Actes des 18e journées sur les

Approches Formelles dans l'Assistance au Développement de Logiciels

Édités par David Chemouil et Thomas Lambolais

12–14 juin 2019, ENSEEIHT, Toulouse

Préface

L'atelier francophone AFADL sur les Approches Formelles dans l'Assistance au Développement de Logiciels, se tiendra pour sa dix-huitième édition les 12, 13 et 14 juin 2019 à Toulouse. Cette année encore, cet atelier est organisé conjointement avec les journées du GDR GPL. L'atelier AFADL rassemble de nombreux acteurs académiques et industriels francophones intéressés par la mise en œuvre des techniques formelles aux divers stades du développement des logiciels et/ou des systèmes. Il a pour objectif de mettre en valeur les travaux récents effectués autour de thèmes comme la preuve, la vérification, le test, l'analyse de code, etc., permettant de développer des applications sûres, allant de l'ingénierie des exigences du système à sa vie opérationnelle. AFADL reste un lieu privilégié pour les échanges et les discussions. En encourageant un format court, et grâce à des types de soumissions variés, comportant des articles courts, des démonstrations d'outils, des présentations de projets, des résumés longs de travaux déjà publiés et des travaux de doctorants, AFADL permet d'entrevoir l'activité de recherche de la communauté francophone autour des méthodes formelles pour le développement des logiciels. Cette année, 23 travaux ont été retenus pour être présentés, ce qui permet de parcourir un panel riche et varié allant des thématiques centrales d'AFADL à certains groupes de travail du GDR-GPL, qui auront leurs sessions dédiées :

- LTP : Langages, Types et Preuves
- MFDL : Méthodes Formelles dans le Développement de Logiciels
- AFSEC : Approches Formelles pour les Systèmes Embarqués Communicants
- MTV2 : Méthodes de Test pour la Validation et la Vérification

Nous remercions les membres du comité de programme et les coordinateurs des groupes LTP, MFDL/AFSEC et MTV2 pour leur travail qui a contribué à produire un programme dense et de qualité, ainsi que tous les auteurs qui ont soumis un article. Merci aussi aux orateurs invités: François Pottier, Érik Martin-Dorel et Pierre Roux (LTP) et Burkhart Wolff (MTV2). Sans toutes ces personnes, AFADL ne pourrait pas exister. Nous remercions les membres du comité d'organisation des journées du GDR-GPL 2019 qui ont pris en charge tous les aspects logistiques. Nous remercions enfin Easychair pour le service rendu.

10 juin 2019
Toulouse

David Chemouil
Thomas Lambolais

Comité de programme

Yamine Aït Ameur	IRIT/INPT-ENSEEIHT
Sandrine Blazy	University of Rennes 1 - IRISA
Oscar Carrillo	CPE Lyon
David Chemouil	ONERA
Sylvain Conchon	Universite Paris-Sud
Frederic Dadeau	FEMTO-ST
David Deharbe	ClearSy System Engineering
Lydie Du Bousquet	LIG
Catherine Dubois	ENSIIE-Samovar
Ylies Falcone	Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP, LIG, 38000 Grenoble, France
Alain Giorgetti	FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté
Aurélie Hurault	IRIT - ENSEEIHT
Akram Idani	Laboratoire d'Informatique de Grenoble
Nikolai Kosmatov	CEA List
Regine Laleau	Paris Est Creteil University
Thomas Lambolais	IMT Mines Alès, LGI2P
Arnaud Lanoix	Université de Nantes
Yves Ledru	Laboratoire d'Informatique de Grenoble - Université Grenoble Alpes
Nicole Levy	Cedric, CNAM
Delphine Longuet	Univ. Paris-Sud, LRI
Ioannis Parissis	Univ. Grenoble Alpes - Grenoble INP
Pascal Poizat	Université Paris Nanterre and LIP6
Marie-Laure Potet	Laboratoire Vérimag
Marc Pouzet	LIENS
Antoine Rollet	LaBRI, Bordeaux INP, University of Bordeaux, CNRS
Vlad Rusu	INRIA
Nicolas Stouls	CITI/INSA Lyon
Safouan Taha	CentraleSupelec
Laurent Voisin	Systerel
Virginie Wiels	ONERA / DTIM
Fatiha Zaidi	Univ. Paris-Sud

Relecteur additionel

Pierre Roux	ONERA DTIS

Sommaire

Session AFADL-AFSEC-MFDL

Une analyse de la propriété fondamentale de vivacité du protocole Chord 1
Julien Brunel, David Chemouil and Jeanne Tawa

Approches au Développement Formel de Systèmes Hybrides Basées sur la Preuve : Logique Dynamique et Event-B 5
Guillaume Dupont, Yamine Ait Ameur, Marc Pantel and Neeraj Singh

Une approche basée sur la séparation des préoccupations pour modéliser et vérifier les règles de signalisation d'un système ferroviaire 9
Yves Ledru, Akram Idani, Rahma Ben Ayed, Abderrahim Ait Wakrime and Philippe Bon

Session AFADL-LTP

On est tranquilles pour longtemps – les reçus-temps en logique de séparation 13
François Pottier

Primitive Floats in Coq 15
Guillaume Bertholon, Érik Martin-Dorel and Pierre Roux

Formalisation en Coq des erreurs d'arrondi de méthodes de Runge-Kutta pour les systèmes matriciels 19
Florian Faissole

Session AFADL-MTV2

Test d'un robot agricole en simulation 27
Clément Robert, Thierry Sotiropoulos, Helene Waeselynck, Jérémie Guiochet and Simon Vernhes

Tests de politiques d'adaptation pour systèmes cyber-physiques 29
Jean-Philippe Gros

Towards a Test-and-Proof Framework for C11 in Isabelle/HOL 37
Burkhart Wolff

Sessions AFADL

Ingénierie dirigée par les foncteurs 39
Gabriel Radanne, Anil Madhavapeddy, Jeremy Yallop, Thomas Gazagnaire, Richard Mortier, Hannes Mehnert, Mindy Preston and David Scott

Qbricks, un environnement pour la vérification formelle eninformatique quantique 43
Christophe Chareton, Sebastien Bardin, François Bobot, Valentin Perrelle and Benoît Valiron

Formalisation et validation d'une méthode de construction de systèmes de blocs 51
Jessy Colonval and Henri de Boutray

La logique contre les fantômes: comparaison de deux approches pour la preuve d'un module de listes chaînées 59
Allan Blanchard, Nikolai Kosmatov and Frederic Loulergue

Polygraph: un modèle flot de données avec arithmétique de fréquences 63
Paul Dubrulle, Christophe Gaston, Nikolai Kosmatov, Arnault Lapitre and Stéphane Louise

MetAcsl : spécification et vérification de propriétés de haut niveau 67
Virgile Robles, Nikolai Kosmatov, Virgile Prevosto, Louis Rilling and Pascale Le Gall

Dépliage de Boucles Versus Précision Numérique 71
Nasrine Damouche, Xavier Thirioux, Matthieu Martel and Hanane Benmaghnia

Contribution à la formalisation des propriétés graphiques des systèmes interactifs pour la validation automatique 79
Pascal Béger, Valentin Becquet, Sebastien Leriche and Daniel Prun

Formalisation, vérification et évaluation de stratégies d'élasticité dans le Cloud 89
Khaled Khebbeb, Nabil Hameurlain and Faiza Belala

Unification de la Vérification et de l'Exécution Embarquée de Modèles 93
Valentin Besnard

EMI : Un Interpréteur de Modèles Embarqué pour l'Exécution et la Vérification de Modèles UML 101
Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault and Ciprian Teodorov

Spécification incrémentale d'un système d'aide au diagnostic de l'épuisement professionnel : problématique et revue de littérature 105
Marion Kissous, Thomas Lambolais, Anne-Lise Courbis, Gérard Dray and Sophie Martin

Modélisation du Domaine au Sein d'une Méthode Formelle d'Ingénierie des Exigences . . . 117
Tueno Fotso Steve Jeffrey

Intégration d'outils tiers de preuve automatique dans Atelier B 127
Lilian Burdy, David Déharbe and Ronan Saillard

Une analyse de la propriété fondamentale de vivacité du protocole Chord*

Julien Brunel, David Chemouil et Jeanne Tawa
ONERA DTIS & Université fédérale de Toulouse
F-31055 Toulouse
prénom.nom@onera.fr

Résumé

Chord est un protocole qui fournit une table de hachage distribuée évolutive sur un réseau peer-to-peer sous-jacent et qui constitue une cible intéressante pour la spécification et la vérification formelles. Les travaux antérieurs ont principalement porté sur des preuves automatiques des propriétés de *sûreté* ou des preuves manuelles de la correction complète du protocole (une propriété de *vivacité*). Dans cet article, nous rendons compte de l'analyse automatique de la correction de Chord au moyen du langage Electrum (développé dans des travaux antérieurs) sur de petites instances de réseau. En particulier, nous avons pu trouver différents cas problématiques dans des travaux antérieurs et montrer que le protocole n'était pas correct tel que décrit alors. Nous avons résolu tous ces problèmes et fourni une version du protocole pour laquelle nous n'avons pas trouvé de contre-exemple en utilisant notre méthode.

1 Introduction

Les systèmes peer-to-peer sont des systèmes distribués sans organisation hiérarchique ou contrôle centralisé. Chord [LNBK02] est l'un des systèmes peer-to-peer les plus populaires. C'est un protocole et algorithme fournissant une table de hachage distribuée (DHT). Une DHT stocke des paires clé-valeur en attribuant des clés à différents nœuds du réseau. Chord traite de la localistion efficace et robuste de données dans un tel réseau. Quand Chord a été initialement présenté, trois qualités principales ont été soulignées : sa simplicité, sa performance et sa correction. Bien que les deux premières affirmations soient valides, démontrer la correction de

*Résumé de [BCT18]. Travaux partiellement financés par le Fonds Européen de Développment Régional (FEDER) à travers le Programme Opérationnel pour la Compétitivité et l'Internationalisation (COMPETE2020), au travers de la *Fundação para a Ciência e a Tecnologia* (FCT) dans le cadre du projet POCI-01-0145-FEDER-016826 et dans le cadre du projet FORMEDICIS de l'Agence Nationale de la Recherche (ANR-16-CE25-0007).

Chord se révèle être une tâche difficile, comme le montrent les nombreux travaux de P. Zave, dont le plus récent est [Zav17].

Dans Chord, chaque nœud dispose d'un identifiant et peut atteindre d'autres nœuds au moyen de pointeurs sur eux. Les nœuds et leurs pointeurs forment une topologie qui est essentielle pour assurer la localisation correcte des données dans le réseau. Comme des nœuds peuvent rejoindre ou quitter le réseau (ou échouer) à tout moment, la topologie est en constante évolution. Un aspect fondamental du protocole réside dans la définition d'opérations de *maintenance* en charge de la réparation de la topologie du réseau, de sorte à ce que les données stockées dans n'importe quel nœud continuent à être accessibles depuis tout autre noeud, malgré les pannes, les arrivées et les départs.

Ainsi, la correction de Chord concerne la topologie du réseau. En fait, les nœuds et leurs pointeurs "successeur" doivent former un *anneau*, de telle sorte que chaque nœud soit accessible depuis n'importe quel autre nœud. Puisque les nœuds peuvent rejoindre et quitter le réseau, la topologie en anneau ne peut pas toujours être assurée. C'est pourquoi la propriété de correction de Chord est exprimée comme suit : *si, à partir d'un instant donné, il n'y a plus d'arrivée, départ ou échec, le réseau est alors assuré de revenir à terme dans une topologie en anneau et d'y rester.* La correction de Chord ne concerne donc pas seulement la structure du réseau, mais aussi son évolution temporelle : il s'agit en fait d'une propriété de *vivacité*. Cette double nature constitue l'une des raisons de la difficulté à démontrer la correction.

Nous avons utilisé Electrum [1] [MBC+16] pour analyser Chord. Electrum est un langage de spécification basé sur la logique temporelle linéaire du premier ordre, dans lequel les propriétés structurelles et temporelles peuvent facilement être définies et vérifiées. Il est inspiré d'Alloy [Jac12] pour ses aspects structurels et de la logique temporelle linéaire (LTL) pour ses concepts temporels.

2 Le protocole Chord

Dans un réseau Chord, chaque nœud a un identifiant (le *hash* à m bits de son adresse IP). Les couples de clés et des données associées sont stockées dans les nœuds. Chaque nœud maintient une *liste de successeurs*, des pointeurs sur d'autres nœuds. Nous appelons le premier élément de cette liste le *successeur*. L'objectif de disposer d'une liste de successeurs au lieu d'un seul est d'être robuste aux pannes : si un noeud quitte le réseau, son prédécesseur a, normalement, toujours des successeurs dans le réseau. En outre, chaque nœud a également un pointeur sur son *prédécesseur*, nécessaire pour l'exécution des opérations de maintenance.

Quand un réseau est structuré en anneau vis-à-vis de la relation induite par les pointeurs *successeurs* et que l'ordre des identifiants est conforme à l'ordre des pointeurs *successor*, alors chaque nœud est accessible depuis n'importe quel autre nœud, *i.e.* toutes les données sont accessibles depuis n'importe quel noeud. Nous disons qu'un tel réseau est dans un état *idéal*.

1. Cf. https://haslab.github.io/Electrum.

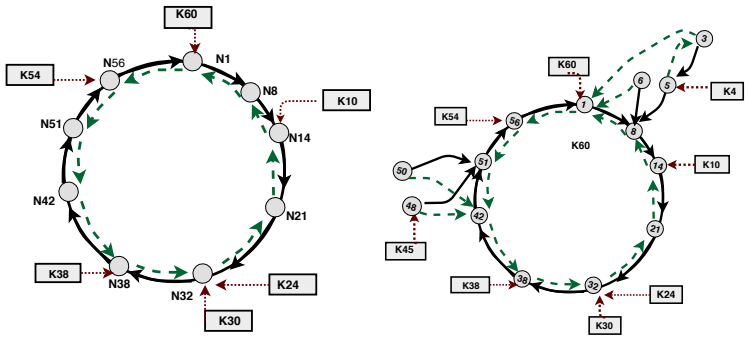


FIGURE 1 – Réseaux idéal et valide (pointeurs “successeur” en gras, pointeurs “prédécesseur” en tireté, et associations clé/données en pointillé).

La structure d’anneau ne peut pas être assurée en permanence. Par exemple, les nœuds rejoignant un anneau créent un *appendice*. Les opérations de maintenance visent alors à revenir à une structure en anneau.

Les auteurs de Chord ont décrit des propriétés du réseau qui assurent la transmission correcte des données [LNBK02]. Ils définissent notamment l’état *idéal*, déjà introduit informellement, et un état imparfait dans une certaine mesure, qu’on dit *valide* [Zav17] (cf. fig. 2).

3 Spécification et vérification

Nous avons spécifié le protocole Chord au moyen du langage Electrum [2]. Notre modèle est inspiré de celui, en Alloy, de P. Zave [Zav17], mais il en diffère grandement sur la forme en raison d’un idiome de modélisation différent. Le modèle traite par ailleurs de Chord avec des nœuds à deux successeurs seulement. De manière plus importante, l’emploi d’Electrum (avec une extension [BCC+18] pour spécifier les événements plus simplement), en particulier de sa couche de logique temporelle, permet de construire un modèle plus abstrait, ne nécessitant pas de spécifier d’invariants pour l’analyse (la recherche d’invariants a été une tâche ardue relatée par P. Zave dans diverses publications). Elle a permis de trouver rapidement plusieurs petites erreurs dans la modélisation de P. Zave et a par ailleurs permis de traiter la propriété de vivacité, tandis que la modélisation en Alloy ne permettait de traiter automatiquement que les propriétés de sûreté.

Nous avons aussi trouvé un problème plus important qui a nécessité l’extension du modèle au moyen d’un nouvel événement (par rapport à celui de P. Zave) mais qui, finalement, correspondait à une proposition des auteurs de Chord dans un de leurs articles. Enfin nous avons dégagé des propriétés d’équité raisonnables nécessaires à l’établissement de la correction.

2. Le modèle complet est disponible ici : https://doi.org/10.5281/zenodo.1322052.

La vérification a été effectué sur un PC sous GNU/Linux équipé d'un processeur Intel Xeon E5-2699 fournissant 512 Go de RAM. En fonction des analyses à effectuer, nous nous sommes basés sur les *model-checkers* bornés et non-bornés proposés dans Electrum AnalyZer. Le premier repose sur une traduction en problème SAT mise en œuvre par Electrum lui-même ou sur un algorithme fourni par nuXmv [CCD+14]. Le second repose sur un algorithme de nuXmv.

Nous d'abord vérifié que notre modèle est cohérent (il admet une instance) et que toutes les branches d'action sont réalisables. Toutes ces analyses ont abouti positivement en moins de 10 s. en utilisant le mode BMC d'Electrum Analyzer. La correction, elle, peut être vérifiée par l'analyseur borné pour des réseaux avec 4-6 nœuds et un horizon de 10 pas (4 nœuds pour un horizon de 15), tandis que l'analyseur non-borné donne un résultat pour les réseaux de 4 membres seulement (le tout avec un *time-out* de 5 h.).

Références

[BCC+18] Julien Brunel, David Chemouil, Alcino Cunha, Thomas Hujsa, Nuno Macedo, and Jeanne Tawa. Proposition of an action layer for electrum. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 397–402. Springer, 2018.

[BCT18] Julien Brunel, David Chemouil, and Jeanne Tawa. Analyzing the Fundamental Liveness Property of the Chord Protocol. In *Formal Methods in Computer-Aided Design*, Austin (USA), October 2018.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In *Computer Aided Verification - 26th International Conference, CAV 2014.*, pages 334–342, 2014.

[Jac12] Daniel Jackson. *Software Abstractions : logic, language, and analysis*. MIT press, 2012.

[LNBK02] David Liben-Nowell, Hari Balakrishnan, and David Karger. Analysis of the evolution of peer-to-peer systems. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 233–242. ACM, 2002.

[MBC+16] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha, and Denis Kuperberg. Lightweight Specification and Analysis of Dynamic Systems with Rich Configurations. In *Foundations of Software Engineering*, 2016.

[Zav17] Pamela Zave. Reasoning about identifier spaces : How to make chord correct. *IEEE Transactions on Software Engineering*, 43(12) :1144–1156, Dec 2017.

Approches au Développement Formel de Systèmes Hybrides Basées sur la Preuve : Logique Dynamique et Event-B

G. Dupont, Y. Aït-Ameur, M. Pantel, N. K. Singh
INPT-ENSEEIHT/IRIT, University of Toulouse, France
{guillaume.dupont,yamine,marc.pantel,nsingh}@enseeiht.fr

Résumé

Les systèmes hybrides sont la fusion d'un programme discret avec un environnement analogique. Ils occupent aujourd'hui une part importante de notre environnement et réalisent des tâches très diverses, qui sont parfois critiques. La formalisation et la preuve de tels systèmes est un problème important qui doit être étudié si l'on veut instiguer de la confiance en ces systèmes auprès de ses utilisateurs. Dans ce papier, nous nous intéressons à deux approches *correctes par construction* et basées sur la preuve. Event-B [1] – avec Rodin [2] – ainsi que la logique différentielle dynamique [8] – avec KeYmaera [9] – sont appliquées à une même étude de cas puis comparées. De cette étude nous tirons ensuite une approche générique à la modélisation des systèmes hybrides avec Event-B.

1 Étude de Cas

Nous nous intéressons à une étude de cas développée par Quesel et al. dans [9] : une voiture roule à la position p et avec une vitesse et une accélération v et a respectivement. Le but est de concevoir un contrôleur qui fait freiner automatiquement la voiture avant un panneau stop, qui se trouve à la position SP.

L'accélération est le seul paramètre sur lequel le contrôleur peut agir. Pour des raisons de simplification, elle change de manière discrète et peut prendre la valeur 0 (vitesse stable), $A > 0$ (vitesse en augmentation) et $-b$, $b > 0$ étant la force de freinage de la voiture. La vitesse de la voiture est par ailleurs physiquement comprise entre 0 et V_{max}.

Tant que la voiture est suffisamment loin de l'objectif, l'utilisateur peut changer d'accélération à loisir ; mais lorsque la voiture s'approche de SP, le contrôleur prend le relais et amorce le freinage du véhicule. La propriété de sûreté du système est donc la suivante : $\forall t \in \mathbb{R}^+ \cdot p(t) < SP$.

Le système obéit à l'équation différentielle $\dot{v}(t) = a, \dot{p}(t) = v(t)$.

2 Approche avec $\mathrm{d}\mathcal{L}$/KeYmaera

Quesel et al. développent dans [9] un programme hybride répondant à l'étude de cas (c.f. : Figure 1).

Le principe général de ce programme hybride est de jouer alternativement et indéfiniment le contrôleur ($ctrl$) et le système sous contrôle ($plant$). Le contrôleur se contente de faire le choix de l'accélération de la voiture en fonction de la distance à SP. La partie processus continu implémente simplement

les équations différentielles données dans la Section 1. Ces équations sont données avec un *domaine d'évolution* (après le symbole $\&$) qui dénote dans quel domaine le système est sensé évoluer.

Dans le cas présent, le domaine est divisé en deux : un domaine où $safe$ est vrai et un autre où il est faux ; cela permet de signifier que dans le cas où le système entre dans une zone non sûre le contrôleur prend la main sur le système continu.

$init$	$\rightarrow$	$[(ctrl; plant)^*]\,(req)$	(1)
$init$	$\equiv$	$v \geq 0 \wedge A > 0 \wedge B > 0 \wedge safe$	(2)
$safe$	$\equiv$	$p + \frac{v^2}{2B} < SP$	(3)
$ctrl$	$\equiv$	$(?safe; a := A) \cup (?v = 0; a := 0) \cup (a := -B)$	(4)
$plant$	$\equiv$	$(p' = v, v' = a \& v \geq 0 \wedge p + \frac{v^2}{2B} \leq SP)$	(5)
		$\cup (p' = v, v' = a \& v \geq 0 \wedge p + \frac{v^2}{2B} \geq SP)$	
req	$\equiv$	$p \leq SP$	(6)

FIGURE 1 – Hybrid program for the self-driven car

Pour prouver que l'invariant de sûreté (req) est respecté par le système, KeYmaera propose de prouver la ligne (1), à savoir *si les conditions initiales ($init$) sont vraies alors quelle que soit l'exécution de ce dernier, req demeure vrai*. Cette quantification sur les exécutions du système est exprimée à l'aide la modalité $[\cdot]$.

3 Approche avec Event-B/Rodin

Dans cette section, nous donnons les grandes lignes de l'approche que nous avons déployée pour modéliser cette étude de cas.

3.1 Extension de la Méthode

La méthode Event-B est basée sur la théorie des ensembles et la logique du premier ordre. Bien que très expressif, ce cadre est assez limitant dès lors qu'il faut manipuler des concepts tels que les réels et la continuité. Afin d'aider à la définition d'extensions à Event-B, [3, 4] ont introduit la notion de *théorie*, un regroupement de types, axiomes, opérateurs et théorèmes qui peuvent être utilisés dans divers modèles Event-B.

Notre travail s'appuie grandement sur ce mécanisme, et nous avons défini à travers lui tous les concepts dont nous avions besoin : continuité, équations différentielles, solvabilité, etc.

3.2 Principe

Observons qu'un système hybride se compose de deux parties : un contrôleur (programme discret que nous voulons écrire) et un processus continu (ou *plant*, le phénomène physique que l'on veut contrôler). Nous décomposons le système en 4 catégories de comportements :

— les **transitions** : changement interne du contrôleur, décision de l'utilisateur
— les **détections** (*sensing*) : changement du contrôleur dû à l'évolution du processus continu
— les **commandes** (*actuation*) : action du contrôleur sur le processus continu
— l'**environnement** : changement du processus continu dû à des perturbations issues de l'environnement (par exemple : vent, pluie, etc.)

De cette observation on déduit un *modèle générique* Event-B qui se compose d'un événement pour chaque catégorie de comportement. À cela s'ajoute une modélisation explicite de l'état discret (en tant que variable discrète) et de l'état continu (en tant que fonction du temps).

Le temps est également modélisé sous la forme d'une variable t en lecture seule, dont le passage est simulé par un événement particulier, `Progress`, qui s'assure que le temps est globalement croissant.

Ce modèle générique s'instancie à l'aide du raffinement. Le raffinement de donnée est utilisé pour concrétiser les états continus et discrets, et le raffinement d'événements est utilisé pour construire les événements réels du système à partir des événements génériques.

3.3 Sémantique

La sémantique classique d'Event-B consiste à envisager les modèles comme des systèmes de transition en entrelaçant les événements qui le composent. Chaque événement est instantané et fait progresser le système d'un état à un autre.

Mais cette sémantique se révèle insuffisante pour traiter des systèmes continus, qui ont une forte composante temporelle. Aussi, nous étendons cette sémantique par l'ajout d'un type d'événement particulier : les événements continus. Contrairement à leurs pendants discrets (dont l'exécution est absolument instantanée), les événements continus s'exécutent pendant un temps prolongé.

Pour être précis, les événement continus concernent le système sous contrôle. Il s'exécutent tant que le contrôleur n'a pas d'événement discret à faire. Dès lors qu'un événement discret est activé, le contrôleur réalise cet événement immédiatement (préemption) avant de revenir à un nouvel événement continu (dépendant probablement du nouvel état du contrôleur).

