables.

[1] Cf. `https://spreecommerce.org/`

Figure 1: Extrait des logs Spree commerce API

3 Evaluer et optimiser la couverture de l'usage par les tests

Le projet Philae a étudié, développé et expérimenté cinq types de traitements des logs, constituant des composants d'une boîte à outils pour générer des tests fonctionnels automatisés à partir de l'analyse de l'usage :

- Conversion des logs en traces et visualisation des traces sous la forme de workflow et capacité d'exploration des traces[2];
- Codage des traces sous forme de vecteurs pour l'utilisation des données par les algorithmes ML[1] (en utilisant des techniques tel que word2vec issu du TALN – Traitement Automatique du Langage Naturel) ;
- Clustering des traces pour regrouper les traces par patterns d'usage et évaluation des clusters pour mesurer la représentativité des patterns vis-à-vis des comportements du logiciel[3] (par couverture de code en particulier) ;
- Mesure et visualisation de la couverture des traces d'usage par les traces de tests (cf. figure 2), et identification des clusters à couvrir ou mieux couvrir pour augmenter la couverture de l'usage par les tests ;
- Génération de tests représentatifs sur les clusters avec des techniques d'apprentissage machine génératives et génération de scripts exécutables dans un framework open-source d'automatisation des tests sur API[4].

Les techniques d'apprentissage machine utilisées sont, d'une part des techniques dites classiques (telles que les algorithmes de clustering et les arbres de décision), et d'autre part des modèles de Deep Learning, en particulier issus du TALN, tels que les Transformers.

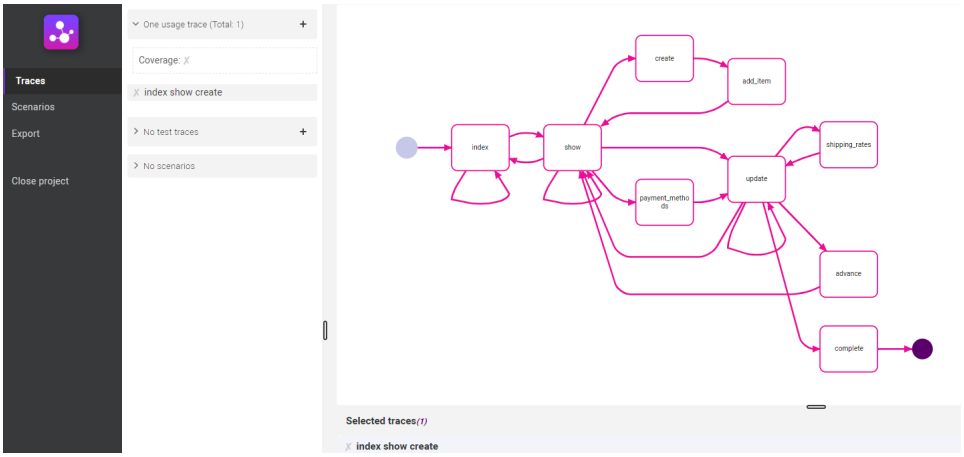


Figure 2: Visualisation des traces d'usage sous la forme de workflow

La figure 2 illustre la visualisation graphique sous la forme d'un workflow représentant un ensemble de traces capturées à partir des logs. C'est une capture d'écran de l'outil Philae de visualisation des traces d'usage.

4 Détecter dans les traces d'exécution de tests des anomalies sur ces tests

Le projet Philae s'est intéressé à l'analyse des logs obtenus lors de phases de tests intensives, comme les tests de non-régression ou les tests d'endurance, lesquels visent à vérifier le bon comportement du système pendant des temps longs, pour y détecter d'éventuelles fautes liées à des fuites mémoires ou d'autres accumulations de consommations de ressources. L'exploitation de ces volumes très importants de logs, en l'absence de modèles formels du comportement pouvant servir d'oracles est difficile, car il faut y repérer les comportements déviants et essayer d'en retrouver les causes possibles. Nous nous plaçons dans le cas de systèmes pour lesquels on dispose de 2 types de journaux :

- Un log des actions du système, tel qu'enregistré sur une API ou via une IHM ou un robot de test. Les événements enregistrés dans ce log le sont à une fréquence élevée qui peut être de l'ordre de la milliseconde ou de la seconde selon le niveau d'observation.

- Un échantillonnage régulier de métriques sur l'état global du système, à intervalle beaucoup plus espacé (de l'ordre de la dizaine de secondes à plusieurs minutes), analysant la consommation de ressources du système d'exploitation, telle que la consommation CPU, la consommation de mémoire, les défauts de page, les accès au réseau, etc.

L'absence de modèle formel du comportement est compensée par la présence de cette deuxième journalisation du système qui permet de retrouver des anomalies de fonctionnement à partir de l'impact sur les ressources utilisées par le système.

L'approche proposée par le projet pour la détection et la prédiction d'anomalies est illustrée sur la figure 3.

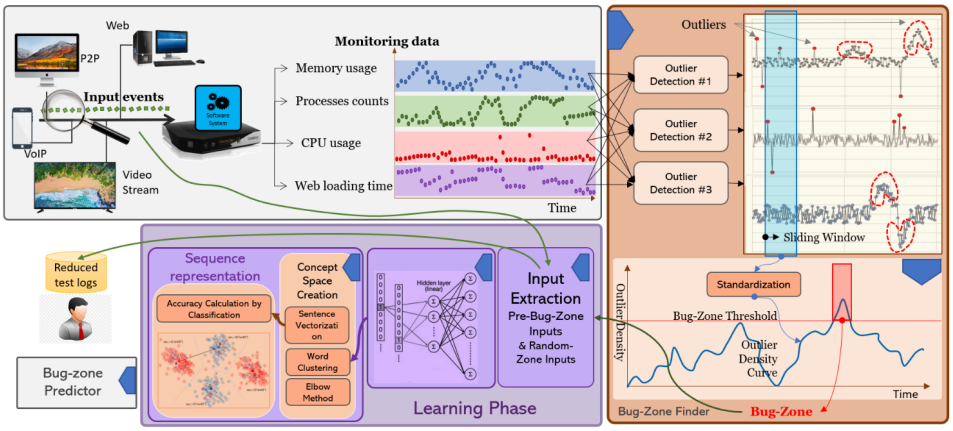


Figure 3: Détection et prédiction d'anomalie à partir de l'échantillonnage de l'état du système

Dans une première étape, des zones à risque (bug zone) sont automatiquement identifiées, par la présence corrélée de plusieurs métriques s'écartant de valeurs médianes pendant un certain intervalle (avec des seuils de détection qui peuvent être ajustés par les utilisateurs). Dans un deuxième temps, on va analyser les événements du journal des actions qui se sont produits en amont de la zone à risque. On représente les séquences d'événements comme des phrases d'un langage, et on va apprendre automatiquement à distinguer ces types de séquences de ceux de comportements normaux appris en dehors de zones à risque. On utilise pour cela des techniques également issues du TALN. L'identification des zones à risque réduit considérablement la charge des équipes de développement et validation pour le dépouillement des tests et des retours de production, et l'apprentissage automatique des séquences induisant des anomalies de comportement permet de prédire et potentiellement de prévenir les défaillances.

Sur l'étude de cas fournie par Orange, une première évaluation a permis de montrer qu'il était possible d'identifier et de prédire 70% des cas de réinitialisation du système, ce qui est proche du taux des réinitialisations sporadiques (liées à des pannes du système), sachant que les 30% restant semblent correspondre principalement sinon essentiellement à des réinitialisations forcées pour des raisons de gestion des équipements de tests.

5 Répertoire de code et de données en open-source Philae

Le projet PHILAE a vocation à mettre à disposition une boite à outil ”PHILAE Tool Box” en open source comme livrable de fin de projet. Il existe à ce jour 24 outils qui desservent 3 niveaux d'architecture au sein du projet PHILAE : 1. Outils de haut niveau dit ”Business Services”, 2. Outils de niveau médiant dit ”Builders” et finalement 3. Outils de bas niveau dit ”Helpers”.

Les outils permettent de mettre en place plusieurs techniques d'extraction des données (One-Hot encoding, Auto-encoding, Bag-of-words, ...), des techniques de clustering (Hidden Markov Model, MeanShift Clustering, ...), des techniques de réduction et d'évaluation de traces ainsi que la génération de scripts de test.

Les outils sont en ce moment en phase de migration d'un répertoire GitLab privé vers un répertoire publique GitHub accessible à cette adresse : `https://github.com/PHILAE-PROJECT`

Tous les outils seront disponibles avec des exemples fonctionnels en version finale du projet Philae en septembre 2022.

6 Conclusion

Le projet Philae a exploré différentes techniques d'analyse de logs pour extraire des traces de tests en vue de leur analyse et de la génération de scripts de tests avec des modèles d'apprentissage automatique. Les logs de test sont aussi analysés pour détecter des anomalies sur ces tests, permettant de les corriger ou de les optimiser.

Une boîte à outils open source permet de partager avec la communauté les différents composants développés, ainsi que deux cas exemple, avec les jeux de données associés, permettant la reproductibilité des résultats obtenus.

References

[1] Bahareh Afshinpour, Roland Groz, Massih-Reza Amini, Yves Ledru, and Catherine Oriat. Reducing regression test suites using the word2vec natural language processing tool. In Bimlesh Wadhwa, Shailey Chawla, Benjamin Gan, Eng Lieh Ouh, Pornsiri Muenchaisri, Saurabh Tiwari, and Santosh Singh Rathore, editors, *Joint Proceedings of SEED & NLPaSE co-located with 27th Asia Pacific Software Engineering Conference 2020, Singapore [Virtual], December 1, 2020*, volume 2799 of *CEUR Workshop Proceedings*, pages 43–53. CEUR-WS.org, 2020.

[2] Elodie BERNARD, Julien BOTELLA, Fabrice AMBERT, Bruno LEGEARD, and Mark UTTING. Tool support for refactoring manual tests. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 332–342, 2020.

[3] Vahana Dorcie, Fabrice Bouquet, and Frédéric Dadeau. Clustering of usage traces for regression test cases selection. In *Artificial Intelligence in Software Testing (AIST'22)*, 2022.

[4] Mark Utting, Bruno Legeard, Frédéric Dadeau, Frédéric Tamagnan, and Fabrice Bouquet. Identifying and generating missing tests using machine learning on execution traces. In *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 83–90, 2020.

Session doctorant : Analyse automatisée de binaires à la recherche de vulnérabilités matérielles *

Theo De Castro Pinto[1,2]
[1] Serma SAFETY & SECURITY
[2] Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI, UMR 5800, Talence, France

Résumé

Les cartes à puce et autres appareils embarqués mettent les développeurs face à un problème de taille : les attaques physiques. Parmi celles-ci, certaines visent à modifier le comportement de la puce (temporairement ou non) grâce à une perturbation physique (laser, glitch ou autre). Ces dernières, qui peuvent être multiples au cours d'une seule exécution, sont difficiles à détecter et même si des contre-mesures existent, elles peuvent être insuffisantes ou disparaître à la compilation. Pour ces raisons une analyse des binaires générés est nécessaire, afin de détecter la présence (ou non) de vulnérabilités face à ces attaques. Cette analyse est généralement effectuée à la main. Ce papier présente un outil d'aide à la recherche de vulnérabilités matérielles pour du code binaire. Il s'appuie sur l'outil Binsec. Il permet de détecter des vulnérabilités dans un binaire en fonction d'un schéma d'attaque et d'une cible à atteindre. Il s'agit de résultats préliminaires obtenus lors d'un stage de Master. Ce papier présente aussi quelques problématiques de recherche qui seront abordées au cours de la thèse.

1 Introduction

En informatique, une vulnérabilité est une faille de sécurité exploitable par un attaquant et pouvant mener à l'altération du comportement du logiciel ou à la divulgation d'informations secrètes. Il existe des vulnérabilités dites logicielles (buffer overflow, ROP, JOP [9], ...) et des vulnérabilités dites matérielles, ce sont ces dernières qui sont considérées dans ce papier. Ces vulnérabilités peuvent être exploitées via des attaques par injection de faute (en utilisant des LASER, des glitch de tension, ...). Ce travail se focalise sur les systèmes embarqués.

Une attaque matérielle peut avoir de nombreuses conséquences. Il est possible de classer ces effets en quatre catégories atomiques : les *bit set* (un bit du programme est mis à 1), les *bit reset* (un bit du programme est mis à 0), les *bit flip* (un bit du programme est inversé) et le *rejeu d'une instruction précédente*. Ces différents effets peuvent être combinés ce qui donne lieu à des attaques plus sophistiquées, par exemple, un branchement conditionnel inversé, une instruction transformée ou sautée. La conséquence de tout cela est que des comportements non-attendus peuvent être ajoutés via des attaques par injection de faute.

Il est donc nécessaire de limiter les vulnérabilités matérielles dans les programmes, notamment embarqués, cependant c'est une tâche ardue. Les développeurs doivent imaginer un maximum d'attaques possibles afin d'intégrer des contre-mesures [1] qui pourront limiter le nombre de vulnérabilités exploitables. Un exemple typique de contre-mesure est la redondance des opérations dans le code. Il existe des organismes (tels que des CESTIs) qui évaluent la sécurité des logiciels embarqués et tentent d'y trouver des vulnérabilités. Actuellement, les évaluations consistent en des revues de code (source en conjonction avec le code assembleur fourni par le client) à la main (faites par des experts) et ces dernières ont le désavantage d'être très longues et difficiles.

Il y a donc un besoin d'analyse automatisée dans ce milieu. L'objectif de ce projet est de développer un outil permettant d'assister les experts dans leur recherche de vulnérabilités matérielles et d'en automatiser une grande partie. Les experts ont accès au code source et assembleur, mais le code assembleur est difficile à analyser pour un humain (en raison de sa taille et de son intrication). Pourtant, la compilation peut

* Ce projet est soutenu par l'ANRT via le financement CIFRE numéro 2021/1673.

1. Des sécurités permettant de détecter des attaques en faute.

```
1  cmp  r0, r1
2  bne  4
3  call  critical_function
4  halt
```

FIGURE 1 – Code assembleur non sécurisé

```
1  cmp  r0, r1
2  bne  6
3  cmp  r0, r1
4  bne  6
5  call  critical_function
6  halt
```

FIGURE 2 – Code assembleur un peu sécurisé

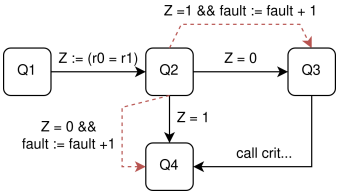


FIGURE 3 – Automate avec mutations correspondant au code en listing 1

souvent retirer des contre-mesures, donc l'analyse du code assembleur est inévitable et l'automatisation de ce procédé est une solution utile.

Différentes approches ont été proposées, par exemple certaines se basent sur de la vérification formelle de code par rapport à des propriétés (Frama-C [5] [10], BMC [3], ...), d'autres analysent [12] et ajoutent [8] automatiquement des contre-mesures. Une autre approche, qui va nous intéresser dans ce papier, consiste à détecter les vulnérabilités (via par exemple de l'exécution symbolique). Dans cette dernière catégorie, on peut citer FIVD [7] qui utilise de l'exécution concrète et des tests physiques ou bien Lazart [11] qui permet de détecter l'existence de vulnérabilités permettant d'atteindre un bout de code donné.

Notre méthodologie est une extension de celle proposée par Lazart. Ce dernier est un outil qui analyse du code LLVM (compilé à partir du code source) afin d'y détecter des vulnérabilités par injection de faute. Nous avons donc développé un outil qui applique et étend cette méthodologie aux binaires, notamment via l'utilisation de Binsec [6]. Nous avons développé un prototype dont les premiers résultats tendent à montrer qu'il permet de détecter plus de vulnérabilités que Lazart (la comparaison a été faite via des tests sur Wookey [1]).

2 Contributions et résultats préliminaires

Afin de clarifier quels sont les enjeux et comment fonctionne l'outil, des exemples de codes sont donnés en figures 1 et 2. Ces codes sont fonctionnellement équivalent. Dans l'idée `r0` est une valeur secrète (un code pin par exemple) et `r1` une entrée utilisateur. Un attaquant ne connaît donc pas le pin mais en exploitant des vulnérabilités matérielles, il est possible d'atteindre la fonction critique. En effet, une attaque en faute peut par exemple transformer l'instruction `bne 4` en `nop`, et dans ce cas la comparaison précédente est inutile et toute valeur donnée en entrée mènera à l'appel de la fonction. Le code 2 est donc plus sécurisé face à une attaque en faute que le code 1.

Dans le code en figure 2, cette attaque n'est pas suffisante grâce à la contre-mesure ajoutée (redondance du test), mais bien sûr, il est possible qu'un attaquant fasse deux attaques en faute dans le même binaire, dans ce cas il lui est possible de transformer les deux instruction `bne` en `nop`, ce qui aura les mêmes conséquences que l'attaque présentée dans le premier code. Enfin, il est intéressant de noter qu'en réalité, un compilateur peut supprimer ce type de redondance, notamment si les optimisations sont activées. Il est donc intéressant de faire une analyse au niveau binaire, car c'est plus représentatif du programme réellement exécuté. Cela permet de détecter, en particulier, des vulnérabilités qui n'existent pas au niveau du code source, et qui sont dues à des optimisations trop agressives.

Dans ces deux exemples, il est facile de trouver des vulnérabilités mais les experts qui analysent des binaires font face à des codes sources de plusieurs milliers de lignes et des codes assembleurs encore plus longs. Ce n'est donc pas envisageable de trouver toutes les vulnérabilités, ni même toutes les vulnérabilités exploitables dans des conditions réelles (celles qui sont vraiment problématiques).

L'approche que nous proposons est la suivante :

1. Transformer le binaire en un langage intermédiaire (DBA [2]) qui correspond à un automate (voir figure 3).

2. Ajouter des mutations dans cet automate, ce qui correspond à ajouter des transitions, voir par exemple la figure 3, ce sont les transitions en pointillés. Dans ce cas, le saut conditionnel `bne` a la possibilité d'être inversé lors de l'exploration de l'automate. La variable `fault` permet de compter le nombre de fautes injectées.
3. Explorer symboliquement l'automate (grâce à Binsec [6]) afin d'atteindre une cible (typiquement une instruction assembleur) pré-définie avec comme pré-condition que les valeurs secrètes ne sont pas connues.
4. Analyser les chemins atteignant la cible et en réduire le nombre car beaucoup de chemins générés sont en réalité équivalents. Un humain doit ensuite choisir quels chemins semblent correspondre à une vulnérabilité exploitable, il est donc nécessaire d'en avoir le moins possible.

L'étape 1 se fait grâce à l'outil Binsec (un outil d'analyse de binaires), le DBA étant le langage intermédiaire utilisé par Binsec. Cet outil a été choisi car Binsec est activement utilisé dans la recherche de vulnérabilités dans du code binaire. Il s'agit de plus d'un outil open source ce qui nous a permis de le modifier afin d'obtenir des informations supplémentaires lors de l'exécution symbolique.

L'étape 2 est plus complexe, car il est nécessaire de limiter le nombre de transitions rajoutées dans l'automate. Bien sûr il est possible de créer un automate très complexe tel que toutes les attaques imaginables sont possibles à tout moment mais l'exécution symbolique ne pourrait jamais explorer efficacement un tel automate.

Afin de limiter le nombre de transitions, il est nécessaire d'une part de définir un modèle de faute, c'est-à-dire quelles attaques sont autorisées dans la recherche, et d'autre part de déterminer à quels nœuds de l'automate il est utile ou non d'ajouter des transitions (certains nœuds n'ont absolument aucun effet sur la cible à atteindre). Le modèle de faute choisi est l'inversion des tests : tout branchement peut être transformé en son complément et le choix des nœuds découle de la méthode proposée par Lazart [11].

Un graphe de flot de contrôle est extrait du binaire, où les nœuds sont des blocs d'instructions se terminant par un branchement (et n'en contenant pas d'autre). L'approche naïve consiste à ajouter des transitions fautées à la fin de chacun de ces blocs mais cela est très coûteux lors de l'exécution symbolique et inutile dans certains cas. Nous utilisons une heuristique qui ajoute des mutations seulement lorsque cela est utile. En effet, s'il n'existe pas de chemin entre un nœud donné et la cible, il n'est pas muté, de la même manière si tous les chemins issus d'un nœud donné atteignent la cible à un moment, il n'est pas muté (car ces mutations n'ont aucune incidence sur l'atteignabilité de la cible.). Tous les autres nœuds le sont, donc des transitions sont rajoutées dans l'automate aux points correspondant (bien sûr lors de l'exécution ces transitions ne sont pas nécessairement empruntées à chaque fois).

Avec cette stratégie, il est possible d'obtenir des résultats sensiblement meilleurs que ceux avancés par Lazart : sur un exemple classique de l'état de l'art, Wookey [1], Lazart détecte deux vulnérabilités, et l'outil que nous avons développé en détecte trois, dont une non documentée par Lazart (cependant cette vulnérabilité n'est peut être pas exploitable, cela n'a pas pu être vérifié). Néanmoins, les performances temporelles de l'outil sont plus mitigées, ce qui est notamment dû au problème de l'explosion combinatoire liée à l'exécution symbolique. Enfin, lors du développement de notre outil, Lazart utilisait uniquement un modèle de faute se basant sur l'inversion de branchements. L'outil développé quant à lui peut travailler avec n'importe quel modèle de faute (mais pas nécessairement de manière efficace).

3 Discussions et perspectives

Malgré les résultats encourageants, l'outil développé reste peu utilisable dans des conditions réelles. Le problème majeur reste le temps d'exécution et le faible nombre d'études réalisées sur des cas concrets. Au cours de la thèse, l'objectif va donc être le passage à l'échelle. Il y a différents points centraux qui seront explorés afin de rendre l'exécution symbolique plus rapide. La première piste est d'utiliser de meilleures heuristiques d'exploration. En effet, il existe de très nombreux chemins d'exécution à explorer dans un binaire de taille réelle (il n'est jamais possible de l'explorer entièrement), il est utile de trouver une mesure qui permet d'explorer les chemins les plus "intéressants" en premier.

Le deuxième objectif est d'utiliser l'interprétation abstraite [4]. Son utilisation peut être double, d'une part elle peut être utilisée afin d'obtenir des résumés de fonction (celles qui ne contiennent aucune mutation) afin de ne plus les exécuter symboliquement. D'autre part, elle peut aussi être utilisée durant l'exécution symbolique afin d'accélérer les calculs (au risque d'augmenter le nombre de faux positifs).