Les événements de commande (*actuation*) et d'environnement (et *a fortiori* tout événement qui modifie l'état continu) sont des événements continus.

4 Analyse

L'approche de Platzer s'attache à modéliser le système à un niveau fonctionnel proche du programme contrôleur lui-même. La structure même de l'approche rend la décomposition et la construction étape par étape assez difficile, même si des formes de raffinement semblent voir le jour [7].

La puissance de $\mathrm{d}\mathcal{L}$ et des programmes hybrides en général réside plus dans le système de preuve qui l'accompagne que dans les modèles qu'ils permettent d'écrire. En particulier, un certain nombre de concepts sont occultés des programmes hybrides (typiquement, les problèmes de solvabilité des équations différentielles) et sont en fait manipulés en arrière-plan de certaines règles d'inférence.

De son côté, Event-B, de par l'extrême simplicité de son système formel (théorie des ensembles et logique du premier ordre) offre une très grande expressivité, ce qui permet de gérer au niveau du modèle les spécificités du monde continu. Cependant, la manipulation de structures complexes (en particulier au travers du plug-in Theory) rend les preuves beaucoup plus complexes.

Grâce au raffinement qui est au coeur de la méthode, il est possible d'utiliser Event-B pour formaliser des systèmes hybrides à un haut niveau d'abstraction, ce qui permet d'explorer divers schémas d'implémentation (différents types de discrétisation, schémas à contrôleurs multiples, etc.).

5 Conclusion et Travaux Futurs

L'approche présentée a été appliquée avec succès à deux études de cas dans [6] et [5]. Les modèles complets de ces deux études de cas sont disponibles à l'adresse `https://www.irit.fr/~Guillaume.Dupont/models.php`.

Le côté générique de l'approche permet d'explorer divers possibilités quant à la conception de systèmes hybrides. En particulier, différents patrons de conception de contrôleurs peuvent être exprimés :

un ou plusieurs contrôleurs, un ou plusieurs systèmes, types spécifiques d'équations différentielles (linéaires, autonomes, etc.) et ainsi de suite.

Le coeur de la méthode étant le raffinement, il est également possible d'envisager d'autres types de raffinement lié au monde continu, par exemple au niveau de la discrétisation.

Remerciements

Ce travail a été financé par l'ANR-17-CE25-0005 (projet DISCONT `http://discont.loria.fr`) de l'Agence Nationale de la Recherche (ANR).

Références

[1] Abrial, J.R. : Modeling in Event-B : System and Software Engineering. Cambridge University Press, New York, NY, USA, 1st edn. (2010)

[2] Abrial, J.R., Butler, M., Hallerstede, S., Hoàng, T.S., Mehta, F., Voisin, L. : Rodin : an open toolset for modelling and reasoning in Event-B. International Journal on Software Tools for Technology Transfer 12(6), 447–466 (2010)

[3] Abrial, J.R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., Voisin, L. : Proposals for mathematical extensions for Event-B. Tech. rep. (2009)

[4] Butler, M., Maamria, I. : Practical theory extension in Event-B. In : Liu, Z., Woodcock, J., Zhu, H. (eds.) Theories of Programming and Formal Methods, Lecture Notes in Computer Science, vol. 8051, pp. 67–81. Springer Berlin Heidelberg (2013)

[5] Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K. : Hybrid systems and event-b : A formal approach to signalised left-turn assist. In : New Trends in Model and Data Engineering. pp. 153–158. Springer International Publishing (2018)

[6] Dupont, G., Aït-Ameur, Y., Pantel, M., Singh, N.K. : Proof-based approach to hybrid systems development : Dynamic logic and event-b. In : Butler, M., Raschke, A., Hoang, T.S., Reichl, K. (eds.) Abstract State Machines, Alloy, B, TLA, VDM, and Z. pp. 155–170. Springer International Publishing, Cham (2018)

[7] Loos, S.M., Platzer, A. : Differential refinement logic. In : Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science. pp. 505–514. LICS '16, ACM, New York, NY, USA (2016)

[8] Platzer, A. : Differential dynamic logic for hybrid systems. Journal of Automated Reasoning 41(2), 143–189 (2008)

[9] Quesel, J.D., Mitsch, S., Loos, S., Aréchiga, N., Platzer, A. : How to model and prove hybrid systems with keymaera : a tutorial on safety. International Journal on Software Tools for Technology Transfer 18(1), 67–91 (2016)

Une approche basée sur la séparation des préoccupations pour modéliser et vérifier les règles de signalisation d'un système ferroviaire

Yves Ledru[1,2], Akram Idani[1,2], Rahma Ben Ayed[2], Abderrahim Ait Wakrime[2], and Philippe Bon[2,3]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble, France
{yves.ledru,akram.idani}@imag.fr
[2] Institut de Recherche Technologique Railenium, F-59300, Famars, France
{rahma.ben-ayed,abderrahim.ait-wakrime}@railenium.eu
[3] Univ Lille Nord de France, IFSTTAR, COSYS, ESTAS, F-59666 Villeneuve d'Ascq Cedex, France
philippe.bon@ifsttar.fr

Cet article est un résumé étendu de l'article *"A separation of concerns approach for the verified modelling of railway signalling rules"* [9], accepté pour publication à la conférence RssRail 2019 (International Conference on Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification).

Les systèmes ferroviaires sont des systèmes critiques dont la sûreté a été étudiée depuis de nombreuses années. La sûreté résulte d'une combinaison de dispositifs physiques (freins, signaux,...), de règles (par exemple des règles de signalisation) et d'une coopération entre agents humains (conducteurs de trains, agent de circulation,...). Des technologies nouvelles sont déployées pour améliorer ces systèmes ferroviaires. Par exemple, GNSS (Global Navigation Satellite System) utilise le système GPS pour connaître la position du train. De nouvelles normes européennes, comme ERTMS/ETCS (European Rail Traffic Management System/European Train Control System) ont été proposées pour remplacer les systèmes de signalisation existants. Dans ce contexte, de nouvelles règles de signalisation doivent être conçues pour prendre en compte ces nouveaux équipements. Ces règles ont un caractère critique qui requiert des activités de vérification et de validation pour garantir leur sûreté.

Les méthodes formelles, et en particulier la méthode B [1], sont utilisées dans ce domaine depuis plus de 25 ans. L'utilisation de la méthode B pour des études

de cas industrielles s'est soldée par plusieurs réussites remarquables [12], parmi lesquelles, le développement du métro Météor à Paris [3] ou la modernisation du métro de New York [13]. Au fil des années, une communauté scientifique s'est bâtie et a grandi autour de projets européens comme FMERail, ou de conférences comme RSSRail [7]. En 2018, la conférence ABZ [6], qui réunit notamment la communauté de la méthode B, a proposé à ses participants de traiter un cas d'étude qui modélisait ERTMS/ETCS level 3. Dans [2], nous avons utilisé une combinaison d'UML et d'Event-B pour cette étude de cas.

Dans [4, 5] nous avons proposé de structurer les spécifications B de règles de signalisation en nous inspirant de la description des systèmes d'information sécurisés en SecureUML [11]. Nous structurons notre modèle en un premier modèle qui décrit le comportement du système ferroviaire en l'absence de règles de circulation, c'est-à-dire un système qui est essentiellement régi par les lois de la physique : les trains doivent suivre les rails et sont arrêtés en cas d'accident. Un deuxième modèle décrit les règles de signalisation qui régissent le comportement des trains et des agents. Dans ce modèle, les trains doivent respecter les signaux et on peut démontrer que les règles garantissent l'absence d'accidents. Ce modèle est proche de la définition d'une politique de contrôle d'accès car il accorde à des agents, humains ou logiciels, des permissions d'accéder aux objets du diagramme de classes.

Cette approche promeut une séparation des préoccupations qui traite d'une part les objets qui constituent le système et d'autre part les règles qui permettent aux agents de les manipuler. L'outil B4MSecure [8] est utilisé en support de cette approche. Il permet de traduire des diagrammes SecureUML en spécifications B. Nous utilisons également l'outil ProB [10] pour animer et valider la spécification [4] et le démonstrateur de l'atelier B pour prouver les obligations de preuve liées aux invariants qui garantissent la sûreté ferroviaire [5].

Dans [4], nous avons appliqué cette approche pour décrire les échanges d'informations entre un train et le centre de contrôle. Dans cet article [9], nous complétons cette approche sur les points suivants:

1. *Méthode formelle "légère" (lightweight formal method).* Notre approche passe par plusieurs étapes de vérification pour vérifier des propriétés d'atteignabilité, ou garantir la préservation d'invariant (absence d'accidents). Nous utilisons le model-checker ProB pour automatiser ces vérifications. Il permet de vérifier les propriétés d'atteignabilité de notre modèle en quelques minutes. Par contre, la vérification de la préservation d'invariant peut prendre plusieurs heures.

2. *Prise en compte des erreurs humaines.* Un système ferroviaire est piloté par des agents humains (conducteurs de train, agent de circulation). Dès lors, cela rend possible des erreurs humaines, par exemple quand un agent est fatigué. Ces erreurs humaines constituent des violations des règles. Dans cet article, nous prenons en compte ces comportements en les modélisant et

en évaluant leurs conséquences. Il est également possible de modéliser des contre-mesures pour empêcher ces erreurs de se produire ou limiter leurs conséquences.

3. Nos travaux précédents [5] ont étudié les protocoles qui régissent les échanges entre le train et l'agent de circulation. Dans cet article, nous prenons en compte un système plus large qui comprend notamment la topologie des voies. De plus, nos activités de vérification font appel au model-checking, alors que dans nos travaux précédents, nous faisions appel à la preuve ou à l'animation.

Remerciements Ce travail est financé par le projet NExTRegio de l'IRT Railenium. Les auteurs remercient SNCF Réseau pour son soutien. Nous remercions également German Vega pour le support de B4MSecure.

References

[1] Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (1996)

[2] Ait Wakrime, A., Ben Ayed, R., Collart Dutilleul, S., Ledru, Y., Idani, A.: Formalizing railway signaling system ERTMS/ETCS using UML/Event-B. In: MEDI 2018. pp. 321–330. Springer (2018). https://doi.org/10.1007/978-3-030-00856-7_21

[3] Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Météor: A successful application of B in a large project. In: FM'99. LNCS, vol. 1708, pp. 369–387. Springer (1999). https://doi.org/10.1007/BFb0053352

[4] Ben Ayed, R., Collart Dutilleul, S., Bon, P., Idani, A., Ledru, Y.: B formal validation of ERTMS/ETCS railway operating rules. In: ABZ 2014. LNCS, vol. 8477, pp. 124–129. Springer (2014). https://doi.org/10.1007/978-3-662-43652-3_10

[5] Ben Ayed, R., Collart Dutilleul, S., Bon, P., Ledru, Y., Idani, A.: Formalismes basés sur les rôles pour la modélisation et la validation des règles d'exploitation ferroviaires. Technique et Science Informatiques **34**(5), 495–521 (2015). https://doi.org/10.3166/tsi.34.495-521

[6] Butler, M.J., Raschke, A., Hoang, T.S., Reichl, K.: Abstract State Machines, Alloy, B, TLA, VDM, and Z - 6th Int. Conference, ABZ 2018, LNCS, vol. 10817. Springer (2018). https://doi.org/10.1007/978-3-319-91271-4

[7] Fantechi, A., Lecomte, T., Romanovsky, A.: Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certifica-

tion - 2nd Int. Conf., RSSRail 2017, LNCS, vol. 10598. Springer (2017). https://doi.org/10.1007/978-3-319-68499-4

[8] Idani, A., Ledru, Y.: B for modeling secure information systems - the B4MSecure platform. In: ICFEM 2015. LNCS, vol. 9407, pp. 312–318. Springer (2015). https://doi.org/10.1007/978-3-319-25423-4_20

[9] Ledru, Y., Idani, A., Ben Ayed, R., Ait Wakrime, A., Bon, P.: A separation of concerns approach for the verified modelling of railway signalling rules. In: International Conference on Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - RssRail 2019. LNCS, vol. 11495. Springer (Jun 2019). https://doi.org/https://doi.org/10.1007/978-3-030-18744-6_11

[10] Leuschel, M., Butler, M.J.: ProB: an automated analysis toolset for the B method. STTT **10**(2), 185–203 (2008). https://doi.org/10.1007/s10009-007-0063-9

[11] Lodderstedt, T., Basin, D.A., Doser, J.: SecureUML: A UML-based modeling language for model-driven security. In: UML 2002, LNCS 2460. Springer (2002). https://doi.org/10.1007/3-540-45800-X_33

[12] Sabatier, D.: Using formal proof and B method at system level for industrial projects. In: RSSRail 2016. LNCS, vol. 9707, pp. 20–31. Springer (2016). https://doi.org/10.1007/978-3-319-33951-1_2

[13] Sabatier, D., Burdy, L., Requet, A., Guéry, J.: Formal proofs for the NYCT line 7 (flushing) modernization project. In: ABZ 2012. LNCS, vol. 7316, pp. 369–372. Springer (2012). https://doi.org/10.1007/978-3-642-30885-7_34

On est tranquilles pour longtemps
Les reçus-temps en logique de séparation

François Pottier

Au cours de cet exposé, je rappellerai d'abord brièvement comment la logique de séparation, étendue avec une notion simple de **crédit-temps**, permet de vérifier à la fois la correction fonctionnelle et la complexité en temps d'un programme. Puis je présenterai une notion duale de **reçu-temps**. Je montrerai qu'elle permet d'argumenter formellement que certains événements indésirables, comme le dépassement de capacité d'un compteur entier, ne peuvent pas survenir avant un temps très grand. En d'autres termes, certains programmes sont *sûrs pour longtemps*.

Primitive Floats in Coq

Guillaume Bertholon Érik Martin-Dorel Pierre Roux

Abstract

Some mathematical proofs involve intensive computations, for instance: the four-color theorem, Hales' theorem on sphere packing (formerly known as the Kepler conjecture) or interval arithmetic. For numerical computations, floating-point arithmetic enjoys widespread usage thanks to its efficiency, despite the introduction of rounding errors.

Formal guaranties can be obtained on floating-point algorithms based on the IEEE 754 standard, which precisely specifies floating-point arithmetic and its rounding modes, and a proof assistant such as Coq, that enjoys efficient computation capabilities. Coq offers machine integers, however floating-point arithmetic still needed to be emulated using these integers.

A modified version of Coq is presented that enables using the machine floating-point operators. The main obstacles to such an implementation and its soundness are discussed. Benchmarks show potential performance gains from two to three orders of magnitude.

1 Motivation

The proof of some mathematical facts can involve a numerical computation in such a way that trusting the proof requires trusting the numerical computation itself. Thus, being able to efficiently perform this kind of proofs inside a proof assistant eventually means that the tool must offer efficient numerical computation capabilities.

Floating-point arithmetic is widely used in particular for its efficiency thanks to its hardware implementation. Although it does not generally give exact results, introducing rounding errors, rigorous proofs can still be obtained by bounding the accumulated errors. There is thus a clear interest in providing an efficient and sound access to the processor floating-point operators inside a proof assistant such as Coq.

1.1 Proofs Involving Numerical Computations

We give below a few examples of proofs involving floating-point computations.

As a first example, consider the proof that a given real number $a \in \mathbb{R}$ is non-negative. One can exhibit another real number r such that $a = r^2$ and apply a

lemma stating that all squares of real numbers are nonnegative. Typically, one could use the square root $\sqrt{a}$.

A similar method can be applied to prove that a matrix $A \in \mathbb{R}^{n\times n}$ is positive semidefinite[1] as one can exhibit R such that[2] $A = R^T R$. Such a matrix can be computed using an algorithm called Cholesky decomposition. The algorithm succeeds, taking neither square roots of negative numbers nor divisions by zero, whenever A is positive definite[3].

When executed with floating-point arithmetic, the exact equality $A = R^T R$ is lost but it remains possible to bound the accumulated rounding errors in the Cholesky decomposition such that the following theorem holds under mild conditions.

Theorem (Corollary 2.4 in [5]) *For $A \in R^{n\times n}$, defining $c := \frac{(n+1)\epsilon}{1-2(n+1)\epsilon}\,\mathrm{tr}(A) + 4n\,(2(n+1) + \max_i A_{i,i})\,\eta$, if the floating-point Cholesky decomposition succeeds on $A - c\,I$, then A is positive definite. ϵ and η are tiny constants given by the floating-point format used.*

A formal proof in Coq of this theorem can be found in a previous work [4]. Thus, an efficient implementation of floating-point arithmetic inside the proof assistant leads to efficient proofs of matrix positive definiteness. This can have multiple applications, such as proving that polynomials are nonnegative by expressing them as sums of squares [3] which can be used in a proof of the Kepler conjecture [1].

Interval arithmetic constitutes another example of proofs involving numerical computations. Sound enclosing intervals can be easily computed in floating-point arithmetic using directed roundings, towards $\pm\infty$ for lower or upper bounds. The Coq.Interval library [2] implements interval arithmetic and could benefit from efficient floating-point arithmetic.

More generally, there are many results on rigorous numerical methods [6] that could see efficient formal implementations provided efficient floating-point arithmetic is available inside proof assistants.

1.2 Objectives

The Coq proof assistant has built-in support for computation, which can be used within proofs, and recent progress have been done to provide efficient integer computation (relying on 63-bit machine integers).

The overall goal of this work is to implement efficient floating-point computation in Coq, relying directly on machine `binary64` floats, instead of emulating floats with pairs of integers. Experimentally, that latter emulation in Coq incurs a slowdown of about three orders of magnitude with respect to an equivalent implementation written in OCaml.

[1] A matrix $A \in \mathbb{R}^{n\times n}$ is said positive semidefinite when for all $x \in \mathbb{R}^n$, $x^T A\,x \geq 0$.

[2] Since, when $A = R^T R$, one gets $x^T A\,x = x^T \left(R^T R\right) x = (Rx)^T\,(Rx) = \|Rx\|^2 \geq 0$.

[3] A matrix $A \in \mathbb{R}^{n\times n}$ is said positive definite when for all $x \in \mathbb{R}^n \setminus \{0\}$, $x^T A\,x > 0$.

1.3 Outline

After providing some background on proof-by-reflection and floating-point arithmetic, this talk will focus on the implementation itself, with the interface that it exposes, several design choices or technicalities and some pitfalls to avoid. Finally some benchmarks will be presented.

The implementation is available in the `primitive-floats` branch of the following repository:
`https://github.com/validsdp/coq/tree/primitive-floats`

References

[1] Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 8(1):1–24, 2015.

[2] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, October 2016.

[3] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017.

[4] Pierre Roux. Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *J. Autom. Reasoning*, 57(2):135–156, 2016.

[5] Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46:433–452, 2006.

[6] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

Formalisation en Coq des erreurs d'arrondi de méthodes de Runge-Kutta pour les systèmes matriciels

Florian Faissole

Inria, Université Paris-Saclay, F-91120 Palaiseau
LRI, CNRS & Univ. Paris-Sud, F-91405 Orsay
Email : florian.faissole@inria.fr

Résumé

Bien que les équations différentielles ordinaires soient omniprésentes dans la modélisation de systèmes physiques ou biologiques, leur résolution exacte est parfois fastidieuse voire impossible. L'utilisation de méthodes numériques, comme les méthodes de Runge-Kutta, permet d'obtenir des solutions approchées. Nous exhibons et formalisons en Coq des bornes sur les erreurs d'arrondi induites par l'implémentation en arithmétique à virgule flottante de méthodes de Runge-Kutta appliquées à des systèmes linéaires matriciels en tenant compte d'éventuels dépassements de capacité inférieurs.

1 Introduction

Les équations différentielles ordinaires modélisent de nombreux phénomènes physiques, biologiques ou économiques. Il n'existe cependant pas toujours de méthode directe pour les résoudre. Des méthodes numériques itératives ont été mises au point pour résoudre ces problèmes de façon approchée. L'essentiel des travaux scientifiques autour des méthodes numériques consiste à construire des méthodes relativement peu coûteuses permettant d'obtenir une approximation précise de la solution exacte et qui puissent s'appliquer à une classe importante de problèmes. Parmi ces méthodes de résolution, les méthodes de Runge-Kutta sont parmi les plus fréquemment utilisées (voir section 3). Les domaines d'application des équations différentielles étant parfois critiques (*e.g.* en aéronautique ou pour la modélisation de comportements robotiques), leur résolution est un objet d'étude intéressant pour des travaux de vérification formelle [13, 14, 5].

L'implémentation de ces méthodes en arithmétique à virgule flottante provoque des erreurs d'arrondi pouvant s'accumuler au cours des itérations. Ces erreurs étant dans la plupart des cas négligeables face aux erreurs de méthode, elles ont peu été étudiées par les numériciens. Des travaux ont néanmoins proposé des méthodes pour majorer les erreurs d'arrondi par des approches probabilistes tirant parti d'éventuelles compensations d'erreurs [10] ou en utilisant des techniques à base d'intervalles [8, 1] ou de modèles de Taylor [3, 17]. Cependant, les bornes exhibées ne tiennent compte ni des caractéristiques mathématiques détaillées des méthodes de Runge-Kutta, ni de la forme de l'implémentation choisie. Nous proposons une approche à grains fins (*i.e.* en décomposant l'analyse d'erreur au niveau des opérations élémentaires) qui tire parti de ces caractéristiques, ce qui rapproche nos travaux des résultats présentés par Fousse [9] pour les méthodes d'intégration numérique

ou par Boldo [4] qui borne et formalise en Coq les erreurs d'arrondi induites par un schéma de résolution de l'équation des ondes. Roux [18] propose une analyse d'erreur d'arrondi en Coq pour plusieurs algorithmes numériques impliquant des opérations matricielles, comme la décomposition de Cholesky. Cette formalisation repose sur une spécification de l'arithmétique à virgule flottante définie *via* un type `Record` et particulièrement adaptée à l'analyse d'erreur. Cette spécification est par ailleurs satisfaite par le format *binary64* de Flocq [7], une bibliothèque Coq d'arithmétique des ordinateurs.

Boldo, Faissole et Chapoutot [6] ont exhibé (sans garantie formelle) des bornes sur les erreurs d'arrondi induites par l'implémentation de méthodes de Runge-Kutta appliquées à des systèmes linéaires unidimensionnels (systèmes de la forme $\dot{y} = \lambda y$ où $\lambda \in \mathbb{R}$). Dans cet article, nous généralisons cette approche au cas des systèmes linéaires multidimensionnels, où λ est remplacé par une matrice carrée A à coefficients réels. Les opérations matricielles étant plus complexes que les opérations sur les nombres réels, les résultats prennent en compte un nombre plus important de paramètres, à commencer par la dimension du système. Nous proposons de plus une formalisation de ces résultats dans l'assistant de preuves Coq [1] en combinant la bibliothèque Flocq [7] et les matrices de la bibliothèque MathComp [15]. Nous tenons compte d'éventuels dépassements graduels de capacité inférieurs (*underflows*), dont la contribution impacte les bornes d'erreurs exhibées. En revanche, nous ne tenons pas compte des dépassements de capacité supérieurs (*overflows*). Notre approche diffère légèrement de celle adoptée par Roux [18] car nous utilisons directement la formalisation des formats de nombres flottants de la bibliothèque Flocq, sans passer par une spécification dédiée de l'arithmétique à virgule flottante.

La section 2 présente quelques rappels d'arithmétique à virgule flottante. La section 3 est dédiée à la présentation du problème et de notre méthodologie (basée sur la distinction entre erreurs locales et globales). Dans la section 4, nous présentons les éléments de base de la formalisation Coq. Dans les sections 5 et 6, nous bornons respectivement les erreurs locales et globales des méthodes de Runge-Kutta avant de conclure en section 7.

2 Prérequis d'arithmétique à virgule flottante

2.1 Formats de représentation des nombres flottants

Les formats de représentation des nombres à virgule flottante ainsi que les opérations sur ces nombres sont régis par la norme IEEE-754 [12, §3]. En analyse d'erreur, une définition relativement haut niveau (sans décrire la représentation bit à bit des nombres flottants) est suffisante. Un format flottant $\mathbb{F}$ est représenté par un tuple $(\beta, p, e_{\min}, e_{\max})$ où l'entier $\beta \geqslant 2$ est la base, $p \in \mathbb{N}$ est la précision, $e_{\min} \in \mathbb{Z}$ et $e_{\max} \in \mathbb{Z}$ sont les valeurs minimales et maximales de l'exposant. Un nombre flottant dans $\mathbb{F}$ est soit une valeur exceptionnelle parmi $+\infty$, $-\infty$ ou NaN, soit une valeur égale à $\pm d_0.d_1 \cdots d_{p-1} \times \beta^e$ (avec d_i des chiffres dans la base β) et tel que $e_{\min} \leq e \leq e_{\max}$. Dans ce qui suit, $\beta = 2$.

La Figure 1 présente la répartition des nombres flottants sur l'axe réel. Les nombres flottants dits normalisés vérifient $d_0 = 1$ et $e_{\min} \leq e \leq e_{\max}$. Le plus petit nombre flottant normalisé positif est $\xi = 2^{e_{\min}}$ et le plus grand nombre flottant normalisé est $\Omega = (2 - 2^{1-p})2^{e_{\max}}$. Lorsqu'un nombre flottant est plus petit que ξ, il est dit dénormalisé et on parle de dépassement de capacité inférieur (*underflow*). Nous avons alors $e = e_{\min}$ et $d_0 = 0$. On note $\eta = 2^{e_{\min}-p+1}$ le plus petit nombre flottant dénormalisé positif. Lorsque le résultat

1. Les fichiers sont disponibles en ligne : `https://www.lri.fr/~faissole/CoqRK`

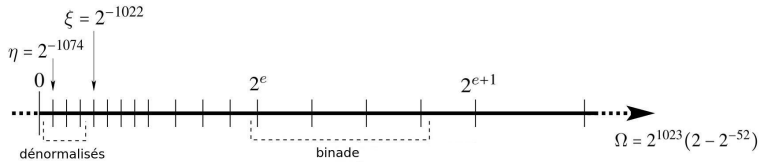


FIGURE 1 – Représentation des nombres flottants (format *binary64*) sur l'axe réel

d'une opération flottante n'est pas exactement représentable dans le format considéré, il est nécessaire d'arrondir le résultat vers un nombre flottant. Plusieurs modes d'arrondi sont définis dans la norme IEEE-754 [12], dont l'arrondi au plus proche. Lorsque le résultat est le point-milieu entre deux flottants consécutifs, une régle de bris d'égalité (*tie-breaking rule*) permet de choisir vers lequel arrondir, par défaut le nombre flottant pair est choisi.

Dans cet article, nous considérons un format flottant $\mathbb{F}$ prenant en compte les *underflows* (sauf mention du contraire). Nous considérons un mode d'arrondi au plus proche avec une règle de bris d'égalité quelconque, noté $\circ$. Nous noterons $\oplus$, $\ominus$, $\otimes$ et $\oslash$ les opérations arrondies correspondant aux opérations usuelles $+, -, \times$ et $/$, $\circ[\cdots]$ signifie que toutes les opérations à l'intérieur des crochets sont arrondies. Pour une même opération, le parenthésage à droite est implicite en l'absence de parenthèses, ainsi $\circ[ab + c] = (a \otimes b) \oplus c$ et $\circ[a + b + c] = a \oplus (b \oplus c)$. La convention s'étend aux opérations entre matrices et vecteurs. En particulier, pour $v \in \mathbb{R}^n$ un vecteur de taille n et $A, B \in \mathbb{R}^{n\times n}$ des matrices carrées de taille n, $\circ[A \times v]$ et $\circ[A \times B]$ font respectivement référence au produit de A par v et au produit de A par B en évaluant les sommations de la droite vers la gauche.

2.2 Fondements de l'analyse d'erreurs d'arrondi

Les analyses d'erreur que nous présentons sont basées sur quelques définitions et résultats fondamentaux (voir Higham pour une présentation détaillée de ces résultats [11, §3]). Nous commençons par présenter un résultat bornant l'erreur relative commise lors de l'arrondi d'un nombre réel de valeur absolue supérieure à ξ : si $x \in \mathbb{R}$ est tel que $\xi \leqslant |x|$, alors $\frac{|\circ(x)-x|}{|x|} \leqslant u = \frac{1}{2}\beta^{1-p}$. La quantité u, fondamentale en analyse d'erreur, est appelée unité d'arrondi. Par exemple, dans le format double précision *binary64*, $u = 2^{-53}$. Lorsque le nombre réel x est plus petit que ξ, l'erreur relative commise en arrondissant x peut devenir très grande, mais nous pouvons prouver que l'erreur absolue est bornée par $\frac{\eta}{2}$. Nous pouvons en déduire ce que Higham appelle modèle standard de l'arithmétique à virgule flottante [11, §2.2.], *i.e.* pour $x, y \in \mathbb{F}$, $\Diamond \in \{+, -, \times, /\}$, il existe $\delta, \varepsilon \in \mathbb{R}$ tels que $\circ(x \Diamond y) = (x \Diamond y)(1 + \varepsilon) + \delta$, $|\varepsilon| \leqslant u$, $|\delta| \leqslant \frac{\eta}{2}$ et $\varepsilon\delta = 0$. Si $\Diamond \in \{+, -\}, \delta = 0$.

Nous utiliserons régulièrement la notation $\gamma_d = \frac{du}{1-du}$ ($d \in \mathbb{N}$, $du < 1$) [11, §3.1.].

2.3 Flocq : bibliothèque d'arithmétique des ordinateurs en Coq

Pour formaliser nos résultats, nous utilisons Flocq, une bibliothèque Coq d'arithmétique des ordinateurs développée par Boldo et Melquiond [7] comportant une représentation abstraite des nombres flottants : les nombres dans un format donné sont des sous-ensembles des nombres réels `R` de la bibliothèque standard de la forme $m \cdot \beta^e$ où m et e sont des entiers. Par exemple, le format `FLX` correspond aux nombres flottants sans bornes sur les exposants : seule la précision p est prise en compte, avec la condition $|m| < \beta^p$. Le format `FLT` prend en compte les dépassements graduels de capacité inférieurs et est paramétré par

p et $e_{\min}$, les nombres flottants dans ce format vérifiant $|m| < \beta^p$ et $e \geq e_{\min}$. L'ensemble des résultats présentés dans cet article sont valides pour le format `FLX`. Nous avons ensuite étendu l'ensemble des résultats génériques pour le format `FLT`, *i.e.* l'ensemble des résultats à l'exception de l'instanciation aux méthodes d'Euler et de RK2.

3 Présentation du problème et méthodologie

Les méthodes de Runge-Kutta sont itératives. Elles consistent à discrétiser un intervalle de temps $[0, t_N]$ en un ensemble de points $t_0, \ldots, t_N$ et à construire itérativement les solutions approchées $y_0, \ldots, y_N$ en ces points. Ce sont des méthodes explicites à un pas constant h : la solution au temps $t_{n+1} = t_n + h$ est obtenue à partir de la solution au temps t_n.

Les systèmes linéaires multidimensionnels sont parmi les plus fréquemment rencontrés dans les domaines liés à des problèmes physiques. Les équations différentielles associées sont de la forme $\dot{y} = Ay \quad \text{avec} \quad y \in \mathbb{R}^d \quad \text{et} \quad A \in \mathbb{R}^{d\times d}$. L'application d'une méthode de Runge-Kutta explicite sur ce système linéaire conduit à une relation de récurrence de la forme $y_{n+1} = R(hA)y_n$ avec R un polynôme en hA (de la forme $\sum_i \alpha_i (hA)^i$, $\alpha_i \in \mathbb{R}$). Prenons l'exemple de deux méthodes classiques, les méthodes d'Euler et de RK2. Les polynômes associés à ces méthodes sont $R_{\text{Euler}}(hA) = (I + hA)$ et $R_{\text{RK2}}(hA) = (I + hA + 0{,}5h^2A^2)$. Le polynôme R est appelé fonction de stabilité linéaire pour les méthodes de Runge-Kutta car, dans le cas des systèmes linéaires, la condition $|||R(hA)|||_\infty < 1$ caractérise une classe de couples système-méthode dit stables (voir section 4.1 pour une définition de la norme vectorielle $\|\cdot\|_\infty$ et de sa norme matricielle subordonnée $|||\cdot|||_\infty$). La fonction R est purement mathématique et ne caractérise pas l'implémentation des méthodes en arithmétique à virgule flottante, qui suppose le choix d'un algorithme, noté $\widetilde{R}$ et qui dépend de trois paramètres : le pas h (considéré comme représentable dans le format flottant, contrairement à ce qui est fait dans [6]), une matrice $\widetilde{A}$ correspondant à la matrice "arrondie de A" (matrice obtenue après arrondi de chaque coefficient de A) ainsi que la solution $\widetilde{y_n}$ calculée à l'itération précédente (lors du calcul de $\widetilde{y_{n+1}}$). Ainsi, une implémentation de la méthode de Runge-Kutta est de la forme : $\widetilde{y_0} = \circ(y_0)$ et pour tout n, $\widetilde{y_{n+1}} = \widetilde{R}(h, \widetilde{A}, \widetilde{y_n})$. À un même polynôme R correspondent plusieurs implémentations $\widetilde{R}$ possibles [2]. Dans cet article, nous nous intéressons à une évaluation "à la Horner" des méthodes numériques (décrite ci-dessous pour Euler et RK2 respectivement) :

$$\widetilde{y_{n+1}} = \circ\left[\widetilde{y_n} + (h\widetilde{A})\widetilde{y_n}\right], \qquad \widetilde{y_{n+1}} = \circ\left[\widetilde{y_n} + (h\widetilde{A})\left(\widetilde{y_n} + (\tfrac{h}{2}\widetilde{A})\widetilde{y_n}\right)\right].$$

Nous souhaitons borner l'erreur d'arrondi globale induite par l'implémentation d'une méthode de Runge-Kutta après n itérations, *i.e.* $E_n = \|\widetilde{y_n} - y_n\|_\infty$. Afin d'y parvenir, nous commençons par étudier les erreurs dites locales. L'erreur locale commise à l'itération n ne quantifie que les erreurs commises par les calculs de l'itération courante et peut être définie par $\varepsilon_0 = \|\widetilde{y_0} - y_0\|_\infty$ (que nous supposons connue) et pour tout $n \in \mathbb{N}, \quad \varepsilon_{n+1} = \|\widetilde{R}(h, \widetilde{A}, \widetilde{y_n}) - R(hA)\widetilde{y_n}\|_\infty$. Si les résultats ont du être adaptés, la méthodologie est similaire à celle utilisée dans le cas unidimensionnel [6]. Elle consiste dans un premier temps à construire une borne sur les erreurs locales relatives par applications mécaniques et consécutives du Lemme 1 (voir section 5). Ensuite, cette borne sur les erreurs locales est utilisée pour borner l'erreur globale par application directe du Théorème 2 (voir section 6).

2. En effet, les opérations flottantes ne sont pas associatives et dépendent de l'ordre d'évaluation. Par ailleurs, les calculs peuvent être décomposés, factorisés, etc.

4 Éléments de base de la formalisation

4.1 Vecteurs et matrices

Les systèmes étudiés font intervenir des vecteurs et des matrices à coefficients réels. Nous utilisons essentiellement des vecteurs colonnes de taille d (dont le type Coq est noté `cV_d`) et des matrices carrées de taille d (de type `M_d`).

Nous procédons à une analyse par normes des erreurs d'arrondi, ce type d'analyse reposant sur des normes sous-multiplicatives [11, §6]. Les normes vectorielles $\|\cdot\|_1$ et $\|\cdot\|_\infty$ sont particulièrement adaptées à l'analyse d'erreurs et les normes matricielles subordonnées à ces normes sont faciles à exprimer en fonction des coefficients de la matrice. Par ailleurs, nous nous basons sur des résultats de Higham qui sont valables pour ces deux normes [11]. Nous travaillons ici avec la norme $\|\cdot\|_\infty$, définie comme $\|v\|_\infty = \max_i |v_i|$ pour $v \in \mathbb{R}^d$ et dont la norme subordonnée est définie par $|||A|||_\infty = \max_i \sum_j |A_{i,j}|$ pour $A \in \mathbb{R}^{d\times d}$. Ces normes sont formalisées à l'aide de grands opérateurs (*bigops*) de MathComp [2], un mécanisme permettant d'itérer des opérations comme l'addition ou le maximum :

```
Definition vec_norm {d} : cV_d → R := fun v ⇒ \big[Rmax / 0]_(i < d) Rabs (v i).
Definition mat_norm {d} : M_d → R :=
                fun A ⇒ \big[Rmax / 0]_(i < d) (\big[Rplus / 0]_(j < d) Rabs (A i j)).
```

Nous prouvons qu'il s'agit bien de normes et que $|||\cdot|||_\infty$ est sous-multiplicative, *i.e.* pour $v \in \mathbb{R}^d, A \in \mathbb{R}^{d\times d}$, $\|Av\|_\infty \leqslant |||A|||_\infty \|v\|_\infty$ (`mx_vec_norm_submult` en Coq) et pour $A, B \in \mathbb{R}^{d\times d}$, $|||AB|||_\infty \leqslant |||A|||_\infty |||B|||_\infty$ (`mx_norm_submult` en Coq). La sous-multiplicativité permet de démontrer des bornes sur les erreurs d'arrondi des produits matrice-vecteur (`mx_vec_prod_error`) et matrice-matrice (`mx_prod_error`) :

$$\|\circ[A \times v] - Av\|_\infty \leqslant \gamma_d |||A|||_\infty \|v\|_\infty + \frac{d}{2}(1+\gamma_{d-1})\eta.$$
$$|||\circ[A \times B] - AB|||_\infty \leqslant \gamma_d |||A|||_\infty |||B|||_\infty + \frac{d^2}{2}(1+\gamma_{d-1})\eta.$$

Nous prouvons ces propriétés pour des sommations évaluées de droite à gauche. Ces bornes restent cependant vraies quelque soit l'ordre d'évaluation [11, §3.5.].

4.2 Méthodes de Runge-Kutta

Nous définissons un type `Sc` (pour Schéma), dont les éléments définissent des relations de récurrence entre les vecteurs y_n et y_{n+1} et caractérisent de ce fait l'application des méthodes de Runge-Kutta à des systèmes multidimensionnels (de dimension d) :

```
Definition Sc (d : nat) : Type := (R → R) → cV_d → cV_d.
```

Sur la donnée d'une fonction $\mathcal{W} \in \mathbb{R} \to \mathbb{R}$ (par exemple, il peut s'agir d'un arrondi ou de la fonction identité) et d'un vecteur $y_n \in \mathbb{R}^d$, un élément de type `Sc` renvoie un vecteur $y_{n+1} \in \mathbb{R}^d$. On peut évaluer l'application de la méthode numérique `M : Sc` sur n itérations à partir de la condition initiale `y0` (`y0_tilde` étant le vecteur `y0` dont toutes les composantes ont été "arrondies" par la fonction $\mathcal{W}$) :

```
Definition meth_iter (M:Sc) n (y0 : cV_d) (W : R → R) := iter n (M W) y0_tilde
```

Nous définissons formellement les erreurs locales et globales (en valeur absolue) :

```
Definition error_loc (M : Sc) n (y0 : cV_d) (W : R → R) (*ε_{n+1} = ‖R̃(h, Ã, ỹ_n) − R(hA)ỹ_n‖_∞*)
  := vec_norm (M W (meth_iter M n y0 W) − M (fun x ⇒ x) (meth_iter M n y0 W)).
Definition error_glob (M : Sc) n (y0 : cV_d) (W : R → R)          (*E_n = ‖ỹ_n − y_n‖_∞*)
  := vec_norm (meth_iter M n y0 W − meth_iter M n y0 (fun x ⇒ x)).
```

5 Erreurs locales

Nous démontrons un résultat générique pour borner l'erreur d'arrondi locale commise lors d'une itération de la méthode. Ce résultat permet de traiter toute méthode à un pas explicite, d'ordre quelconque, implémentée suivant une évaluation "à la Horner" (voir Section 3)

Lemme 1. *Soit* $d \in \mathbb{N}^*$, $y \in \mathbb{R}^n$, $C_1, C_2, C_3, D_1, D_3 \in \mathbb{R}_+$, $A_1, A_2, A_3 \in \mathbb{R}^{n\times n}$, $X_1, X_3 \in \mathbb{F}^d$, $\widetilde{A_2} \in \mathbb{F}^{n\times n}$. *Soit* $\rho = C_2u|||A_3|||_\infty + C_3u|||A_2|||_\infty + C_2C_3u^2$.
Soit $\mathscr{C} = C_1u + \rho + u\,(|||A_1|||_\infty + C_1u) + (u + (1+u)\gamma_d)(\rho + |||A_2|||_\infty|||A_3|||_\infty)$.
Soit $\mathscr{D} = (1+u)\left(\frac{d}{2}(1+\gamma_{d-1}) + D_1 + D_3(1+\gamma_d)(C_2 + |||A_2|||_\infty)\right)$.
Supposons que :
- $\|X_1 - A_1y\|_\infty \leqslant C_1u\|y\|_\infty + D_1\eta$.
- $|||\widetilde{A_2} - A_2|||_\infty \leqslant C_2u$.
- $\|X_3 - A_3y\|_\infty \leqslant C_3u\|y\|_\infty + D_3\eta$.

Alors $\|X_1 \oplus (\widetilde{A_2} \otimes X_3) - (A_1 + A_2A_3)y\|_\infty \leqslant \mathscr{C}\|y\|_\infty + \mathscr{D}\eta$.

En Coq, le lemme correspondant au Lemme 1 est nommé `build_bound_mult_loc`. La démonstration de ce résultat repose sur une analyse d'erreur en avant relativement naïve. La quantité $X_1 \oplus (\widetilde{A_2} \otimes X_3)$ correspond à un pas de l'évaluation "à la Horner" en arithmétique à virgule flottante (c'est-à-dire $\widetilde{R}(h, \widetilde{A}, \widetilde{y_n})$) et $(A_1 + A_2A_3)y$ correspond à l'évaluation mathématique $R(hA)\widetilde{y_n}$. Ces évaluations étant construites à partir d'évaluations de Horner plus simples, on peut reconstruire pas à pas une borne sur l'erreur locale de l'expression entière. En l'absence de dépassement de capacité inférieur, $\mathscr{D} = 0$.

Prenons l'exemple de la méthode RK2 (sans prise en compte de l'*underflow*). On instancie le Lemme 1 avec $y = \widetilde{y_n}$, $X_1 = \widetilde{y_n}$, $A_1 = I$, $\widetilde{A_2} = \circ\left[h\widetilde{A}\right]$, $A_2 = hA$, $X_3 = \circ\left[\widetilde{y_n} + \frac{h}{2}\widetilde{A}\widetilde{y_n}\right]$ et $A_3 = I + \frac{hA}{2}$. On peut vérifier que l'expression $\|X_1 \oplus (\widetilde{A_2} \otimes X_3) - (A_1 + A_2A_3)y\|_\infty$ correspond à l'erreur locale ε_{n+1} pour la méthode RK2. Il faut alors exhiber C_1, C_2 et C_3. Il apparaît de façon évidente que $C_1 = 0$ car $\|X_1 - A_1y\|_\infty = \|\widetilde{y_n} - I\widetilde{y_n}\|_\infty = 0$. Pour exhiber C_2, on borne $|||h\widetilde{A} - hA|||_\infty$, ce qui est assez naturel par dépliage de la définition de $|||\cdot|||_\infty$ et en appliquant des résultats génériques d'analyse d'erreurs d'arrondi. Enfin, il reste à exhiber C_3, *i.e.* à borner $\|X_3 - A_3y\|_\infty$. Pour y parvenir, on réapplique le Lemme 1 avec $y = X_1 = X_3 = \widetilde{y_n}$, $A_1 = A_3 = I$, $A_2 = \frac{hA}{2}$ et $\widetilde{A_2} = \circ\left[\frac{h}{2}\widetilde{A}\right]$. Il est trivial d'exhiber C_1 et C_3, C_2 étant obtenu de la même manière qu'à l'étape précédente.

Nous avons utilisé cette méthodologie pour borner les erreurs locales des méthodes d'Euler et de RK2 en *binary64* (en négligeant les *underflows*). Pour la méthode d'Euler :

$$\forall n \in \mathbb{N}^*, \varepsilon_n \leqslant (u + (u + 3{,}12\gamma_d)h|||A|||_\infty)\,\|y_{n-1}\|_\infty. \tag{1}$$

Le lemme Coq associé est nommé `Euler_loc_FLX`. Pour la méthode RK2 :

$$\forall n \in \mathbb{N}^*, \varepsilon_n \leqslant \left(u + (11{,}3u + 2{,}56\gamma_d)(h|||A|||_\infty + h^2|||A|||_\infty^2)\right)\|y_{n-1}\|_\infty. \tag{2}$$

Le lemme Coq associé est nommé `RK2_loc_FLX`. La méthodologie peut paraître fastidieuse mais il est néanmoins possible de partiellement l'automatiser. Tout d'abord, à chaque sous-application du Lemme 1, nous utilisons la tactique `eapply` de Coq, qui va permettre d'inférer automatiquement les valeurs des paramètres y, X_1, A_1, $\widetilde{A_2}$, A_2, X_3 et A_3. Il ne reste alors plus qu'à instancier manuellement les valeurs respectives de C_1, C_2 et C_3 puis à prouver les 3 hypothèses du lemme. Les bornes obtenues s'avèrent relativement lisibles (voir Équations (1) et (2)) et ne font pas apparaître de termes d'ordre supérieur (en u^2, u^3, etc). Nous avons en effet majoré ces termes à chaque étape de la construction de la borne d'erreur en utilisant la tactique automatique `interval` [16], ce qui nécessite des hypothèses sur u : en *binary64*, on a $u^2 \leqslant 2^{-53}u$ mais des hypothèses plus faibles suffisent, *e.g.* $u^2 \leqslant 0{,}01u$.

6 Erreurs globales

Dans de précédents travaux [6], un théorème général permet de construire de façon systématique une borne sur l'erreur globale d'une méthode à partir des bornes exhibées sur les erreurs locales précédentes. Nous généralisons ce résultat au cas multidimensionnel :

Théorème 2. Passage des erreurs locales à l'erreur globale
Soit $C, D \geqslant 0$. *Supposons que pour tout* $n \in \mathbb{N}^*, \varepsilon_n \leqslant C\|\widetilde{y_{n-1}}\|_\infty + D\eta$ *et que* $0 < C + |||R(hA)|||_\infty < 1$ *(condition de stabilité). Alors :*

$$\forall n, E_n \leqslant (C + |||R(hA)|||_\infty)^n \left(\varepsilon_0 + \frac{nC\|y_0\|_\infty}{C + |||R(hA)|||_\infty}\right) + nD\eta.$$

En Coq, le théorème correspondant au Théorème 2 est nommé `error_loc_to_glob`. Si l'énoncé du théorème est très proche de celui démontré dans [6], la démonstration l'est également. En effet, le choix de mener une analyse par normes a permis de généraliser le résultat appliqué aux systèmes scalaires unidimensionnels pour qu'il s'étende aux systèmes linéaires multidimensionnels, *i.e.* matriciels, en remplaçant les valeurs absolues par des normes. À partir des résultats exhibés dans la section 5, nous bornons l'erreur globale correspondant aux méthodes d'Euler et de Runge-Kutta d'ordre 2 par application directe du Théorème 2. Il suffit en effet de remplacer la variable C par la valeur correspondante dans le Théorème 2, *e.g.* $u + (u + 3{,}12\gamma_d)h|||A|||_\infty$ pour la méthode d'Euler.

7 Conclusion et perspectives

Nous avons proposé une méthodologie générique pour borner les erreurs d'arrondi associées à une certaine classe de méthodes numériques en tenant compte des dépassements de capacité inférieurs. Par rapport à nos travaux antérieurs [6], nous proposons une généralisation aux systèmes multidimensionnels *via* une analyse par normes. Cette généralisation est non-triviale car elle repose sur des résultats fondamentaux d'analyse matricielle, mais la méthodologie appliquée reste similaire. Nous instancions nos résultats à deux méthodes classiques (Euler et RK2) et exhibons des bornes qui, dans le cas de méthodes stables, sont quasi-linéaires en le nombre d'itérations. Par ailleurs, nous formalisons l'ensemble du raisonnement dans Coq. La formalisation des résultats génériques représente environ 3300 lignes de Coq en `FLX` et 3600 lignes en `FLT`. L'instanciation aux méthodes d'Euler et de RK2 représente environ 1200 lignes de Coq.

Nous envisageons de factoriser une partie des résultats commune aux formats `FLX` et `FLT`. Flocq contient en effet des résultats liant ces deux formats [7]. De plus, bien que nous nous basions sur la bibliothèque Mathematical Components, la formalisation n'utilise pas les tactiques propres à `ssreflect`, qui permettraient probablement de réduire la taille des démonstrations. Nous pourrions également combiner nos preuves à des définitions et résultats formalisés par Roux dans de précédents travaux [18], comme la notation γ (et ses propriétés), ainsi que certains résultats fondamentaux, *e.g.* une borne sur l'erreur d'arrondi du produit scalaire de deux vecteurs. Une autre perspective pour ce travail est l'extension de la méthodologie à d'autres classes de méthodes, comme les méthodes implicites et multi-pas, ou à d'autres classes de systèmes, comme les systèmes affines ou non-linéaires. Une autre perspective envisagée est la combinaison des erreurs d'arrondi et des erreurs de méthode. Pour évaluer la finesse de la borne obtenue, nous pourrions, comme dans le cas unidimensionnel [6], comparer numériquement la borne d'erreur à l'erreur d'arrondi effectivement commise. Enfin, nous envisageons de vérifier que la borne d'erreur d'arrondi obtenue est négligeable par rapport à l'erreur de méthode effectivement commise.

Références

[1] J. Alexandre dit SANDRETTO et A. CHAPOUTOT : Validated explicit and implicit Runge-Kutta methods. *Reliable Computing*, 22, 2016.

[2] Y. BERTOT, G. GONTHIER, S. O. BIHA et I. PASCA : Canonical big operators. *In International Conference on Theorem Proving in Higher Order Logics*, pages 86–101. Springer, 2008.

[3] M. BERZ et K. MAKINO : Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. *Reliable Computing*, 4(4):361–369, 1998.

[4] S. BOLDO : Floats & Ropes : a case study for formal numerical program verification. *In 36th International Colloquium on Automata, Languages and Programming*, volume 5556 de *LNCS - ARCoSS*, pages 91–102, Rhodos, Greece, juillet 2009. Springer.

[5] S. BOLDO, F. CLÉMENT, J.-C. FILLIÂTRE, M. MAYERO, G. MELQUIOND et P. WEIS : Wave Equation Numerical Resolution : a Comprehensive Mechanized Proof of a C Program. *Journal of Automated Reasoning*, 50(4):423–456, avril 2013.