Dans les cas pratiques, l'architecture utilisée est souvent ARMv7-M, qui correspond à du THUMB-2. Cependant, la traduction vers du DBA est lourde à cause de certaines instructions (notamment l'instruction `IT`), il existe cependant d'autres représentations intermédiaires qui pourraient alléger la traduction (par exemple : VEX [13]). Une troisième idée repose donc sur l'utilisation d'une représentation intermédiaire un peu modifiée (mélange de DBA et de techniques issues de différentes représentations intermédiaires).

Enfin, il est crucial de formaliser différents aspects du domaine, notamment les binaires et les modèles de faute. Il s'agira donc de définir formellement la syntaxe et la sémantique des différents objets manipulés. De plus, la cible à atteindre et les chemins "intéressants" ne sont actuellement pas des concepts formels ce qui rend la recherche parfois hasardeuse.

Références

[1] ANSSI. Wookey. `https://wookey-project.github.io/publi.html`. Accessed : 24-05-2022.

[2] Sébastien Bardin, Philippe Herrmann, Jérôme Leroux, Olivier Ly, Renaud Tabary, and Aymeric Vincent. The bincoa framework for binary code analysis. In *International Conference on Computer Aided Verification*, pages 165–170. Springer, 2011.

[3] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[4] Patrick Cousot. Interprétation abstraite. *Technique et science informatique*, 19(1-2) :3, 2000.

[5] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *International conference on software engineering and formal methods*, pages 233–247. Springer, 2012.

[6] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. Binsec/se : A dynamic symbolic execution toolkit for binary-level analysis. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 653–656. IEEE, 2016.

[7] Nisrine Jafri. *Formal fault injection vulnerability detection in binaries : a software process and hardware validation*. PhD thesis, Université Rennes 1, 2019.

[8] Pantea Kiaei, Cees-Bart Breunesse, Mohsen Ahmadi, Patrick Schaumont, and Jasper Van Woudenberg. Rewrite to reinforce : Rewriting the binary to apply countermeasures against fault injection. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 319–324. IEEE, 2021.

[9] Yongje Lee, Jinyong Lee, Ingoo Heo, Dongil Hwang, and Yunheung Paek. Integration of rop/jop monitoring ips in an arm-based soc. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 331–336. IEEE, 2016.

[10] Thibault Martin, Nikolai Kosmatov, and Virgile Prevosto. Verifying redundant-check based countermeasures : A case study. 2022.

[11] Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. Lazart : A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, pages 213–222. IEEE, 2014.

[12] Pablo Rauzy and Sylvain Guilley. A formal proof of countermeasures against fault injection attacks on crt-rsa. *Journal of Cryptographic Engineering*, 4(3) :173–185, 2014.

[13] ANGR team. Intermediate representation - angr. `https://docs.angr.io/advanced-topics/ir`. Accessed : 24-05-2022.

AFADL & LVP

Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup

Pierre Castéran[1], Hugo Gimbert*[1], Claire Mathieu[2], and Gérald Point[1]

[1]LaBRI, CNRS, Université de Bordeaux, France
[2]IRIF, CNRS, France

Parcoursup est la plateforme nationale de préinscription en première année de l'enseignement supérieur en France. Chaque année, plus de 1 000 000 candidats toutes procédures confondues formulent des vœux sur Parcoursup, où plus de 19 500 formations de l'enseignement supérieur sont proposées.

La procédure principale de Parcoursup se décompose en plusieurs phases. Initialement, les candidats font des voeux pour les différentes formations proposées. Ensuite chaque formation classe une partie des candidats ayant effectué des voeux dans cette formation [1]. Enfin les candidats reçoivent des propositions.

La loi ORE[2] prévoit l'application de taux encadrant la proportion de candidats boursiers et de candidats hors-secteur dans les formations concernées [3].

La mise en oeuvre des taux minimum de candidats boursiers et des taux minimum de candidats résidents dans l'académie est traitée en partie de manière automatisée, à l'échelle nationale, par un algorithme mis en oeuvre par le Service à Compétence Nationale Parcoursup (SCN Parcoursup). Le contexte de cette automatisation et le fonctionnement de cet algorithme sont détaillés dans le "Document de présentation des algorithmes de Parcoursup" [9, Section 4] et les algorithmes et leur implémentation open-source sont consultables sur le dépôt de code dédié [10].

L'algorithme calcule un *ordre d'appel* des candidats, qui définit l'ordre dans lequel les propositions seront faites aux candidats. Si les taux minimum de boursiers et de résidents étaient de 0%, l'ordre d'appel serait tout simplement l'ordre du classement pédagogique des candidatures, tel que transmis par la formation. Dès lors qu'il convient de respecter ces taux, l'ordre d'appel est obtenu à partir du classement pédagogique en faisant remonter des boursiers et/ou des résidents dans le classement afin de garantir le respect des dits taux.

*Les auteurs de ces travaux ont pu bénéficier de plusieurs subventions du MESRI sur la période 2018-2021.

[1]Plus précisément chaque formation dans laquelle le nombre de candidature est supérieur au nombre de places disponibles.

[2]Loi du 8 mars 2018 relative à l'Orientation et à la Réussite des Étudiants

[3]cf sections V et VI de l'article L. 612-3 du code de l'éducation.

Dans ce document on présente une partie des travaux[4] de vérification logicielle effectués sur cet algorithme:

- spécification des algorithmes,
- preuve mathématique de la conformité des algorithmes à leur spécification,
- preuve formelle en `Why3` de la conformité d'une implémentation à la spécification.

Ce travail de preuve a débuté avec l'équipe Toccata du LRI [5], en collaboration avec le second auteur de cet article et avec un soutien financier du Ministère de l'Enseignement Supérieur, de la Recherche et de l'Innovation (MESRI). Les travaux effectués par le LRI ont permis de réaliser la preuve de certaines propriétés d'une implémentation `WhyML` de l'algorithme à un taux, ainsi qu'une preuve partielle d'une implémentation alternative en `Java` [1, 2].

Nous avons complété les travaux du LRI en deux temps. Tout d'abord nous nous sommes intéressés aux parties de la spécification qui n'avaient pas été prouvées. Lors de cette étape nous avons été confrontés à un lemme, présenté à la section 3, qui nous semblait difficile à prouver en `WhyML`; nous avons alors délégué cette preuve à `COQ` (par la suite, le lemme a été prouvé directement en `WhyML`).

Dans un second temps nous avons réalisé les preuves pour l'algorithme à deux taux (celui en production). Ne pouvant prouver directement le code `Java`, nous nous sommes attachés à traduire l'algorithme pratiquement ligne à ligne ainsi qu'à réduire au maximum les annotations `WhyML`. L'outil `COQ` n'a pas été utilisé lors de cette étape.

Développant les preuves dans un cadre d'intégration continue sur une forge logicielle, nous nous sommes astreints à obtenir une preuve entièrement automatique en limitant `Why3` à l'utilisation d'`Alt-Ergo`. Nous avons utilisé `Why3` de manière interactive mais sans faire appel à ses stratégies[6], en ajoutant progressivement des assertions et lemmes `WhyML`, jusqu'à obtenir une validation automatique par `Alt-Ergo` des obligations de preuves générées par par `Why3`.

D'un point de vue quantitatif le projet représente un peu plus de 8000 lignes de `WhyML`. L'algorithme annoté représente uniquement, environ, 400 lignes de code. La preuve du lemme de la section 3 a demandé environ 1300 lignes de `COQ` (sans le contexte produit par `Why3`).

Travaux connexes. Les méthodes formelles sont également appliquées à d'autres algorithmes de décision publiques, notamment la sécurisation du calcul des impôts et des allocations sociales grâce au langage CATALA [8] ou les systèmes de régulation des heures de conduite des chauffeurs routiers [6]. Plus globalement, une

[4] Le détail de ces travaux est consultable dans le rapport technique [7] et disponible en libre accès dans le dépôt du projet[12].

[5] https://toccata.gitlabpages.inria.fr/toccata/

[6] Les stratégies de `Why3` ressemblent aux tactiques de `COQ`.

Q1) Pour tout $k \in 1 \dots |C|$, $d_1, \dots, d_k$ contient au moins $\lceil q_B * k \rceil$ candidats boursiers ; ou sinon, aucun candidat parmi $d_{k+1} \dots d_{|C|}$ n'est boursier.

Q3) Un candidat résident boursier qui a le rang r dans le classement pédagogique n'est jamais doublé par personne et a un rang inférieur ou égal à r dans l'ordre d'appel.

P4) Supposons $q_R = 0$. Comparé au classement pédagogique, l'ordre d'appel minimise le nombre d'inversions (distance de Kendall-tau), parmi ceux qui garantissent la propriété Q1.

Figure 1: Une partie de la spécification du calcul de l'ordre d'appel.

réflexion internationale appelée *Rules as Code* interroge les conditions de mise en oeuvre et d'élaboration des lois dont l'application est partiellement ou totalement automatisée [11].

1 Spécification et calcul des ordres d'appel

Le document de présentation des algorithmes de Parcoursup [9, Section 4] précise un certain nombre de propriétés que l'algorithme est tenu de respecter. Pour des raisons de lisibilité, la spécification disponible dans [9, Section 4] est exprimée de manière semi-formelle.

L'ordre d'appel est représenté par une permutation $d_1, \dots, d_{|C|}$ de l'ensemble C des candidats apparaissant dans le classement pédagogique. Certaines des propriétés de la spécification sont présentées dans la Figure 1. La propriété Q1 garantit le respect du taux boursier. La propriété P4 spécifie dans quelles proportions un candidat peut perdre des places lors du calcul de l'ordre d'appel.

Dans une formation soumise à deux taux q_B (boursiers) et q_R (résidents), l'algorithme consiste à énumérer les candidats à partir du candidat mieux classé, en les intégrant un à un dans l'ordre d'appel. Ce faisant, on surveille le respect des taux. L'algorithme est présenté figure 2, dans le cas particulier où le taux minimum de résidents q_R est fixé à 0%. On note q_B le taux minimum de boursiers, avec $0 \leq q_B \leq 1$.

Illustrons cet algorithme avec un exemple. Dans cette formation, le taux minimum de résidents est fixé à $q_R = 0$, le taux minimum de boursiers est de $q_B = 20\%$, et la formation a classé ses candidats dans l'ordre

$$(C_1, C_2, B_3, C_4, C_5, C_6, C_7, C_8, C_9, B_{10}, C_{11})$$

où seuls deux candidats B_3 et B_{10} sont boursiers. Alors l'ordre d'appel sera $(B_3, C_1, C_2, C_4, C_5, B_{10}, C_6, C_7, C_8, C_9, C_{11})$. En effet, d'une part le taux résident n'est jamais contraignant. D'autre part le taux boursier est contraignant lors du choix des candidats d'indices 1 et 6, ce qui permet à B_3 et B_{10} de "remonter" dans

1. Notons n le nombre de candidats apparaissant dans le classement pédagogique de la formation.
2. Pour chaque entier k de 1 à n, dans cet ordre, le candidat C_k de rang k dans l'ordre d'appel est calculé de la manière suivante. On a déjà sélectionné les candidats $C_1, \ldots, C_{k-1}$ dans l'ordre d'appel, et parmi eux il y a b boursiers. On dit que le taux minimum boursiers est contraignant si $b/k < q_B$. On considère tous les autres candidats, pris dans l'ordre pédagogique. Pour choisir C_k, parmi ceux-là:
   - Si le taux n'est pas contraignant, on prend le premier candidat.
   - Si le taux est contraignant, on prend le premier candidat boursier s'il y en a, le premier candidat sinon.

Figure 2: Algorithme de calcul de l'ordre d'appel dans les formations soumises au seul taux boursier.

l'ordre d'appel. Remarquons que le choix du candidat d'indice 11 est également contraignant, mais à ce stade du calcul de l'ordre d'appel tous les boursiers ont déjà été sélectionnés.

2 Vérification de l'algorithme

Les travaux de vérification logicielle effectués sur cet algorithme (voir [7, 12]) comprennent la spécification des algorithmes, la preuve mathématique de la conformité des algorithmes à leur spécification, la preuve formelle de la conformité d'une implémentation à la spécification, la vérification à l'exécution de la spécification.

La preuve formelle s'applique à une implémentation de l'algorithme dans le langage `WhyML`, en s'appuyant sur la plateforme `Why3` pour la preuve de programme [3, 4]. `Why3` est une plateforme pour la preuve de programme, qui s'appuie sur des prouveurs externes automatiques ou interactifs [3]. Elle permet de prouver des programmes écrits en `WhyML`, un langage de spécification et de programmation. Une fois prouvé, le programme peut être automatiquement traduit en un programme Caml correct par construction.

L'implémentation en `WhyML` de l'algorithme n'est pas l'implémentation `Java` effectivement exécutée par le SCN Parcoursup. Cette mesure garantit donc l'absence de bugs dans l'implémentation `WhyML` mais pas dans l'implémentation `Java`. Le langage `WhyML` se prête lui même peu à une exploitation par le SCN Parcoursup. Actuellement c'est l'implémentation `Java` qui est exploitée, avec des mesures de vérification à l'exécution, dont une vérification en double-aveugle contre une autre implémentation en PL/SQL.

Des mesures de *vérification à l'exécution* intégrées dans l'implémentation `Java`

va permettent de certifier que chaque exécution du logiciel est conforme à sa spécification, tant qu'elle ne lève pas une exception. Cette mesure est complémentaire au travail de preuve formelle effectuée sur l'implémentation WhyML. A première vue, le niveau de confiance donné par les mesures de *vérification à l'exécution* est moindre que celui d'une preuve formelle, car cette mesure de vérification à l'exécution écarte la possibilité d'une erreur dans les exécutions passées du programme, mais ne garantit rien sur les exécutions futures avec de nouvelles données d'entrée. Toutefois, la vérification à l'exécution s'applique à l'implémentation Java effectivement exploitée par le SCN Parcoursup, et elle a pu être mise en place rapidement dès la première année, très en avance de phase par rapport au développement des preuves formelles.

3 Preuve formelle de l'implémentation WhyML

Dans cette section on donne un aperçu de l'implémentation WhyML et des preuves formelles associées, qui sont disponibles sur le dépôt [12].

L'implémentation de l'algorithme à deux taux en WhyML est une traduction assez proche de l'implémentation Java, consultable dans le module algo.mlw.

L'essentiel de la preuve de l'algorithme consiste à prouver que les invariants et propriétés de l'algorithme restent satisfaits à chaque itération. La preuve est décomposée en plusieurs fichiers indépendants, correspondants aux différents invariants. Cette modularité permet de conserver une définition de l'algorithme la plus lisible possible (cf. algo.mlw). De plus cette isolation des différentes parties de la preuve améliore également la performance des solveurs automatiques en isolant les différents contextes dans lesquels les différents invariants sont prouvés.

Traitement modulaire des invariants. Nous avons essayé autant que possible de séparer en plusieurs modules et fichiers l'implémentation WhyML proprement dite, la définition des différentes propriétés de la spécification, la définition des différentes invariants, et les lemmes permettant de prouver les invariants. Prenons pour exemple la fonction selectionnerMeilleurVoeu[7] dont le contrat spécifie que (voir figure 1):

- si avant l'appel à cette fonction les invariants étaient vérifiés par g, oak (c-à-d. l'ordre courant) et filesAttente (lignes 75-76);
- alors (lignes 77-78) le candidat choisi, result, est tel que les invariants restent satisfaits pour g, oak augmenté de result et filesAttente de laquelle result a été retiré. Pour les variables modifiables (c-à-d. annotées mutable en WhyML), la post-condition fait référence à la valeur des variables à la fin de la fonction.

[7] Le code de la fonction selectionnerMeilleurVoeu est donné dans le module algo_selection_meilleur_voeu.mlw

```
69    let selectionnerMeilleurVoeu
70        (contrainteTauxBoursier : bool)
71        (contrainteTauxResident : bool)
72        (g: groupe_classement)
73        (oak : Q.t voeu)
74        (filesAttente :  enummap) : voeu
75      requires {[@expl:InvAlgo2]
76            invariants_algo2 g oak filesAttente }
77      ensures {[@expl:InvAlgo2(oak.res)]
78           invariants_algo2 g (snoc oak result) filesAttente }
```

Figure 1: Contrat de la fonction de sélection du meilleur candidat.

La preuve de la post-condition du contrat par `Why3` nécessite l'utilisation du lemme `invariants_algo_snoc` déclaré dans le module algo_invariants.mlw. Ce lemme est un simple corollaire des lemmes qui prouvent que chaque propriété ou invariant est satisfait après l'appel à `selectionnerMeilleurVoeu`. Ces lemmes sont prouvés dans des modules séparés portant le nom de la propriété ou de l'invariant concerné; par exemple q1.mlw ou iq4_3.mlw.

Traduction de types `Java`. L'implémentation `Java` commence par le partitionnement du classement pédagogique en quatre files (`java.util.Queue`) de candidats stockées dans une table (`java.util.EnumMap`) indexée par les quatre types de candidats:

```
EnumMap<VoeuClasse.TypeCandidat, Queue<VoeuClasse>> filesAttente
```

Nous avons modélisé cette structure de donnée avec un type `WhyML` *ad hoc* contenant explicitement les quatre files. Cette structure est déclarée dans le module enummap.mlw de la preuve.

```
type enummap =
{
  bds : Q.t voeu;
  bhs : Q.t voeu;
  nbds : Q.t voeu;
  nbhs : Q.t voeu;
}
```

Par exemple la file stockée dans le champ `bds` contient les boursiers du secteur, l'équivalent de la file `Java`

```
filesAttente[BOURSIER_DU_SECTEUR]
```

Nous avons adapté des méthodes de la classe `java.util.EnumMap` à notre structure comme par exemple les méthodes `get` et `put`.

Preuve d'un lemme clé. La propriété (P4) de l'algorithme à un taux s'appuie sur un lemme clé concernant le nombre d'inversions dans un vecteur.

Le *nombre d'inversions* ou indice de Kendall-Tau est défini pour toute permutation $(e_k)_{k\in 1\ldots n}$ de $\{1,\ldots,n\}$ par

$$\begin{aligned}\text{Inversions}((e_k)_{k\in 1\ldots n}) =\mid \{&i \in 1\ldots n, j \in 1\ldots n,\\ &(i<j \wedge \text{rang}(e_i) > \text{rang}(e_j)) \vee (i>j \wedge \text{rang}(e_i) < \text{rang}(e_j))\} \mid\end{aligned}$$

La preuve de (P4) passe par le lemme suivant.

Lemma 1. *Soit $(e_k)_{k\in 1\ldots n}$ une permutation de $1\ldots n$ et $0 \le i_0 < j_0 \le n$ une inversion de ce vecteur. Soit e' le vecteur obtenu en échangeant e_{i_0} et e_{j_0}. Alors le nombre d'inversions de e' est inférieur à celui de e.*

La preuve de ce lemme a été réalisée en `COQ` et en `Why3`. En `Why3`, le lemme s'énonce comme suit

```
let lemma  inversions_dec (s : seq voeu) (p : pair)
  requires { 0 <= p.i && 0 <= p.j }
  requires { inversion s p }
  ensures { nb_inversions (swap s p) < nb_inversions s }
= ()
```

`COQ` est un système de gestion des preuves formelles [5]. Il fournit un langage formel permettant d'écrire des définitions mathématiques, des algorithmes exécutables et des théorèmes, et fournit également un environnement de preuves semi-interactive vérifiées par ordinateur. Les applications typiques sont la certification des propriétés des langages de programmation (e.g. le projet de certification du compilateur CompCert, la "Verified Software Toolchain for verification of C programs", ou le framework Iris pour la logique de séparation concurrente), la formalisation des mathématiques (e.g. la formalisation complète du théorème de Feit-Thompson theorem), et l'enseignement.

L'utilisation des deux approches basées sur `COQ` d'une part et `Why3` d'autre part a permis de les comparer. L'utilisation de `COQ` est adaptée à la preuve des propriétés abstraites, en s'exploitant sur les riches librairies du langage. Les modules `Why3` générant les obligations de preuve `COQ` doivent être réduits à l'essentiel, sous peine de produire de nombreux axiomes inutiles pour le théorème visé. Une preuve `COQ` fournit un très haut niveau de confiance, sans dépendance à des solveurs externes. La preuve `COQ` utilise un angle différent de la preuve en `Why3`. L'utilisation de `Why3` offre un bon degré d'automatisation du traitement des cas et sous-cas. Dans les deux approches, le traitement des sommes partielles sur des sous-ensembles de $\mathbb{Z}$ nécessite de forger quelques outils *ad-hoc*.

Conclusion

Un autre algorithme crucial pour le fonctionnement de Parcoursup est l'algorithme qui calcule quotidiennement quelles propositions doivent être envoyées aux candidats. Tout comme le calcul des ordres d'appel, cet algorithme est documenté dans le "Document de présentation des algorithmes de Parcoursup" [9, Sections 3,5,6] et

l'implémentation `Java` est publiée sur le dépôt public du code de Parcoursup [10]. Des mesures de vérification à l'exécution ont été implémentées en `Java` pour garantir la sécurité du code de l'algorithme des propositions mais cet algorithme n'a pas encore été prouvé formellement, notamment l'unicité du résultat pour les formations avec internat, une propriété importante qui implique l'indépendance du résultat à l'ordre d'énumération des données d'entrée.

La preuve formelle de l'implémentation `Java` a été partiellement effectuée [2] mais pas encore finalisée. Pour avancer dans cette direction, il y a au moins deux possibilités. Premièrement contourner les difficultés inhérentes à l'analyse du code `Java` en générant des implémentations de référence en `Java` à partir des implémentations `WhyML`, en commençant par le code effectuant la vérification à l'exécution. C'est un travail en cours, dans le cadre d'une collaboration entre le LRI et le LaBRI. Deuxièmement définir un fragment syntaxique de `Java` qui se prête à une interprétation en `WhyML` avec une extension de l'outil `jmltowhy3` développé au LRI [2].

References

[1] Léo Andrès. Vérification par preuve formelle de propriétés fonctionnelles d'algorithme de classification. Research report, Université Paris Sud (Paris 11) - Université Paris Saclay, https://hal.inria.fr/hal-02421484/file/internship-report-parcoursup.pdf, August 2019.