[6] S. BOLDO, F. FAISSOLE et A. CHAPOUTOT : Round-off Error Analysis of Explicit One-Step Numerical Integration Methods. *In 24th IEEE Symposium on Computer Arithmetic*, pages 82–89, London, United Kingdom, juillet 2017.

[7] S. BOLDO et G. MELQUIOND : *Computer Arithmetic and Formal Proofs*. ISTE Press - Elsevier, décembre 2017.

[8] O. BOUISSOU et M. MARTEL : GRKLib : a guaranteed Runge-Kutta library. *In International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2006.

[9] L. FOUSSE : Multiple-precision correctly rounded Newton-Cotes quadrature. *ITA*, 41(1):103–121, 2007.

[10] P. HENRICI : *Error propagation for difference methods*. The SIAM series in applied mathematics. John Wiley, 1963.

[11] N. J. HIGHAM : *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd édition, 2002.

[12] IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, Aug 2008.

[13] F. IMMLER et J. HÖLZL : Numerical analysis of ordinary differential equations in Isabelle/HOL. *In Interactive Theorem Proving*, volume 7406 de *LNCS*, pages 377–392. Springer Berlin Heidelberg, 2012.

[14] F. IMMLER et C. TRAUT : The flow of ODEs : Formalization of Variational Equation and Poincaré Map. *Journal of Automated Reasoning*, 62(2):215–236, Feb 2019.

[15] A. MAHBOUBI et E. TASSI : Mathematical Components. `https://math-comp.github.io/mcb/`, 2017.

[16] G. MELQUIOND : Proving bounds on real-valued functions with computations. *In International Joint Conference on Automated Reasoning, IJCAR 2008*, Sydney, Australia.

[17] M. NEHER, K. R. JACKSON et N. S. NEDIALKOV : On Taylor model based integration of ODEs. *SIAM Journal on Numerical Analysis*, 45(1):236–262, 2007.

[18] P. ROUX : Formal Proofs of Rounding Error Bounds - With Application to an Automatic Positive Definiteness Check. *J. Autom. Reasoning*, 57(2):135–156, 2016.

Test d'un robot agricole en simulation

Clément Robert[1], Thierry Sotiropoulos[1], Hélène Waeselynck[1], Jérémie Guiochet [1], and Simon Vernhes[2]

[1]LAAS-CNRS, Université de Toulouse, Toulouse
[2]Naïo Technologies, Toulouse

Résumé

Ce document est le résumé étendu d'un article en cours de soumission à une revue. Le titre de l'article d'origine est : "The virtual lands of Oz : testing an agribot in simulation".

Mots-clés – Etude de cas industrielle, test du logiciel, simulation, génération de mondes, système autonome, robot agricole

Les missions d'un robot autonome sont typiquement testées par des expérimentations sur le terrain. Cette approche est très coûteuse et peut présenter des risques. Afin d'explorer plus de situations opérationnelles à moindre coût et sans encourir de risque, nos travaux développent des tests en simulation : le robot accomplit ses missions dans des mondes virtuels. Cet article présente une étude de cas industrielle sur la faisabilité et l'efficacité d'une telle approche.

Le système étudié est Oz, un robot agricole de désherbage développé par la société Naïo Technologies (voir la figure 1). Son logiciel a été testé dans des champs de légumes virtuels, en utilisant un simulateur 3D basé sur Gazebo. L'étude de cas a permis de se confronter à plusieurs défis : la génération aléatoire d'environnements 3D complexes, la vérification automatisée du comportement observé (oracle de test), et la fidélité imparfaite de la simulation par rapport à un test dans le monde réel. Nous présentons l'approche de test en simulation que nous avons développée, et comparons les résultats avec ceux d'expérimentations sur le terrain.

Les tests en simulation sont générés à partir d'un modèle de mondes qui comprend : (i) une vue structurée des éléments à générer (les rangées de légumes, le terrain 3D, ...) sous forme d'un diagramme UML, (ii) des contraintes sur les valeurs des paramètres de génération de ces éléments, sous forme d'une grammaire attribuée. La génération procède en deux temps. On produit d'abord une configuration valide de valeurs de paramètres (i.e., un mot de la grammaire), puis on utilise ces valeurs pour produire le contenu des mondes, dans un format directement compréhensible par le simulateur.

(a) Experimentation sur le terrain (b) Test dans un champ virtuel

FIGURE 1 – Le robot Oz en action

L'oracle de test est constitué d'un ensemble de détecteurs, chacun ciblant une propriété spécifique. On cherche à détecter des comportements anormaux tels que des collisions, le franchissement d'un seuil de vitesse, des erreurs de perception, ou encore des comportements inappropriés à la phase de mission en cours.

Les résultats de l'étude montrent le fort potentiel du test en simulation pour alléger les expérimentations coûteuses. En effet, les tests en simulation s'avèrent efficaces en dépit d'une reproduction basse fidélité de la physique du robot. Ils révèlent la plupart des fautes trouvées par les expérimentations dans le monde réel, y compris la faute ayant causé la majorité des défaillances sur le terrain. Ils révèlent également une nouvelle faute qui n'avait pas été mise en évidence jusque-là. Par contre, la simulation peut introduire des défaillances parasites qui ne se produiraient pas dans le monde réel.

Tests de politiques d'adaptation pour systèmes cyber-physiques

Jean-Philippe Gros
Institut FEMTO-ST, Univ. Bourgogne Franche-Comté, CNRS
15B avenue des Montboucons, 25030 Besançon, Cedex, France

Abstract

Cet article est dédié à la validation des politiques d'adaptation en utilisant une approche de tests à partir de modèles; à la mise en place d'un verdict basé à la fois sur la vérification à l'exécution et aux propriétés temporelles; à la détection d'inconsistances entre les politiques d'adaptation et les reconfigurations implémentées dans le système. Nous proposons un moyen d'établir un verdict de tests basé sur le respect des politiques d'adaptation ainsi que les mesures de couverture des règles. Ces verdicts nous donnent des informations sur les règles pour détecter d'éventuelles reconfigurations qui n'auraient pas dû avoir lieu, des règles avec une priorité haute qui ne sont jamais déclenchées ou des règles avec une priorité faible qui sont déclenchées trop souvent, des inconsistances dans les règles, ou une mauvaise interprétation des priorités. Les verdicts sont obtenus en analysant les traces d'exécution du système qui est stimulé par un modèle d'usage à transitions probabilistes. Afin d'illustrer notre approche, nous utilisons un système de peloton de voitures autonomes.

1 Introduction

Les systèmes adaptatifs gagnent en importance au fil des années, on les retrouve dans les véhicules intelligents, les infrastructures adaptatives (e.g smartgrids), les robots et de manière générale dans l'industrie. Ces systèmes s'adaptent en fonction des évènements internes et externes qu'ils rencontrent par le biais de reconfigurations dynamiques. Ces reconfigurations dynamiques changent l'architecture de systèmes adaptatifs [4] et sont guidées par des politiques d'adaptation. Une politique d'adaptation a pour rôle de guider le système en indiquant la pertinence de déclencher ses reconfigurations. Une politique d'adaptation est constituée de reconfigurations et d'un ensemble de règles de déclenchement des reconfigurations. Chaque règle étant composée de gardes sur la configuration du système, de priorités à l'activation et de propriétés temporelles. Les priorités, exprimées par de valeurs floues comme dans [5] indiquent la pertinence d'appliquer une reconfiguration.

Lors du développement d'un système adaptatif, il est possible de mal interpréter les choix spécifiés par les politiques d'adaptation. Actuellement, des méthodes de vérification [10] existent afin d'établir qu'un système se comporte correctement du point de vue des propriétés (temporelles) du système. Cependant, il n'existe pas de garantie sur le respect des politiques d'adaptation. Plus précisément, nous souhaitons nous assurer que l'ensemble des politiques d'adaptation ne présente pas d'incohérence. Ensuite, il faut s'assurer que ces politiques d'adaptation soient valides en terme de cohérence et de respect des propriétés fonctionnelles. Dans une autre mesure, il est possible de se convaincre de l'efficacité du système en validant des propriétés non-fonctionnelles. La validation des propriétés non-fonctionnelles consisterait à évaluer les choix des reconfigurations faites en rejouant la même séquence d'évènements. Le fait est que les propriétés et les règles de politiques d'adaptation

ne peuvent évaluer les reconfigurations du système à la volée mais plutôt a posteriori. Afin d'établir de tels verdicts, il est nécessaire d'établir une technique de validation de conformité pour exercer le système à produire des traces pertinentes.

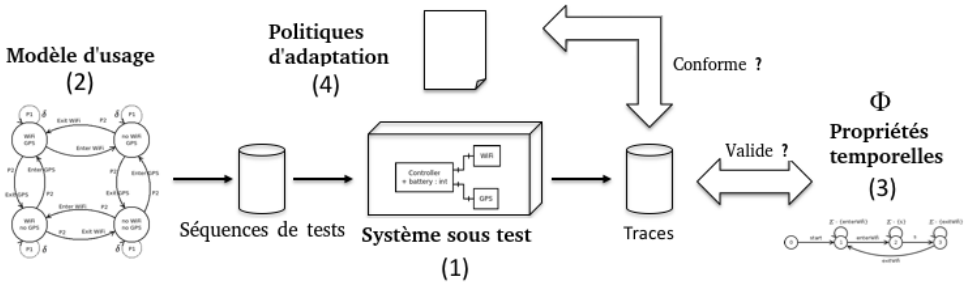


Figure 1: Approche par tests à partir de modèles pour valider les politiques d'adaptation

Afin de satisfaire ces problématiques, nous proposons une approche à partir de modèles dans la Figure. 1, qui se base sur les méthodes proposées dans [1]. Nous proposons de générer des cas de tests tout en établissant une méthode de verdict de test permettant de s'assurer du bon comportement du système sous test.

Les travaux [10] répondent à une première partie de la problématique en vérifiant la validité des propriétés temporelles en analysant rétroactivement les traces produites par le système (3). À partir de ces travaux, nous souhaitons nous assurer que les traces produites par système sous test (1) couvrent suffisamment les comportements possibles du système. Pour cela, nous proposons des critères de couverture qui s'appuient sur les propriétés temporelles (3) et les politiques d'adaptation (4). Les traces utilisées visant à satisfaire les critères de couverture sont obtenues en stimulant le système à l'aide d'un modèle d'usage (2). Ce modèle d'usage génère des séquences d'évènements grâce à un automate probabiliste. Les traces d'exécution sont alors analysées : les propriétés temporelles (3), sont vérifiées pendant l'exécution du système sous test (1). Cette étape s'effectue en utilisant une technique d'analyse du système basée sur les travaux de [8]. Les politiques d'adaptation (4) sont utilisées pour s'assurer que le système effectue les reconfigurations au bon moment. Pour cela, nous vérifions que les reconfigurations souhaitées sont bien déclenchées et qu'aucune reconfiguration non souhaitée n'est déclenchée. Nous nous intéressons également à la cohérence de la politique d'adaptation, pour cela, en se basant sur les critères de couverture, il nous est possible de détecter qu'une règle n'est jamais satisfaite. En outre de ces aspects de cohérence, nous proposons un verdict sur le respect des priorités. Ce verdict permet de détecter lorsqu'une reconfiguration avec une priorité haute n'est pas ou peu déclenchée, ou qu'une reconfiguration avec une faible priorité est trop souvent déclenchée.

Cet article est organisé de la façon suivante : La Section 2 présente le contexte scientifique et définit les notions et notations utilisées dans ce document. Dans la Section 3, nous décrivons notre approche de tests à partir de modèles ainsi que les critères de couverture associés. Nous conclurons et présenterons les futurs axes de recherche de ce travail dans la Section 4.

2 Contexte

Cette section présente le contexte dans lequel nous avons mené nos études. Celles-ci se sont portées sur un système de convoi de véhicules autonomes. Le comportement de ce système est vérifié par des propriétés temporelles et le déclenchement des reconfiguration guidé par des politiques d'adaptation.

2.1 Le convoi de véhicules autonomes

Dans ce système, les véhicules sont organisés en peloton ou individuellement. Dans chaque peloton, on trouve un leader qui se situe à l'avant. Un véhicule seul peut demander à rejoindre un peloton ou créer un nouveau peloton en se groupant avec un autre véhicule seul. Le système est soumis à des évènements internes, en effet, chaque véhicule au sein d'un peloton peut demander à sortir soit parce qu'il a atteint sa destination soit parce qu'il arrive à court d'énergie. Le véhicule désigné leader peut changer soit parce qu'un véhicule a plus d'autonomie que lui soit parce que sa destination est plus éloignée que le véhicule leader actuel. Des évènements externes peuvent avoir lieu, un conducteur peut décider de reprendre la main et quitter le convoi ou un nouveau véhicule peut arriver à portée du peloton pour une éventuelle fusion.

Nous considérons deux séquences différentes comme illustré dans la Figure 2, la séquence d'évènements externes (ou cas de tests) générée par le modèle d'usage et le chemin de reconfiguration (ou traces d'exécution) généré par le système en réponse aux évènements externes. Les séquences et chemins de reconfiguration sont régis par un tic d'horloge assurant que les évènements et les reconfigurations sont échantillonnés selon la même fréquence. Dans le cas d'un convoi de véhicules autonomes, plusieurs évènements peuvent avoir lieu. L'évènement *join* intervient quand des véhicules sont à portée pour se rejoindre, le système peut alors faire rejoindre plusieurs véhicules *acceptJoin* ou ne rien faire *refuseJoin*. Lorsqu'un utilisateur décide volontairement de quitter le peloton, l'évènement *quit* intervient, le système réagit à cet évènement par un (*acceptQuit*). Le système peut également réagir à des évènements internes et choisir de changer de leader avec la reconfiguration (*getRelay*). Si aucune reconfiguration n'a eu lieu entre deux tics d'horloge, on observe une reconfiguration *run* sur le chemin de reconfiguration.

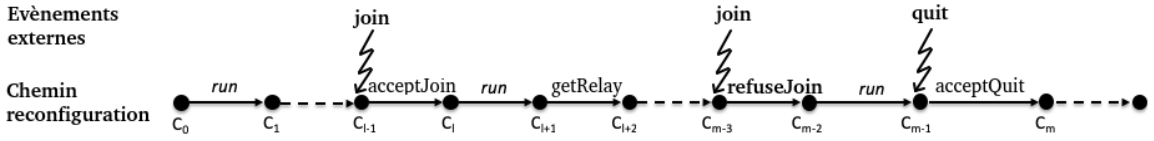


Figure 2: Séquence d'évènements et chemin de reconfiguration

2.2 Propriétés Temporelles

Dans cette section, nous présentons les propriétés temporelles FTPL[1] introduites dans [6]. Le langage FTPL est instancié avec des évènements du système pour exprimer une propriété temporelle. La sémantique de FTPL que nous utilisons se base sur les travaux de [7]. Nous proposons deux exemples liés à notre cas de peloton de véhicules autonomes :

$\varphi 1$: **after** *CreatePlt* **before** *DeletePlt* **always** *PlatoonId.VehicleNb* > 2
Cette propriété s'applique pour chaque convoi de véhicules. Elle vérifie qu'après la création d'un convoi il y a toujours au moins deux véhicules jusqu'à la suppression du peloton.

$\varphi 2$: **after** *Join* **before** *Quit* **always** *VehicleId.Battery* > 5 and *VehicleId.Distance* > 2
Cette propriété s'applique pour chaque *VehicleId*. Elle s'assure qu'après l'entrée du véhicule dans un convoi, les niveaux de batterie et distance restante du véhicule sont supérieurs à respectivement 5% et 2km, jusqu'à la sortie du véhicule.

[1]FTPL vient de la fusion entre TPL (Temporal Pattern Language) et le préfixe 'F' comme 'First order logic'.

Comme nous le montrent ces exemples, une propriété FTPL est constituée entre autres de mots clés temporels (**after, before, always, eventually**), d'évènements (*CreatePl*, *DeletePl*) et de propriétés de configuration (*Distance* > *20*, *Battery* > *5*, *VehicleNb* > *2*). Ces propriétés s'inscrivent dans le contexte global de la route qui inclut les voitures et leurs convois distingués grâce aux identifieurs (*VehicleId* et *PlatoonId*). Ces propriétés assurent à l'exécution que le système se comporte correctement. Toutefois, il est possible que certaines règles soient évaluées à potentiellement vrai (resp. potentiellement faux) dans le cas d'un **always** (resp. **eventually**) qui est évalué à vrai (resp. faux) jusqu'à un instant t mais qui demande d'être évalué à vrai jusqu'à (resp. vrai au moins une fois avant) un instant t'.
Les propriétés sont utilisées pour exprimer des exigences sur le système mais elles seront aussi dans la définition des politiques d'adaptation que nous présentons maintenant.

2.3 Politiques d'adaptation

Des politiques d'adaptation ont été définies dans [2, 5, 7], nous les avons redéfinies afin de les adapter à notre modèle. Dans notre approche, les opérations de reconfiguration sont gardées par des séquences spécifiques d'évènements.
Reprenons notre exemple de convoi de véhicules autonomes, afin de garantir la propriété $\varphi 2$, nous devons réfléchir aux reconfigurations à effectuer. C'est le rôle des politiques d'adaptation et c'est au niveau de leurs règles de reconfiguration que ces reconfigurations sont décidées. Deux exemples de règles de reconfiguration sont données en Figure 3.

```
when (after Join before Quit and VehicleId.battery < 33)
   if (state = leader ) then
        utility of PassRelay is high

when (after Join before Quit and VehicleId.battery > Leader.battery)
   if (state = platooned ) then
        utility of GetRelay is medium
```

Figure 3: Deux règles de reconfiguration du convoi de véhicules autonomes

Ces règles de reconfiguration stipulent, dans un premier temps, la propriété FTPL à valider, elle est décrite après l'utilisation du mot-clé `when` qui délimite l'intervalle d'évènements durant lequel la reconfiguration peut avoir lieu. Dans un second temps, les règles de reconfiguration contiennent une garde sur la configuration du système, qui est décrite après l'utilisation du mot-clé `if` qui définit la configuration courante du système qui peut entraîner une reconfiguration. Pour finir, le nom de la reconfiguration R_N et son utilité F est renseignée sous forme de valeur floue (`high, medium, low`) après le mot clé *utility* afin d'associer une utilité avec un nom d'opération de reconfiguration.
Ces deux exemples s'appliquent aux véhicules et servent à déterminer les moments de passage de relais entre le véhicule leader et un autre véhicule du convoi. Dans le premier cas, la reconfiguration `PassRelay` se déclenche lorsque le leader n'a plus assez de batterie et dans ce cas il doit être rétrogradé. Dans le second cas, la reconfiguration `GetRelay` se déclenche lorsque l'autonomie du véhicule courant est supérieure à celle du leader.
L'implémentation des politiques d'adaptation et plus particulièrement les règles de reconfiguration sont laissées à la discrétion des développeurs qui sont susceptibles de mal interpréter la spécification. Une mauvaise implémentation peut mener le système à se reconfigurer lorsque ce n'est pas nécessaire, ne pas se reconfigurer ou choisir la mauvaise reconfiguration et engendrer une violation de propriété. Le processus de tests décrit dans la prochaine section a pour but de répondre à ce problème en validant que les politiques d'adaptation sont correctement écrites et retranscrites à l'exécution du système.

3 Validation de l'implémentation vis à vis des politiques d'adaptation

Dans cette section, nous présentons les contributions apportées lors de la thèse dont l'ensemble est résumé Fig. 1. Les politiques d'adaptation influencent le système, il est donc nécessaire de s'assurer que le système implémente correctement ces politiques d'adaptation. Cela consiste à vérifier de manière indépendante que le système se reconfigure avec les bonnes reconfigurations au bon moment. Le système doit également toujours répondre aux propriétés fonctionnelles. Nous proposons également un moyen de nous assurer de la conformité des priorités avec la spécification.

Une des difficultés de cette approche est que les vérifications de propriétés sont faites à posteriori et durant l'exécution. En effet, les propriétés temporelles doivent atteindre leur état final pour être évaluées. De la même façon, les reconfigurations déclenchées par le système guidées par les politiques de reconfiguration peuvent mener à une erreur mais après un certain délai. Pour répondre à ces problématiques, nous proposons une analyse à l'exécution des traces produites par le système. Afin de garantir la pertinence de ces traces, nous élaborons des critères de couverture garantissant que le système a bien exploré les propriétés et les politiques d'adaptation. Tout d'abord, nous présentons une méthode de génération de tests à partir d'un modèle d'usage indépendant en simulant l'environnement extérieur au système. Ce modèle a pour but de produire les traces pertinentes pour les critères de couverture définis. Pour finir, nous expliciterons les verdicts de test mis en place et plus précisément les verdicts sur le respect des politiques d'adaptation.

3.1 Critères de couverture

Nos critères de couverture pour les propriétés FTPL s'appuient sur la définition d'un automate associé à ses propriétés et adaptent les précédents travaux de [3].

Definition 1 (Automate de propriétes FTPL)**.** *Soit un automate noté A_φ composé du quadruplet $\langle Q, q_0, Q_f, T\rangle$ avec Q l'ensemble des états, $q_0 \in Q$ est l'état initial, $Q_f \subset Q$ est l'ensemble des états finaux, et $T : Q \times Ev_\varphi \to Q$ est la fonction de transition avec Ev_φ l'ensemble des évènements possibles dans la propriété.*

Nous allons nous appuyer sur les exemples de la Sect. 2.2 et particuliérement la propriété :

after *Join* **before** *Quit* **always** *VehicleId.Battery* > 5 and *VehicleId.Distance* > 2
La création de l'automate visible Fig. 4 est déterminée par les mots-clés temporels et leurs évènements associés. Notre automate est donc constitué des transitions *join* et *quit*. ϵ représente l'ensemble des transitions possibles. L'état final est atteint lorsque le scénario de la propriété a été complété, dans notre exemple lorsqu'un véhicule est entré et sorti du convoi.

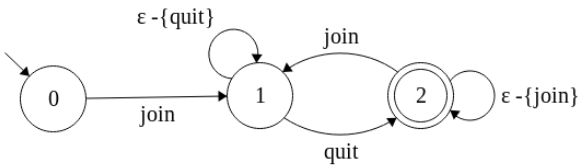


Figure 4: Automate de la propriété φ_1

Definition 2 (Couverture des propriétés FTPL)**.** *Soit une propriété φ et A_φ son automate associé. Un état de l'automate A_φ est dit* couvert *si tous ses couples de transitions entrantes et sortantes ont été*

parcourus par le système. À noter qu'une transition sortante d'un état et entrante dans le même état ne compte pas en tant que transition sortante ou entrante. Ce critère de couverture permet d'assurer une meilleure couverture qu'en se basant uniquement sur le parcours de chaque état ou transition. Une propriété est alors dite couverte *lorsque tous les états de son automate associé ont été couverts.*

Dans notre exemple Fig. 4, pour que A_φ soit *activé*, les trois couples de transitions $0 \xrightarrow{join} 1 \xrightarrow{Quit} 2$, $1 \xrightarrow{Quit} 2 \xrightarrow{Join} 1$ et $2 \xrightarrow{Join} 1 \xrightarrow{Quit} 2$ doivent être sensibilisés.

Definition 3 (Couverture des politiques d'adaptation)**.** *Une règle de politique d'adaptation est dite* activable *si sa garde (partie* `if`*) et sa propriété de déclenchement (partie* `when`*) sont évaluées à vrai et si l'opération de reconfiguration associée est exécutée par le système. Une politique d'adaptation est dite* couverte *lorsque toutes ses règles ont été activées.*

Maintenant que les critères de couverture ont été définis, nous présentons le modèle d'usage permettant de les satisfaire.

3.2 Modèle d'usage et génération de tests

Le modèle d'usage a pour but de produire des séquences d'évènements externes qui exercent le système. Les systèmes adaptatifs réagissent d'une part aux évènements externes et d'autre part aux évolutions de leurs variables internes en accord avec les politiques d'adaptation qui guident à quel moment déclencher les reconfigurations. En d'autres termes, le système se comporte à la manière d'une boîte noire ce qui rend la définition de son modèle impossible. C'est pourquoi nous considérons un modèle d'usage du système sous test. Ce modèle d'usage ne va pas représenter le comportement du système mais simuler l'environnement dans lequel le système évolue. Ce type de modèle a pour but de spécifier les différents évènements qui peuvent avoir lieu dans l'environnement. Pour rappel, nous considérons les systèmes régis par des tics d'horloge, et donc, en plus des évènements externes, nous définissons la notion de *délai*. Cela se traduit par une absence d'évènement externe durant une certaine période, mais de son côté, le système continue d'évoluer et de mettre à jour son état interne. L'automate dans la Figure 5 représentant le modèle d'usage est défini de la même façon qu'un automate de propriété temporelle mais à une différence : la notion de *délai* est présente dans l'automate du modèle alors que l'automate de propriété ne garde que les évènements qui agissent sur sa propriété. L'automate génère pendant l'exécution (en ligne) ce qui permet à l'automate d'être connecté au système. En effet, en plus d'envoyer les séquences d'évènements externes au système, l'automate peut observer les évènements internes du système. Ces concepts sont illustrés Figure 5.