[2] Benedikt Becker, Jean-Christophe Filliâtre, and Claude Marché. Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup. Technical report, Université Paris-Saclay, https://hal.inria.fr/hal-02447409/file/main.pdf, January 2020.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3. http://why3.lri.fr.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer*, 17(6):709–727, 2015.

[5] COQ. Site web de COQ. https://coq.inria.fr.

[6] formalvindications. Site web de formalvindications. https://formalvindications.com.

[7] Hugo Gimbert, Pierre Castéran, Claire Mathieu, and Gérald Point. Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup. Research report, CNRS ; Université de Bordeaux ; LaBRI - Laboratoire Bordelais de Recherche en Informatique ; IRIF, October 2021.

[8] Liane Huttner and Denis Merigoux. Traduire la loi en code grâce au langage de programmation Catala. In *Intelligence artificielle et finances publiques*, Nice, France, October 2020.

[9] MESRI. Document de présentation des algorithmes de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/-/blob/master/doc/presentation_algorithmes_parcoursup_2021.pdf, 2021.

[10] MESRI. Dépôt public du code de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/, 2021.

[11] OCDE. Rapport RaC de l'OCDE. https://oecd-opsi.org/projects/rulesascode/.

[12] Hugo Gimbert, Pierre Castéran, Claire Mathieu, Gérald Point. Dépôt des preuves des algoritihmes de Parcoursup Why3 et COQ. https://gitub.u-bordeaux.fr/parcoursup/why3.

Typage avancé de langages dynamiques

Mickaël LAURENT (Université Paris Cité) *

1 Introduction

Avec l'essor du Web, la quantité de code écrit en JavaScript a considérablement augmenté ces dernières années. Afin d'augmenter la sûreté de tels programmes, on veut pouvoir typer avec précision des expressions utilisant des *typecases*, i.e. des branchements conditionnés par le résultat d'un test de type (à l'éxecution) d'une variable ou expression. Ce motif de programmation est omniprésent en JavaScript car il permet d'implémenter la surcharge des fonctions. Des solutions au problème existent, avec des langages comme TypeScript [12] ou Flow[4] qui étendent JavaScript avec un système de types statique. Nous souhaitons cependant aller plus loin que ces langages, sur plusieurs points. Considérons par exemple le code TypeScript suivant

```
function foo(x : number | string) {
    return (typeof(x) === "number")? x+1 : x.trim();
}
```

On reconnait ici une fonction JavaScript classique à la différence près que le paramètre `x` est annoté par un type. Ce code utilise la fonction prédéfinie `typeof` du langage, qui renvoie une chaine de caractères décrivant le type dynamique de son argument. Afin de pouvoir typer statiquement ce programme, il faut déduire, lors du typage de la première branche `x+1`, que `x` est un nombre. De plus, il faut déduire, lors du typage de la seconde branche `x.trim()`, que `x` est une chaine de caractères. Ainsi, les différentes occurrences de la variable `x` devront être typées différemment selon le contexte dans lequel elles se trouvent : on appelle cette discipline de typage *l'occurrence typing*[11].

L'objectif est de créer un système de types respectant les critères suivants :

— intégrer de *l'occurrence typing* afin de pouvoir typer l'exemple ci-dessus.

— fournir des types précis. Par exemple, on veut pouvoir dire que le type de la fonction ci-dessus est $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String})$, désignant les fonctions pouvant prendre en entrée à la fois des entiers et des chaines de caractères, et renvoyant dans le premier cas un entier et dans le deuxième cas une chaine de caractères. Ces intersections de types fonctionnels permettent de typer précisemment les fonctions surchargées, c'est-à-dire les fonctions réalisant des opérations différentes selon le type de leurs paramètres.

— fournir une inférence de types, afin que l'utilisateur n'ai pas besoin d'écrire lui-même des annotations de type. Par exemple, dans le code ci-dessus, le domaine de la fonction doit pouvoir être inféré automatiquement sans l'annotation de type (ce qui est impossible en TypeScript ou Flow, ces derniers ne proposant pas d'inférence de types).

— être suffisamment général pour pouvoir typer avec précision des expressions plus complexes, notamment contenant des tests de type sur une expression arbitraire. Par exemple, lors du typage d'une expression de la forme `typeof(f(x)) == "number")? ... : ...`, on veut pouvoir déduire pour chaque branche des informations de type à la fois sur `x`, sur `f` et sur `f(x)`.

Pour parvenir à ce résultat, nous utilisons des *types ensemblistes* avec du sous-typage sémantique (remplissant de ce fait le deuxième critère présenté ci-dessus).

*Thèse commencée en Septembre 2020 et encadrée par Giuseppe CASTAGNA (CNRS, Université Paris Cité) et Kim NGUYEN (Université Paris-Saclay)

2 Types ensemblistes

Les types ensemblistes que nous utilisons sont ceux définis par FRISCH, CASTAGNA et BENZAKEN [6] et FRISCH, CASTAGNA et BENZAKEN [7]. Leur syntaxe est la suivante :

$$\textbf{Types} \quad t \quad ::= \quad b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \texttt{Empty} \mid \texttt{Any}$$

Chaque type dénote un ensemble de valeurs de notre langage. Ainsi, le type `Empty` ne contient aucune valeur, tandis que le type `Any` dénote l'ensemble de toutes les valeurs. On note $[\![t]\!]$ l'ensemble de valeurs dénoté par un type t : on a par exemple $[\![\texttt{Empty}]\!] = \{\}$, $[\![\texttt{True}]\!] = \{\texttt{true}\}$, $[\![\texttt{Bool}]\!] = \{\texttt{true}, \texttt{false}\}$, etc.

Grâce à cette inteprétation ensembliste des types, on peut définir entre eux une relation de sous-typage dite *sémantique* :

$$t_1 \leq t_2 \stackrel{def}{\Leftrightarrow} [\![t_1]\!] \subseteq [\![t_2]\!]$$

Parmi les travaux utilisant ces types ensemblistes, on compte notamment :

— Le système du langage CDuce[9], décrit par FRISCH [5], qui propose un système puissant mais fonctionnant sur un lambda-calcul explicitement typé. Il propose un pattern matching très général, dont les *typecases* sont un cas particulier, mais ne fait pas d'occurrence typing dessus. Ce système nous servira néanmoins de base.

— Le système décrit par PETRUCCIANI [8] qui permet d'inférer le type des fonctions, mais sans intersection de types fonctionnels. Il ne permet donc pas de typer les fonctions surchargées avec précision. Il ne propose pas d'occurrence typing non plus.

3 Système de types

Le langage que nous utilisons pour notre système de types est le lambda-calcul usuel, muni d'une construction additionnelle pour les *typecases* :

$$\begin{array}{lrcl} \textbf{Expressions} & e & ::= & c \mid x \mid \lambda x.e \mid ee \mid (e,e) \mid \pi_i e \mid \ (e \in t)\ ?\, e : e \\ \textbf{Values} & v & ::= & c \mid \lambda x.e \mid (v,v) \end{array} \tag{1}$$

Nous utilisons comme base les règles de typage usuelles du lambda-calcul simplement typé. Comme nous utilisons des types ensemblistes avec sous-typage, nous avons également besoin d'une règle de subsomption [1] :

$$[\to\text{I}]\ \frac{\Gamma, x : t_1 \vdash e : t_2}{\Gamma \vdash \lambda x.e : t_1 \to t_2} \qquad [\to\text{E}]\ \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2} \qquad [\leq]\ \frac{\Gamma \vdash e : t \quad t \leq t'}{\Gamma \vdash e : t'}$$

Ce qui nous intéresse maintenant, c'est le typage des *typecases*. Pour cela, nous utilisons deux règles simples :

$$[\in_1]\ \frac{\Gamma \vdash e : t \qquad \Gamma \vdash e_1 : t_1}{\Gamma \vdash \ (e \in t)\ ?\, e_1 : e_2 : t_1} \qquad [\in_2]\ \frac{\Gamma \vdash e : \neg t \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \ (e \in t)\ ?\, e_1 : e_2 : t_2}$$

La première concerne uniquement le cas où le test est toujours vrai (et donc la première branche sera toujours choisie), et la deuxième le cas où le test est toujours faux (et donc la deuxième branche sera toujours choisie). Malheureusement, ces deux règles seules ne suffisent pas dans le cas général, où l'expression testée e peut avoir un type qui intersecte à la fois t et $\neg t$.

Afin de pouvoir typer le cas général, nous avons besoin d'une règle capable de décomposer le type s de l'expression testée en deux types disjoints, $s \wedge t$ et $s \wedge \neg t$, afin de pouvoir par la suite appliquer les règles $[\in_1]$ et $[\in_2]$ dans chaque cas séparemment. Nous utilisons pour cela la règle d'élimination de l'union :

$$[\vee]\ \frac{\Gamma \vdash e' : t_1 \vee t_2 \quad \Gamma, x : t_1 \vdash e : t \quad \Gamma, x : t_2 \vdash e : t}{\Gamma \vdash e\{e'/x\} : t}$$

Cette règle peut être appliquée sur les expressions testées, mais pas seulement : elle peut être appliquée sur n'importe quelle sous-expression, permettant ainsi de mettre en corrélation le type de cette sous-expression avec le résultat d'un test.

1. Il s'agit de la règle $[\leq]$. Les règles pour les paires, variables et constantes, quant à elles, ont été omises.

Par exemple, reprenons la fonction `foo` de l'introduction. Pour rappel, elle a le type $(\texttt{Int} \to \texttt{Int}) \wedge (\texttt{String} \to \texttt{String})$. Essayons maintenant de typer l'expression suivante (pour x ayant le type $\texttt{Int} \vee \texttt{String}$) :

$$((\texttt{foo}\ x) \in \texttt{Int})\ ?\ (\texttt{foo}\ x) + x + 1 : 42$$

Afin de typer la première branche, on doit déduire du résultat du test non seulement que $\texttt{foo}\ x$ est un entier, mais également que x est un entier (car si ce n'était pas le cas, $\texttt{foo}\ x$ aurait renvoyé une valeur de type `String`).

Pour cela, on peut simplement appliquer la règle [$\vee$] sur x afin d'étudier séparemment le cas où x a le type `Int` du cas où il a le type `String`. Lorsque x a le type `Int`, alors $(\texttt{foo}\ x)$ a également le type `Int` et ainsi le test est toujours vrai : nous pouvons alors appliquer la règle [$\in_1$] et typer la première branche. Lorsque x a le type `String`, le test est cette fois toujours faux et nous pouvons alors conclure avec la règle [$\in_2$].

L'avantage de ce système est qu'il est très simple et cependant très puissant : la règle d'élimination de l'union permet, en séparant les types de sous-expressions choisies, de faire tous les raffinements possibles de notre environnement lors du typage des branches d'un *typecase*. Nous avons montré que ce système est *sûr*, dans le sens où une expression typable avec ce système se réduit toujours à une valeur du même type ou alors diverge.

En revanche, l'inconvénient est que ce système n'est pas algorithmique :

— La règle [$\vee$] rend le système non dirigé par la syntaxe, car elle peut s'appliquer dans n'importe quel contexte et substituer n'importe quelle(s) sous-expression(s).

— Les règles [$\to$I] et [$\vee$] sont non-analytiques, dans le sens où l'on doit deviner, respectivement, le type t_1 de la variable abstraite et la décomposition $t_1 \vee t_2$ du type de la sous-expression choisie.

Pour résoudre le premier point, nous avons introduit une forme normale pour les expressions, que nous avons appelé *Maximal Sharing Canonical form* (MSC). Le principe de la forme MSC est de déplacer toutes les sous-expressions de notre expression initiale dans des définitions séparées, tout en factorisant le code le plus possible (i.e., deux sous-expressions syntaxiquement équivalentes vont partager la même définition). Par exemple, l'expression $((\texttt{foo}\ x) \in \texttt{Int})\ ?\ (\texttt{foo}\ x) + x + 1 : 42$ devient :

$$\begin{array}{l}
\texttt{bind}\ u = x + 1\ \texttt{in} \\
\texttt{bind}\ v = \texttt{foo}\ x\ \texttt{in} \\
\texttt{bind}\ w = v + u\ \texttt{in} \\
\texttt{bind}\ z = (v \in \texttt{Int})\ ?\ w : 42\ \texttt{in} \\
z
\end{array}$$

L'intéret est que désormais, toutes les sous-expressions sont définies les unes après les autres, et on peut donc appliquer la règle [$\vee$] une fois sur chacun de ces `bind` sans perdre de généralité. Ainsi, notre nouvelle règle [$\vee$] devient :

$$[\vee\text{-MSC}]\ \frac{\Gamma \vdash_{\mathcal{I}} e_1 : \bigvee_{j \in J} t_j \quad {}^{(\forall j \in J)}\ \Gamma, x{:}t_j \vdash_{\mathcal{I}} e_2 : s_j}{\Gamma \vdash_{\mathcal{I}} \texttt{bind}\, x = e_1\, \texttt{in}\, e_2 : \bigvee_{j \in J} s_j}\ J \neq \varnothing$$

Pour résoudre le second point, qui est de choisir la bonne décomposition lors de l'application de [$\vee$] et le bon type pour la variable abstraite lors de l'application de [$\to$I], nous ajoutons des annotations dans nos expressions en forme MSC : chaque lambda-abstraction et chaque `bind` se voit enrichi d'annotations indiquant, en fonction de l'environnement actuel, les types à considérer.

Avec ces expressions en forme MSC annotées, nous pouvons définir un système de types algorithmique équivalent au système de types initial : toute expression typable dans le système initial, une fois mise en forme MSC, peut être annotée de telle sorte à devenir typable dans le système algorithmique. Inversement, si une expression en forme MSC est typable par le système algorithmique, alors l'expression d'origine est typable par le système initial.

La dernière étape pour pouvoir typer une expression du langage source est donc de fournir un algorithme permettant d'inférer ces annotations. Nous avons proposé un tel algorithme qui, bien que non complet, donne en général de meilleurs résultats que les approches actuelles de la littérature (notamment [10]). Cet algorithme fonctionne itérativement en raffinant au fur et à mesure les annotations de notre expression en forme MSC : il va pour cela regarder les tests de type qui sont faits ainsi que les applications de fonctions surchargées, et en déduire des décompositions à faire jusqu'à ce que l'expression devienne typable, ou au contraire, que l'algorithme détecte qu'il ne pourra pas la rendre typable.

Nous avons réalisé un prototype implémentant le système de type présenté dans cette section (ainsi que l'algorithme d'inférence). Il peut être testé en ligne à cette adresse : `https://typecaseunion.github.io/`. Ce prototype a été implémenté en OCaml (environ 4000 lignes à l'heure actuelle), en utilisant la bibliothèque de types ensemblistes du langage CDuce[9].

4 Extensions en cours et futures

Le travail présenté jusqu'ici a donné lieu à une publication à la conférence POPL [1]. Cependant, le système de types et le système d'inférence des annotations que nous y avons présenté a quelques limitations. Nous en avons déjà dépassé certaines, mais il reste plusieurs points à améliorer.

4.1 Inférence des arguments fonctionnels

Le système d'inférence des annotations n'est pas capable d'inférer des types fonctionnels pour les paramètres d'une lambda-abstraction. Nous sommes actuellement en train de retravailler ce système d'inférence pour permettre cela. Par exemple, si on modifie notre fonction `foo` de la manière suivante :

```
function foo(f, x) {
    return (typeof(x) === "number")? f(x)+1 : x.trim();
}
```

alors on veut pouvoir inférer le type $((\texttt{Int} \to \texttt{Int}) \times \texttt{Int} \to \texttt{Int}) \wedge (\texttt{Any} \times \texttt{String} \to \texttt{String})$.

4.2 Polymorphisme

Notre système de types actuel ne supporte pas le polymorphisme. Ainsi, la fonction identité sera typée par notre système avec le type $\texttt{Any} \to \texttt{Any}$, ce qui est peu précis. En se basant sur les travaux [2, 3], l'ajout du polymorphisme à notre système devrait résoudre un certain nombre de problèmes de précision. Par exemple, si on modifie la seconde branche de `foo` de la manière suivante :

```
function foo(x) {
    return (typeof(x) === "number")? x+1 : x;
}
```

alors on veut pouvoir inférer le type $\forall\alpha.\ (\texttt{Int} \to \texttt{Int}) \wedge (\alpha \setminus \texttt{Int} \to \alpha \setminus \texttt{Int})$.

4.3 Effets de bord

Nous travaillons actuellement avec un lambda-calcul pur. L'ajout d'effets de bord nécessite dans un premier temps de revoir la règle d'élimination de l'union : en effet, cette dernière ne devrait pas pouvoir s'appliquer sur des sous-expressions non pures, car deux expressions non pures syntaxiquement équivalentes peuvent se réduire à deux valeurs différentes. Par exemple, remplaçons dans notre fonction `foo` la variable x par une application $f(x)$, avec f de type $\texttt{Any} \to (\texttt{Int} \vee \texttt{String})$:

```
function foo(x) {
    return (typeof(f(x)) === "number")? f(x)+1 : f(x).trim();
}
```

Si on suppose que f est pure, alors cette expression peut être typée par $\texttt{Any} \to (\texttt{Int} \vee \texttt{String})$. En revanche, sans la garantie que f est pure, alors ce type n'est plus correct car l'appel à $f(x)$ de la première branche peut donner une valeur dans `String`, malgré le fait que ce même appel a donné une valeur dans `Int` lors du précédent test.

Au niveau du système algorithmique, c'est le partage réalisé lors de la mise en forme MSC qui doit être régulé : deux sous-expressions impures ne doivent pas être représentées par la même variable. Cela nécessite de détecter au préalable les expressions impures, et ceci le plus finement possible. Cette analyse étant préalable au typage, elle devra se faire directement sur une expression non-typée.

Références

[1] Giuseppe CASTAGNA et al. “On Type-Cases, Union Elimination, and Occurrence Typing”. In : *Proc. ACM Program. Lang.* 6.POPL (jan. 2022). DOI : 10.1145/3498674. URL : https://doi.org/10.1145/3498674.

[2] Giuseppe CASTAGNA et al. “Polymorphic Functions with Set-Theoretic Types. Part 1 : Syntax, Semantics, and Evaluation”. In : *Proceedings of the 41st Annual ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’14. Jan. 2014, p. 5-17. DOI : 10.1145/2676726.2676991.

[3] Giuseppe CASTAGNA et al. “Polymorphic functions with set-theoretic types. Part 2 : local type inference and type reconstruction”. In : *Proceedings of the 42nd Annual ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL ’15. Jan. 2015, p. 289-302. DOI : 10.1145/2676726.2676991.

[4] *Flow*. Facebook URL : https://flow.org/.

[5] Alain FRISCH. “Théorie, conception et réalisation d’un langage de programmation adapté à XML”. Thèse de doct. Université Paris Diderot, déc. 2004. URL : http://www.cduce.org/papers/frisch_phd.pdf.

[6] Alain FRISCH, Giuseppe CASTAGNA et Véronique BENZAKEN. “Semantic Subtyping”. In : *LICS ’02, 17th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2002, p. 137-146. DOI : 10.1109/LICS.2002.1029823.

[7] Alain FRISCH, Giuseppe CASTAGNA et Véronique BENZAKEN. “Semantic subtyping : dealing set-theoretically with function, union, intersection, and negation types”. In : *Journal of the ACM* 55.4 (sept. 2008), 19 :1-19 :64. ISSN : 0004-5411. URL : http://doi.acm.org/10.1145/1391289.1391293.

[8] Tommaso PETRUCCIANI. “Polymorphic Set-Theoretic Types for Functional Languages”. Thèse de doct. Joint Ph.D. Thesis, Università di Genova et Université Paris Diderot, mar. 2019. URL : https://tel.archives-ouvertes.fr/tel-02119930.

[9] *The CDuce Compiler*. CDuce URL : https://www.cduce.org.

[10] Sam TOBIN-HOCHSTADT et Matthias FELLEISEN. “Logical types for untyped languages”. In : *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP ’10. Baltimore, Maryland, USA : ACM, 2010, p. 117-128. ISBN : 978-1-60558-794-3. DOI : 10.1145/1863543.1863561. URL : http://doi.acm.org/10.1145/1863543.1863561.

[11] Sam TOBIN-HOCHSTADT et Matthias FELLEISEN. “The Design and Implementation of Typed Scheme”. In : *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL ’08. San Francisco, California, USA : ACM, 2008, p. 395-406. ISBN : 978-1-59593-689-9. DOI : 10.1145/1328438.1328486.

[12] *TypeScript*. Microsoft URL : https://www.typescriptlang.org/.

Knit&Frog: Pattern matching compilation for custom memory representations (doctoral session)

Thaïs Baudon[1], Gabriel Radanne[2], and Laure Gonnord[3]

[1]ENS Lyon/LIP
[2]Inria/LIP
[3]Grenoble-INP/LCIS & LIP

Abstract

Initially present only in functional languages such as OCaml and Haskell, Algebraic Data Types have now become pervasive in mainstream languages, providing nice data abstractions and an elegant way to express functions though *pattern-matching*. Numerous approaches have been designed to compile rich pattern matching to cleverly designed, efficient decision trees. However, these approaches are specific to a choice of *internal memory representation* which must accommodate garbage-collection and polymorphism.

ADTs now appear in languages more liberal in their memory representation such as Rust. Notably, Rust is now introducing more and more optimizations of the memory layout of Algebraic Data Types. As memory representation and compilation are interdependent, it raises the question of pattern matching compilation in the presence of non-regular, potentially customized, memory layouts.

In this article, we present Knit&Frog, a framework to compile pattern-matching for monomorphic ADTs, *parametrized* by an arbitrary memory representation. We propose a novel way to describe choices of memory representation along with a validity condition under which we prove the correctness of our compilation scheme. The approach is implemented in a prototype tool `ribbit`.

1 Introduction

Algebraic Data Types (ADTs) are an essential tool to model data and information. They allow to group together information in a consistent way through the use of records, also called product types, and to organize options through the use of variants, also called sum types. Product and sum types together allow to enforce invariants present in the domain under study and provide convenient tools to inspect these data-types through pattern matching.

Combined, these features offer numerous advantages:

- Model data in a way that is close to the programmer's intuition, abstracting away the details of the memory representation of said data.
- Safely handle data by ensuring via pattern-matching that its manipulation is well-typed, exhaustive and non-redundant.
- Optimize manipulation of data thanks to rich constructs understood by the compiler.