L'exemple des voitures autonomes utilise trois évènements externes marqués par des transitions en

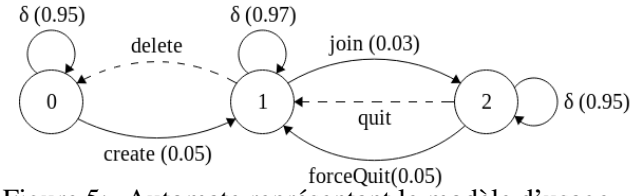


Figure 5: Automate représentant le modèle d'usage

lignes pleines : (*create*) crée un véhicule en l'ajoutant sur la route, (*join*) est déclenché lorsque le véhicule est à portée d'un ou plusieurs véhicules et (*forceQuit*) est déclenché lorsqu'un véhicule veut sortir de son convoi. Les évènements marqués par des transitions en tiret sont des évènements internes observables par le modèle d'usage, ils servent à garantir la cohérence du modèle. En effet,

l'évènement (*delete*) est déclenché lorsqu'un véhicule arrive à destination et ne peut avoir lieu avant la création de ce véhicule. De la même façon, un véhicule peut quitter le convoi avec l'évènement (*quit*) suite à une reconfiguration déclenchée par le système, dans ce cas l'automate d'usage peut à nouveau déclencher l'évènement (*join*).
A noter que l'automate est discret, et donc le *délai* représente une unité de temps. Dans ces conditions, pour générer une séquence d'évènements, nous utilisons un algorithme de type 'Markov random walk' [9] qui consiste à laisser la séquence être générée par parcours aléatoire de l'automate probabiliste. Le processus de génération continue de générer des évènements jusqu'à atteindre les critères de couverture. Avec ces cas de tests, le système est stimulé et peut déclencher des changements internes ou des reconfigurations dans le système. Le système produit une trace de reconfiguration qui est analysée selon le respect des politiques d'adaptation et propriétés FTPL. Cette analyse, les métriques et leurs verdicts sont décrits maintenant.

3.3 Verdicts de tests

L'établissement d'un verdict de test s'appuie sur deux artefacts. La première étape consiste à vérifier les propriétés FTPL. La seconde étape s'occupe de vérifier les conditions d'exécution des reconfigurations. Ce travail se base sur les travaux [8] sur la vérification des propriétés FTPL.

Verdict sur les propriétés FTPL Intuitivement, une propriété FTPL *passe le verdict* pour toute séquence de longueur *i*, si l'état à l'indice *i* est le dernier état final de la propriété FTPL et que la propriété est vérifiée. Le verdict est basé sur le dernier état final de la séquence et donc toute séquence sans état final est dite *inconclusive*. La propriété est dite *violée* s'il existe un état ne respectant pas la propriété peu importe si l'état est final ou non.

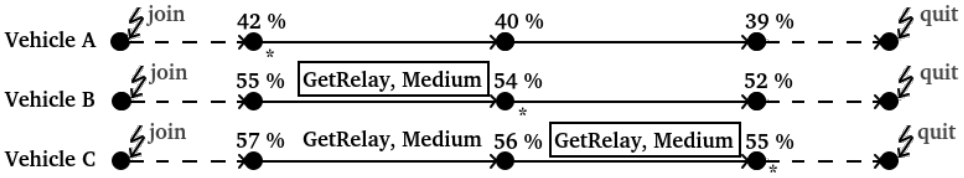


Figure 6: Séquence de reconfigurations observables

Nous définissons à partir des exemples présentés dans la Secion 2.3 notre deuxième partie de verdict basé cette fois sur les politiques d'adaptation en essayant de détecter les reconfigurations non souhaitées.
La Figure 6 montre un exemple de traces produites par trois véhicules au sein d'un même convoi. Les évènement externes *join* et *quit* sont indiqués de la même façon que dans la Figure 2, les reconfigurations internes (*GetRelay*) sont associées à une priorité (*Medium*). À chaque état, le niveau d'énergie restante est indiqué par un pourcentage et le signe $*$ signifie que le véhicule est *leader*. Entre chaque état, les reconfigurations possibles ainsi que leur priorité sont affichées. La reconfiguration choisie par le système est encadrée.
On appelle *eligible* une règle de reconfiguration qui est activable. On appelle *actual* règle de reconfiguration activée. La propriété qui en découle vérifie que la reconfiguration *actual* est présente dans la liste des reconfigurations éligibles. Dans notre exemple, GetRelay est *eligible* pour les véhicules B et C donc, on s'attend à voir une telle reconfiguration être activée, contrairement à PassRelay définie Figure 3 qui déclencherait une erreur si elle était activée dans ce cas de figure.
En complément de ce verdict, nous proposons des métriques sur les règles de reconfiguration : $\#elig$ est donné par le nombre de fois qu'elle a été désignée *eligible* et $\#actual$ est donné par le nombre de

fois qu'une règle est désignée *actual*. Ces métriques nous permettent d'effectuer des ratios permettant de comparer le taux de déclenchement. Si une règle à priorité élevée possède un taux de déclenchement plus faible qu'une règle à priorité basse c'est qu'il existe potentiellement une incohérence dans le déclenchement des reconfigurations ou la définition des politiques d'adaptation.

4 Conclusion et axes de recherche

Ce papier a présenté une approche de tests à partir de modèles qui a pour but de vérifier qu'un système adaptatif implémente fidèlement les politiques d'adaptation spécifiées indépendamment. Cette approche se base sur un modèle d'usage générant des séquences de tests permettant de diriger le système sous test. La validation du système repose sur plusieurs verdicts : le respect des propriétés temporelles ainsi que l'évaluation de la légitimité des reconfigurations. Étant donné que nos mesures et vérifications sont effectuées à l'exécution, la génération des séquences de tests peut être effectuée avant ou pendant l'exécution. Cette approche a été implémentée sur un programme Java modélisant le convoi de véhicules autonomes décrit dans ce papier. Les résultats obtenus valident, pour cet exemple l'approche décrite. Les travaux futurs se dirigent vers une validation des propriétés non fonctionnelles du système en analysant quels paramètres peuvent être modifiés pour mieux répondre à des questions d'efficacité. Nous préparons des expérimentations sur d'autres cas d'étude.

References

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

[2] F. Chauvel, O. Barais, N. Plouzeau, I. Borne, and J.-M. Jézéquel. Expression qualitative de politiques d'adaptation pour Fractal. In Y. Aït Ameur, editor, *2ème Conf. sur les Architectures Logicielles (CAL 2008), 3-7 Mars 2008, Montréal, Québec, Canada*, volume RNTI-L-2 of *Revue des Nouvelles Technologies de l'Information*, page 119. Cépaduès-Éditions, 2008.

[3] F. Dadeau, K. Cabrera Castillos, and J. Julliand. Coverage criteria for model-based testing using property patterns. In A.K. Petrenko and H. Schlingloff, editors, *MBT 2014, 9th Workshop on Model-Based Testing, Satellite workshop of ETAPS 2014*, volume 141, pages 29–43, Grenoble, France, apr 2014.

[4] R. De Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N.M. Villegas, T. Vogel, et al. Software engineering for self-adaptive systems: A second research roadmap. In *Software Engineering for Self-Adaptive Systems II*, pages 1–32. Springer, 2013.

[5] J. Dormoy and O. Kouchnarenko. Event-based adaptation policies for Fractal components. In *AICCSA 2010, ACS/IEEE Int. Conf. on Computer Systems and Applications*, pages 1–8, Hammamet, Tunisia, may 2010.

[6] J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In L. Barbosa and M. Lumpe, editors, *FACS*, volume 6921 of *LNCS*, pages 200–217. Springer Berlin Heidelberg, 2012.

[7] O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS, 10th Int. Symp. on Formal Aspects of Component Software*, volume 8348 of *LNCS*, pages 234–253. Springer, 2014.

[8] O. Kouchnarenko and J.-F. Weber. Decentralised evaluation of temporal patterns over component-based systems at runtime. In I. Lanese and E. Madelaine, editors, *Formal Aspects of Component Software*, volume 8997 of *LNCS*, pages 108 – 126, Bertinoro, Italy, sep 2015. Springer.

[9] A. Sinclair. *Algorithms for Random Generation and Counting: A Markov Chain Approach*. Birkhauser Verlag, Basel, Switzerland, Switzerland, 1993.

[10] Jean-François Weber. *Guider et contrôler les reconfigurations de systèmes à composants: Reconfigurations dynamiques: modélisation formelle et validation automatique. (Guide and control component systems-based system reconfigurations: Dynamic reconfigurations: formal modelling and automatic validation).* PhD thesis, University of Franche-Comté, Besançon, France, 2017.

Towards a Test-and-Proof Framework for C11 in Isabelle/HOL

Burkhart Wolff

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
Burkhart.Wolff@lri.fr

Abstract

We report on an integration of a novel C11 Frontend into Isabelle/HOL enabling different semantic backends (AutoCorres, Securify, IMP2, Clean, ...). We discuss the challenges of a Generic Framework ranging from IDE to Proof-Support, and show how a small semantic backend can be hooked into our Framework enabling both deductive program verification as well as program-based Test-Generation.

Ingénierie dirigée par les foncteurs
Développement d'unikernels à large échelle

Gabriel Radanne[a], Anil Madhavapeddy[b], Jeremy Yallop[b], Thomas Gazagnaire[d], Richard Mortier[b], Hannes Mehnert[c], Mindy Preston[c], et David Scott[e]

[a]University of Freiburg
[b]University of Cambridge
[c]Robur
[d]Tarides
[e]Docker, Inc

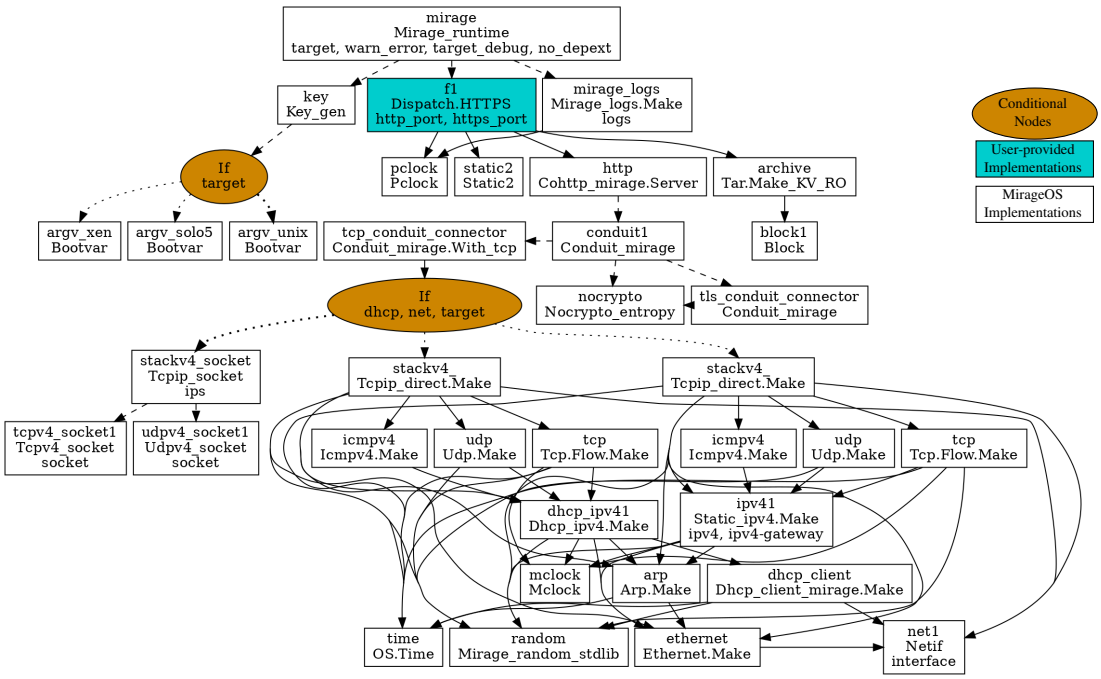


FIGURE 1 – Graphe de configuration pour un serveur Web MirageOS – https://mirage.io

Ceci est un résumé long pour l'article « Programming Unikernels in the Large via Functor Driven Development » [Radanne et al., 2019].

Une source majeure de complexité dans le développement d'application modernes est l'exécution sur un nombre croissant de plates-formes : les systèmes d'exploitation conventionnels, tels Linux et Windows, let systèmes mobiles tels Android ou iOS, les systèmes embarqués tels ARMs ou les microcontroleurs RISC-V, ou encore les navigateurs Web qui exposent des machines virtuelles Javascript ou Webassembly. Concevoir un modèle de programmation pour de tels environnements hétérogènes est difficile, étant donné leurs différences en termes de gestion de la mémoire, d'isolation, d'entrée/sortie et d'ordonnancement. La plupart des tentatives pour adapter les modèles communs (c.a.d, POSIX) à ces environnements ont menés à la création de bibliothèques système "mini-libc" qui forment un plus petit dénominateur commun. Cette solution est peu satisfaisante : il serait préférable de générer des exécutables spécialisés pour chaque cible de compilation afin d'utiliser pleinement ses fonctionnalités

spécifiques. Idéalement, le programmeur utiliserait un ensemble d'interfaces modulaires permettant aux applications de dépendre de différentes fonctionnalités qui seraient ensuite fournies par chaque plate-forme.

Une étape importante pour atteindre cet objectif est l'utilisation de "bibliothèques systèmes d'exploitation" qui décomposent les composants d'un noyau monolithique en bibliothèques qui sont ensuite assemblées avec le code de l'application lui-même [Leslie et al., 1996, Engler et al., 1995]. Quand un tel noyau est lié avec une application et un chargeur de démarrage, on obtient un *unikernel*, une application indépendante qui contient ses propres pièces de système d'exploitation [Madhavapeddy et al., 2013]. L'unikernel est également spécialisé pour une plate-forme donnée, ce qui fournit également des bénéfices significatifs en termes de performance et de taille [Madhavapeddy et al., 2015, Manco et al., 2017].

Ces dernières années ont vu l'apparition de nombreux framework pour unikernel dans différents langages de programmation de haut niveau. Le nombre de bibliothèques implémentant un composant donné a également augmenté en conséquence, menant à une nouvelle difficulté pratique : *Comment éviter aux développeurs de sélectionner l'ensemble exact de bibliothèques nécessaire par la plate-forme cible ?* L'approche courante qui consiste à dépendre d'une interface monolithique complète (par exemple, pour OCaml, le module `Unix`), ne passe pas à l'échelle dans les environnements hétérogènes modernes.

Cet article décrit comment nous avons résolu ces difficultés en utilisant les puissantes capacités d'abstraction et de modularité d'OCaml dans le cadre du framework unikernel MirageOS. MirageOS est un ensemble de bibliothèques système d'exploitation développé depuis 2006 et a été largement déployé dans des projets industriels tels Xen [Scott et al., 2010, Gazagnaire and Hanquez, 2009] et Docker. Au cours des 13 dernières années, MirageOS s'est enrichi et fournit un ensemble très divers de plate-formes cibles, y compris des virtualisateurs tels Xen [Barham et al., 2003], KVM [Kivity et al., 2007] et Muen [Buerki and Rueegsegger, 2013], des exécutables Unix et Windows conventionnels, et des cibles de compilation plus expérimentales tels que Javascript ou les systèmes embarqués sur des architectures RISC-V ou ARM.

Le défi principal pour maintenir cette panoplie de cibles de compilation à été d'empêcher les programmeurs OCaml, usuellement méticuleux dans l'utilisation de l'abstraction, d'accéder aux interfaces systèmes monolithiques, telles que le module `Unix` de la bibliothèque standard, qui lierait irrémédiablement l'application à un environnement d'exécution fixé. À la place, MirageOS utilise le système de modules ML pour permettre aux programmeurs d'abstraire l'utilisation des différentes fonctionnalités systèmes (par exemple la gestion du temps, du réseau, du stockage ou de l'entropie). Au lieu d'appeler la `libc`, le code de l'application est paramétré en terme des fonctionnalités systèmes utilisés via les modules paramétrés, ou foncteurs, d'OCaml. L'outil MirageOS va ensuite fournir les bibliothèques appropriées pour la plate-forme cible. Ces bibliothèques peuvent simplement invoquer les appels systèmes Unix, ou alors réimplémenter complètement des composants noyau tel TCP/IP pour les plateformes ne disposant pas de système d'exploitation traditionnel, comme les systèmes embarqués.

Avec cette technique, le développeur écrit une application qui peut être efficacement compilée vers tous ces différents environnements simplement en rendant ses dépendances explicites via la paramétrisation. Les bases de code résultantes sont hautement structurées et faciles à compiler vers des cibles de déploiement futures. Par exemple, le site web de MirageOS (lui-même écrit à l'aide de MirageOS) est présenté Figure 1 et est constitué d'un module utilisateur, représenté en cyan, et de différents composants issus de l'écosystème MirageOS ainsi que des noeuds conditionnels permettant d'adapter la configuration aux cibles de compilation. En particulier, la partie basse du diagramme représente trois piles réseau. Le choix de la pile réseau concrète est fait par l'outil `mirage` pendant l'étape de configuration. Si l'utilisateur veut tester son application sur une machine de développement UNIX, une implémentation « pass-through » basée sur les sockets est utilisée. Si l'utilisateur veut une application indépendante avec un DHCP, une reimplémentation complète de la pile réseau est utilisée.

Nous baptisons cette approche « Ingénierie dirigée par les Foncteurs » et faisons les contributions

suivantes :

— nous décrivons une structure d'application portable qui encourage les programmeurs à spécifier leurs dépendances systèmes via les modules d'OCaml : structures, signatures et foncteurs (fonctions de modules) ;

— nous montrons comment utiliser des techniques de méta-programmation pour générer le code liant configuration, compilation et déploiement de l'application. Pour ce faire, nous utilisons un langage dédié embarqué pour exprimer les dépendances entre l'application et les implémentations concrètes des plateformes cibles ;

— Nous évaluons notre utilisation des modules OCaml à large échelle dans le cadre de MirageOS.

Un serveur de fichier modulaire

Pour démontrer notre approche, nous considérons le cas d'un serveur de fichiers modulaire. Un serveur de fichiers nécessite deux composants système : un système de stockage de fichier et un accès réseau. Afin de rendre notre approche plus modulaire, Figure 2 présente une implémentation d'un serveur de fichiers ou ces deux composants ont été abstraits. Le développeur peut alors fournir différentes implémentations. Par exemple, Figure 3 présente deux implémentations de la signature `Network` : la première via une connexion TCP directe, la deuxième via une pile HTTP. Chaque implémentation contient sa propre fonction d'initialisation adaptée à ses besoins : `TPCIP.create`, par exemple, demande un port sur lequel écouter. De même, le stockage de fichier peut être implémenté via un système de fichier, stocké directement en mémoire, via un serveur FTP, etc.

```
module type Store = sig
  type t
  val read: t -> string -> string
end
module type Network = sig
  type t
  val listen:t -> (string -> string) -> unit
end
```

```
module Make (S: Store) (N: Network) = struct
  let start storage network =
    N.listen network (S.read storage)
end
```

FIGURE 2 – `Server_modular` – Un serveur de fichier modulaire

```
module TCPIP : sig
  include Network
  val create : int -> t
end
```

(a) `TCPIP` implémente `Network`.

```
module HTTP (N : Network) : sig
  include Network
  val create : N.t -> t
end
```

(b) `HTTP` implémente `Network`.

FIGURE 3 – Examples d'implémentations réseau

Afin de composer et compiler l'application, le développeur pourrait simplement choisir un ensemble d'implémentations et écrire le code d'assemblage manuellement. Ceci est peu flexible : changer une implémentation nécessiterai alors de réécrire le code-glu associé. De plus, ce code glu, bien que très facile à écrire, croit rapidement en complexité avec la taille de l'application (tels que celle présentée en Figure 1).

Plutôt que de procéder à l'assemblage manuel, nous proposons d'utiliser *Functoria*, un outil de configuration pour assembler des applications modulaires. Functoria fournit un outil en ligne de commande qui prend en argument l'ensemble des paramètres de l'application :

— `functoria configure --store direct --fs /my/files -p 42`
configure l'application pour servir "/my/files" sur un socket au port 42.

— `functoria configure --store memory --fs /my/files -p 80`
configure l'application pour créer un serveur HTTP au port 80 qui stocke "/my/files" en mémoire.

Pendant la configuration, functoria génère le fichier `main.ml` qui contient le code-glu d'assemblage des modules et d'initialisation de l'application. Il suffit ensuite d'invoquer `make` pour compiler l'application. Functoria via également interagir avec le gestionnaire de paquet pour installer toutes les dépendances nécessaires (par exemple, l'implémentation d'un serveur HTTP dans le deuxième cas). Tout ceci est décrit via un fichier de configuration présenté Figure 4 et représenté visuellement en Figure 5.

```
let default_network : network impl =
  let is_http =
    Key.(pure (fun x -> x = 80 || x = 8080) $ value port)
  in if_ is_http (http $ tcpip) tcpip

let make_server =
  foreign "Server_modular.Make"
    (store @-> network @-> job)

let my_server =
  make_server $ default_store "data/" $ default_network
```

FIGURE 4 – `config.ml`

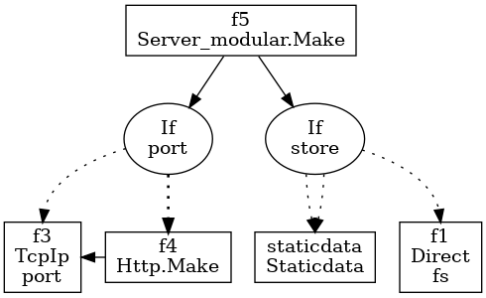


FIGURE 5 – Graphe de configuration

Dans le cadre de cet exposé, nous présenterons notre approche en pratique, décrirons les détails du langage de configuration et donnerons un aperçu des difficultés théoriques sous-jacentes, suivant l'article complet [Radanne et al., 2019]. Nous utiliserons aussi cette occasion pour présenter le projet MirageOS, qui est relativement peu connu de la communauté française des méthodes formelles.

Références

P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5) :164–177, Oct. 2003. ISSN 0163-5980. doi : 10.1145/1165389.945462.

R. Buerki and A.-K. Rueegsegger. Muen : An x86/64 separation kernel for high assurance. 2013. URL https://muen.sk/.

D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel : an operating system architecture for application-level resource management. In *Proc. 15th ACM Symposium on Operating Systems Principles (SOSP)*, pages 251–266, Copper Mountain, Colorado, USA, 1995. ACM. ISBN 0-89791-715-4. doi : 10.1145/224056.224076. URL http://doi.acm.org/10.1145/224056.224076.

T. Gazagnaire and V. Hanquez. Oxenstored : An efficient hierarchical and transactional database using functional programming with reference cell comparisons. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 203–214, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-332-7.

A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. KVM : the Linux Virtual Machine Monitor. In *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07*, 2007.

I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. T. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal of Selected Areas in Communications*, 14(7) :1280–1297, 1996.

A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels : Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1870-9. doi : 10.1145/2451116.2451167.

A. Madhavapeddy, T. Leonard, M. Skjegstad, T. Gazagnaire, D. Sheets, D. Scott, R. Mortier, A. Chaudhry, B. Singh, J. Ludlam, J. Crowcroft, and I. Leslie. Jitsu : Just-In-Time Summoning of Unikernels. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 559–573, Oakland, CA, 2015. USENIX Association. ISBN 978-1-931971-218. URL https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/madhavapeddy.

F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi : 10.1145/3132747.3132763. URL http://doi.acm.org/10.1145/3132747.3132763.

G. Radanne, T. Gazagnaire, A. Madhavapeddy, J. Yallop, R. Mortier, H. Mehnert, M. Preston, and D. Scott. Programming unikernels in the large via functor driven development. 2019. URL https://arxiv.org/abs/1905.02529.

D. Scott, R. Sharp, T. Gazagnaire, and A. Madhavapeddy. Using functional programming within an industrial product group : Perspectives and perceptions. *SIGPLAN Not.*, 45(9) :87–92, Sept. 2010. ISSN 0362-1340. doi : 10.1145/1932681.1863557.

Qbricks, un environnement pour la vérification formelle en informatique quantique*

Christophe Chareton, Sébastien Bardin, François Bobot, Valentin Perelle
CEA, LIST, Université Paris-Saclay, France
firstname.lastname@cea.fr

Benoît Valiron
LRI, CentraleSupélec, Université Paris-Saclay, Orsay, France
benoit.valiron@lri.fr

1 Introduction

L'informatique quantique est un domaine actif de recherche, avec comme enjeu un possible bouleversement des capacités de calculs dans certains domaines clés (optimisation, cryptographie, simulation physique, etc). Alors que les progrès sont sensibles côté matériel, il est temps de s'intéresser aux chaînes de développement logiciel qui accompagneront ces machines. Un premier pas dans cette direction est le développement de langages de programmation quantique [1, 7, 11]. Les utilisateurs et développeurs de ce type de langages ont la tâche difficile de créer des langages et des programmes qu'ils ne peuvent pas tester ou déboguer facilement. En effet, de par la nature du calcul quantique, toute observation d'un registre au cours d'une exécution est probabiliste, et par ailleurs elle détruit inévitablement l'état du registre qu'elle observe. Il n'est donc pas possible de tester un programme par des exécutions partielles ou des tests de valeurs sur des variables en cours d'exécution. Enfin, l'informatique quantique est par nature destinée à implémenter des algorithmes à des échelles non simulables classiquement.

Assurer la correction des programmes quantiques est donc un défi d'importance cruciale.