Despite these promises, Algebraic Data Types were initially only present in functional programming languages such as OCaml and Haskell. Recently, they have gained a foothold in more mainstream languages such as Ada, Typescript, Scala, Rust and even soon Java. They are however still lacking in high-performance, lower-level languages. One difficulty faced by language designers who want to add pattern matching to their language is that compiling a rich pattern matching language to efficient code is a non-trivial task, which is not commonly available in shared compiler frameworks such as LLVM.

Indeed, such frameworks only provide optimizations for C-like switches on integers (or integer-like enumerations). Additionally, existing works on pattern matching (Maranget, 2008; Wadler, 1987; Sestoft, 1996) provide very efficient compilation schemes, but are geared towards memory representations found in GC-managed functional languages such as OCaml and Haskell: uniform representations with liberal usage of boxing. Highly non-uniform data representations such as the ones found in C++ do not easily fit.

More generally, the descriptive nature of ADTs should enable compilers to aggressively optimize the representation of terms. The simplest example is the `Option` type, which is either `Some value` or `None`. If the value in question is an integer from 0 to 10, `None` can easily be represented as 11. This optimization is regularly done by programmers manually, forgoing the guarantees provided by languages. More complex optimizations on nested and rich data-types are even more error-prone. These transformations could easily be done automatically by the compiler. This trove of optimization potential has been brushed on in recent versions of Rust, but remains largely unexplored.

Finally, on top of the previously mentioned difficulty of adapting pattern matching compilation schemes to irregular memory representations, the correctness of such compilation schemes is also delicate to establish when the terms can be arbitrarily shaped.

The wider goal of this thesis is to enable the use of ADTs in high-performance applications. This requires optimized, possibly custom memory representations for ADTs, either constructed automatically or user-provided. However, current pattern matching compilation approaches mandate a fixed representation (often designed for high-level, garbage-collected languages) and thus lack the required flexibility. In addition, a memory-aware pattern matching compilation approach could enable further performance gains by tweaking the decision tree according to individual representation quirks. This requires a more flexible pattern matching compilation approach, able to handle various memory representations.

In this article, we lay the groundwork for highly optimized pattern matching in any language with arbitrary non-uniform memory representation. We present Knit&Frog, a compilation framework for rich pattern languages with arbitrary memory representation of monomorphic terms:

- We provide a characterization of memory representations for terms of Algebraic Data Types as two main operations: **Knit** which builds a term into its memory representation, and **Frog** which deconstructs the memory representation to identify the underlying term.
- We provide a compilation scheme parameterized by the memory representation. This scheme is adapted from Maranget (2008), the best implementation of pattern matching known thus far.
- We provide a sufficient condition for a memory representation to yield itself to pattern matching and prove end-to-end correctness of our compilation scheme in this case.

We first give some motivation examples of memory representations and pattern-matching compilation as motivation, before hinting at the scientific content of our presentation.

2 Algebraic Data Types and their memory representation

Let us now explore the different memory representation choices for Algebraic Data Types (ADTs, for short), and how their values are manipulated. In this section, we present various examples of types and programs and discuss how they are currently represented in existing languages, and how they might be. For illustration, we will look at the output of two languages which implement ADTs: Rust and OCaml. While these languages have some common lineage, they have a very different attitude towards code emission: OCaml is a GC-managed language which factors predictability and regularity. Rust on the other hand favors performance and absolute control over low-level details. These differences result in drastically different choices in memory representation, hence the need for a sophisticated pattern matching compiler that is flexible enough to take these differences into account. To illustrate this, let's consider an example of code manipulating *red-black trees*.

```rust
//Colors for the Red-Black Trees
enum Color { Red, Black, }

//Red-Black Trees of content T
enum RBT<T> {
  Node (Color, T, &RBT<T>, &RBT<T>),
  Empty,
}
```

```rust
match c, v, t1, t2 {
  Black, z, &Node(Red, y, &Node(Red, x, a, b), c), d
  | Black, z, &Node(Red, x, a, &Node(Red, y, b, c)), d
  | Black, x, a, &Node(Red, z, &Node(Red, y, b, c), d)
  | Black, x, a, &Node(Red, y, b, &Node(Red, z, c, d))
  => Node(Red, y, &Node(Black, x, a, b), &Node(Black, z, c, d)),
  a, b, c, d => Node (a, b, c, d),
}
```

Figure 1: Example of Red-Black trees and their balancing operation in Rust

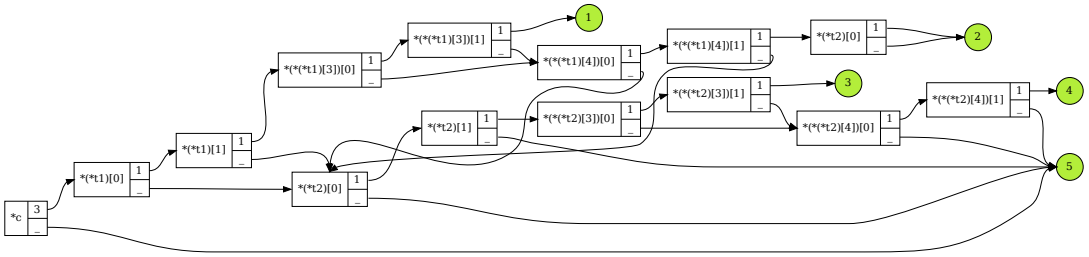


Figure 2: Output of our `ribbit` compiler for the code of Fig. 1

This output is a (decision tree) whose root is on the left. Decision (square) nodes compute values from their input (low-level representation), then branch depending on this value. Round nodes denote the final returned value of the procedure (the numbering of the matched branch).

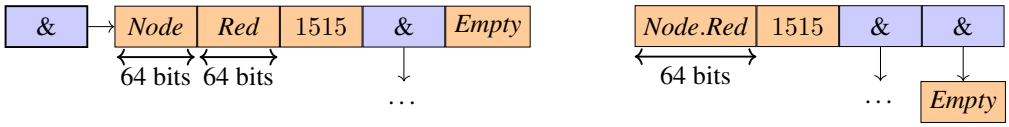


Figure 3: OCaml and Rust memory representations of $\mathcal{T}$ = `Node(Red, 1515, &..., &Empty)`

The optimized decision trees produced by such a flexible compiler should still respect, and even take advantage of, the memory representation. A Rust version is depicted in Fig. 1. This type definition expresses trees as recursive data structures, and uses a tuple for the various fields in the `Node` case. For instance, $\mathcal{T}$ = `Node(Red, 1515, &Node(Black, 42, &Empty,&Empty), &Empty)` is a tree of type `RBT<Int>`.

Red-Black trees famously rely on a fairly complex balancing step, which redistributes the colors depending on the internal invariant of the data structure. Thanks to nested pattern matching, this step can be expressed very compactly, as shown in Fig. 1. This pattern matching inspects four arguments at the same time: the current color `c`, the current value `v` and the sub-trees `t1` and `t2`.

Since this pattern matching is at the core of a performance-sensitive data structure, we naturally want it to be as efficient as possible. This is why many pattern matching implementations come with clever heuristics and techniques to output optimized decision trees (Kosarev u. a., 2020; Maranget, 2008; Sestoft, 1996). The resulting code is highly non-trivial, as can be seen in Fig. 2.

The OCaml-equivalent type of `RBT<u64>` would use 6 memory words per `Node`, as illustrated in Fig. 3. However, type `RBT<u64>` is represented by the Rust compiler using only 4 memory words per `Node`: a node is then made of a word with a bit marking it as non-0 (to avoid confusion with `None`) and the color bit, followed by the 64-bit integer and the two pointers. It is however possible to be even more compact: the Linux kernel uses a hand-crafted representation of Red-Black trees that exploits bit-stealing to use one less word (Wilke, 2016): since pointers are word-aligned, colors can be stored in the lower bits of each pointer. These complex manually-tuned optimizations are common in low-level C programming, but come at a high cost: programmers forgo the use of modern safer constructs such as Algebraic Data Types and pattern matching. Furthermore, they now need to design the decision tree themselves, and make

sure their choice of representation contains enough information to distinguish, for instance, between the `Empty` and `Node` cases. Moreover, the emission of code for such mangled objects in memory is delicate (notably, the bit-stealing technique in question is undefined behavior in the C standard).

Ideally, we would like to still use high-level pattern languages, while the compilation process adapts and exploits the optimized representation. Such optimized representation would be found automatically by the compiler, or specified by the programmer, transparently from the rest of the code which only mentions constructors of the Algebraic Data Type. We make a first step towards this goal by developing Knit&Frog, a pattern matching compilation technique **parameterized by the memory representation**.

3 Complete article

In the complete paper, we make the following contributions:

1. a source pattern language featuring variables and or-patterns and a target decision tree language with switches on memory expressions that include pointer dereferencing operations and array accesses;
2. a formal definition of memory representations consisting of three ingredients: a mapping REPR$^\bullet$ from values to memory values; a function KNIT$^\bullet$ that rebuilds the type-appropriate representation of a subterm from the memory representation of a parent term and a function FROG$^\bullet$ that decodes a memory value to discriminate between different variants of a type;
3. a compilation scheme inspired by Maranget (2008) parametrized by these three ingredients;
4. a validity criterion ensuring that KNIT$^\bullet$ and FROG$^\bullet$ correctly manipulate memory representations of values and a proof of correctness of our algorithm;
5. several examples of both toy and realistic memory representations;
6. `ribbit`, a prototype implementation of our approach.

The complete article is currently under submission; an ArXiV link will be included in the final version of this extended abstract.

During our presentation, we will detail our approach to formalizing memory representations and how to compile pattern matching accordingly, with illustrating examples. We will finish our presentation with future perspective, in particular our plan to generate new optimized representations of Algebraic Data Types which fulfill our validity criterion.

References

[Kosarev u. a. 2020] KOSAREV, Dmitry ; LOZOV, Petr ; BOULYTCHEV, Dmitry: Relational Synthesis for Pattern Matching. In: S. OLIVEIRA, Bruno C. d. (Hrsg.): *Processings of the 18th Programming Languages and Systems Asian Symposium, APLAS 2020, Fukuoka, Japan, 2020* Bd. 12470, Springer, 2020, S. 293–310. – URL https://doi.org/10.1007/978-3-030-64437-6_15

[Maranget 2008] MARANGET, Luc: Compiling pattern matching to good decision trees. In: SUMII, Eijiro (Hrsg.): *Proceedings of the ACM Workshop on ML, 2008, Victoria, BC, Canada, September 21, 2008*, ACM, 2008, S. 35–46. – URL https://doi.org/10.1145/1411304.1411311

[Sestoft 1996] SESTOFT, Peter: ML pattern match compilation and partial evaluation. In: *Partial Evaluation*. Springer, 1996, S. 446–464

[Wadler 1987] WADLER, Philip: Efficient compilation of pattern-matching. In: *The implementation of functional programming languages* (1987)

[Wilke 2016] WILKE, Pierre: *Formally verified compilation of low-level C code. (Compilation formellement vérifiée de code C de bas-niveau)*, University of Rennes 1, France, Dissertation, 2016. – URL https://tel.archives-ouvertes.fr/tel-01483676

AFADL

xDSLs dirigés par les Modèles Formels
Tour d'horizon de l'outil Meeduse

Akram Idani, German Vega

Univ. Grenoble Alpes, Grenoble INP, CNRS, LIG
F-38000 Grenoble France
{nom.prenom}@univ-grenoble-alpes.fr

Résumé

Meeduse est un atelier de conception formelle de langages dédiés domaine (DSLs). Il permet de travailler sur la correction d'un DSL au moyen d'outils de raisonnements automatisés. La force de l'outil provient de ProB (un animateur et un model-checker de la méthode B) et d'EMF (un environnement IDM bien établi). Cet article passe en revue deux usages utiles de Meeduse : (*i*) exécution et déboggage, et (*ii*) transformation de modèles.

Mots clés : DSLs, Méthode B, Animation, Transformation de modèles.

1 Introduction

Plusieurs langages de modélisation dédiés domaine (DSLs), tels que les automates de D. Harel, les réseaux de Pétri ou les diagrammes de séquences d'UML décrivent les aspects comportementaux d'un système. Ils disposent ainsi d'une sémantique dite opérationnelle qui leur confère un caractère exécutable − d'où l'appellation xDSLs[1]. Les travaux IDM qui se sont intéressés à ce type de DSLs, ont donné lieu à des langages d'action (*e.g.* Kermeta, fUML) ainsi que des outils divers (*e.g.* xMOF, USE). Grâce à son "exécutabilité", un xDSL permet de simuler le comportement du système bien avant les activités de codage. L'objectif en est d'aborder les activités de vérification et validation (V&V) tôt dans le processus de développement. Dans [3] nous avons montré qu'une approche formelle peut s'avérer incontournable lors de la définition de la sémantique du xDSL. En effet, si celle-ci n'est pas rigoureusement définie, les activités de V&V peuvent être grandement impactées. Aussi, une approche formelle permet-elle de neutraliser les risques d'erreur émanant du langage et ré-hausser de fait son intérêt.

Malheureusement, les ateliers de conception de DSLs (appelés Language Workbenches − LWBs) ne supportent pas les méthodes formelles. Cette constatation repose sur les études de Kosar et al., [7] et Iung et al., [6] qui ont finement étudié et analysé les LWBs de 2006 jusqu'à 2019. La première étude (2006-2012) indique clairement qu'il y a un besoin urgent d'identifier les raisons et trouver les solutions. La deuxième étude (2012-2019) ne se réfère qu'au test comme activité de vérification et montre que sur les 59 outils analysés seuls 9 proposent un support au test. Dans nos travaux, nous avons cherché à donner une réponse pragmatique à cette problématique grâce à l'usage de la méthode B [1] et ses outils de preuve, d'animation et de model-checking. Nous avons développé Meeduse[2], un LWB qui permet d'instrumenter formellement un xDSL en décrivant sa sémantique en B et en fournissant l'outillage de vérification nécessaire. Une première preuve de concept a été présentée à AFADL'2018 − l'outil n'était qu'à l'état embryonnaire. Depuis, il a évolué, a gagné en maturité et a fait l'objet de plusieurs publications [2, 3, 4]. Il a aussi été appliqué avec succès sur plusieurs études de cas et a remporté deux prix au challenge TTC'19 (best verification et audience award). Dans cet article, nous passons en revue deux usages utiles de Meeduse : (*i*) exécution et déboggage, et (*ii*) transformation de modèles.

[1] xDSL: Executable Domain-Specific Language.
[2] `http://vasco.imag.fr/tools/meeduse/`

2 Exécution et déboggage

L'exécution et le déboggage – termes communément utilisés dans le jargon IDM – servent à observer le comportement d'un modèle dans le but d'identifier et comprendre des erreurs éventuelles. Dans le domaine des méthodes formelles on évoque souvent le terme animation interactive avec un objectif de vérification plus large que le déboggage ; cela peut couvrir la décomposition de prédicats, la résolution de contraintes à la volée, l'abstraction d'états, etc. Il existe deux approches pour assurer l'exécution et le déboggage d'un DSL : les approches interprétées, et les approches compilées. Dans le cadre des approches interprétées, l'état d'un modèle ainsi que les étapes d'exécution qui modifient cet état, sont définis au niveau du modèle au travers d'un langage spécifique. C'est donc le modèle lui-même qui est directement exécuté, analysé et déboggé. Cependant, les fonctionnalités de vérification de ces techniques sont très limitées. Dans le cadre des approches compilées, le modèle est traduit dans un formalisme cible exécutable dans le but de bénéficier des outils sous-jacents. La limitation majeure de ce deuxième type d'approches est que l'exécution se fait dans l'espace technologique du formalisme cible, sans aucune rétro-action au niveau du modèle source.

Meeduse met en œuvre une approche compilée, avec pour espace technologique cible celui de la méthode B. D'une part, il repose sur ProB [8] et bénéficie par conséquent de la richesse de ses fonctionnalités ; et d'autre part, il apporte la brique manquante car il réalise les rétro-actions requises au niveau du modèle source. L'approche de l'outil est présentée dans la Figure 1. A partir du méta-modèle du DSL, Meeduse génère une machine B fonctionnelle (appelée Static Semantics) qui représente ses aspects structurels. La sémantique opérationnelle du langage peut alors être définie dans une machine B à part (appelée Dynamic Semantics) qui inclut la machine fonctionnelle. Ce niveau sémantique permet de travailler sur la correction du DSL au moyen d'outils de preuve comme l'AtelierB.

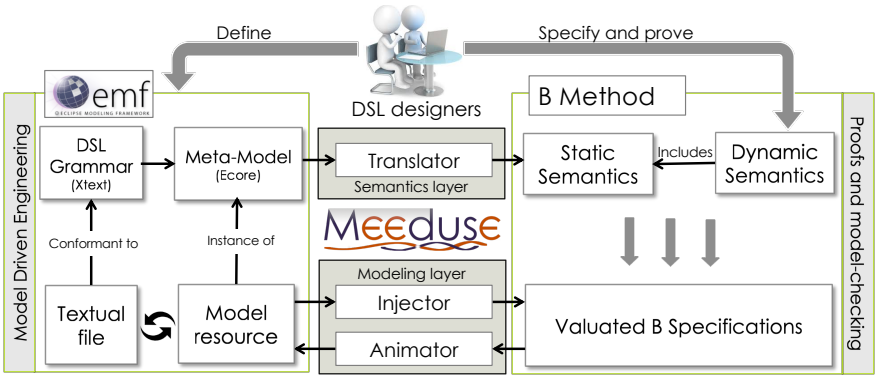


Figure 1: Concepts majeurs de Meeduse (Figure issue de [2])

Pour réaliser l'exécution de modèles, Meeduse embarque ProB et l'utilise de manière complètement transparente pour l'utilisateur final. A partir d'un modèle conforme au méta-modèle du DSL, Meeduse produit un raffinement de la spécification fonctionnelle en y introduisant les données du modèle. On obtient ainsi une spécification valuée et sémantiquement équivalente au modèle en entrée. La Figure 2 donne un exemple : (1) la machine *MeeduseTuto* est extraite à partir d'un méta-modèle contenant une classe **`Counter`** avec un attribut **`value`** de type entier ; et (2) le raffinement *MeeduseTuto_r* est extrait d'un modèle constitué d'une instance de cette classe où **`value`** est égal à 5. Dans cet exemple, on a considéré que seul l'attribut changera de valeur lors de l'animation.

Ayant cette spécification valuée, ProB entre en action pour calculer les opérations déclenchables dans l'état courant. A chaque pas de l'animation, Meeduse calcule la différence entre l'état avant et l'état après l'animation, et applique les modifications sous-jacentes au modèle source produisant ainsi son exécution.

```
MACHINE MeeduseTuto
SETS
   COUNTER
CONCRETE_CONSTANTS
   Counter
PROPERTIES
   Counter ∈ F (COUNTER)
CONCRETE_VARIABLES
   value
INVARIANT
   value ∈ Counter → NAT
INITIALISATION
   value :∈ Counter → NAT
END
```

```
REFINEMENT MeeduseTuto_r
REFINES MeeduseTuto
CONSTANTS
   COUNTER1
PROPERTIES
   COUNTER1 ∈ COUNTER ∧
   Counter = {COUNTER1}
INITIALISATION
   value := {(COUNTER1 ↦ 5)}
END
```

Figure 2: Modèles et méta-modèles dans Meeduse

3 Transformation de modèles

Dès lors qu'on parvient à exécuter un modèle EMF en entrée au moyen de ProB, il devient possible d'instrumenter en B plusieurs paradigmes de l'IDM, tels que la fusion, la composition, ou la transformation de modèles. En 2019, nous avons participé à la compétition TTC (Transformation Tool Contest) et nous avons montré la viabilité de l'outil dans ce cadre [5]. Meeduse a été utilisé non seulement pour montrer comment vérifier grâce au langage B, une transformation de modèles, mais aussi pour l'exécuter et produire les modèles cibles. L'outil n'a pas été conçu comme étant dédié à la transformation de modèles. Néanmoins, il offre la possibilité d'exécuter plusieurs modèles qui dépendent les uns des autres ; et c'est justement cette possibilité qui le rend bien adapté à la transformation de modèles.

La traduction d'un méta-modèle qui utilise des concepts d'un ou plusieurs autres méta-modèles donne lieu à une machine B unique qui inclut les structures de données de tous les méta-modèles. Ce même principe est appliqué au niveau modèle. En conséquence, pour instrumenter une transformation de modèle dans Meeduse, il suffit de s'inspirer du schéma de la Figure 3. Ayant un méta-modèle source $\mathrm{MM_{Source}}$ et un méta-modèle cible $\mathrm{MM_{Cible}}$, on introduit un méta-modèle $\mathrm{MM_{transfo}}$ qui utilise $\mathrm{MM_{Source}}$ et $\mathrm{MM_{Cible}}$. Étant donnée que la spécification B qui découle de $\mathrm{MM_{transfo}}$ couvre tous les concepts utiles (ceux de $\mathrm{MM_{Source}}$ et ceux de $\mathrm{MM_{Cible}}$), alors une transformation de modèle peut être spécifiée au travers d'opérations B dans cette même spécification. L'animation est réalisée sur la base d'une instance de $\mathrm{MM_{transfo}}$ qui désigne le modèle source à transformer. Quant au modèle cible, il est produit au fur et à mesure de l'animation des opérations B calculées par ProB.

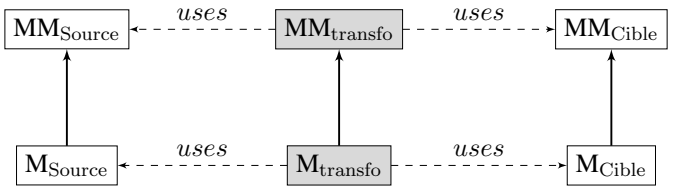


Figure 3: Architecture d'une transformation de modèles dans Meeduse

Cette approche a plusieurs bénéfices :

- Définir au moyen d'invariants B, les propriétés qu'une transformation doit préserver ;
- Spécifier les règles de transformation sous forme d'opérations et prouver leur correction vis-à-vis des propriétés invariantes ;

- Expliciter les concepts utiles à la trace de la transformation dans $\mathrm{MM}_{\mathrm{transfo}}$, et les inclure dans la spécification formelle.

4 Conclusion

Meeduse se présente aujourd'hui comme étant le seul atelier de conception formelle de DSLs. En effet, les investigations existantes [6, 7] pointent l'absence de méthodes formelles dans les LWBs. Notre approche repose sur la méthode B et ses outils, et couvre aussi bien les DSLs graphiques (conçus via Sirius ou basés sur UML comme dans le cadre de Papyrus) que les DSLs textuels (définis au moyen d'une grammaire Xtext). Outre cette contribution au monde des DSLs, Meeduse peut apporter des bénéfices immédiats aux nombreuses recherches, réalisées par la communauté B, pour la formalisation de notations semi-formelles. Nous pouvons citer, sans être exhaustifs, [10] où les auteurs proposent une formalisation de la sémantique opérationnelle des diagrammes de séquences d'UML2.x, ou encore [9] présentant une sémantique run-to-completion aux diagrammes d'états/transition, etc. Non seulement les sémantiques opérationnelles que ces travaux proposent pourraient être instrumentées et exécutées dans Meeduse mais aussi les transformations sous-jacentes. En effet, l'extraction de spécifications B à partir d'un modèle source n'est autre qu'une transformation d'un méta-modèle source vers un méta-modèle (ou une grammaire) de B. Grâce à Meeduse il devient possible d'écrire ces transformations en B lui-même, prouver leur correction, et les exécuter pour produire le modèle B escompté.

References

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] A. Idani. Meeduse: A tool to build and run proved DSLs. In *16th International Conference on Integrated Formal Methods (IFM)*, volume 12546 of *LNCS*, pages 349–367. Springer, 2020.

[3] A. Idani. Formal model-driven executable DSLs: Application to Petri-Nets. *International Journal on Innovations in Systems and Software Engineering (ISSE)*, 18(1), 2022.

[4] A. Idani. The B Method meets MDE: Survey, progress and future. In *16th Int. Conference on Research Challenges in Information Science (RCIS)*, volume 446, pages 495–512. Springer, 2022.

[5] A. Idani, G. Vega, and M. Leuschel. Applying formal reasoning to model transformation: The meeduse solution. In *Proceedings of the 12th Transformation Tool Contest, co-located with STAF'2019, Software Technologies: Applications and Foundations*, volume 2550, pages 33–44, 2019.

[6] A. Iung, J. Carbonell, L. Marchezan, E. M. Rodrigues, M. Bernardino, F. P. Basso, and B. Medeiros. Systematic mapping study on domain-specific language development tools. *Empirical Software Engineering*, 25(5):4205–4249, 2020.

[7] T. Kosar, S. Bohra, and M. Mernik. Domain-specific languages: A systematic mapping study. *Information and Software Technology*, 71:77–91, 2016.

[8] M. Leuschel and M. Butler. ProB: an automated analysis toolset for the B method. *Software Tools for Technology Transfer (STTT)*, 10(2):185–203, 2008.

[9] K. Morris, C. F. Snook, T. S. Hoang, G. C. Hulette, R. Armstrong, and M. J. Butler. Formal Verification of Run-to-Completion Style Statecharts Using Event-B. In *14th European Conference on Software Architecture*, volume 1269 of *CCIS*, pages 311–325. Springer, 2020.

[10] I. Mouakher, F. Dhaou, and J.-C. Attiogbé. Event-Based Semantics of UML 2.X Concurrent Sequence Diagrams for Formal Verification. *Journal of Computer Science and Technology*, 37(1):4–28, 2022.

An Incremental Model-Based Design Methodology to Develop CPS with SysML/OCL/Reo

PhD student: Perla Tannoury
Supervisors: Dr. Samir Chouali & Dr. Ahmed Hammad
perla.tannoury@femto-st.fr, schouali@femto-st.fr, ahammad@femto-st.fr
Univ. Bourgogne Franche-Comté, FEMTO-ST Institute/CNRS, Besançon

April 2022

Abstract

Modeling Cyber-Physical Systems (CPS) remains a challenge due to their interconnected networks of heterogeneous embedded systems that operate in a physical environment. In this paper, we introduce a new modeling approach that relies on SysML, OCL, and Reo to capture the different aspects of CPS, including requirements, architecture, and interaction protocols. The novelty of our approach relies in the combination of SysML and Reo to handle the complexity of CPS architecture and protocols, in the design step by proceeding incrementally. Furthermore, we define OCL constraints to specify rules to be respected to model consistently CPS.
Keywords: CPS. SysML. Reo. OCL. Meta-Model. Incremental Design.

1 Introduction

Cyber-physical systems (CPSs) are a couple of computational systems (Cyber) with sensors and actuators (Physical), executing in varying spatial and temporal contexts while exhibiting diverse behaviors across runs. The relationship between the logical components and the physical ones is very complex. Therefore, many obstacles arise through their combination. To address these problems, a design methodology is needed on the one hand in a modeling language for modeling and analyzing all CPS facets and on the other hand in an appropriate tool to verify and simulate the modeled system.

The System Modeling Language (SysML), aims to support systems that exhibit hybrid phenomena, such as CPSs, by combining the continuous phenomena of physical systems with the discrete phenomena of software systems. SysML has many advantages in modeling a system's architecture, behavior, and requirements. However, CPS are usually developed by assembling components incrementally, which leads to complex behavior of the final system. SysML is limited to correctly specifying such behaviors, hence the need to use a powerful coordination language, to specify compositionally complex interactions between CPS components.

Reo [1], is a powerful coordination language that captures the foundations of the system. Reo helps the CPS designer to formally specify how, when, and under what conditions data can flow from a component's input to output ports. It makes it simple to design and analyze the dynamics of scalable systems, as well as to automatically validate conformance and ensure interoperability, which can be used

to overcome the limitations of SysML in modeling CPSs. In addition, Reo has a graphical notation, which allows us to use it to enrich SysML models.

To the best of our knowledge, we are not aware of a comprehensive work on CPS modelization that combines SysML and Reo together. Most of previous works in [2, 4, 3] either focus on SysML or on Reo but never on both. SysML alone models the architecture and internal communication flow of components. Reo alone models the interaction protocols of the components. Meanwhile, the combination of SysML and Reo offers detailed and precise internal communication of components by modeling their architectures and protocols.

In this paper, we introduce a new incremental design methodology approach "SysReo", that combines SysML, OCL, and Reo for modeling CPS in the design step. We first define SysReo Meta-models allowing, on one hand, to define the extended SysML Block Definition Diagram (ExtBDD) to specify CPS hierarchical structure, on the other hand, to define Reo Internal Block Diagram (Reo IBD) by combining SysML and Reo notations to model interconnection between CPS components and their exogenous protocols. ExtBDD and Reo IBD allow to model CPS architecture and to capture their complex protocols. Then, we define OCL constraints on SysReo models to ensure the consistency of SysReo models between successive levels of modeling, in the context of our incremental modeling approach. The main purpose of our incremental model-based design methodology is to tackle CPSs design challenges and to express their interactions explicitly.

The paper is structured as follows. Section 2 describes the proposed modeling approach of SysReo models. Section 3 concludes the paper and presents the future works.

2 Modeling approach based on SysML/OCL/Reo

In this section, we present our model-based design methodology. First, we show the steps to follow to design incrementally CPS with SysML, OCL, and Reo. Then we define the meta-models that were exploited in the first step.

Approach steps: We present our modeling approach as shown in Figure 1. First, the modeler starts by specifying a requirement diagram (RD) to analyze and organize CPS requirements. The second phase is dedicated to specify CPS architecture, where the modeler defines the system's components as blocks using ExtBDD (discussed below). Then, in the third phase, the modeler uses an incremental approach starting from an abstract specification of the global internal system, described by Reo IBD. The latter is built gradually by selecting the appropriate components in such a way that their composition respects the constraints defined in the abstract specification. The fourth phase is dedicated to link requirement diagram to ExtBDD and Reo IBD. Finally, the modeler uses our OCL rules on ExtBDD and Reo IBD to precise and to detail the abstract diagrams, and to ensure the consistency of SysReo models.

SysReo meta-models: Figure 2 represents the meta-models of SysReo.The latter are described below.

ExtBDD: Figure 2A represents the meta-model of ExtBDD that is used to model the system's overall architecture. It is based on the block definition diagram, which provides an abstract representation of components and their connections. A block contains proxy ports and internal operations. Each proxy port is linked to an input/output interface block that contains provided (input) or required (output) services of a component. Operations that are defined in a block can be translated into functions which represent the behavioral aspect of the system.

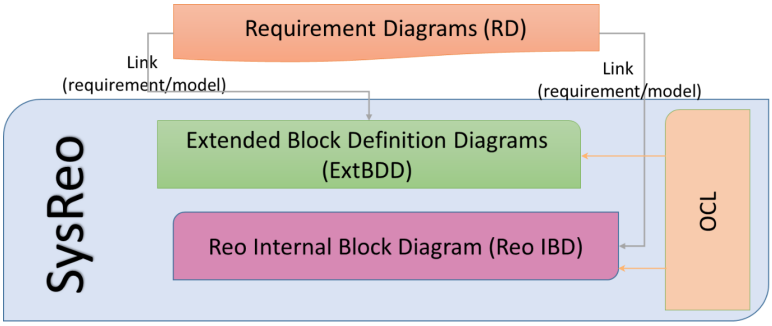


Figure 1: Modeling approach overview.

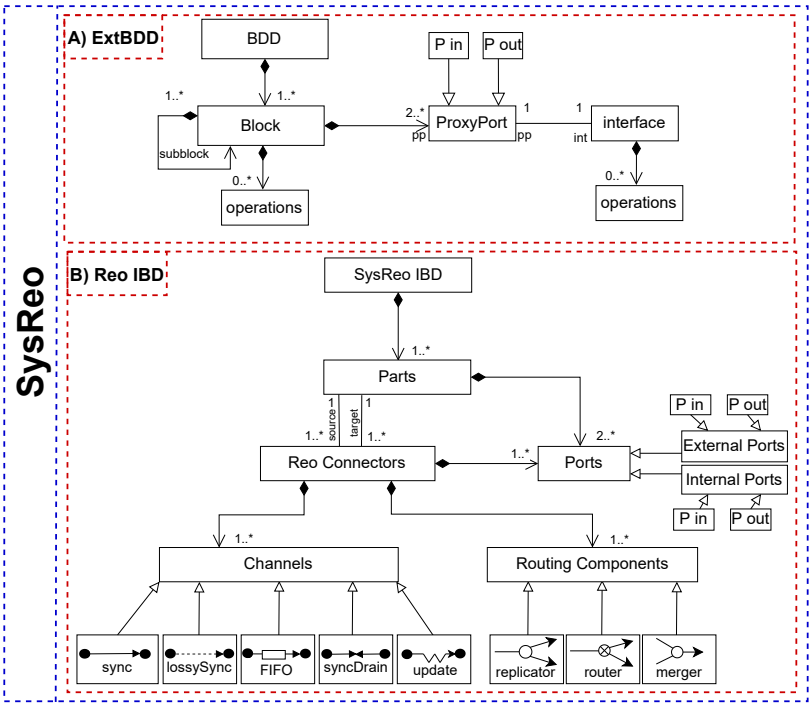


Figure 2: Architecture modeling approach of SysReo.

Reo IBD: Figure 2B captures the meta-model of Reo IBD that is based on ExtBDD to assemble the parts that compose the main block. It aims to model the second part of CPS architecture: connection between parts, specified as exogenous Reo protocols. Reo IBD is composed of Parts and Ports, where each part models a CPS component. The parts are ”black-box” components composed of proxy ports. Proxy ports are used to exchange data between internal parts. The parts communicate with each other using one or many Reo connectors. These Reo connectors can be basic channels (sync, FIFO, syncDrain...) or routing components (replicator, merger, router...). Table 1 represents some constraints on ExtBDD and Reo IBD. For example, the first OCL in Table 1 is used to show that all the operations in a block must exist in the set of operations of its sub-blocks. The second one is

used to precise that a Part (component) has at least one offered and one required services in the system (input/output), therefore the number of its ports should be at least ”2”.

Table 1: OCL Well-formedness constraints of SysReo.

SysReo	Description	OCL
ExtBDD	All operations of a Block must exist in the set of operations of its subblocks	`context Block inv: self.pp.int.operations-> forAll (p:operations implies p in self. subblock.pp.int.operations)`
Reo IBD	Each Part must have at least two Ports (input , output)	`context Part inv : Part.Ports -> size()`$\geq$`2`

3 Conclusion and future works

In this paper, we first introduce a novel approach, ”SysReo”, which is a composition of SysML, OCL, and Reo to enable faithful modeling of a CPS. Then, we used OCL to precise constraints on SysReo models. Therefore, from the Reo level, the modeler can easily construct (possible) large scalable CPS. With SysML, the modeler can describe, on a single model, different aspects of a CPS and analyze the behavior, structure, and requirements of the components. As future works, we plan to: (1) Consider more SysML diagrams to enrich our methodology (parametric diagrams, sequence diagrams...). (2) Define formal semantics of SysReo models to verify formally OCL constraints and to verify safety properties on CPS. (3) Illustrate our work on the case study of the Communication-Based Train Control (CBTC) system to prove the feasibility of our approach.

References

[1] Farhad Arbab. Reo: a channel-based coordination model for component composition. *Mathematical Structures in Computer Science*, 14(3):329–366, 2004.

[2] Ping Huang, Kaiqiang Jiang, Chunlin Guan, and Dehui Du. Towards modeling cyber-physical systems with sysml/marte/pccsl. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 264–269. IEEE, 2018.

[3] Esther Palomar, Xiaohong Chen, Zhiming Liu, Sabita Maharjan, and Jonathan Bowen. Component-based modelling for scalable smart city systems interoperability: A case study on integrating energy demand response systems. *Sensors*, 16(11):1810, 2016.

[4] Jian Xie, Wenan Tan, Zhibin Yang, Shuming Li, Linquan Xing, and Zhiqiu Huang. Sysml-based compositional verification and safety analysis for safety-critical cyber-physical systems. *Connection Science*, pages 1–31, 2021.

Spécification semi-formelle et formelle d'une application de téléréhabilitation : retour d'expérience

Farid Arfi[1], Anne-Lise Courbis[2], Thomas Lambolais[2], François Bughin[3], Maurice Hayot[3]

[1]Univ. Montpellier, farid.arfi@umontpellier.fr
[2]Euromov DHM, Univ. Montpellier, IMT mines Ales, prenom.nom@mines-ales.fr
[3]PhyMedExp, Univ. Montpellier, INSERM, CNRS, CHRU de Montpellier, Montpellier, France,
f-bughin@chu-montpellier.fr, m-hayot@chu-montpellier.fr

Résumé

Nous nous intéressons aux étapes de recueil des besoins et de spécification d'une application de téléréhabilitation de patients atteints de maladies chroniques respiratoires. Après avoir réalisé des entretiens avec les experts impliqués dans la définition des processus de réhabilitation, un cahier des charges a été établi sous forme textuelle et sous forme de modèles UML. Nous avons constaté que la modélisation UML est un réel apport pour la maîtrise d'ouvrage mais elle doit être accompagnée d'une étape de modélisation formelle et de vérification pour analyser les processus et vérifier des propriétés. Ici nous nous concentrons sur les propriétés de vivacité et de sûreté. Nous avons choisi UPPAAL comme outil de simulation et de vérification formelle. Nous montrons sur cette étude de cas quels sont les apports de chacune des étapes de modélisation informelle et formelle pour toutes les parties prenantes du projet.

Mots-clés : vérification formelle, UML, automates temporisés, UPPAAL, téléréhabilitation.

1 Introduction

Le projet m-Rehab [1] concerne la téléréhabilitation de patients atteints de maladies chroniques respiratoires telles que la bronchopneumopathie chronique obstructive (BPCO) et le syndrome d'apnées-hypopnées obstructives du sommeil (SAHOS) [23, 7]. La réhabilitation qualifie le processus qui vise à amener un patient à l'état de santé, physique et psychique, le plus proche possible de celui qui était le sien avant de tomber malade. La téléréhabilitation est un domaine de la télésanté, qui utilise les technologies de l'information et de la communication pour fournir des services de réhabilitation clinique à distance [13].

Le projet m-Rehab vise à réaliser un essai clinique sur des patients atteints de maladies chroniques respiratoires dont la réhabilitation est assurée par une équipe interdisciplinaire grâce à une application mobile, nommée également m-Rehab pour *mobile-Rehabilitation*. Cette application est multi-utilisateur. Sont concernés les patients et les *Personnels de Santé et de Soin* (PSS). Nous nous intéressons ici à la phase d'expression des besoins et de spécification à laquelle nous avons été associés et qui a conduit à définir des documents de référence pour le partenaire en charge du développement.

Nous avons conduit les interviews et la transcription des besoins ont été réalisés, jouant ainsi le rôle d'assistance à la maîtrise d'ouvrage (AMOA), afin de constituer un cahier des charges formé de représentations textuelles informelles, mais également de représentations en UML. Les modèles UML ont été établis sous le logiciel Modelio [15]. Parallèlement au développement qui s'est déroulé sur deux années, nous avons réalisé des analyses formelles de processus, dont la criticité était jugée importante, afin de mettre en évidence des *anomalies* de spécification et de les corriger durant le développement de l'application. Cette étape d'analyse formelle, riche d'enseignements, est ci-après présentée sur un extrait de l'application : la gestion d'une alerte déclenchée par le patient.

Au paragraphe 2, nous présentons une synthèse de la phase d'analyse des besoins du projet m-Rehab ainsi qu'un extrait du modèle UML. Nous y détaillerons le processus d'alerte ainsi que sa modélisation et sa vérification formelle. Les principaux travaux de modélisation et de vérification formelle de processus médicaux sont cités au paragraphe 3. Nous ferons un bilan de ce retour d'expérience au paragraphe 4 en identifiant les apports de chacune des étapes pour chacune des parties prenantes du projet.

1. Ce projet est financé par les fonds FEDER Occitanie. Les partenaires sont : PhyMedExp (Univ. Montpellier, Fr), MRM (Univ. Montpellier, Fr), Euromov-DHM (Univ Montpellier, IMT Mines Ales, Fr), et le groupe KORIAN (Fr).

2 Étude de cas extraite du projet m-Rehab

2.1 Présentation de l'application m-Rehab

Le cahier des charges de m-Rehab a été établi par une succession d'entretiens menés avec les experts du domaine. Comme nous l'avons mentionné, la réhabilitation est un processus interdisciplinaire. En plus des processus administratifs (inscription par exemple) et de gestion des données du patient, trois processus sont proposés par m-Rehab, appelés : *parcours Santé*, *parcours Nutrition* et *parcours APA* (Activité Physique Adaptée).

Les experts en réhabilitation sont de formations diverses (médecins pneumologues, nutritionnistes, spécialistes en APA) et ne partagent pas toujours le même vocabulaire. De plus, ils ne connaissent pas précisément les activités de réhabilitation des autres spécialités. Il est donc nécessaire qu'à l'issue du recueil des besoins, on puisse avoir à la fois une vue synthétique et détaillée de ce qui est proposé aux patients. Durant sa téléréhabilitation, le patient sera équipé d'objets connectés (balance, tensiomètre et montre) et suivra des parcours personnalisés à travers l'application m-Rehab. Il aura des contacts en présentiel ou en visio avec les PSS formant son équipe de soin. Il aura un contact privilégié avec une personne, appelée *Care Manager*, qui assure le lien Patient/PSS, par téléphone, visio-conférence ou mail, et qui est en charge de l'orienter s'il a besoin d'information ou si certains marqueurs ne progressent pas. Le métier de *Care Manager* est nouveau et il est co-construit par les experts : les fonctions attendues sont définies par chacun des PSS qui spécifie ce que le *Care Manager* peut apporter à son parcours.

Chaque expert va ainsi définir pour sa spécialité le parcours des patients. Tout parcours s'appuie sur des contenus pédagogiques et des activités qui donneront lieu systématiquement à une évaluation et un feedback. Les processus ont été définis dans un premier temps sous forme textuelle en retranscrivant les entretiens et un diagramme de classes a été élaboré et partagé avec tous les experts de façon à mettre en évidence les concepts. Ce diagramme de classes a été un support de discussion, et a servi également à établir un dictionnaire des termes de façon à ce que toutes les parties prenantes partagent le même vocabulaire sans ambiguïté. On compte environ 300 classes organisées en package représentant les processus. La figure 1 montre une partie du diagramme de classes permettant de mettre en évidence les concepts présentés dans cette introduction.

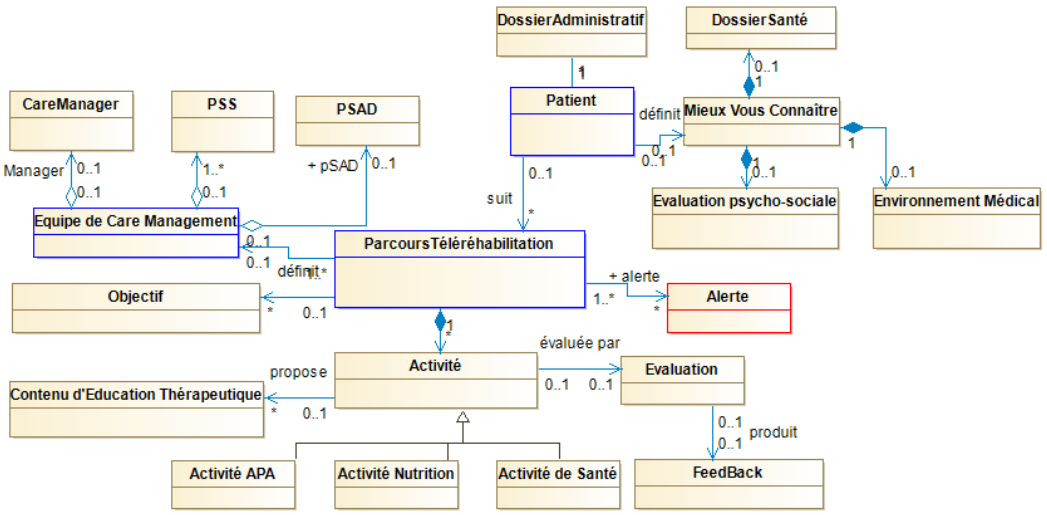


FIGURE 1 – Modèle du domaine : diagramme de classes de m-Rehab (extrait)

Nous allons nous intéresser au processus d'*Alerte* qui apparaît dans le diagramme de classes comme étant associé au Parcours du Patient. Nous en donnons sa spécification textuelle et sa modélisation UML. Nous étudierons ensuite sa modélisation formelle pour effectuer une vérification.