État de l'art. Différentes approches ont été proposées pour répondre à ce défi. Une première approche consiste à étendre des techniques de *model-checking* au domaine quantique [6, 15]. Cette approche utilise des spécifications non paramétrées, elle est donc limitée dans son utilisation par la taille des registres quantiques. Une seconde approche est le langage Qwire [11, 12], un langage fonctionnel embarqué dans l'assistant de preuves Coq. Qwire bénéficie du puissant système de types dépendants de Coq. Un premier problème pour cette approche est le manque d'automaticité : dans Coq, les preuves sont vérifiées automatiquement, mais elles doivent être

*Les travaux présentés dans cet article ont été financés par l'Agence Nationale de la Recherche (ANR), projet SoftQPro, ANR-17-CE25-0009

intégralement écrites par l'utilisateur. La deuxième difficulté vient de la combinaison des types et des spécifications : les annotations de types sont complexes et les preuves doivent être données pour permettre au programme de compiler. Une dernière approche réside dans l'extension de la logique de Hoare (QHL) à la programmation quantique [4, 14]. QHL [14] interprète les opérateurs de densité comme des *prédicats quantiques* qui sont des atomes pour une logique de type Hoare. Des efforts sont actuellement en cours pour la génération automatique d'invariants dans QHL [16] et la preuve de théorèmes pour QHL en Isabelle/HOL [9]. Le défaut de l'approche QHL réside dans son formalisme de spécification, très restrictif puisque les prédicats exprimables y sont limités aux opérateurs positifs.

Objectifs. Notre but est de proposer une approche de vérification formelle des programmes quantiques répondant aux différentes limitations perçues dans les approches existantes. Plus spécifiquement nous voulons :

— des spécifications intuitives, séparant clairement l'écriture d'un programme et sa spécification,

— des outils pour automatiser autant que possible l'écriture de ces spécifications.

Approche et premiers résultats. Nous développons *Qbricks*, un environnement purement fonctionnel de programmation et de vérification. La vérification étant paramétrée par la taille des registres de données, cette approche a le même coût quelle que soit la taille de ces registres et ne pose donc pas de problème de passage à l'échelle. *Qbricks* est un langage dédié embarqué dans un outil robuste et éprouvé dans différents contextes industriels : Why3 [3, 5], conçu pour prouver des propriétés de programmes classiques. Nous présentons ici de manière générale le modèle de calcul quantique standard (Section 2) et notre méthodologie (Section 3). Pour *Qbricks*, nous avons implémenté la sémantique standard, matricielle, ainsi qu'une sémantique alternative proposée en 2018 et qui repose sur le formalisme des sommes de chemins [2]. Nous avons comparé l'usage de ces deux sémantiques et prouvé formellement leur équivalence (Section 4). À titre de preuve de concept, nous avons par ailleurs développé et prouvé la correction dans chacune de ces deux sémantiques d'une des primitives majeures du calcul quantique : la transformée de Fourier quantique (Section 5).

2 Informatique quantique et modèle Qram hybride

Information quantique. Là où un bit d'un ordinateur classique est toujours dans un état parmi deux possibles (typiquement noté **0** et **1**), son équivalent quantique, un *qubit* peut être préparé et manipulé dans un état qui est une superposition de ces deux états. Plus précisément, la convention d'écriture désignant respectivement les états **0** et **1** par $|0\rangle$ et $|1\rangle$, l'état d'un qubit est représenté par la forme

$$qb_1 := a_0 \, |0\rangle + a_1 \, |1\rangle \tag{1}$$

où a_0 et a_1 sont des valeurs complexes respectant la relation $|a_0|^2 + |a_1|^2 = 1$. Cette particularité révèle son intérêt pour des registres de données comportant plusieurs qubits, à travers le phénomène de la *superposition d'états* : alors que l'état d'un registre classique à n bits correspond à

un élément dans l'ensemble de taille 2^n de toutes les chaînes de n bits, un registre quantique à n qubits correspond à une *combinaison linéaire* (à nombres complexes) de chaînes de n bits : on peut donc avoir les 2^n chaînes de n bits en superposition en même temps. L'état d'un registre quantique s'écrit donc, en étendant la notation employée ci-dessus à des registres $|k\rangle_n$ encodant des entiers k en n qubits, sous la forme suivante :

$$\sum_{k=0}^{2^n-1} a_k \, |k\rangle_n$$

Il en résulte un accroissement potentiellement exponentiel de la quantité d'information stockable dans un registre quantique et manipulable par un ordinateur quantique. Les lois de la physique quantique, auxquelles est soumis le support d'information pour un ordinateur quantique, contraignent cependant sévèrement le calcul. Notamment :

— les opérations possibles sont restreintes à une certaine classe, qui correspond algébriquement aux opérateurs unitaires ;

— l'information contenue dans un registre quantique n'est pas accessible directement, mais via une opération de mesure, qui transforme un registre quantique (ou une partie d'un registre quantique) superposé en registre classique, en le projetant de manière probabiliste sur un des termes de sa superposition. La quantité d'information renvoyée est donc linéaire par rapport à la taille du registre.

C'est cette seconde contrainte qui rend impossible le débogage du code au sens classique. Il n'est pas possible d'arrêter une exécution en cours pour l'analyser, sauf à détruire la superposition d'états du registre. Il faut donc générer intégralement les séquences de calcul quantique avant de les lancer pour des exécutions d'un seul tenant.

Circuits quantiques. Les opérations possibles sur un registre quantique sont approximées, à partir d'un ensemble fini donné de portes élémentaires, par deux compositions :

— la séquence, qui correspond à la succession des opérations sur un registre,

— la composition parallèle, qui permet d'agir sur différentes parties d'un registre.

On forme ainsi un *circuit quantique*, qui correspond à une instruction structurée de calcul pour un ordinateur quantique. Un exemple en est donné par la Figure 1. Il s'agit du circuit pour la Transformée de Fourier Quantique (QFT), dont nous reparlerons dans la Section 4. L'instance de la Figure 1 agit sur un registre à six qubits, représentés par les lignes horizontales indicées. Nous avons par ailleurs souligné, par des tracés pointillés, la structure récursive de cette construction. Les sous-circuits imbriqués correspondent respectivement aux instances de la transformée de Fourier quantique pour des registres à 5,4,3,2 puis 1 qbits. Deux opérations agissent sur ces qubits :

— une opération unaire, la transformation de Hadamard (H), qui opère une superposition d'états sur un qubit,

— une opération binaire, la rotation contrôlée (CR_k), dont le paramètre k indique l'angle de rotation $\frac{2\pi}{2^k}$. Par exemple, la porte dessinée en traits épais indique une rotation d'angle $\frac{\pi}{2}$ du qubit 2 si le qubit 5 est dans l'état $|1\rangle$.

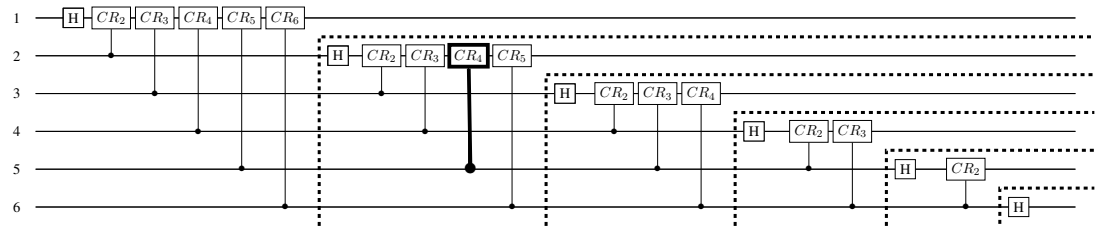


FIGURE 1: Circuit pour la transformée de Fourier quantique

Modèle de calcul hybride (co-processeur quantique). Dans le modèle standard d'architecture quantique, le modèle *Qram* [8], des instructions de calcul sont envoyées sous forme de circuit quantique depuis un ordinateur classique vers un co-processeur quantique. Ce co-processeur effectue le calcul encodé par ce circuit et renvoie les résultats à l'ordinateur classique qui les traite. L'ensemble du contrôle de flot est donc géré par l'ordinateur classique. L'interaction homme-machine s'effectue également entièrement avec l'ordinateur classique.

Dans cet échange d'informations et ce processus de calcul, maîtriser précisément la relation entre un circuit quantique envoyé au co-processeur quantique et la fonction liant les sorties de ce circuit à ses entrées est donc d'une importance cruciale. C'est à ce besoin que nous souhaitons répondre par le développement du formalisme Qbricks.

3 Méthodologie

Notre idée maîtresse est de développer un langage minimal (*Qbricks*) de construction de circuits quantiques avec une sémantique formelle qui interprète ces circuits comme des fonctions transformant des registres quantiques. Cette sémantique étant définie récursivement sur la structure des circuits, on peut calculer, pour chaque circuit, son interprétation sémantique. À l'aide de macros adéquates, on construit ainsi, dans une syntaxe minimale de construction de circuits, des circuits tels que celui de la Figure 1. Ces constructions peuvent éventuellement être paramétrées, de manière à correspondre à des *familles de circuits*.

Sémantique duale. Parallèlement nous développons des outils d'interprétation sémantique pour *Qbricks*. Pour chacun des termes du langage, il s'agit d'interpréter formellement les calculs qu'il permet. Pour *Qbricks* nous développons parallèlement deux sémantiques formelles distinctes : la sémantique matricielle, qui est la sémantique standard, et la sémantique des sommes de chemins. Nous y reviendrons dans la Section 4.

Vérification. En vérification déductive de programmes, les programmes sont décorés par des assertions telles que des pré- ou postconditions ou des invariants de boucles. Pour l'utilisateur, ces décorations constituent des contrats assurant le respect des postconditions par les *outputs* du programme, pour tout *input* qui en vérifie les préconditions. Inférer la satisfaction des postcon-

ditions en sortie de la satisfaction des préconditions en entrée constitue alors une obligation de preuve pour le développeur.

C'est cette méthodologie que nous adoptons dans le développement de *Qbricks*, en annotant les programmes de construction de circuits par des spécifications sur la sémantique formelle. Les annotations constituent donc des contrats de correction du calcul effectué par le circuit créé par le programme.

Why3. L'outil Why3 [3] est un environnement de spécification qui fournit un langage de type ML à la fois pour la programmation et pour l'écriture des spécifications. À partir d'un programme spécifié, Why3 génère un ensemble d'obligations de preuves qui doivent être remplies afin de certifier ce programme. Ces preuves peuvent être manipulées à travers une interface graphique dédiée. Why3 fournit aussi un assistant de preuves interactif. Pour prouver un théorème, on peut ainsi appeler un ensemble de prouveurs SMT automatiques (CVC, Z3, Alt-ergo,etc), accessibles depuis l'interface. Il est aussi possible d'appeler des lemmes pour fournir aux prouveurs des étapes de la preuve, ainsi que de directement simplifier les objectifs de preuve par différentes commandes de transformation de termes.

4 Sémantique matricielle et sommes de chemins

La sémantique standard interprète les circuits quantiques en termes de matrices. On donne ci-dessous cette sémantique pour les opérations de base :

$$M_{Had} := \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \qquad M_{Id} := \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \qquad M_{CRk} := \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & e^{\frac{2i\pi}{2^k}}\end{pmatrix} \qquad M_{Swap} := \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

La composition séquentielle est alors interprétée par le produit matriciel, et la composition parallèle par le produit tensoriel de matrices. On représente de plus un registre quantique par un vecteur – $\begin{pmatrix}1\\0\end{pmatrix}$ pour le registre $|0\rangle$ et $\begin{pmatrix}0\\1\end{pmatrix}$ pour le registre $|1\rangle$ par exemple –. Un calcul quantique effectué par un circuit C de sémantique matricielle M_c sur un registre V est alors interprété par la multiplication matricielle $M_c \cdot V$. Ci-dessous à titre d'illustration, nous représentons l'action d'un circuit élémentaire constitué de la seule porte de Hadamard sur le qubit qb_1 de l'équation 1 :

$$M_{Had} \cdot qb_1 = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \cdot \left(a_0\begin{pmatrix}1\\0\end{pmatrix} + a_1\begin{pmatrix}0\\1\end{pmatrix}\right) = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix} \cdot \begin{pmatrix}a_0\\a_1\end{pmatrix} = \frac{1}{\sqrt{2}}\begin{pmatrix}a_0 + a_1\\ a_0 - a_1\end{pmatrix}$$

Nous avons implémenté la sémantique matricielle pour *Qbricks* en Why3. Travailler avec cette sémantique nous a cependant conduit à chercher une expression sémantique plus abstraite et donc plus adéquate pour les preuves automatiques. La manipulation des matrices présente en effet plusieurs inconvénients, parmi lesquels :

— les matrices sont utilisées dans ce domaine pour représenter, via la multiplication matricielle à un vecteur représentant le registre d'entrée du calcul, des fonctions transformant

des registres quantiques. On leur préférera une expression directe des fonctions des vecteurs vers les vecteurs ;

— les matrices sont une représentation pour des fonctions bornées, de $(\mathbb{N} \times \mathbb{N})$ dans $\mathbb{C}$. La gestion des indices peut s'avérer fastidieuse et parasiter le développement. On préférera donc manipuler des objets moins contraints ;

— enfin, comme l'illustrent à leur niveau les matrices M_{CR_k} et M_{Swap} présentées ci-dessus, les matrices utilisées dans le calcul quantique peuvent être très creuses. On leur préférera des structures de données plus compactes et qui ne tiennent pas compte des zéros.

Récemment, Matthew Amy a proposé une interprétation sémantique du calcul quantique qui remplit ces objectifs [2]. Les circuits y sont directement représentés comme des fonctions de transformation des registres quantiques. La multiplication d'une matrice par un vecteur y est remplacée par une décomposition en somme de produits de vecteurs. On représente ainsi les circuits comme des sommes de vecteurs, appelées des sommes de chemins. On en donne ici l'expression générale pour un registre $|j\rangle_n$ de taille n encodant un entier j :

$$\textit{Path-sum}\ (\ |j\rangle_n) = \frac{1}{2^r} \sum_{l=0}^{2^r-1} e^{\frac{2\pi i(p(j,l))}{c}}\ |k(j,l)\rangle_n$$

où r et c sont des constantes entières et p et k sont des fonctions de $(\mathbb{N} \times \mathbb{N})$ dans $\mathbb{N}$. Il ne s'agit bien sûr que de suggérer une intuition du gain apporté. Il apparaît en effet clairement que cette expression prend en charge une fois pour toutes des éléments structurels communs à tous les circuits quantiques. Par ailleurs cette expression ne contient qu'un unique opérateur de somme, là où l'interprétation par les matrices ajoute une somme imbriquée pour chaque multiplication (donc, selon la sémantique, pour chaque transition d'une séquence). Enfin, les constantes r et c et les fonctions p et k ont des expressions simples et peuvent être chacune définie récursivement sur la structure du circuit concerné.

Nous avons donc implémenté également cette sémantique pour *Qbricks*.

5 Cas d'étude : la transformée de Fourier

En l'état actuel de la recherche, les algorithmes quantiques reposent sur un ensemble de *routines* de calcul. L'une des principales est la *transformée de Fourier quantique*. Cette opération est réalisée par le circuit montré dans la Figure 1. Elle permet de transformer un registre superposé $|x\rangle_n = \sum_{k=0}^{2^n-1} x_k\ |k\rangle_n$ en le registre

$$\textit{QFT}\ (\ |x\rangle_n) = \frac{1}{2^n} \sum_{k=0}^{2^n-1} \sum_{j=0}^{2^n-1} x_j e^{\frac{2\pi i * jk}{2^n}}\ |k\rangle_n \tag{2}$$

en un nombre polynomial d'étapes de calcul ($\frac{n(n+1)}{2}$ étapes pour le circuit de la Figure 1). Cette transformation (voir [10] pour une présentation détaillée) est au cœur de nombreux algorithmes quantiques emblématiques, tels que l'algorithme de Shor pour la factorisation de nombres premiers [13] ou l'estimation de phase.

La Figure 1 illustre le caractère récursif de ce circuit. Il est possible de l'implémenter de manière paramétrée, de façon à définir une famille de circuits indicés par la taille du registre quantique. C'est ce type d'invariants d'échelle que nous exploitons à travers *Qbricks*, afin de fournir des méthodes de développement certifiées quelle que soit la taille d'une instance de calcul.

Résultats obtenus. Afin d'éprouver notre méthodologie, nous avons implémenté cette transformation dans *Qbricks*. Nous avons ensuite prouvé sa correction quant à l'expression de l'équation 2, séparément pour la sémantique matricielle et pour la sémantique des sommes de chemins. Cette construction et ces preuves sont paramétrées. Elles sont donc insensibles à l'échelle.

À notre connaissance, il s'agit de la première proposition pour une preuve formelle de correction d'un calcul quantique écrite dans un langage général de génération de circuits quantiques.

Dans l'état actuel, l'ensemble contient environ 10 000 lignes de codes, principalement pour les bibliothèques relatives à la sémantique du calcul quantique (opérations matricielles de sommes, produit tensoriel, etc., opérations binaires, opérations sur les nombres complexes, etc.). Il contient environ 300 définitions et 1000 preuves de lemmes intermédiaires et théorèmes.

6 Conclusion

Nous développons *Qbricks*, un langage noyau pour la programmation et la vérification de programmes quantiques. Nous nous appuyons pour cela sur l'outil Why3, qui permet à la fois l'écriture de programmes dans un langage fonctionnel de type ML, l'écriture d'annotations pour ces programmes et des techniques de preuves formelles du respect de ces annotations. Nous avons implémenté ce langage ainsi que l'ensemble des librairies nécessaires pour sa sémantique (algèbre, théorie des ensembles, opérations binaires,etc). Nous avons implémenté deux sémantiques pour *Qbricks* : d'une part, la sémantique matricielle, standard pour l'interprétation des programmes quantiques et d'autre part, la sémantique des sommes de chemins, proposition récente, avantageuse pour la manipulation des structures de données et la preuve formelle. Nous avons prouvé la correction du calcul pour une routine fondamentale du calcul quantique : la transformée de Fourier quantique. La preuve de correction de la transformée de Fourier quantique a été développée indépendamment pour chacune des deux sémantiques de *Qbricks*. Nous avons par ailleurs prouvé intégralement en Why3 l'équivalence entre ces deux sémantiques. Nous disposons ainsi à la fois d'un cadre formel adapté à la vérification déductive (la sémantique des sommes de chemins), et d'une méthode de traduction des résultats obtenus dans la sémantique standard pour l'informatique quantique (la sémantique matricielle).

Dans un futur proche, nous prévoyons d'étendre notre travail sur la transformée de Fourier à l'algorithme d'estimation de phases. Nous disposerons alors d'un algorithme quantique entièrement vérifié.

Remerciements. Nous remercions Chantal Keller et Dongho Lee pour la relecture attentive de cet article et les discussions constructives durant sa rédaction.

Références

[1] The Q# programming language. `https://docs.microsoft.com/en-us/quantum/language/?view=qsharp-preview`, 2017.

[2] Matthew Amy. Towards large-scale functional verification of universal quantum circuits. *CoRR*, 2018.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3 : Shepherd Your Herd of Provers. In *Boogie 2011 : First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011.

[4] Yuan Feng, Runyao Duan, Zheng-Feng Ji, and Mingsheng Ying. Proof rules for the correctness of quantum programs. *Theor. Comput. Sci.*, 386(1-2) :151–166, 2007.

[5] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013*, pages 125–128, 2013.

[6] Simon J. Gay, Rajagopal Nagarajan, and Nikolaos Papanikolaou. QMC : A model checker for quantum systems. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 543–547, 2008.

[7] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper : a scalable quantum programming language. In *PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 333–342, 2013.

[8] E. Knill. Conventions for quantum pseudocode. Los Alamos National Laboratory. Technical report, 1996.

[9] Tao Liu, Yangjia Li, Shuling Wang, Mingsheng Ying, and Naijun Zhan. A theorem prover for quantum hoare logic and its applications. *arXiv :1601.03835*, 2016.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information : 10th Anniversary Edition*. Cambridge University Press, New York, NY, USA, 10th edition, 2011.

[11] Jennifer Paykin, Robert Rand, and Steve Zdancewic. QWIRE : a core language for quantum circuits. *ACM SIGPLAN Notices*, 2017.

[12] Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE practice : Formal verification of quantum circuits in coq. *arXiv :1803.00699*, 2018.

[13] Peter W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5), 1997.

[14] Mingsheng Ying. Floyd–hoare logic for quantum programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(6) :19, 2011.

[15] Mingsheng Ying, Yangjia Li, Nengkun Yu, and Yuan Feng. Model-checking linear-time properties of quantum systems. *ACM Trans. Comput. Log.*, 15(3) :22 :1–22 :31, 2014.

[16] Mingsheng Ying, Shenggang Ying, and Xiaodi Wu. Invariants of quantum programs : characterisations and generation. *ACM SIGPLAN Notices*, 2017.

Formalisation et validation d'une méthode de construction de systèmes de blocs*

Jessy Colonval† et Henri de Boutray‡
Institut FEMTO-ST, Université de Bourgogne Franche-Comté, CNRS, Besançon, France

Résumé

Dans le domaine des mathématiques expérimentales, les programmes utilisés pour obtenir divers résultats suivent rarement les bonnes pratiques du génie logiciel, ce qui rend ces résultats difficiles à reproduire et à évaluer. Cet article présente une formalisation d'une méthode de construction de structures combinatoires (appelées "systèmes de blocs") et une validation de leurs propriétés.

1 Introduction

Dans le domaine des mathématiques expérimentales, les chercheurs développent des programmes informatiques pour découvrir de nouveaux résultats. Cependant, les publications scientifiques de ces résultats ne sont pas toujours accompagnées par ces programmes, ou sinon ces derniers ne sont pas suffisamment accessibles, structurés et documentés pour permettre aux lecteurs de les exécuter pour reproduire ces résultats. Or, la reproductibilité est un enjeu scientifique majeur, qui doit aussi s'appliquer à la partie calculatoire des mathématiques [SBB+12], en particulier pour faciliter la vérification et l'extension des résultats publiés.

Un théorème mathématique est souvent un énoncé général qui doit être démontré, car le calcul ne permet de valider qu'un fragment limité d'une infinité de cas couverts par cet énoncé. Ce théorème peut être vérifié par un tiers, dans un assistant de preuve, si sa publication est accompagnée par une démonstration formelle. Cependant, la construction d'une démonstration formelle est une tâche ardue, car elle nécessite une formalisation de tous les concepts mathématiques sous-jacents au théorème, et la maîtrise d'un assistant de preuve. La démonstration formelle du *odd order theorem* [GAA+13] en est un bon exemple : elle a mobilisé de nombreux chercheurs sur une longue période. Comparativement, le test d'un grand nombre d'instances d'un théorème est souvent beaucoup plus simple à mettre en œuvre et peut apporter un niveau de confiance suffisant dans la correction de ce théorème.

Nous proposons d'appliquer aux mathématiques calculatoires le test intensif, ainsi que d'autres bonnes pratiques de programmation, comme la structuration, la factorisation, la documentation et la distribution libre du code source. Nous visons en particulier les théorèmes portant sur une infinité de structures mathématiques munies d'une taille. Dans ce cas, nous proposons de vérifier le théorème pour toutes les structures, par taille croissante, jusqu'à une taille maximale jugée suffisante pour donner confiance en la correction du théorème.

*Financé par le projet ISITE-BFC I-QUINS (contrat ANR-15-IDEX-03) du programme français "Investissements d'Avenir".

†Master recherche.

‡Doctorant.

Ce travail s'inscrit dans le cadre de recherches des géométries finies dites *quantiques*, car liées à la contextualité quantique. Planat et al. [PGHS15] ont montré comment construire ces géométries à partir de groupes de permutations, mais sans publier de programme pour cette construction. Ces géométries sont des cas particuliers de *systèmes de blocs* (en anglais *block designs*, définis dans la partie 2) et l'article de Planat et al. présente une méthode pour les construire.

Une recherche bibliographique sur l'origine de cette méthode nous a menés à une référence antérieure [KM02], qui présente une méthode plus simple, qui construit des systèmes de blocs à partir de groupes de permutations primitifs. Cet article est complété par un programme, mais ce dernier ne contient pas de code pour valider cette méthode. Nous formalisons cette méthode, puis sa validation par énumération. Plus précisément, nous validons que les systèmes de blocs construits selon cette méthode ont tous les caractéristiques annoncées dans une proposition de cet article. Nous utilisons l'environnement Magma [BCP97], composé d'un langage impératif structuré et d'une vaste bibliothèque de fonctions mathématiques, en particulier de la théorie des groupes et des *designs*. Notre code source est distribué librement sur GitHub [1].

La partie 2 présente notre implémentation de la méthode de construction de systèmes de blocs de Key et Moori [KM02]. La partie 3 traite du test intensif de cette méthode.

2 Construction de systèmes de blocs

Key et Moori [KM02] définissent une méthode de construction de systèmes de blocs par la proposition suivante :

> **Proposition 1.** *Let G be a finite primitive permutation group acting on the set Ω of size n. Let $\alpha \in \Omega$, and let $\Delta \neq \{\alpha\}$ be an orbit of the stabilizer G_α of α. If $\mathcal{B} = \{\Delta^g : g \in G\}$ [...] then $\mathcal{B}$ forms a self-dual 1-$(n, |\Delta|, |\Delta|)$ design with n blocks [...].*
>
> (Extrait de la proposition 1 de la page 3 de [KM02])

La partie 2.1 explique cette proposition, mais le lecteur peu familier de ces notions peut en admettre le contenu. La partie 2.2 présente notre implémentation du contenu calculatoire de cette proposition.