2.2 Extrait du cahier des charges : création et gestion d'une Alerte du Parcours Santé

Nous nous focalisons sur un extrait du cahier des charges dans lequel un médecin a décrit le processus de traitement de l'alerte pouvant être lancée par un patient présentant le syndrome d'apnée du sommeil (SAHOS).

> « *Le patient peut envoyer une alerte s'il rencontre des problèmes avec son dispositif de Pression Positive Continue. Le* Care Manager *et le PSAD (Prestataire de Santé à Domicile) sont alors avertis. Le PSAD intervient généralement sous trois jours. Au-delà de ces jours, on demande alors au Patient si son état s'est amélioré. Deux réponses sont possibles : problème résolu ou état non stabilisé. Si le problème est résolu, l'alerte est clôturée. Dans le cas contraire, on questionne le Patient sur son état tous les deux jours pendant 10 jours maximum tant que le problème n'est pas résolu. Quand le problème est résolu, l'alerte est clôturée. Sinon, le Care Manager et le médecin sont avertis : le médecin doit intervenir dans un intervalle d'un jour et l'alerte est alors clôturée. Si le PSAD ne passe pas chez le patient après trois jours du lancement d'alerte, une notification est envoyée au Care Manager et au médecin qui doit intervenir comme dans le cas décrit précédemment.* »

Ce processus est représenté dans un premier temps par un diagramme de séquences. Un extrait en est donné en figure 2 correspondant au cas où le PSAD intervient. Deux situations sont mises en évidence : (i) le problème est résolu après le passage du PSAD ou dans un délai de 10 jours (blocs en vert) et (ii) l'état du patient n'est pas stabilisé (bloc en rouge) : une notification est alors envoyée au *Care Manager* et au médecin afin qu'ils contactent le patient. L'intérêt de ce diagramme est d'avoir une représentation graphique comme support de discussion avec le médecin qui a émis le besoin et pouvoir modifier facilement le processus.

Nous allons formaliser ce processus pour l'analyser et nous allons voir qu'il reste quelques anomalies ne permettant pas de l'implanter sous cette forme.

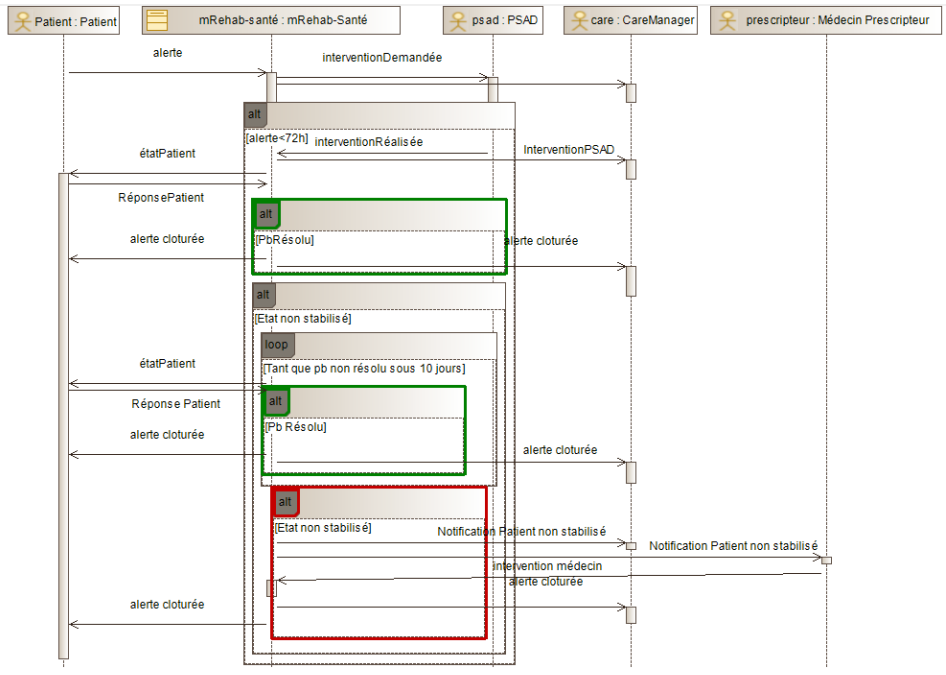


FIGURE 2 – Diagramme de séquences de la gestion d'une Alerte (extrait)

2.3 Modélisation et vérification d'une alerte du parcours Santé

Les spécifications textuelles souffrent généralement de plusieurs types d'anomalies : (1) incomplétudes : informations insuffisantes ou manquantes ; (2) ambiguïtés sur les termes utilisés ou la description des parcours ; (3) incohérences : des résultats différents sont obtenus avec les mêmes données des patients selon

l'interprétation que l'on fait du processus. C'est une anomalie de type incomplétude qui sera traitée dans ce paragraphe.

Bien que la modélisation UML permette de relever un certain nombre d'anomalies lors des échanges avec les parties prenantes, d'autres peuvent persister étant donnée sa nature semi-formelle. Nous avons recours ici à des étapes de modélisation formelle et de vérification pour analyser et corriger la spécification. Nous avons choisi le modèle d'automates temporisés et son outil de vérification UPPAAL [5] pour les raisons suivantes : (1) les méthodes de vérification des propriétés de vivacité et de sûreté basées sur les automates temporisés ont été bien étudiées et développées dans UPPAAL ; (2) le cahier des charges du projet m-Rehab fait référence à des événements temporisés et périodiques : les automates temporisés sont appropriés pour modéliser ce type d'événements.

2.3.1 Modélisation du processus de traitement d'alertes par des automates temporisés

Le processus de traitement d'alertes énoncé précédemment au paragraphe 2.2 est spécifié sous forme d'automates temporisés (cf. figure 3). Nous retrouvons dans cette figure les cinq processus intervenant dans le diagramme de séquences de la figure 2. Les déclarations des horloges, des variables et des canaux de synchronisation des processus sont définies dans la figure 4.

On y voit que lorsque le Patient déclenche une alerte (Alerte ! dans le processus (b)) celle-ci est synchronisée avec le processus Santé (cf. Alerte ? dans le processus (a)) qui va de façon immédiate, grâce à l'état engagé AlerteRecue, se synchroniser par diffusion avec le Care et le PSAD : cf. PbPatient ! en sortie de l'état AlerteRecue dans (a) et PbPatient ? en sortie (i) de l'état Off dans (c) et (ii) de l'état On dans (e). Le PSAD peut répondre à la demande sous trois jours : cf. $x < 3$ invariant de l'état ResoudrePb dans processus (c), et sa transition sortante synchronisée sur InterventionPSAD ! avec le processus (a). Le processus (a) passe alors dans l'état AttenteReponse1 en attendant la réponse du patient sur son état. Le processus (b) peut fournir la réponse au processus (a) lors de la synchronisation sur le canal EtatPB. Cette réponse est choisie de façon aléatoire par le processus (b), où 0 (resp. 1) représente l'état non stabilisé (resp. résolu). Par la suite, le processus (a) analyse la réponse du patient grâce à la variable globale Etat servant de moyen de communication entre les processus (a) et (b) lors de la synchronisation sur EtatPB .

2.3.2 Spécification et vérification d'exigences

Plusieurs exigences sont énoncées dans le cahier des charges. Nous en avons extrait six. Elles ont émergé suite à des discussions avec le médecin, afin de nous assurer de la bonne compréhension des processus. La première propriété est indépendante du domaine : c'est une propriété attendue de tout processus réactif continu. Ceci correspond aux descriptions informelles ci-dessous. Nous donnons pour chaque propriété sa formalisation sous l'outil UPPAAL, qui utilise un sous-ensemble de la logique TCTL (*Timed Computation Tree Logic*) [2] :

P_1 : *Absence de blocage*
$A\Box$ not deadlock.

P_2 : *Le médecin ne recevra pas de notification d'alerte si celle-ci est en cours de traitement par le PSAD.*
$A\Box$ PSAD.ResoudreProbleme imply !Medecin.RecevoirNotification.

P_3 : *Toute alerte déclenchée finira toujours par être clôturée.*
Patient.AlerteOn $\longrightarrow$!Patient.AlerteOn.

P_4 : *Après avoir lancé une alerte, le patient aura de façon sûre une intervention d'un professionnel de santé (PSAD ou médecin) dans un délai de quatre jours maximum.*
Sante.AlerteRecue $\longrightarrow$ Intervention && $x \leqslant 4$.

P_5 : *Il existe des interventions qui permettent de résoudre le problème.*
$E\Diamond$ Intervention && Etat == Resolu.

P_6 : *Malgré l'intervention le problème peut ne pas être résolu.*
$E\Diamond$ Intervention && !Etat == Resolu.

Notons que P_5 et P_6 sont des propriétés du domaine, ou des connaissances métier, et non des exigences sur le système.

2. Pour des formules d'état ϕ et ψ décrites en logique propositionnelle, les notations $A\Box\,\phi$, $E\Diamond\,\phi$ et $\phi \longrightarrow \psi$ désignent respectivement le fait que ϕ soit toujours vraie, qu'il existe une branche dans laquelle ϕ deviendra vraie, et qu'en toute circonstance, ϕ conduira à ψ. Voir par exemple [5] pour une présentation plus complète.

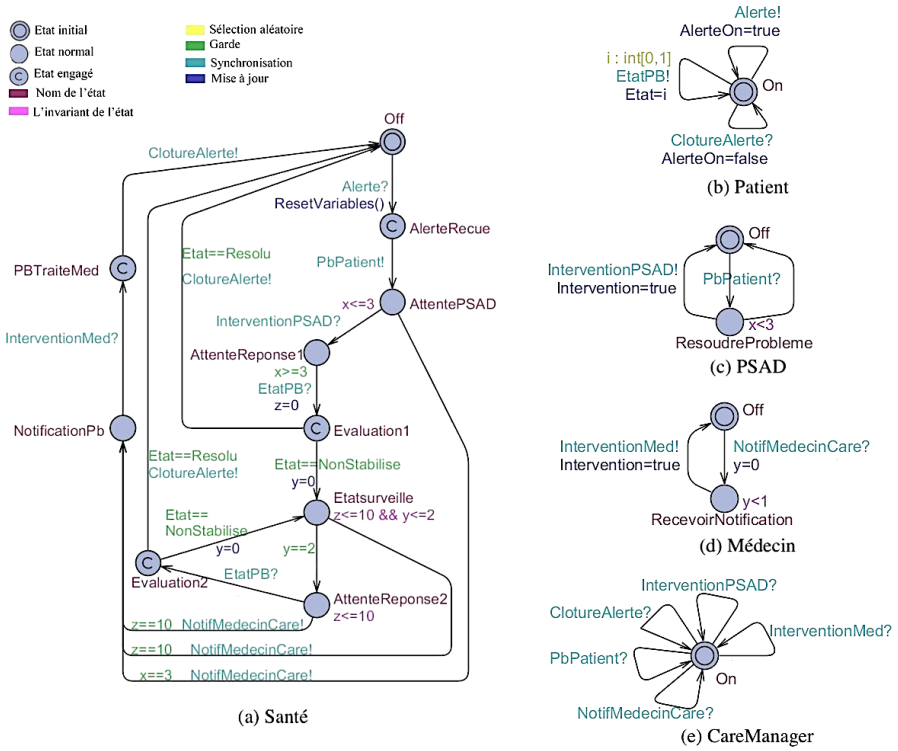


FIGURE 3 – Modélisation par automates temporisés de la gestion d'alerte.

Après vérification, toutes les propriétés sont satisfaites, à l'exception de la propriété P_3. En voici l'explication : après intervention du PSAD, le patient étant libre de répondre à la question concernant son état de santé à partir du troisième jour du lancement d'alerte, il peut ne pas répondre à cette question ; le processus (a) reste alors en attente de la synchronisation avec le processus (b) sur le canal EtatPB dans l'état AttenteReponse1. Cette erreur nous a conduits à interroger les médecins de façon à compléter la spécification : ils proposent qu'un délai de deux jours soit laissé au patient pour répondre sur son état de santé. Cette précision engendre une modification de la garde de la transition de synchronisation sur le canal EtatPB en sortie de l'état AttenteReponse1 qui devient : $x \geqslant 3$ && $x < 5$. Le même raisonnement est fait sur l'état AttenteReponse2 : la garde de sa transition de sortie synchronisée sur le canal EtatPB devient : $y < 4$.

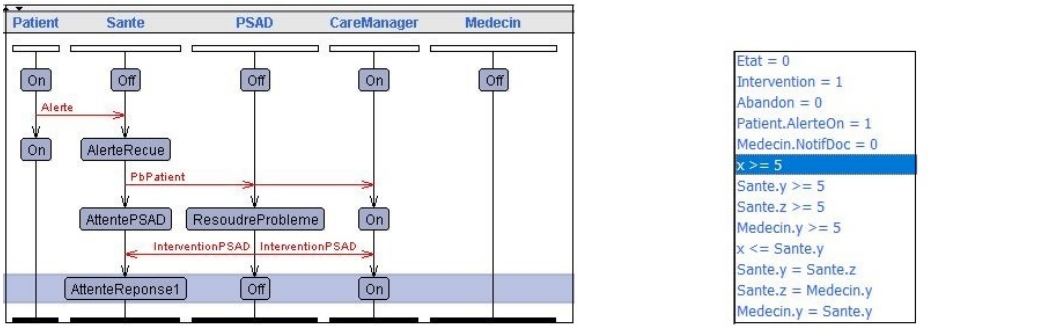
```
// global declarations:
broadcast chan InterventionMed,ClotureAlerte,PbPatient,NotifMedecinCare,InterventionPSAD;
chan Alerte,EtatPB;
clock x;
int Etat;
const int Resolu=1,NonStabilise=0;
bool Intervention;
void ResetVariables() { Intervention=false;x=0;Abandon=false; return;}

clock y,z;    // local declarations: Sante
bool AlerteOn // local declaration: Patient
clock y;     // local declaration: Medecin
```

FIGURE 4 – Déclaration des canaux de synchronisation, des variables et des horloges.

Avec cette modification, la propriété P_1 n'est plus satisfaite : il y a un blocage provenant de la synchronisation avec le patient sur le canal EtatPB. En effet, comme le patient n'est pas obligé de se synchroniser exactement pendant l'intervalle préconisé, s'il rate cette synchronisation, il ne peut plus fournir la réponse attendue par le système. La figure 5 montre un contre-exemple généré par UPPAAL où P_1 n'est pas satisfaite, illustré par un diagramme de séquences conduisant au blocage de l'état AttenteReponse1 du processus Santé (figure 5(a)), et la valeur des variables et des horloges correspondantes (figure 5(b)).

Cette erreur nous a conduits encore une fois à interroger les médecins et compléter le cahier des charges. Ils demandent alors que, dans le cas où le patient ne répond pas dans l'échéance proposée, une notification



(a) Diagramme de séquence de blocage

(b) Valuation des horloges et des variables de blocage

FIGURE 5 – Un contre-exemple montrant que la propriété P_1 n'est pas satisfaite.

soit envoyée au *Care Manager* pour l'informer de la situation. Le *Care Manager* prend alors l'initiative de contacter le patient (non modélisé ici) et l'alerte sera clôturée.

Cette précision nous a permis finalement de corriger le modèle formel et de vérifier l'ensemble des propriétés souhaitées. La figure 6 montre les nouvelles versions des processus Santé et CareManager de la figure 3 permettant de satisfaire toutes les propriétés énoncées. Si le patient ne se synchronise pas au niveau de l'état AttenteReponse1 pendant l'échéance attendue ($x \geqslant 3$ && $x < 5$), il est nécessaire de provoquer une sortie immédiate de l'état. Pour cela, un invariant est ajouté sur l'état AttenteReponse1 : $x \leqslant 5$ et une transition de synchronisation est ajoutée sur le canal NotifCareAband vers l'état PasReponsePatient avec pour garde $x == 5$. Un raisonnement similaire est appliqué au niveau de l'état AttenteReponse2 pour être conforme à la nouvelle demande.

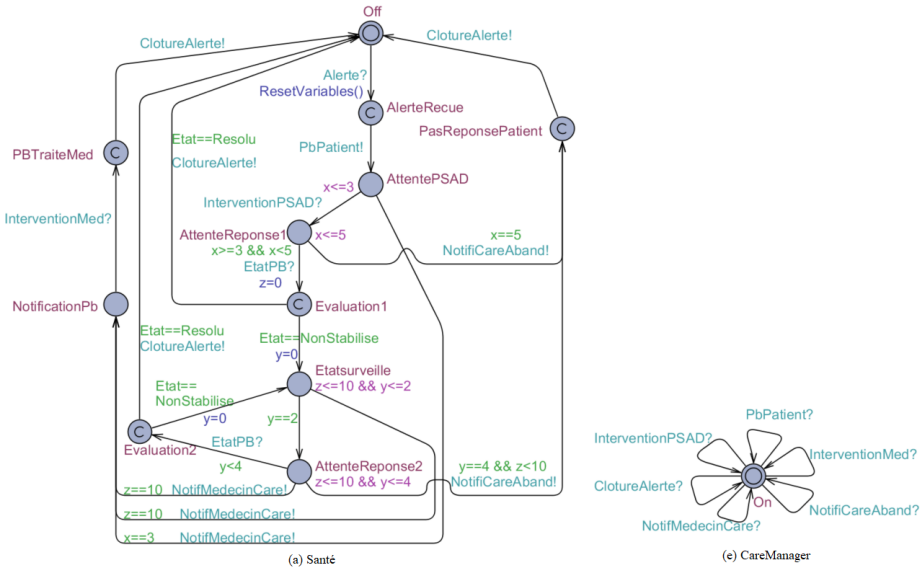


FIGURE 6 – Correction des automates (a) et (e) de la figure 3.

3 Travaux du domaine

Nous présentons ici des travaux concernant les approches de modélisation de processus médicaux puis les approches permettant de faire une vérification formelle des processus soit à partir de modèles formels, soit par transformation de modèles.

Il existe des approches de modélisation spécifiques à l'expression de *protocoles médicaux* telles que Asbru [20] et GLARE [21]. L'état de l'art de [22] et des travaux postérieurs montrent que UML a été largement utilisé pour modéliser des parcours cliniques. On notera l'existence d'un projet de développement

d'un profile UML pour le domaine médical appelé *Healthcare System Specifications* et le *HL7 Development Framework* visant à définir un standard pour l'interopérabilité des modèles de santé [9].

L'utilisation des ces approches contribue à clarifier et faciliter la validation des processus par les experts du domaine médical ou les concepteurs. Néanmoins, elles ne permettent de garantir que les processus modélisés ont le comportement désiré, et une vérification formelle est nécessaire. La revue systématique de [6] qui a été réalisée en 2018 montre que les techniques de vérification formelle sont de plus en plus utilisées dans le domaine médical que ce soit pour la vérification, la validation ou encore la génération de code. Nous avons complété cette bibliographie et nous constatons que la plupart des méthodes formelles sont appliquées sur des dispositifs médicaux critiques plutôt que sur des processus médicaux. Par exemple, nombreuses sont les références qui utilisent les automates temporisés et l'outil UPPAAL (ou la technique de *model-cheking*) pour la vérification de dispositifs. Nous citons deux des dispositifs les plus étudiés dans la littérature : le stimulateur cardiaque (Pacemaker) [11, 12] et la pompe à perfusion (Infusion pump) [3, 10]. Concernant les processus médicaux, certains auteurs s'intéressent à l'utilisation des méthodes formelles pour assurer le bon fonctionnement des interactions dans un système médical [14, 18], d'autres par exemple portent leurs travaux sur la confidentialité et la sécurité des données de santé [1, 2].

Quelques travaux portent sur l'utilisation de méthodes formelles pour aider au développement d'applications de santé [4, 19] : les auteurs montrent l'intérêt de l'utilisation du formalisme Z et VDM-SL pour détecter des anomalies de spécification et réduire le coût du processus de développement.

Pour combler l'écart entre les approches de modélisation semi-formelles et les méthodes formelles, des travaux appliqués au domaine médical proposent des transformation de modèles. Citons la transformation des modèles Asbru vers des modèles algébriques dédiés au prouveur de théorèmes KIV [20], ou, dans le cas d'UML, les transformations des machines d'états [17], des diagrammes de classes [16] et diagrammes d'activités [8] vers les automates temporisés de l'outil UPPAAL.

4 Bilan : retour d'expérience

L'application m-Rehab concerne des utilisateurs que sont les patients et experts variés dont aucun n'a la connaissance détaillée de tous les processus, et pour lesquels les services offerts sont différents.

Étant donné le nombre important d'exigences à prendre en compte et la complexité des parcours de réhabilitation proposés aux patients, il était nécessaire d'avoir en premier lieu une approche de modélisation informelle. La représentation des besoins de m-Rehab sous forme textuelle accompagnée de modèles UML a permis de représenter et manipuler tous les concepts de façon unifiée. Le modèle rassemble le lexique (une centaine de termes), les types de données (environ 120), les concepts (300 classes) et leurs relations, la typologie des parcours (Santé, Nutrition, APA et parcours administratif) et pour certains, leur description sous forme de diagrammes de séquences ou de machines à états UML.

Il apparaît que l'utilisation de langages du domaine de l'informatique n'est pas une barrière pour communiquer avec la maîtrise d'ouvrage. L'approche de modélisation UML rend tangible une expertise d'un domaine afin de l'affiner et le corriger par une collaboration entre concepteurs et experts. Le cahier des charges final est ainsi plus clair et précis qu'un document purement textuel. Globalement, UML a permis d'atteindre les objectifs suivants : établir un lexique du domaine et partager les mêmes concepts avec un minimum d'ambiguïté ; avoir un support de discussion pour affiner les besoins des PSS, chacun pouvant voir son processus à haut niveau et les éléments impliqués dans ce processus ; donner aux PSS une vision à la fois détaillée et transverse de la téléréhabilitation ; offrir une co-construction des fonctions d'un rôle nouveau : la vue synthétique des fonctions du *care manager* a permis aux PSS de co-construire ces fonctions. De plus, l'utilisation d'UML permet d'adopter une organisation modulaire pour ensuite mettre en place une approche incrémentale et itérative de développement. Il a ainsi été possible de travailler sur une partie des spécifications sans nécessairement connaître l'ensemble. Ce processus réduit le coût et la complexité de la modélisation.

De façon plus surprenante, la modélisation formelle n'est pas non plus une barrière stricte : les animations et les contre-exemples d'UPPAAL ont rendu compréhensibles des concepts difficiles à appréhender par des non experts de la modélisation par automates temporisés, avec des modèles visuels jugés clairs et précis, ainsi qu'en identifiant des sources d'anomalies qui étaient restées cachées.

En revanche, nous notons que, même informellement, il a été difficile de faire énoncer des propriétés aux experts : même si ces propriétés correspondent à des exigences, les experts ne raisonnent pas en ces termes mais plutôt en décrivant leur savoir-faire. Seule l'AMOA peut les reformuler ainsi.

Cette expérience a mis en évidence la nécessité d'allier modélisation formelle et informelle pour définir des processus validés par les experts du domaine et répondant à des propriétés de sûreté ou de vivacité, et ceci y compris pour des applications de santé jugées *a priori* non critiques.

Remerciements *Les auteurs remercient le comité scientifique de m-Rehab et ses membres : Bronia Ayoub, Julie Boiché, Anne-Sophie Cases, Blandine Chapel, Gérard Dray, Nelly Heraud, Nathalie Jean, Pierre Jean, Christophe Latrille, Jordan Michel, Pascal Pomies, Roxana Ologeanu-Taddei.*

Références

[1] Rima Addas and Ning Zhang. Formal security analysis and performance evaluation of the linkable anonymous access protocol. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 8407 LNCS :500–510, 2014.

[2] Flora Amato and Francesco Moscato. A model driven approach to data privacy verification in e-health systems. *Transactions on Data Privacy*, 8 :273–296, 2015.

[3] David Arney, Raoul Jetley, Paul Jones, Insup Lee, and Oleg Sokolsky. Formal methods based development of a pca infusion pump reference model : Generic infusion pump (gip) project. In *2007 Joint Workshop on High Confidence Medical Devices, Software, and Systems and Medical Device Plug-and-Play Interoperability (HCMDSS-MDPnP 2007)*, pages 23–33. IEEE, 2007.

[4] Muhammad Waqar Azeem, Muhammad Ahsan, Nasir Mehmood Minhas, and Khadija Noreen. Specification of e-health system using z : A motivation to formal methods. In *International Conference for Convergence for Technology-2014*, pages 1–6. IEEE, 2014.

[5] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on uppaal 4.0. *Department of computer science, Aalborg university*, 2006.

[6] Silvia Bonfanti, Angelo Gargantini, and Atif Mashkoor. A systematic literature review of the use of formal methods in medical software systems. *Journal of Software : Evolution and Process*, 30(5), 2018.

[7] Narelle S Cox, Simone Dal Corso, Henrik Hansen, Christine F McDonald, Catherine J Hill, Paolo Zanaboni, Jennifer A Alison, Paul O'Halloran, Heather Macdonald, and Anne E Holland. Telerehabilitation for chronic respiratory disease. *Cochrane Database of Systematic Reviews*, (1), 2021.

[8] Zamira Daw, Rance Cleaveland, and Marcus Vetter. Formal verification of software-based medical devices considering medical guidelines. *International Journal of Computer Assisted Radiology and Surgery*, 9 :145–153, 2014.

[9] HL7-FHIR. `https://www.hl7.org/fhir/`, avril 2022.

[10] Raoul Jetley, S Purushothaman Iyer, Paul L Jones, and William Spees. A formal approach to pre-market review for medical device software. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 169–177. IEEE, 2006.

[11] Zhihao Jiang, Miroslav Pajic, Rajeev Alur, and Rahul Mangharam. Closed-loop verification of medical devices with model abstraction and refinement. *International Journal on Software Tools for Technology Transfer*, 16(2) :191–213, 2014.

[12] Zhihao Jiang, Miroslav Pajic, and Rahul Mangharam. Cyber–physical modeling of implantable cardiac medical devices. *Proceeding of IEEE Special Issue on Cyber-Physical Systems*, 100(1) :122–137, 2011.

[13] Dahlia Kairy, Pascale Lehoux, Claude Vincent, and Martha Visintin. A systematic review of clinical outcomes, clinical process, healthcare utilization and costs associated with telerehabilitation. *Disability and rehabilitation*, 31(6) :427–447, 2009.

[14] Marrium Khalid, Hamra Afzaal, Shoaib Hassan, Nazir Ahmad Zafar, Saba Latif, and Aniqa Rehman. Automated uml-based formal model of e-health system. In *2019 13th International Conference on Mathematics, Actuarial Science, Computer Science and Statistics (MACS)*, pages 1–6. IEEE, 2019.

[15] Modelio. `https://www.modelio.org/`, avril 2022.

[16] Mehdi Nobakht and Dragos Truscan. An approach for validation, verification, and model-based testing of {UML}-based real-time systems. *ICSEA 2013, The Eighth International Conference on Software Engineering Advances*, pages 79–85, 2013.

[17] Kasi Periyasamy, Yiwei Zou, and Sachin Padhye. A framework for verification of uml statechart diagrams. *29th International Conference on Computers and Their Applications, CATA 2014*, pages 167–174, 2014.

[18] Usman Pervez, Osman Hasan, Khalid Latif, Sofiene Tahar, Amjad Gawanmeh, and Mohamed Salah Hamdi. Formal reliability analysis of a typical fhir standard based e-health system using prism. In *2014 IEEE 16th International Conference on e-Health Networking, Applications and Services (Healthcom)*, pages 43–48. IEEE, 2014.

[19] Hafiz Muhammad Tahir, Muhammad Nadeem, and Nazir Ahmad Zafar. Specifying electronic health system with vienna development method specification language. In *2015 National Software Engineering Conference (NSEC)*, pages 61–66. IEEE, 2015.

[20] Annette Ten Teije, Mar Marcos, Michel Balser, Joyce van Croonenborg, Christoph Duelli, Frank van Harmelen, Peter Lucas, Silvia Miksch, Wolfgang Reif, Kitty Rosenbrand, et al. Improving medical protocols by formal methods. *Artificial intelligence in medicine*, 36(3) :193–209, 2006.

[21] Paolo Terenziani, Stefania Montani, Alessio Bottrighi, Mauro Torchio, Gianpaolo Molino, and Gianluca Correndo. The glare approach to clinical guidelines : Main features. *Studies in health technology and informatics*, 101 :162–6, 2004.

[22] Christos Vasilakis, Dorota Lecnzarowicz, and Chooi Lee. Application of unified modelling language (uml) to the modelling of health care systems : An introduction and literature survey. *Developments in Healthcare Information Systems and Technologies : Models and Methods*, pages 275–287, 2011.

[23] Danielle SR Vieira, Francois Maltais, and Jean Bourbeau. Home-based pulmonary rehabilitation in chronic obstructive pulmonary disease patients. *Current opinion in pulmonary medicine*, 16(2) :134–143, 2010.

Etude comparative des méthodes pour la vérification des systèmes cyber-physiques basés machine learning

Arthur Clavière[1], Laura Altieri-Sambartolomé[1], Eric Asselin[1], Christophe Garion[2] et Claire Pagetti[3]

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

Cet article est un résumé étendu de l'article "Verification of machine learning based cyber-physical systems: a comparative study", accepté à HSCC 2022 (25th ACM International Conference on Hybrid Systems: Computation and Control).

Contexte Malgré un fort potentiel, les algorithmes issus d'apprentissage automatique *e.g.,* les réseaux de neurones, demeurent inutilisés dans les systèmes embarqués critiques. Le principal frein à cette utilisation réside dans la difficulté à certifier ces algorithmes. Cette difficulté est essentiellement liée au problème de la spécification du comportement attendu pour ces algorithmes. En effet, si une telle spécification pouvait être obtenue facilement, l'apprentissage ne présenterait plus aucun intérêt : la spécification serait suffisante et l'apprentissage ne serait plus nécessaire. Comment alors vérifier un réseau de neurones vis-à-vis d'une spécification qui n'existe pas, ou tout du moins n'est pas complète ?

Récemment, plusieurs travaux ont proposé une approche *système* permettant de contourner le problème de la spécification des réseaux de neurones en formulant le problème de vérification non pas au niveau du réseau de neurones mais au niveau du système qui l'utilise. Cet article s'intéresse à cette approche, en considérant une classe de système particulière : un système cyber-physique (CPS) où le contrôleur est un *classificateur* basé sur plusieurs réseaux de neurones. Ce type de système combine une partie physique en temps continu avec un contrôleur en temps discret, qui produit une commande parmi un ensemble fini à chaque exécution. Afin de choisir parmi cet ensemble fini de commandes, le contrôleur dispose d'une collection de réseaux de neurones. Un seul de ces réseaux est exécuté à chaque fois : le choix du réseau à exécuter est fonction de la commande précédente. L'étude de ce type de système est pertinente puisque plusieurs cas d'usage décrits dans la littérature entrent dans ce cadre, notamment dans le domaine aéronautique [5]. Le fait de choisir parmi plusieurs réseaux en fonction de l'état du contrôleur (*i.e.,* la commande précédente) permet d'avoir des réseaux plus petits et qui s'exécutent plus rapidement, chose importante dans un contexte embarqué.

Différents travaux se sont intéressés à la vérification d'un tel système. Dans cet article, nous proposons une revue détaillée de ces différentes méthodes, via les contributions suivantes :

Modélisation Un modèle est proposé pour le système d'intérêt, s'appuyant sur le formalisme des automates hybrides. Ce modèle prend en compte des hypothèses réalistes, notamment l'exécution non instantanée du contrôleur et des réseaux de neurones associés. Le problème de vérification est formulé comme un problème d'atteignabilité : montrer que l'ensemble des états atteignables du système est disjoint de l'ensemble des états d'erreurs.

Description des méthodes de vérification En s'appuyant sur le modèle proposé, les méthodes de vérification existantes sont présentées et les hypothèses sous-jacentes sont mises en évidence. Parmi ces méthodes, une approche par programmation linéaire en nombres entiers à été proposée et implémentée dans l'outil VENMAS [1]. Cette approche présente l'avantage d'offrir une représentation exacte du problème mais s'applique uniquement au cas où la dynamique du système est linéaire. Une autre approche, disponible dans l'outil NNV [6], propose de calculer une sur-approximation des états atteignables par le système en s'appuyant sur plusieurs domaines abstraits, à savoir les zonotopes et les *star sets*. Parallèlement au développement de cette dernière méthode, une approche similaire a été développée, implémentée dans l'outil SAMBA [3]. Celui-ci construit une sur-approximation qui s'avère être moins précise (mais potentiellement suffisante) et propose aussi plusieurs heuristiques pour accélérer la résolution du problème de vérification. Cette approche s'appuie d'une part sur des techniques de simulation garantie [2] pour évaluer la dynamique continue du système, et d'autre part des techniques d'interprétation abstraite dédiées à l'analyse de réseaux de neurones [7, 8, 4].

Etude expérimentale Afin de comparer expérimentalement les méthodes susmentionnées, nous avons considéré un ensemble de cas d'étude incluant deux systèmes d'anti-collision aéronautiques (VCAS et ACAS Xu), correspondant à des systèmes réalistes et représentatifs des potentielles futures utilisations des réseaux de neurones dans le domaine de l'aviation, ainsi qu'un pendule inversé placé sur un chariot en mouvement (Cartpole). Ces trois cas d'étude présentent différentes caractéristiques, pouvant influencer la performance de la vérification : différents types de dynamique, un nombre plus ou moins élevé de réseaux de neurones, eux-mêmes de taille plus ou moins grande. Pour chacun de ces cas d'étude, nous avons considéré un ensemble de problèmes de vérification, de difficulté plus ou moins élevée, afin d'évaluer le compromis entre *précision* et *passage à l'échelle* offert par chaque méthode de vérification. Ces problèmes de vérification ont été construit en considérant (1) un état initial pour le système, (2) une incertitude sur cet état et (3) un horizon temporel donné pour l'évaluation de la propriété d'intérêt. La difficulté du

problème de vérification est déterminée par (a) sa *criticité* : l'état initial choisi peut conduire, ou non, le système dans un état indésirable sans action du contrôleur, (b) sa *nature* : l'incertitude considérée peut concernée l'ensemble des variables d'états ou uniquement certaines d'entre-elles, et (c) son *horizon temporel*, plus ou moins grand.

Un extrait des résultats est donné en figure 1, où, pour chaque cas d'étude, on donne le pourcentage de problèmes qui ont pu être vérifiés par l'outil, ainsi que le temps cumulé nécessaire à la résolution de ces problèmes. Le meilleur outil est donc situé en haut à gauche sur ces diagrammes. Ces résultats mettent en évidence la moins bonne performance de VENMAS qui, du fait de sa plus grande précision, passe mal à l'échelle sur les problèmes difficiles du VCAS (grands horizons temporels), en plus de n'être applicable ni sur le ACAS Xu ni sur le Cartpole. Par ailleurs, les performances relatives de NNV et SAMBA dépendent du cas d'étude considéré : par exemple, sur le VCAS, NNV offre une approximation plus précise, précision déterminante pour éviter une explosion combinatoire du nombre de chemins explorés, d'où sa meilleure performance.

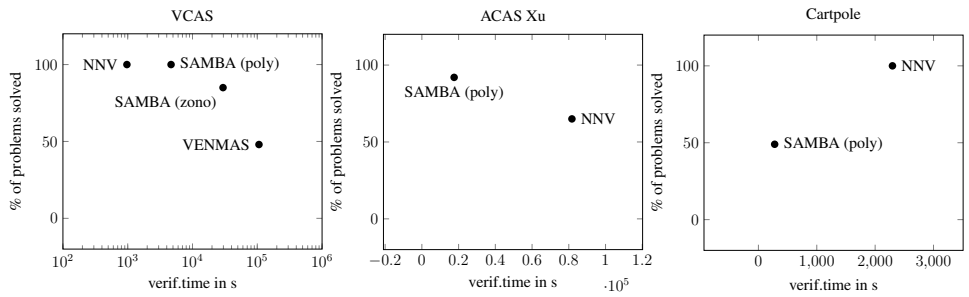


Figure 1: Comparaison des outils de vérification sur 3 cas d'étude: VCAS, ACAS Xu et Cartpole.

Enseignements tirés Le premier enseignement concerne l'applicabilité des méthodes formelles pour la vérification de CPSs basés machine learning : 97.3% des 225 problèmes de vérification considérés ont pu être résolus par l'un des outils. Un deuxième enseignement porte sur le coût de l'approche système étudiée dans cette article qui, si elle permet de s'affranchir de la problématique liée la spécification des réseaux de neurones, s'avère aussi beaucoup plus couteuse que l'analyse d'un réseau de neurones *isolé* (avec NNV, 95% du temps de résolution est consacré à l'analyse de la dynamique tandis que l'analyse des réseaux de neurones représente les 5% restants). Parmi les enseignements tirés, sont aussi discutées les possibles heuristiques utilisées dans SAMBA pour pallier sa moins bonne précision par rapport à NNV, ainsi qu'une heuristique pour le choix de la méthode la plus adaptée pour un problème de vérification donné.

References

[1] Michael E. Akintunde, Elena Botoeva, Panagiotis Kouvaros, and Alessio Lomuscio. Verifying strategic abilities of neural-symbolic multi-agent systems. In *Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR'20)*, pages 22–32, 2020.

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

[3] Arthur Clavière, Eric Asselin, Christophe Garion, and Claire Pagetti. Safety verification of neural network controlled systems. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pages 47–54, 2021.

[4] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI ^{2}: Safety and robustness certification of neural networks with abstract interpretation. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 3–18, 2018.

[5] Kyle D. Julian, Mykel J. Kochenderfer, and Michael P. Owen. Deep neural network compression for aircraft collision avoidance systems. *Journal of Guidance, Control, and Dynamics*, 42(3):598–608, 2019.

[6] Diego Manzanas Lopez, Taylor Johnson, Hoang-Dung Tran, Stanley Bak, Xin Chen, and Kerianne L. Hobbs. Verification of neural network compression of acas xu lookup tables with star set reachability. In *AIAA Scitech 2021 Forum*.

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

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

Étude de propriétés d'opacité temporisée à l'aide de vérification temporisée paramétrée*

Dylan Marinho
Université de Lorraine, CNRS, Inria, LORIA, F-54000 Nancy

Résumé

Une fuite d'informations peut avoir des conséquences dramatiques sur la sécurité d'un système. Parmi ces failles, une fuite d'informations temporelle se produit lorsqu'un attaquant peut déduire des informations internes confidentielles en basant son attaque sur une observation temporisée du système. Nous modélisons ici le système par un automate temporisé, et considérons qu'un attaquant a accès (uniquement) au temps d'exécution du système. Nous abordons deux problèmes d'opacité temporisée : déterminer les temps d'exécution pour lesquels le système est sécurisé (c'est-à-dire pour lesquels l'attaquant ne peut pas déduire d'information confidentielle) et déterminer si un système est sécurisé pour tous ses temps d'exécution possibles.

1 Introduction

Étant donné un ensemble de chemins qui révèlent un secret, l'opacité caractérise le fait que s'il existe une exécution du système qui révèle le secret, il existe une autre exécution, avec la même observation, qui ne révèle pas ce secret. Notre étude porte sur une forme d'opacité dans laquelle l'observation se fait uniquement sur le temps nécessaire pour atteindre une localité finale.

2 Problèmes étudiés

Dans ce travail, un système est modélisé par un automate temporisé paramétré (*Parametric Timed Automaton (PTA)*) [AHV93]. Un PTA $\mathcal{A}$ est un automate fini, composé d'un ensemble de localités et d'un ensemble d'actions, que l'on étend avec : *i*) un ensemble d'horloges $\mathbb{X} = \{x, y, \ldots\}$, qui sont des variables réelles augmentant linéairement ; *ii*) un ensemble de paramètres $\mathbb{P} = \{p_1, p_2, \ldots\}$. On définit ensuite : *i*) des invariants sur les localités (propriétés à vérifier, sous la forme d'une

*Ce travail est tiré d'un article récemment accepté à ACM TOSEM [And+22], en collaboration avec Étienne André, Didier Lime et Jun Sun. Ce travail est soutenu par le projet ANR-NRF ProMiS (ANR-19-CE25-0015).

(a) Un PTA $\mathcal{A}$ (b) Un TA $v(\mathcal{A})$, où $v(p) = 3$

FIGURE 1 – Un exemple de PTA et de TA

contrainte linéaire, pour rester dans une localité) ; *ii*) des gardes sur les transitions (propriétés à vérifier pour activer une transition) ; *iii*) la réinitialisation d'horloges lors d'une transition. Un exemple de PTA est donné par la figure 1a.

Si l'on considère un PTA $\mathcal{A}$ et une fonction d'évaluation $v : \mathbb{P} \to \mathbb{Q}$, on définit l'automate temporisé $v(\mathcal{A})$ (*Timed automaton – TA*) [AD94] comme étant un PTA dans lequel tous les paramètres sont évalués en fonction de v. Un exemple de TA est donné par la figure 1b.

Un PTA est L/U [Hun+02] si l'ensemble de ses paramètres est partitionné en deux sous-ensembles : celui des paramètres toujours comparés en borne inférieure dans les gardes et invariants, et celui des paramètres toujours comparés en borne supérieure. Le PTA figure 1a est L/U : p est toujours comparé en borne supérieure.

2.1 Opacité temporisée

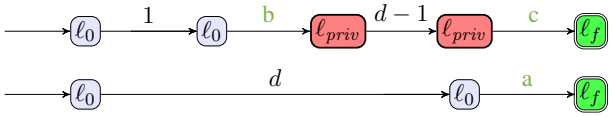
On considère que l'attaquant a accès seulement au temps total d'exécution. Le secret est formalisé par la visite, ou non, d'une localité privée ℓ_{priv} sur le chemin conduisant à la localité finale ℓ_f. On cherche donc à définir qu'un système est sécurisé si l'attaquant ne peut pas déterminer si la localité privée a été visitée.

Définition 1 (Opacité temporisée)**.** Un système est *opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f* pour un temps d s'il existe deux chemins de ℓ_0 à ℓ_f de temps d *i*) l'un passant par ℓ_{priv} et *ii*) l'autre ne passant pas par ℓ_{priv}.

Le système est dit *complètement opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f* s'il est opaque pour tous ses temps d'exécution possibles.

Exemple 1. On considère le TA donné en figure 1b.

— Il existe deux exécutions de durée d pour tout $d \in [2, 3]$:



Le système est donc *opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f* pour toute durée dans $[2, 3]$;

— Mais il est possible d'atteindre ℓ_f avec une exécution de durée $1,5$ ne passant pas par ℓ_{priv}, sans que cela ne soit possible en passant par ℓ_{priv} : le système n'est donc pas complètement opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f.

2.2 Opacité sur les automates temporisés

Un TA étant défini uniquement avec des horloges et des constantes, on s'intéresse au fait *i*) de pouvoir déterminer quels sont les temps d'exécution sécurisés (TOCP) et *ii*) de pouvoir déterminer si le modèle est sécurisé pour tous ses temps d'exécution (FTODP). Les deux problèmes suivants considèrent en entrée un TA $v(\mathcal{A})$, une localité privée ℓ_{priv} et une localité finale ℓ_f.

Calcul d'opacité temporisée (***Timed Opacity Computation Problem – TOCP***)**:**
Calculer les temps d'exécution D pour lesquels $v(\mathcal{A})$ est opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f.

Problème de décidabilité de l'opacité temporisée complète (***Full Timed Opacity Decision Problem – FTODP***)**:**
$v(\mathcal{A})$ est-il complètement opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f ?

2.3 Opacité sur les automates temporisés paramétrés

On définit également des problèmes analogues sur les PTA, dans lesquels l'objectif est de déterminer s'il existe des fonctions d'évaluations qui permettent de vérifier la propriété. Ces problèmes considèrent en entrée un PTA $\mathcal{A}$, une localité privée ℓ_{priv} et une localité finale ℓ_f.

Vide de l'opacité (***Timed Opacity Emptiness Problem – TOEP***)**:**
L'ensemble des fonctions d'évaluations v telles que $v(\mathcal{A})$ est opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f pour un ensemble non vide de temps d'exécution est-il vide ?

Vide de l'opacité complète (***Full Timed Opacity Emptiness Problem – FTOEP***)**:**
L'ensemble des fonctions d'évaluations v telles que $v(\mathcal{A})$ est complètement opaque vis-à-vis de ℓ_{priv} sur le chemin vers ℓ_f est-il vide ?