2.1 Définitions

La proposition 1 s'applique à tout entier naturel n et à tout groupe de permutations G sur un ensemble fini Ω de cardinalité n, identifié ici à l'ensemble $\{1, \ldots, n\}$. Par nature, ce groupe G est fini. L'application de la permutation g de G à l'élément α de l'ensemble Ω est notée $g \bullet \alpha$. L'image d'une partie Δ de Ω par la permutation g de G est notée $\Delta^g =_{\mathsf{def}} \{g \bullet x \ : \ x \in \Delta\}$. Les définitions suivantes de la théorie des groupes et de la théorie des systèmes de blocs permettent de comprendre la suite de la proposition, dans laquelle $|\Delta|$ désigne la cardinalité de l'orbite Δ.

Le groupe de permutations G sur Ω est *transitif* si, pour tous les éléments x et y de Ω, il existe une permutation g de G telle que $g \bullet x = y$. Le groupe G est *primitif* s'il est transitif et s'il ne préserve aucune partition non triviale de Ω (les partitions triviales de Ω sont la partition $\{\Omega\}$, dont le seul élément est Ω, et la partition $\{\{x\} \ : \ x \in \Omega\}$, dont tous les éléments sont des singletons). Le *stabilisateur* $G_\alpha =_{\mathsf{def}} \{g \in G \ : \ g \bullet \alpha = \alpha\}$ d'un élément α de Ω (*sous l'action de* G) est l'ensemble des permutations de G qui laissent α invariant sous leur action. L'orbite $O_x =_{\mathsf{def}} \{g \bullet x \ : \ g \in H\}$ d'un élément x de Ω selon un groupe H de permutations sur Ω est l'ensemble des images de x par les permutations de H.

1. `https://quantcert.github.io/Designs`

Un *bloc* est une partie de l'ensemble Ω. Un *système de blocs* $\mathcal{B}$ est un ensemble de blocs. Une *structure d'incidence* est un triplet $\mathcal{D} = (\mathcal{P}, \mathcal{B}, \mathcal{I})$ où $\mathcal{P} = \{1, \ldots, n\}$ est un ensemble d'éléments fini, $\mathcal{B} = \{1, \ldots, b\}$ numérote un système de blocs sur $\mathcal{P}$ et $\mathcal{I} \subseteq \mathcal{P} \times \mathcal{B}$ est une *relation d'incidence*, qui définit l'appartenance d'un élément à un bloc. Sa structure *duale* est la structure d'incidence $\mathcal{D}^t =_{\mathsf{def}} (\mathcal{B}, \mathcal{P}, \mathcal{I}^t)$ où $(y, x) \in \mathcal{I}^t$ si et seulement si $(x, y) \in \mathcal{I}$. La relation d'incidence permet d'associer un système de blocs à toute structure d'incidence, et de définir le dual d'un système de blocs. Un système de blocs est *symétrique* s'il possède autant d'éléments que de blocs. Il est *auto-dual* (*self-dual* en anglais) s'il est de plus isomorphe à son dual. Un *système de blocs* t-(v, k, λ) est un système de v blocs de cardinalité k, tel que chaque sous-ensemble de λ blocs a exactement t éléments distincts en commun.

La proposition 1 affirme que, pour tout élément α de Ω et pour toute orbite Δ du stabilisateur de α sous l'action de G différente de l'orbite $\{\alpha\}$, le système de blocs $\mathcal{B} =_{\mathsf{def}} \{\Delta^g : g \in G\}$ contient n blocs et présente la régularité d'un système 1-$(n, |\Delta|, |\Delta|)$ auto-dual. L'exemple 1 illustre ces définitions à partir d'un groupe donné.

Exemple 1. *Soit* $n = 5$ *et* $G = \{$ *Id,* $(1,2,3,4,5)$*,* $(1,5,4,3,2)$*,* $(1,3,5,2,4)$*,* $(1,4,2,5,3)$*,* $(2,5)$ $(3,4)$*,* $(1,2)$ $(3,5)$*,* $(1,5)$ $(2,4)$*,* $(1,3)$ $(4,5)$*,* $(1,4)$ $(2,3)$ $\}$ *un groupe de permutations primitif sur* $\Omega =_{\mathit{def}} \{1,2,3,4,5\}$*. Outre la permutation identité, notée Id, ses permutations sont données sous la forme de leur produit de cycles disjoints. Les points fixes, tels que le cycle* (1) *pour la sixième permutation, ne sont pas écrits. Avec* $\alpha = 1$*, le stabilisateur de* G *sur* α *est le groupe* $G_1 = \{Id, (2,5)(3,4)\}$*. Les orbites de* G_1 *sont les ensembles* $\{1\}$*,* $\{2,5\}$ *et* $\{3,4\}$*. Pour* $\Delta = \{2,5\}$*, le système de blocs est* $\mathcal{B} = \{\{2,5\}, \{1,3\}, \{1,4\}, \{2,4\}, \{3,5\}\}$*. C'est un système de blocs* 1-$(5,2,2)$ *sur* 5 *éléments, composé de* 5 *blocs de cardinalité* $k = 2$*, où chaque élément (*$t = 1$*) est présent dans exactement* $\lambda = 2$ *blocs distincts.*

2.2 Implémentation

Magma propose des fonctions permettant de construire des groupes primitifs [BCFS19, `PrimitiveGroups`], des orbites d'un groupe [BCFS19, `Orbits`], le stabilisateur d'un élément selon l'action d'un groupe [BCFS19, `Stabilizer`] et la création d'un système de blocs à partir d'une structure d'incidence [BCFS19, `Design`], mais ne propose pas d'implémentation de la méthode de Key et Moori pour construire des systèmes de blocs à partir d'un groupe de permutations primitif.

L'article [KM02] est complété par un code Magma (dans son annexe) qui ne formalise pas le contenu calculatoire de la proposition 1 par une fonction, mais l'applique seulement à deux cas particuliers de groupes de permutations primitifs. Ainsi, ce code n'est pas structuré. De plus, il ne contient pas d'instruction pour valider la propriété des systèmes construits, annoncée dans la proposition 1. Enfin, les résultats retournés par ce code sont mélangés avec ce dernier, sans être placés dans un bloc de commentaires. Tout ceci rend le code peu convaincant et ré-utilisable.

C'est pourquoi nous proposons une implémentation de la proposition 1 par deux fonctions, reproduites dans les listings 1 et 2. Afin de faciliter leur utilisation, elles sont commentées selon le format Javadoc, car Magma ne propose pas d'outil permettant de générer une documentation [2]. Les variables Magma sont nommées et les fonctions sont structurées de telle sorte que le code ressemble le plus possible au texte de la proposition 1.

Le listing 1 présente une fonction qui calcule l'ensemble des orbites Δ définies dans la proposition 1. Elle prend en paramètre le groupe de permutations primitif G et retourne un tableau associatif `Deltas` des orbites Δ indexées par les α correspondants. Pour tous les éléments α de

2. Il existe cependant un projet GitHub "magdoc" visant à remplir cette fonctionnalité.

l'ensemble $\Omega = \{1, \ldots, n\}$, la boucle calcule dans la variable `Galpha` le groupe stabilisateur G_α et ses orbites dans la variable `orbits`, puis associe à l'index α la séquence des orbites Δ différentes du singleton $\{\alpha\}$. Les orbites Δ sont converties en ensembles grâce à la fonction Magma `IndexedSetToSet` [BCFS19] pour que les résultats soient sous une forme utilisable pour la création de certains objets Magma utiles pour la suite.

```
/**
 * Compute orbits of stabilizers of a primitive group [KM02, Proposition 1].
 *
 * @param G::GrpPerm A primitive group
 * @return Deltas::Assoc An associative array indexed by alpha
 *       and containing the corresponding delta set
 */
AllDelta := function(G)
    n := Degree(G);
    Omega := {1..n};
    Deltas := AssociativeArray();
    for alpha in Omega do
        Galpha := Stabilizer(G, alpha);
        orbits := Orbits(Galpha);
        Deltas[alpha] := { IndexedSetToSet(Delta) : Delta in orbits | Delta ne { alpha } };
    end for;
    return Deltas;
end function;
```

Listing 1 – Fonction de calcul des orbites Δ à partir d'un groupe de permutations primitif.

Le listing 2 présente une fonction qui construit les systèmes de blocs à partir d'un groupe primitif. Elle prend en paramètre un groupe de permutations primitif G et retourne un tableau associatif des systèmes de blocs indexés par l'orbite Δ associée. Les orbites Δ sont construites à l'aide de la fonction précédente, puis chaque système de blocs est construit par l'application de chaque permutation du groupe G à une orbite Δ. Le résultat est associé à l'index Δ dans le tableau associatif `blocks`.

```
/**
 * Builds all block designs from a primitive group [KM02, Proposition 1]
 *
 * @param G::GrpPerm A primitive group
 * @return blocks::Assoc An associative array indexed by delta indexes
 *       and containing corresponding block designs
 */
BlckDsgnsFromPrmtvGrp := function(G)
    Deltas := AllDelta(G);
    blocks := AssociativeArray();
    for alpha in Keys(Deltas) do
        for Delta in Deltas[alpha] do
            blocks[Delta] := { Delta^g : g in G };
        end for;
    end for;
    return blocks;
end function;
```

Listing 2 – Fonction de calcul de systèmes de blocs à partir d'un groupe de permutations primitif.

3 Validation

Une recherche bibliographique a révélé l'existence d'une correction de la proposition 1, par les mêmes auteurs [KM08]. Cette correction ne remet pas en cause le calcul des systèmes de blocs mais seulement leur nature : ils doivent être symétriques, et non pas auto-duaux, comme résumé dans la proposition 2.

Proposition 2. *Let G be a finite primitive permutation group acting on the set Ω of size n. Let $\alpha \in \Omega$, and let $\Delta \neq \{\alpha\}$ be an orbit of the stabilizer G_α of α. If $\mathcal{B} = \{\Delta^g : g \in G\}$ [...] then $\mathcal{D} = (\Omega, \mathcal{B})$ forms a symmetric 1-$(n, |\Delta|, |\Delta|)$ design. [...]*

(Extrait de la proposition 1, page 1 de [KM08])

Cette partie tente de valider les propositions 1 et 2 à l'aide d'un test exhaustif, borné par le nombre de groupes primitifs utilisés. Nous présentons d'abord trois fonctions booléennes qui implémentent les propriétés des systèmes de blocs des propositions 1 et 2.

Le listing 3 présente une fonction booléenne Magma pour caractériser les systèmes de blocs 1-(v, k, λ). Elle convertit l'ensemble d'ensembles `blocks` en une structure d'incidence, pour déterminer avec la fonction Magma `IsDesign` [BCFS19] si c'est un t-système de blocs. Si c'est le cas, elle construit ce système de blocs pour en extraire ses caractéristiques avec la fonction Magma `Parameters` [BCFS19], puis elle vérifie que ces caractéristiques sont bien celles attendues.

```
/**
 * Characterization of t-(v,k,lambda) block designs
 *
 * @param blocks::Set The design blocks
 * @param t::RngIntElt The number of elements distinct in lambda blocks
 * @param v::RngIntElt The number of blocks
 * @param k::RngIntElt The cardinality of blocks
 * @param lambda::RngIntElt The number of blocks contains t elements distinct
 * @return BoolElt Indicates that blocks design is a t-(v,k,lambda) block design
 */
CorrectDesign := function(blocks, t, v, k, lambda)
    incidence := IncidenceStructure<v | blocks>;
    if not IsDesign(incidence, t) then
        return false;
    end if;
    record := Parameters(Design(incidence, t));
    return record`v eq v and record`k eq k and record`lambda eq lambda;
end function;
```

Listing 3 – Fonction caractéristique d'un système de blocs 1-(v, k, λ)

La proposition 1 indique que le système de blocs construit est un système de blocs 1-$(n, |\Delta|, |\Delta|)$ auto-dual. Le listing 4 présente une fonction booléenne Magma pour vérifier cette condition. La fonction a pour paramètres le nombre d'éléments du système de blocs, le système de blocs lui-même et son orbite Δ. Elle retourne `true` si le système de blocs construit est un système de blocs 1-$(n, |\Delta|, |\Delta|)$ (vérifié avec la fonction précédente) et s'il est auto-dual (vérifié avec la fonction Magma `IsSelfDual` [BCFS19].

```
/**
 * Conditions of block designs in [KM02, proposition 1]
 *
 * @param n::RngIntElt The number of points of the design
 * @param blocks::Set The design blocks
 * @param Delta::Set The delta that generated the design blocks
 * @return BoolElt Indicates whether the block design is a 1-(n,|delta|,|delta|) block
 *      design and a block design self-dual
 */
CorrectConstructionKM02 := function(n, blocks, Delta)
    return CorrectDesign(blocks, 1, n, #Delta, #Delta) and IsSelfDual(Design<1, n | blocks>);
end function;
```

Listing 4 – Propriétés des systèmes de blocs de la proposition 1.

Magma implémente une fonction `IsSymmetric` [BCFS19] qui caractérise un système de blocs symétrique, mais selon une définition différente de celle utilisée par Key et Moori [KM02]. La documentation Magma ne le précise pas mais, après quelques tests, il semble que Magma implémente la

définition de symétrie des BIBD. Un *BIBD* (*Balanced Incomplete Block Design*) est un système de blocs 2-(v, k, λ). Un BIBD est symétrique s'il possède autant d'éléments que de blocs [Col10]. Ainsi, Magma ne considère comme symétriques que les systèmes de blocs dont le paramètre t est égal à 2.

Le listing 5 présente deux fonctions booléennes Magma pour vérifier la condition de la proposition 2 selon ces deux interprétations de la définition de symétrie. Les fonctions ont pour paramètres le nombre d'éléments du système de blocs, le système de blocs lui-même et son orbite Δ. La première (resp. seconde) fonction retourne `true` si le système de blocs construit est un système de blocs 1-$(n, |\Delta|, |\Delta|)$ et si c'est un BIBD symétrique (resp. s'il possède le même le nombre de blocs que d'éléments).

```
/**
 * Characterization of 1-(n,|delta|,|delta|) block designs that are BIBD symmetric
 *
 * @param n::RngIntElt The number of points of the design
 * @param blocks::Set The design blocks
 * @param Delta::Set The delta that generated the design blocks
 * @return BoolElt Indicates whether the block design is a 1-(n,|delta|,|delta|) block
 *       design and a BIBD symmetric
 */
CorrectConstructionKM08_MagmaSym := function(n, blocks, Delta)
    if not CorrectDesign(blocks, 1, n, #Delta, #Delta) then
        return false;
    end if;
    return IsSymmetric(Design<1, n | blocks>);
end function;

/**
 * Characterization of 1-(n,|delta|,|delta|) block designs with the same
 * numbers of blocks and elements
 */
CorrectConstructionKM08_KMSym := function(n, blocks, Delta)
    if not CorrectDesign(blocks, 1, n, #Delta, #Delta) then
        return false;
    end if;
    record := Parameters(Design<1, n | blocks>);
    return record`b eq record`v;
end function;
```

Listing 5 – Propriétés des systèmes de blocs de la proposition 2 selon les définitions de symétrie de Magma et de Key et Moori.

Le listing 6 présente une fonction de validation des propositions 1 et 2. Magma propose une base de données composée de tous les groupes de permutations primitifs de degré[3] inférieur ou égal à 2 500, soit 16 916 groupes, et de 7 643 groupes primitifs de degré plus élevé. Ces groupes contiennent entre 1 et 4 095 permutations. La fonction Magma `PrimitiveGroups` [BCFS19] retourne la séquence de ces groupes primitifs, triés par degré croissant, puis par cardinalité (nombre de permutations) croissante. Le code parcourt cette séquence et calcule tous les systèmes de blocs à partir du groupe courant grâce à la fonction présentée dans le listing 2. Pour chaque système de blocs, le code vérifie qu'il respecte les conditions des propositions 1 et 2, avec les deux interprétations de la symétrie pour la proposition 2. Les résultats sont enregistrés dans un fichier avec le nombre d'éléments n, l'index du groupe parent dans la séquence, l'orbite Δ associée et le temps de construction du système de blocs testé. Les calculs ont été effectués sur le calculateur du Mésocentre de calcul de Franche-Comté, pour les 74 premiers groupes primitifs, en 574 107 secondes (environ 7 jours), jusqu'au degré $n = 14$ inclus. C'est le degré maximal atteignable dans le délai d'utilisation du mésocentre, limité à 8 jours.

3. Le degré d'un groupe de permutations sur un ensemble fini Ω est la cardinalité de cet ensemble Ω.

Ces calculs peuvent être reproduits librement pour les 48 premiers groupes primitifs, jusqu'au degré $n = 10$, sur le calculateur en ligne de Magma [4], dont l'usage est limité à 120 secondes.

```
/**
 * Validation of Proposition 1 in [KM02] and [KM08]
 *
 * @param nbGrp::RngIntElt Number of smallest first primitive groups tested
 */
procedure verifProp(nbGrp)
    allG := PrimitiveGroups(:Warning := false);
    assert nbGrp le #allG;

    printf "degree,numGrp,delta,isKM02,isKM08,isKM08_v2,time\n";
    for numGrp := 1 to nbGrp do
        G := allG[i];
        n := Degree(G);

        beginBlck := Realtime();
        allBlck := BlckDsgnsFromPrmtvGrp(G);
        tBlck := Realtime(beginBlck);
        for Delta in Keys(allBlck) do
            block := allBlck[Delta];
            printf "%o,%o,%o,%o,%o,%o,%o\n",
                n, numGrp, Sprintf("\"%o\"", Delta),
                CorrectConstructionKM02(n, block, Delta),
                CorrectConstructionKM08_MagmaSym(n, block, Delta),
                CorrectConstructionKM08_KMSym(n, block, Delta),
                tBlck;
        end for;
    end for;
end procedure;
```

Listing 6 – Code implémentant un test exhaustif borné des propositions 1 et 2.

À partir des 74 premiers groupes de permutations primitifs, ce programme construit 926 systèmes de blocs, soit tous les systèmes de blocs possibles pour tous les groupes de permutations primitifs de degré inférieur ou égal à 13 et deux groupes de degré 14. Le programme n'a trouvé aucun contre-exemple pour les propositions 1 et 2. En revanche, il a permis d'exhiber 398 contre-exemples pour l'interprétation erronée de la proposition 2, un nombre élevé pour une différence minime entre les deux définitions de la symétrie. Le plus petit contre-exemple est le groupe symétrique S_2 avec $\alpha \in \{1, 2\}$.

Exemple 2. *Soit $S_2 = \{Id, (1, 2)\}$ le groupe symétrique sur $\Omega = \{1, 2\}$ et $\alpha = 1$. Le stabilisateur de S_2 de l'élément 1 est le groupe $G_1 = \{Id\}$ et les orbites de ce groupe sont les ensembles $\{1\}$ et $\{2\}$. La seule orbite Δ possible est l'ensemble $\{2\}$. Le système de blocs construit est $\mathcal{B} = \{\Delta^{Id}, \Delta^{(1,2)}\} = \{\{2\}^{Id}, \{2\}^{(1,2)}\} = \{\{2\}, \{1\}\}$. Le système de blocs $\mathcal{B}$ est composé de 2 blocs de taille 1 et 1 élément distinct se trouve dans exactement 1 bloc, donc c'est un système de blocs 1-$(2, 1, 1)$. Ce n'est pas un système de blocs 2-(v, k, λ), ce qui le rend non symétrique selon Magma. Mais il est symétrique selon la définition de Key et Moori, car il possède autant d'éléments que de blocs.*

4 Conclusion

Nous avons donné un exemple d'application de la programmation et du test intensif à la vérification de propriétés mathématiques. Nous avons appliqué le test exhaustif borné à deux propositions de construction de systèmes de blocs, selon deux interprétations différentes, ce qui nous a permis de

4. http://magma.maths.usyd.edu.au/calc/

valider ces propositions et de lever une ambiguïté sur leurs interprétations possibles. Outre la documentation du code, nous avons aussi identifié une bonne pratique : la proximité entre le programme et le théorème qu'il formalise rassure le lecteur sur la cohérence entre les deux.

Une perspective globale serait d'établir et de diffuser d'autres principes généraux pour la reproductibilité et la vérification des publications mathématiques de nature calculatoire, mais ceci n'entre pas dans le cadre de nos sujets de master et de thèse. La suite du travail de master sera d'appliquer ces méthodes à l'article de Planat et al. [PGHS15]. La suite du travail de thèse sera d'intégrer cette contribution dans nos recherches sur les spécifications des langages quantiques.

Remerciements

Nous remercions Alain Giorgetti, Pierre-Alain Masson et Frédéric Holweck, notre encadrant de stage de master recherche et nos directeur, co-directeur et co-encadrant de thèse, qui nous ont aidé et conseillé tout au long de ce travail. Nous remercions aussi Michel Planat qui nous a aidé à comprendre certaines notions complexes.

Références

[BCFS19] W. Bosma, J. Cannon, C. Fieker, and A. Steel. Handbook of Magma functions, edition 2.24-6, 2019.

[BCP97] W. Bosma, J. Cannon, and C. Playoust. The Magma Algebra System I : The User Language. *Journal of Symbolic Computation*, 24(3) :235–265, 1997.

[Col10] C.J. Colbourn. *CRC Handbook of Combinatorial Designs*. CRC Press, 2010.

[GAA+13] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A Machine-Checked Proof of the Odd Order Theorem. In *Interactive Theorem Proving*, volume 7998, pages 163–179. 2013.

[KM02] J.D. Key and J. Moori. Codes, Designs and Graphs from the Janko Groups J_1 and J_2. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 40 :143–159, 2002.

[KM08] J.D. Key and J. Moori. Correction to : Codes, Designs and Graphs from the Janko Groups J_1 and J_2, J.D. Key and J. Moori, JCMCC 40 (2002), 143-159. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 64 :153, 2008.

[PGHS15] M. Planat, A. Giorgetti, F. Holweck, and M. Saniga. Quantum contextual finite geometries from dessins d'enfants. *International Journal of Geometric Methods in Modern Physics*, 12(07) :1550067, 2015.

[SBB+12] V. Stodden, D.H. Bailey, J. Borwein, R.J. LeVeque, and W. Rider. Setting the Default to Reproducible. Technical report, 2012. `http://icerm.brown.edu/topical_workshops/tw12-5-rcem`.

La logique contre les fantômes: comparaison de deux approches pour la preuve d'un module de listes chaînées*

Allan Blanchard[†1], Nikolai Kosmatov[‡2], and Frédéric Loulergue[§3]

[1]Inria Lille – Nord Europe, Villeneuve d'Ascq, France
[2]CEA, List, Software Reliability Lab, 91191 Gif-suf-Yvette, France
[3]School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, USA

Contexte et motivation. Les projets de vérification récents continuent d'offrir de nouveaux défis pour la vérification formelle. L'un d'eux est le module de listes chaînées de Contiki, un système d'exploitation pour l'*internet-des-objets*. Son API est riche, elle permet l'ajout et la suppression à n'importe quelle position dans une liste donnée. Par ailleurs, elle assure l'unicité des éléments lors de l'insertion (en assurant la suppression préalable de l'élément ajouté s'il était déjà présent dans la liste). Finalement, contrairement aux listes chaînées classiques, le module de Contiki ne produit pas d'allocation dynamique car le système d'exploitation ne le permet pas. Le module de listes chaînées est utilisé dans de nombreux autres modules du système, il est donc critique d'un point de vue sûreté et sécurité.

Dans un travail précédent [2], nous avons vérifié ce module avec l'outil FRAMA-C et son greffon WP. Ce travail reposait sur l'usage de tableaux fantômes (*ghost*) comme représentation des listes chaînées, et nous avait permis de prouver toutes les fonctions sauf l'insertion. Cependant, cette approche a ses inconvénients. D'abord, elle nécessite un travail d'annotation conséquent : il faut spécifier de nombreuses propriétés de séparation et écrire beaucoup d'assertions pour guider les prouveurs automatiques. Ceux-ci voient alors leur efficacité réduite car ces assertions augmentent la taille du contexte de preuve, qui devient difficile à manipuler. Vu le nombre important d'assertions que nous avions dû écrire pour prouver des propriétés parfois assez évidentes, la fonction d'insertion nous paraissait difficile à vérifier par cette approche du fait de ses nombreux et complexes comportements.

*Cette soumission est un résume étendu de l'article [1] accepté à SAC-SVT 2019.

†allan.blanchard@inria.fr
‡nikolai.kosmatov@cea.fr
§frederic.loulergue@nau.edu

Finalement, d'un point de vue méthodologique, une vue plus abstraite des listes que celle fournie par un tableau peut sembler préférable.

Méthode de vérification. L'article [1] présente une nouvelle vérification de ce module, réalisée avec les mêmes outils mais reposant cette fois sur l'usage de *listes logiques* fournies par ACSL, le langage de spécification de FRAMA-C. La nouvelle vérification est réalisée par le maintien d'une propriété d'équivalence entre la liste chaînée C et la liste logique qui la représente. Cette liste logique contient les adresses des différents éléments de la liste chaînée. Cette relation d'équivalence est définie à l'aide d'un prédicat inductif. Comme dans le précédent travail, nous avons écrit et prouvé 12 fonctions utilisatrices (et 12 variantes incorrectes), pour montrer que la spécification est utilisable pour prouver du code client du module (resp., que l'on ne peut pas prouver du code incorrect).