On définit également de façon analogue les problèmes de synthèse de paramètres, où l'on cherche à synthétiser toutes les fonctions d'évaluations v pour lesquelles les propriétés susmentionnées sont vérifiées.

3 Résultats théoriques

Nous avons étudié la décidabilité (ou la calculabilité, suivant le problème étudié) de chacun des problèmes présentés dans la section 2. Les différents résultats sont présentés dans le tableau 1.

Nous observons alors que les problèmes définis sur des TA restent décidables, mais deviennent presque tous indécidables dès lors que nous considérons des paramètres. Le seul cas d'étude qui permet de conserver la décidabilité est l'opacité temporisée appliquée aux L/U-PTA. Cependant, nous avons montré que la synthèse est infaisable en pratique. Plus précisément, nous montrons qu'il est impossible de représenter l'ensemble des solutions à l'aide d'un formalisme pour lequel le vide de

TABLEAU 1 – Récapitulatif des résultats théoriques obtenus

Problème		Opacité temporisée	Opacité temporisée complète
Calculabilité (TOCP) Décidabilité (FTODP)	TA	calculable [**And+22**, Prop. 5.2]	décidable [**And+22**, Prop. 5.3]
Vide ((F)TOEP)	PTA	indécidable [**And+22**, Thm. 6.1]	indécidable [**And+22**, Thm. 7.2]
	L/U-PTA	décidable [**And+22**, Thm. 6.2]	indécidable [**And+22**, Thm. 7.4]
Synthèse	L/U-PTA	infaisable [**And+22**, Prop. 6.4]	*de facto infaisable*

l'intersection est décidable : cela exclut donc l'utilisation des unions de polyèdres, un formalisme pourtant souvent utilisé pour la synthèse de paramètres.

4 Expérimentations

Nous avons défini une procédure permettant de déterminer les fonctions d'évaluations qui rendent un PTA opaque. En raison de l'indécidabilité du problème, cette procédure n'a pas de garantie de terminaison. Nous avons ensuite utilisé IMITATOR [And21] (un logiciel de vérification temporisée paramétrée prenant en entrée une extension des PTA) afin de donner une preuve de concept de notre méthode.

Notre approche a été appliquée à des modèles de la littérature [AMP21] ainsi qu'à des programmes Java manuellement traduits en PTA [STAC].

Références

[AD94] Rajeev ALUR et David L. DILL. "A theory of timed automata". In : *Theoretical Computer Science* 126.2 (avr. 1994), p. 183-235. ISSN : 0304-3975. DOI : 10.1016/0304-3975(94)90010-8.

[AHV93] Rajeev ALUR, Thomas A. HENZINGER et Moshe Y. VARDI. "Parametric real-time reasoning". In : *STOC*. San Diego, California, United States : ACM, 1993, p. 592-601. DOI : 10.1145/167088.167242.

[AMP21] Étienne ANDRÉ, Dylan MARINHO et Jaco van de POL. "A Benchmarks Library for Extended Parametric Timed Automata". In : *TAP 2021*. 2021. DOI : 10.1007/978-3-030-79379-1_3.

[And+22] Étienne ANDRÉ, Didier LIME, Dylan MARINHO et Jun SUN. "Guaranteeing Timed Opacity using Parametric Timed Model Checking". In : *TOSEM* (2022).

[And21] Étienne ANDRÉ. "IMITATOR 3 : Synthesis of Timing Parameters Beyond Decidability". In : *CAV 2021*. 2021. DOI : 10.1007/978-3-030-81685-8_26.

[Hun+02] Thomas HUNE, Judi ROMIJN, Mariëlle STOELINGA et Frits W. VAANDRAGER. "Linear parametric model checking of timed automata". In : *JLAP* 52-53 (2002). DOI : 10.1016/S1567-8326(02)00037-1.

[STAC] STAC. URL : https://github.com/Apogee-Research/STAC/.

Vérification formelle d'une carte à puce pour une certification Critères Communs*

Adel Djoudi[1], Martin Hána[2], and Nikolai Kosmatov[3]

[1]Thales Digital Identity & Security, Meudon, France
[2]Thales Digital Identity & Security, Prague, Czech Republic
[3]Thales Research & Technology, Palaiseau, France,
firstname.lastname@thalesgroup.com

1 Contexte et motivation

La sûreté et la sécurité des logiciels critiques sont devenues aujourd'hui des préoccupations majeures. La vérification formelle de programme peut fournir une preuve rigoureuse de l'absence d'erreurs et de vulnérabilités de sécurité. Elle peut assurer des garanties de correction très fortes qui sont particulièrement importantes pour les produits critiques, tels que les cartes à puce. Cependant, la vérification formelle des logiciels industriels reste un défi.

La sécurité d'une carte à puce repose sur un ensemble de propriétés d'isolation qui doivent être assurées par la machine virtuelle de la carte (*JavaCard virtual machine*, ou JCVM). Selon la politique de sécurité [2], une application ne doit pas lire ou écrire les données ou exécuter le code d'une autre application sans son autorisation. Les règles d'accès correspondantes sont en général assurées par un mécanisme de contrôle d'accès dédié, appelé pare-feu (*firewall*).

Ce résumé étendu présente une vérification formelle récente [1] d'une implémentation de machine virtuelle de carte à puce réalisée par Thales à l'aide de la plateforme de vérification Frama-C [3]. Elle a été réalisée en vue d'une certification Critères Communs de niveau EAL6 (pour laquelle le certificat a été délivré en 2021). Les propriétés visées incluent les propriétés de sécurité habituelles, telles que l'intégrité et la confidentialité, qui doivent être assurées par le mécanisme de contrôle d'accès de la JCVM. Le code C de la JCVM a été annoté en utilisant le langage de spécification formelle ACSL [4].

*Cette soumission est un résumé étendu d'un article [1], publié à FM 2021.

2 Approche et contributions

Notre approche de vérification formelle de code C avec l'outil Frama-C et son greffon de vérification déductive WP [3] s'intègre dans un contexte industriel avec des exigences de sécurité très élevées.

Le code vérifié se distingue par une taille relativement importante (plus de 7.000 lignes de code C) avec des opérations bas niveau manipulant des champs de bits et des conversions de pointeurs hétérogènes. Plus de 35.000 lignes d'annotations ACSL ont été introduites directement dans le code source. La spécification en ACSL se distingue à son tour par une taille importante (5 fois la taille du code analysé) et l'expression des propriétés à prouver directement au niveau du code source. En effet, les propriétés considérées, la confidentialité et l'intégrité, sont des propriétés de haut niveau, et leur vérification déductive directement sur le code source nécessite de trouver une approche pour les prouver sous forme d'invariants globaux dans tout le code considéré. Leur spécification et vérification dans ce travail s'appuie sur MetAcsl [5], un autre greffon de Frama-C.

Dans notre publication [1], nous présentons notre approche de vérification et illustrons à travers un exemple la structure de la spécification implémentée avec un sous-ensemble de propriétés. La méthodologie que nous avons mise en place afin de satisfaire les exigences de certification Critères Communs avec des exemples d'exigences et d'annotations ACSL est présentée dans une autre publication [6].

Plus de 50.000 objectifs de preuve ont été générés par Frama-C/WP à partir du code C et des annotations ACSL associées. Plusieurs défis de passage à l'échelle ont été rencontrés et surmontés lors de ce projet afin de pouvoir prouver que le code de la JCVM respecte bien la spécification. L'utilisation de code fantôme (ou code *ghost*) a été déterminante pour trouver le bon compromis entre, d'une part, la complexité et les opérations bas niveau du code et, d'autre part, les capacités des prouveurs automatiques. Le choix précis des lemmes a permis d'augmenter le nombre d'objectifs prouvés automatiquement. Ainsi près de 99% des objectifs de preuves ont été prouvés automatiquement. Néanmoins, la création manuelle de scripts de preuve dans WP (utilisant des tactiques de preuve pour indiquer quelques premières étapes de preuve à effectuer) a été nécessaire afin de prouver le reste des objectifs.

Bien que la plupart des solutions adoptées dans notre projet ne soient pas nouvelles, leur combinaison précise a été essentielle pour mener à bout la preuve complète des propriétés exigées. Lors de ce projet, nous avons également proposé quelques extensions et améliorations de Frama-C dont certaines ont été essentielles pour le succès de ce projet. Nous présentons dans [1] les résultats de preuve détaillés, quelques leçons apprises et les améliorations souhaitées de l'outil de preuve.

3 Conclusion et travaux futurs

Ce résumé étendu présente une étude de cas de vérification formelle entièrement réalisée dans un contexte industriel en vue d'une certification. Ce travail contribue à collecter les bonnes pratiques en spécification et vérification déductive. Il établit un nouvel état de l'art des applications de la vérification déductive sur un code critique de grande taille pour la preuve de propriétés de sécurité. Les leçons tirées de ce projet ouvrent la porte à d'autres améliorations de la méthodologie et des outils de vérification déductive. Comme travaux futurs, nous prévoyons d'introduire la vérification déductive dans un processus d'intégration continue de développement logiciel.

Références

[1] A. DJOUDI, M. HÁNA et N. KOSMATOV. « Formal verification of a JavaCard virtual machine with Frama-C ». In : *the 24th Int. Symp. on Formal Methods (FM 2021)*. T. 13047. Springer, 2021, p. 427-444. DOI : 10.1007/978-3-030-90870-6_23. URL : https://nikolai-kosmatov.eu/publications/djoudi_hk_fm_2021.pdf.

[2] ORACLE. *Java Card System – Open Configuration Protection Profile, Version 3.1*. Rapp. tech. Oracle, 2020.

[3] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI. « Frama-C : A software analysis perspective ». In : *Formal Asp. Comput.* (2015), p. 1-37. DOI : 10.1007/s00165-014-0326-7.

[4] P. BAUDIN, P. CUOQ, J.-C. FILLIÂTRE, C. MARCHÉ, B. MONATE, Y. MOY et V. PREVOSTO. *ACSL : ANSI/ISO C Specification Language*. 2021. URL : https://www.frama-c.com/download/acsl.pdf.

[5] V. ROBLES, N. KOSMATOV, V. PREVOSTO, L. RILLING et P. L. GALL. « MetAcsl : Specification and Verification of High-Level Properties ». In : *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)*. T. 11427. LNCS. Springer, 2019, p. 358-364. DOI : 10.1007/978-3-030-17462-0_22.

[6] A. DJOUDI, M. HÁNA, N. KOSMATOV, M. KŘÍŽENECKÝ, F. OHAYON, P. MOUY, A. FONTAINE et D. FÉLIOT. « A Bottom-Up Formal Verification Approach for Common Criteria Certification : Application to JavaCard Virtual Machine ». In : *Proc. of the 11th European Congress on Embedded Real-Time Systems (ERTS 2022)*. Juin 2022. URL : https://nikolai-kosmatov.eu/publications/djoudi_hkkomff_erts_2022.pdf.

Illustration de spécifications temporisées paramétrées sur des signaux continus*

Étienne André[1] ✉, Masaki Waga[2], Natuski Urabe[3] et Ichiro Hasuo[3,4]

Résumé

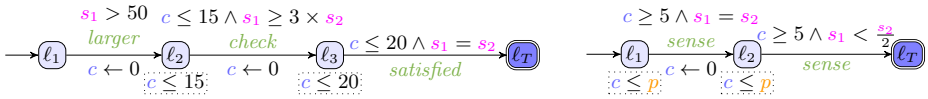
La spécification de propriétés peut s'avérer délicate. Nous proposons ici une approche automatisée pour l'illustration de propriétés données sous forme d'automates étendus avec des contraintes temporelles et des paramètres temporels, et qui peuvent également spécifier des contraintes sur des signaux à valeurs continues. En d'autres termes, étant donné une telle spécification et un automate bornant le comportement admissible de chacun des signaux, notre approche construit des exécutions continues fournissant ainsi des exemples d'exécutions réelles ou impossibles pour la spécification de départ. Dans notre approche, tant la spécification que les automates bornant les comportements admissibles sont donnés sous forme d'une sous-classe des automates hybrides linéaires, plus précisément des automates temporisés étendus par des vitesses d'horloges arbitraires, des contraintes sur les signaux, et des paramètres temporels ; notre méthode génère alors des exécutions concrètes permettant d'illustrer la spécification.

Ce manuscrit court est tiré de l'article récemment accepté [And+22].

1 Introduction

Le *model-checking* a connu de nombreux succès depuis plusieurs décennies (voir par exemple [Kur18]). Néanmoins, son usage dans l'industrie peut être vu comme encore relativement décevant, notamment par rapport aux garanties pourtant très fortes offertes en terme de correction d'un système. C'est d'autant plus vrai pour le model-checking *quantitatif*, qui considère des systèmes étendus avec des quantités telles que les probabilités, le temps, des coûts, etc. Parmi les explications possibles de la pénétration décevante du model-checking dans l'industrie, l'une des raisons est l'expertise poussée demandée par le model-checking aux personnes l'utilisant. Même les personnes expertes dans le domaine sont susceptibles de commettre des erreurs, ce qui donne lieu à des spécifications avec un comportement différent de celui envisagé. Ces erreurs ne pourront ensuite être résolues qu'après une phase de débogage potentiellement fastidieuse.

*Ce travail a bénéficié du soutien du projet ERATO HASUO Metamathematics for Systems Design (N° JPMJER1603), JST et du projet ANR-NRF ProMiS (ANR-19-CE25-0015).



(a) Spécification non paramétrée (b) Spécification paramétrée

FIGURE 1 – Exemples de PTAS

2 Contribution

Nous proposons ici une approche offrant une aide à la spécification, sous forme d'illustration de l'évolution continue de signaux au cours du temps, vérifiant une spécification donnée. Nous introduisons comme formalisme de spécification les automates temporisés paramétrés à signaux (*parametric timed automata with signals*, ou PTAS), une extension des automates temporisés (paramétrés) [AD94 ; AHV93] : nos PTAS bénéficient de la puissance expressive des automates temporisés, où des horloges peuvent être comparées à des constantes, à laquelle nous ajoutons la possibilité de spécifier des contraintes linéaires sur des signaux, telles que « $s_1 \geq 3 \times s_2$ ». Ceci permet d'exprimer des spécifications telles que « lorsque le signal s_1 dépasse 50 alors, en moins de 15 unités de temps, nous avons $s_1 \geq 3 \times s_2$ et ensuite, en moins de 20 unités de temps, les deux signaux sont égaux ($s_1 = s_2$) ». La figure 1a représente le PTAS codant cette spécification (où c est une horloge, et s_1 et s_2 sont des signaux), c.-à-d. que ℓ_T est accessible si et seulement si cette spécification est vérifiée pour une exécution. En outre, nous autorisons des *paramètres temporels* (constantes inconnues), permettant ainsi des spécifications paramétrées combinant des actions discrètes, des contraintes sur les signaux, et des paramètres temporels, telles que « après une première détection à l'aide d'un capteur (action $sense$) se produisant dans un intervalle $[5, p]$, alors nous avons $s_1 = s_2$, puis après une seconde détection dans un intervalle $[5, p]$, nous avons $s_1 < \frac{s_2}{2}$ », où p est un paramètre temporel. Le PTAS codant cette spécification est donné figure 1b. Dans ce cas, notre illustration prendra la forme d'une valeur concrète de p *et* d'une évolution des signaux satisfaisant la spécification.

Afin de borner les comportements possibles des signaux, nous introduisons comme seconde entrée de notre approche des *automates bornant un signal* (*signal bounding automata*, ou SBA). Ces SBA peuvent être déduits d'une connaissance grossière du système considéré ; ils peuvent être également utilisés pour prédéterminer spécifiquement un type de comportement attendu satisfaisant la spécification (par exemple, l'évolution particulière d'un véhicule en cas de freinage, ou au contraire en cas d'accélération). En outre, grâce à nos SBA, nous évitons de générer des exemples d'évolutions de signaux fantaisistes, par exemple avec des changements de valeur arbitrairement rapides. Nos SBA imposent aux signaux de prendre une dérivée arbitraire (mais constante), qui peut évoluer au gré des changements d'états du SBA. Par exemple un SBA pourrait contraindre un signal s à alterner entre une augmentation lente ($\dot{s} = 1$) ou rapide ($\dot{s} = 3$), ou une décélération lente ($\dot{s} = -1$) ou rapide ($\dot{s} = -3$) ; ce SBA est donné figure 2.

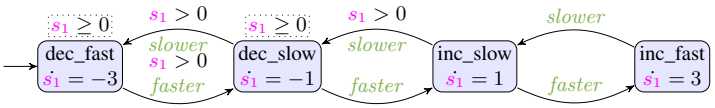


FIGURE 2 – Un exemple de SBA

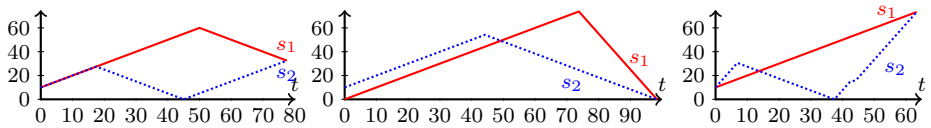


FIGURE 3 – Exemples d'exécutions pour les figures 1a et 2

Nous générons non seulement des exemples d'exécutions positives (correctes), mais aussi négatives (incorrectes, qui ne satisfont *pas* la spécification). En effet, notre paradigme est que, afin d'illustrer une spécification, il peut être nécessaire de produire à la fois des exemples positifs et des exemples négatifs, si possible proches de la « frontière » de la satisfiabilité.

Exemple 1. Soit $\mathcal{A}$ le PTAS figure 1a ; soit $\mathcal{A}_1$ le SBA figure 2, et $\mathcal{A}_2$ le SBA figure 2 où s_1 est remplacé par s_2. Nous supposons que, initialement, $s_1, s_2 \in [0, 10]$. Étant donné le PTAS $\mathcal{A}$ et les deux SBA $\mathcal{A}_1$ et $\mathcal{A}_2$ bornant le comportement de s_1 et s_2, notre approche génère automatiquement plusieurs évolutions possibles des signaux satisfaisant la spécification ; nous donnons trois de ces exécutions en figure 3. Notons que ces trois exécutions représentent trois évolutions très différentes des signaux, avec des valeurs initiales et finales, et des vitesses d'évolution, toutes très différentes.

3 Algorithmes et implémentation

Notre approche résumée en figure 4 se base sur le formalisme des PTAS et des SBA, eux-mêmes définis comme une sous-classe d'un nouveau formalisme dé-

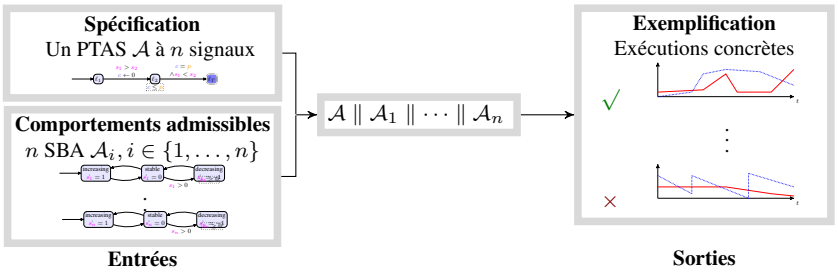


FIGURE 4 – Notre approche

fini comme une sous-classe des automates hybrides rectangulaires [Hen96]. Nous avons défini pour notre nouveau formalisme une syntaxe et une sémantique formelles, et nous avons proposé des algorithmes basés sur une représentation symbolique de l'espace d'états, qui peut être vue comme une extension du graphe des zones tel que défini pour les automates temporisés (paramétrés) [AD94 ; AHV93]. Nos algorithmes proposent d'une part des exécutions concrètes à partir de cette représentation symbolique et, d'autre part (grâce à des heuristiques), des exécutions négatives (impossibles) — sous certaines hypothèses, notamment de déterminisme. Les détails sont disponibles dans [And+22].

Nous avons implémenté nos algorithmes au sein du logiciel IMITATOR [And21] (`v.3.3-alpha` « *Cheese Caramel au beurre salé* »), et avons appliqué notre approche à plusieurs exemples.

Perspectives Les perspectives incluent une extension à des logiques telles que MITL ou STL, mais aussi des garanties de couverture des cas limites.

Références

[AD94] Rajeev ALUR et David L. DILL. "A theory of timed automata". In : *TCS* 126.2 (avr. 1994), p. 183-235. DOI : 10.1016/0304-3975(94)90010-8.

[AHV93] Rajeev ALUR, Thomas A. HENZINGER et Moshe Y. VARDI. "Parametric real-time reasoning". In : *STOC* (16-18 mai 1993). Sous la dir. de S. Rao KOSARAJU, David S. JOHNSON et Alok AGGARWAL. San Diego, California, United States : ACM, 1993, p. 592-601. DOI : 10.1145/167088.167242.

[And+22] Étienne ANDRÉ, Masaki WAGA, Natsuki URABE et Ichiro HASUO. "Exemplifying parametric timed specifications over signals with bounded behavior". In : *NFM* (24-27 mai 2022). Sous la dir. de Klaus HAVELUND, Jyo DESHMUKH et Ivan PEREZ. T. 13260. LNCS. Caltech, Pasadena, CA, USA : Springer, 2022. DOI : 10.1007/978-3-031-06773-0_25.

[And21] Étienne ANDRÉ. "IMITATOR 3 : Synthesis of timing parameters beyond decidability". In : *CAV* (18-23 juil. 2021). Sous la dir. de Rustan LEINO et Alexandra SILVA. T. 12759. LNCS. virtual : Springer, 2021, p. 1-14. DOI : 10.1007/978-3-030-81685-8_26.

[Hen96] Thomas A. HENZINGER. "The Theory of Hybrid Automata". In : *LiCS* (27-30 juil. 1996). Sous la dir. de Moshe Y. VARDI et Edmund M. CLARKE. New Brunswick, New Jersey, USA : IEEE Computer Society, 1996, p. 278-292. DOI : 10.1109/LICS.1996.561342.

[Kur18] Robert P. KURSHAN. "Transfer of Model Checking to Industrial Practice". In : *Handbook of Model Checking*. Sous la dir. d'Edmund M. CLARKE, Thomas A. HENZINGER, Helmut VEITH et Roderick BLOEM. Springer, 2018, p. 763-793. DOI : 10.1007/978-3-319-10575-8_23.