Manipuler des propriétés de listes logiques nécessite de raisonner par induction, une tâche pour laquelle les prouveurs automatiques ne sont pas bons. Dans un tel cas, l'approche classique consiste à ajouter un ensemble de lemmes exprimant des propriétés utilisant ce prédicat inductif, et qui peuvent être directement manipulées par les prouveurs automatiques. La preuve de ces lemmes est faite par induction en utilisant des prouveurs interactifs. Dans le cas des listes, les lemmes permettent par exemple d'exprimer qu'une liste peut être découpée en deux sous-listes ou inversement que deux sous-listes peuvent être fusionnées si elles se suivent.

Nous utilisons le prédicat d'équivalence pour spécifier les différentes fonctions de l'API. Chaque fonction met en relation la représentation logique de la liste chaînée en pré-condition avec la représentation obtenue en post-condition. Cependant, comme le langage ACSL ne permet pas de quantifier universellement ou existentiellement une variable sur l'ensemble d'un contrat, nous construisons la liste logique à chaque état de la mémoire (pré et post-conditions) à l'aide d'une fonction logique définie axiomatiquement. Une partie des lemmes qui étaient exprimés à propos de l'équivalence doivent être dupliqués pour pouvoir exprimer des propriétés similaires sur la construction de la liste logique. Cependant, cela permet de simplifier l'écriture des spécifications et d'éviter la présence de variables quantifiées existentiellement qui rendraient l'usage des contrats plus difficile.

Comparaison de deux approches. Nous comparons la précédente [2] et la nouvelle technique [1] pour le module de listes pour déterminer les avantages et inconvénients de chacune. Nous comparons le nombre d'obligations de preuve, le travail d'annotation et le temps nécessaire pour exécuter les preuves sur l'ensemble des fonctions prouvées dans [2]. Notons que quelques modifications ont été faites sur le travail original pour considérer la même version de FRAMA-C et des prouveurs automatiques (ces modifications ont amélioré les résultats du travail original).

Tout d'abord la taille des contrats dans la version avec listes logiques est de 42% inférieur (de 500 à 290 lignes) et le nombre d'obligations de preuve associées est diminué de 45% (de 274 à 152). La raison principale est le fait que la version avec listes logiques ne nécessite d'exprimer que très peu de propriétés de séparation, à la différence de la version avec tableaux fantômes. La taille des annotations

utilisées pour guider les prouveurs a également diminué de 33% (de 680 à 460 lignes), de même pour le nombre d'obligations (de 399 à 264). Le temps nécessaire à l'exécution complète des preuves a été divisé par 4 dans l'ensemble (de 21min 20s à 5min 30s), et divisé par 2 si l'on s'intéresse au temps moyen par obligation (de 1.6s à 0.7s). Cela rend la preuve des fonctions plus efficace : l'ingénieur doit attendre moins de temps pour avoir des résultats entre chaque tentative. En revanche, la version avec listes logiques nécessite plus de lemmes prouvés interactivement (33 contre 24 dans la version avec tableaux fantômes), et les preuves de ces lemmes sont globalement plus complexes et plus longues.

Même si nous pouvons utiliser la spécification pour prouver du code client, un utilisateur pourrait vouloir se tourner vers la vérification à l'exécution, et avoir besoin de spécifications exécutables, que le plugin E-ACSL de FRAMA-C peut transformer en code C. Nous avons pu montrer que l'approche avec les tableaux fantômes est compatible avec la vérification à l'exécution [3]. Pour l'approche avec listes logiques, ce n'est pas aussi clair, car E-ACSL ne supporte pas ce type.

La fonction d'insertion a été prouvée à l'aide de l'approche par listes logiques. À elle seule, la vérification de cette fonction représente le tiers des spécifications et des obligations de preuve du module (626 lignes, pour 259 annotations). C'est également la seule fonction ayant nécessité l'usage de COQ pour deux assertions.

Perspectives. Dans le futur, nous prévoyons de comparer ces deux versions avec une troisième version reposant sur l'usage d'une fonction d'observation. Dans le cas des listes, cela donne une fonction qui à une liste chaînée et un indice associe l'élément à l'indice correspondant dans la liste. La spécification du comportement des fonctions est alors faite en reliant les anciens indices des éléments aux nouveaux.

Remerciements. Ce travail a été partiellement soutenu par le CPER DATA et le projet VESSEDIA, financé par le programme européen pour la recherche et l'innovation Horizon 2020 selon la convention de subvention No 731453. Les auteurs remercient également l'équipe FRAMA-C pour les outils et le support, ainsi que Patrick Baudin, François Bobot et Loïc Correnson pour nos discussions et leurs conseils.

Références

[1] A. Blanchard, N. Kosmatov, and F. Loulergue. Logic against Ghosts : Comparison of Two Proof Approaches for a List Module. In *34^{th} Annual ACM Symposium on Applied Computing - Software Verification Track (SAC-SVT)*, Limassol, Cyprus, April 2019. ACM. Best "Software Development" Paper Award.

[2] A. Blanchard, N. Kosmatov, and F. Loulergue. Ghosts for Lists :A Critical Module of Contiki verified in FRAMA-C. In *NASA Formal Methods Symposium (NFM)*, LNCS, Newport News, VA, USA, April 2018. Springer.

[3] F. Loulergue, A. Blanchard, and N. Kosmatov. Ghosts for Lists : from Axiomatic to Executable Specifications. In *12^{th} International Conference on Tests & Proofs (TAP)*, LNCS, Toulouse, France, June 2018. Springer.

Polygraph: un modèle flot de données avec arithmétique de fréquences*

Paul Dubrulle†, Christophe Gaston‡, Nikolai Kosmatov§, Arnault Lapitre¶, and Stéphane Louise‖

CEA, List, 91191 Gif-suf-Yvette, France

Contexte. La prédiction de performance des Systèmes Cyber-Physiques (CPS) recouvre différentes caractéristiques du système, en particulier le débit, les latences bout en bout, et l'empreinte mémoire liée aux communications. Les modèles de calcul flot de données sont souvent utilisés pour effectuer ce type d'analyse. Un ordonnancement périodique du système doit exister pour analyser ses performances, et il doit vérifier deux propriétés essentielles. La première, la *cohérence*, requiert que l'empreinte mémoire soit bornée pour une répétition non-bornée de l'ordonnancement. La deuxième propriété, la *vivacité*, requiert l'absence d'interblocage.

Les formalismes flot de données existants ont une expressivité limitée pour la modélisation des CPS, en particulier pour la synchronisation des composants sur une échelle de temps commune. Pour surmonter cette limitation, nous proposons Polygraph, une extension du modèle Synchronous Data Flow (SDF). Dans SDF, les propriétés de cohérence et de vivacité sont décidables. L'extension proposée permet de modéliser précisément les contraintes de synchronisation, tout en préservant la décidabilité de ces deux propriétés.

Formalisme existant. Dans SDF, un système est modélisé comme un graphe orienté représentant les dépendances de données entre les composants du système. Chaque nœud du graphe modélise un *acteur*, une entité abstraite représentant la fonction implémentée par un composant. Chaque arc modélise un *canal* de communication, l'acteur source étant producteur de données consommées par l'acteur cible.

*Cette soumission est un résume étendu de l'article [1] paru à FASE 2019.

†paul.dubrulle@cea.fr
‡christophe.gaston@cea.fr
§nikolai.kosmatov@cea.fr
¶arnault.lapitre@cea.fr
‖stephane.louise@cea.fr

Les données échangées sur un canal sont quantifiées en nombre de *jetons*, un jeton représentant la quantité atomique de données manipulée par une opération d'écriture ou de lecture sur le canal. Chaque acteur reçoit par canal connecté un *taux* de communication entier strictement positif. L'*activation* d'un acteur est un processus atomique durant lequel l'acteur consomme et produit autant de jetons par canal que spécifié par le taux correspondant. L'activation d'un acteur a lieu de manière asynchrone à partir du moment où un nombre suffisant de jetons est présent sur tous ses canaux d'entrée. Chaque canal peut recevoir comme *marquage initial* un nombre de jetons, permettant une activation anticipée d'un acteur au départ du système.

Extension proposée. Dans Polygraph, l'extension à SDF que nous proposons, nous pouvons assigner à chaque acteur une *fréquence* de fonctionnement, correspondant à une contrainte temps-réel imposée au composant modélisé. Un acteur avec une fréquence doit s'activer à cette fréquence, de manière synchrone à une horloge globale commune à tout le système.

Nous pouvons également exprimer les taux comme des nombres rationnels $r = p/q$ de jetons. Un tel taux spécifie que l'acteur produit ou consomme p jetons toutes les q activations, et que le nombre naturel de jetons produit ou consommé par toute activation est r, arrondi soit au supérieur soit à l'inférieur. Ceci permet d'exprimer un sur-échantillonnage ou un sous-échantillonnage sur une entrée ou une sortie d'un acteur, basé sur le rapport entre les fréquences de fonctionnement des acteurs.

Nous étendons également le marquage initial de manière à avoir un nombre rationnel de jetons, et nous permettons l'affectation d'une *phase* aux acteurs avec une fréquence spécifiée, déterminant le nombre de coups d'horloge avant leur première activation.

Nous prouvons que l'on peut décider si un tel modèle vérifie ou non les propriétés de cohérence et de vivacité. Pour ce faire, nous étendons les théorèmes existants pour SDF en prenant en compte les contraintes d'activations synchrones et les phases. Pour la vivacité, nous présentons un algorithme pour vérifier la propriété en pratique, et nous donnons de premiers résultats expérimentaux en s'appuyant sur l'outil DIVERSITY.

Perspectives. Cette définition de Polygraph sera étendue pour la prise en compte de reconfigurations dynamiques du comportement d'un système modélisé, avec le même objectif de préserver la capacité à analyser le modèle statiquement. De plus, les méthodes existantes pour la prédiction de performances des modèles SDF sera étendue pour prendre en compte les spécificités de l'ordonnancement des acteurs d'un polygraphe.

Références

[1] Paul Dubrulle, Christophe Gaston, Nikolai Kosmatov, Arnault Lapitre, and Stéphane Louise. A dataflow model with frequency arithmetic. In *Proc. of the 22nd International Conference on Fundamental Approaches to Software Engineering (FASE 2019) part of the European Joint Conferences on Theory and Practice of Software (ETAPS 2019)*, Prague, Czech Republic, April 2019, pages 369-385. LNCS, vol. 11424.

METACSL : spécification et vérification de propriétés de haut niveau *

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

[1]Institut LIST, CEA, Université Paris-Saclay, Palaiseau, France,
firstname.lastname@cea.fr
[2]DGA, France, louis.rilling@irisa.fr
[3]Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes, CentraleSupélec, Université Paris-Saclay, Gif-Sur-Yvette, France,
pascale.legall@centralesupelec.fr

Contexte La vérification déductive modulaire est une technique couramment utilisée pour prouver qu'un programme respecte un certain nombre de propriétés définies par l'utilisateur. Elle se base sur l'écriture de contrats de fonctions décrivant les hypothèses requises et leur comportement attendu via des *pré-conditions* et des *post-conditions*, dans un langage de spécification formel tel qu'ACSL [2] pour les programmes écrits en C. Pour cela on peut utiliser la plateforme FRAMA-C [3], dotée d'un greffon WP pouvant générer des conditions de vérification, dont la preuve est déléguée à des solveurs SMT ou des outils interactifs tels que Coq.

Problème Cependant il n'est pas toujours aisé d'exprimer par des contrats des propriétés de haut niveau telles qu'on les trouve dans une spécification informelle, où elles s'appliquent en général à plusieurs fonctions. L'encodage sous forme de contrats d'une telle propriété nécessite d'ajouter des clauses dans les contrats des fonctions auxquelles elle s'applique. Or ces clauses n'ont pas de lien explicite entre elles, ce qui fait qu'il peut être difficile de se convaincre qu'un ensemble de contrats (même si chacun est prouvé formellement) reflète la propriété de haut niveau attendue.

Enfin dans le contexte d'une évolution constante du code et de la spécification des programmes, l'absence de mécanisme de spécification et de vérification automatique des propriétés de haut niveau ne permet pas de se convaincre facilement qu'une propriété déjà établie le reste après une mise à jour.

*Cette soumission est un résumé étendu d'un article [1], publié à TACAS 2019.

Ce besoin d'un mécanisme permettant de spécifier et vérifier facilement des propriétés de haut niveau s'est fait ressentir dans des projets passés, tels que la vérification d'un hyperviseur [4], ou encore dans une étude de cas récente concernant un système de gestion de pages mémoire avec une notion de confidentialité. Dans ce système où chaque processus et chaque page mémoire ont leur niveau de confidentialité propre, on aimerait spécifier des propriétés telles que : pour tout processus P avec un niveau de confidentialité n_P et toute page mémoire M avec un niveau n_M : (i) P ne peut pas lire M si $n_P < n_M$, (ii) si M n'est pas allouée, P ne peut pas la modifier.

Contributions Nous proposons une méthode permettant d'une part de spécifier facilement de telles propriétés de haut niveau, que l'on appellera *méta-propriétés*, et d'autre part de les vérifier automatiquement. Cette méthode est implantée dans un greffon de FRAMA-C, appelé METACSL. Elle est ensuite appliquée au cas d'étude sur la confidentialité d'un système de gestion de pages mémoire, qui est implanté en C et entièrement spécifié grâce aux méta-propriétés, lesquelles sont toutes automatiquement prouvées.

Spécification On se propose de définir une méta-propriété comme un triplet (F, P, C) avec (i) un ensemble de fonctions cibles F, (ii) une propriété P exprimée en ACSL et (iii) un *contexte* C qui définit les points dans une fonction où P doit être vérifiée (par exemple « partout dans la fonction », « seulement au début et à la fin de la fonction », « en tout point de la fonction où la mémoire est modifiée », etc.). Étant donné ces trois éléments, une méta-propriété est interprétée comme « Pour toute fonction f dans F, P est valide à tous les points de f définis par C ». Il devient alors possible de spécifier par exemple la propriété (P_1) du cas d'étude via une méta-propriété en prenant $(F, P, C) =$ (« l'ensemble de toutes les fonctions du programme », « lorsque la mémoire est lue », « pour toute page p, si son niveau de confidentialité est supérieur à celui du processus courant, alors la mémoire lue est distincte des données de p »).

Vérification Pour vérifier formellement une propriété (F, P, C) il suffit d'insérer P comme une assertion dans chaque fonction de F aux points définis par C. Ces assertions étant alors exprimées en ACSL classique, on tire parti de l'outillage existant de FRAMA-C pour les prouver automatiquement. L'insertion de P à différents endroits du code contraint la propriété à utiliser uniquement des variables globales ou propres au contexte C (par exemple « la mémoire lue » pour $C =$ « lorsque la mémoire est lue »). Cette méthode donne potentiellement lieu à un grand nombre d'assertions, qui peut être réduit par analyse syntaxique pour ne pas ralentir l'analyse du code instrumenté.

Conclusion et travaux futurs Nous avons montré la pertinence de notre approche en l'appliquant avec succès à une dizaine de propriétés intéressantes sur notre cas d'étude, lesquelles sont vérifiées entièrement automatiquement. Nous envisageons maintenant d'optimiser l'étape de vérification qui génère de nombreuses obligations de preuves et d'établir formellement la correction de cette vérification vis-à-vis de la sémantique donnée aux méta-propriétés. Enfin, il est nécessaire de continuer à expérimenter notre approche sur des cas d'études industriels et aux propriétés plus complexes.

Remerciements Ce travail a été partiellement soutenu par le projet VESSEDIA, financé par le programme européen pour la recherche et l'innovation Horizon 2020 selon la convention de subvention No 731453. Les travaux du premier auteur ont été partiellement financés par une bourse de thèse du Ministère de la Défense.

Références

[1] V. ROBLES, N. KOSMATOV, V. PREVOSTO, L. RILLING et P. LE GALL. « MetAcsl : Specification and Verification of High-Level Properties ». In : *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 2019.

[2] P. BAUDIN, P. CUOQ, J.-C. FILIÂTRE, C. MARCHÉ, B. MONATE, Y. MOY et V. PREVOSTO. *ACSL : ANSI/ISO C Specification Language*. `https://frama-c.com/acsl.html`. 2009.

[3] F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI. « Frama-C : A software analysis perspective ». In : *Formal Aspects of Computing (FAOC)* 27.3 (2015), p. 573-609.

[4] G. KLEIN, J. ANDRONICK, K. ELPHINSTONE, T. MURRAY, T. SEWELL, R. KOLANSKI et G. HEISER. « Comprehensive Formal Verification of an OS Microkernel ». In : *ACM Transactions on Computer Systems (TAOCS)* 32.1 (2014), p. 2.

Dépliage de Boucles Versus Précision Numérique

Nasrine Damouche, Xavier Thirioux
University of Toulouse,
IRIT, France
nasrine.damouche@irit.fr
xavier.thirioux@enseeiht.fr

Hanane Benmaghnia, Matthieu Martel
University of Perpignan,
LAMPS Laboratory, France
hanane.benmaghnia@etudiant.univ-perp.fr
matthieu.martel@univ-perp.fr

Résumé

Les calculs en nombres flottants sont intensivement utilisés dans divers domaines, notamment les systèmes embarqués critiques. En général, les résultats de ces calculs sont perturbés par les erreurs d'arrondi. Dans un scenario critique, ces erreurs peuvent être accumulées et propagées, générant ainsi des dommages plus ou moins graves sur le plan humain, matériel, financier, etc. Il est donc souhaitable d'obtenir les résultats les plus précis possibles lorsque nous utilisons l'arithmétique flottante. Pour remédier à ce problème, l'outil Salsa [7] permet d'améliorer la précision des calculs en corrigeant partiellement ces erreurs d'arrondi par une transformation automatique et source à source des programmes. La principale contribution de ce travail consiste à analyser, à étudier si l'optimisation par dépliage de boucles améliore plus la précision numérique des calculs dans le programme initial. À cours terme, on souhaite définir un facteur de dépliage de boucles, c'est à dire, trouver quand est-ce qu'il est pertinent de déplier la boucle dans le programme.

1 Introduction

Les progrès rapides et incessants de l'informatique ne cessent d'envahir notre vie quotidienne, allant des simples montres connectées, des smartphones à l'industrie spatiale en passant par les voitures qui sont de plus en plus sophistiquées. Ces systèmes sont dit critiques, car une petite erreur de calcul peut engendrer de graves conséquences sur la vie humaine, la finance et le matériel. Il est à noter que les ordinateurs utilisent des nombres à virgule flottante [1] qui n'ont qu'un nombre fini de chiffres. Autrement dit, l'arithmétique des ordinateurs basée sur les nombres flottants fait qu'une valeur ne peut être représentée exactement en mémoire, ce qui oblige à l'arrondir. Généralement, ces erreurs d'arrondi sont faibles mais dans un scenario critique, elles peuvent être accumulées et propagées, générant ainsi des dégâts considérables allant de l'explosion de fusées à des mauvais calculs de résultats aux jeux olympiques. Par conséquent, il est souhaitable d'obtenir les résultats les plus précis possibles lorsque nous utilisons l'arithmétique flottante. Notons, que dans l'arithmétique à virgule flottante, le parenthésage a un impact majeur sur la correction des résultats de calculs. Par exemple, il vaut mieux en général commencer par additionner les petits nombres flottants entre eux et puis les grands flottants afin d'éviter tout problème liée à l'absorption ou à l'annulation. À titre d'exemple, les deux expressions $(1.0 + 100.0^{-20}) - 100.0^{-20}$ et $1.0 + (100.0^{-20} - 100.0^{-20})$ sont équivalentes dans l'arithmétique des nombres réels cependant dans l'arithmétique des ordinateurs (arithmétique flottante), ces deux expressions retournent des résultats différents. La première formule renvoie 0.0 alors que la deuxième formule donne 1.0. Par conséquent, la fiabilité des calculs dans des contextes critiques impose de vérifier et de valider la précision des traitements numériques [18].

Plusieurs outils destinés à valider et à vérifier les programmes ont été développés comme Fluctuat [14], Astrée [6], etc. En revanche, une difficulté est que l'arithmétique des ordinateurs n'est pas intuitive, et donc détecter les erreurs ne suffit pas, il faut pouvoir les corriger. Des outils de réécriture automatique des expressions arithmétiques ont été mis en œuvre. On peut cité Sardana [16], basé sur une représentation intermédiaire nommée les APEG, et Herbie [22], basé sur une heuristique permettant de générer des tests aléatoires. À la différence, l'outil Salsa [7] prend en charge des programmes plus complets avec des affectations, conditionnelles, boucles, fonctions, etc. [8, 10], écrits dans un langage impératif. Salsa corrige partiellement les erreurs d'arrondi en transformant automatiquement en source à

source des programmes. La transformation des programmes repose sur une analyse statique par interprétation abstraite [5] qui fournit des intervalles pour les variables présentées dans les codes sources. Cette transformation est définie à l'aide de règles formelles appliquées dans un ordre déterministe afin d'optimiser les programmes en temps polynomial. D'un point de vue théorique, le programme généré après transformation ne possède pas forcément la même sémantique que celui de départ mais les programmes source et transformé sont mathématiquement équivalents pour les entrées (intervalles) considérées. De plus, le programme transformé est plus précis. La correction de notre approche repose sur une preuve mathématique par induction comparant le programme transformé avec celui d'origine [9]. Les résultats obtenus par Salsa sont très prometteurs. Nous avons montré, à travers une suite de programmes provenant de systèmes embarqués et de méthodes d'analyse numérique, que nous améliorons significativement la précision numérique des calculs en minimisant l'erreur par rapport à l'arithmétique exacte des réels.

La principale contribution de cet article consiste à étudier si l'optimisation par dépliage de boucles améliore plus la précision numérique des calculs dans le programme original. Ce dépliage de boucles est défini par le fait d'écrire le corps de la boucle plusieurs fois dans l'optique de réduire le nombre d'itérations. Cependant, il est strictement indispensable de s'assurer qu'il n'y ait pas de dépendances entre les instructions. Cette technique est mise en œuvre dans tous les compilateurs. De plus, cette étude nous permettra de définir par la suite un facteur de dépliage de boucles. Plus précisément, trouver quand est-ce qu'il est pertinent de déplier le corps de la boucle dans le programme.

Cet article est organisé comme suit. La section 2 rappelle brièvement la norme IEEE754. La section 3 donne un bref aperçu sur le fonctionnement de Salsa. La section 4, détaille la principale contribution de cet article. La section 5, décrit les différents résultats expérimentaux obtenus avec Salsa. La section 6 résume nos travaux et ouvre sur quelques perspectives.

2 Arithmétique des nombres flottants

2.1 La norme IEEE754

La norme IEEE754 est le standard permettant de spécifier l'arithmétique à virgule flottante [1, 21]. Les nombres réels ne peuvent être représentés exactement en mémoire sur machine. A cause des erreurs d'arrondi apparaissant lors des calculs, la précision des résultats numériques est généralement peu intuitive. Un nombre x en virgule flottante, en base b, est défini par : $x = s \cdot m \cdot b^{e-p+1}$, avec, $s \in \{0,1\}$ le signe, m la mantisse et e l'exposant. Le standard IEEE754 décrit quatre modes d'arrondi pour un nombre x à virgule flottante : vers $+\infty$ ($\uparrow_{+\infty}(x)$), vers $-\infty$ ($\uparrow_{-\infty}(x)$), vers 0 ($\uparrow_0(x)$) et plus près ($\uparrow_\sim(x)$). Il est à noter que nos techniques de transformation ne dépendent pas d'un mode d'arrondi précis.

La sémantique des opérations élémentaires définie par le standard IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$ cités précédemment pour $\uparrow_r : \mathbb{R} \to \mathbb{F}$, est donnée par :

$$x \circledast_r y = \uparrow_r (x * y) \ , \tag{1}$$

avec, $\circledast_r \in \{+, -, \times, \div\}$ une des quatre opérations élémentaires utilisées pour le calcul des nombres flottants en utilisant le mode d'arrondi r et $* \in \{+, -, \times, \div\}$ l'opération exacte (opérations sur les réels). Clairement, les résultats des calculs à base des nombres flottants ne sont pas exacts et ceci est dû aux erreurs d'arrondi. Par ailleurs, on utilise la fonction $\downarrow_r : \mathbb{R} \to \mathbb{R}$ permettant de renvoyer l'erreur d'arrondi du nombre en question. Cette fonction est définie par :

$$\downarrow_r (x) = x - \uparrow_r (x) \ . \tag{2}$$

2.2 Calcul de bornes d'erreur

Pour calculer les erreurs se glissant durant l'évaluation des expressions arithmétiques, nous définissons des valeurs non standard faites d'une paire $(x, \mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$, où la valeur x représente un nombre flottant et μ l'erreur exacte liée à x. Plus précisément, μ est la différence exacte entre la valeur réelle et flottante de x comme défini par l'équation (2). La sémantique concrète des opérations élémentaires dans $\mathbb{E}$ est détaillée dans [17].

La sémantique abstraite associée à $\mathbb{E}$ utilise une paire d'intervalles $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, tel que le premier intervalle $x^\sharp$ contient les nombres flottants du programme, et le deuxième intervalle $\mu^\sharp$ contient les erreurs sur $x^\sharp$ obtenues en soustrayant le nombre flottant de la valeur exacte. Cette valeur abstrait un ensemble de valeurs concrètes $\{(x,\mu) : x \in x^\sharp \text{ et } \mu \in \mu^\sharp\}$. Revenons maintenant à la sémantique des expressions arithmétiques dont l'ensemble des valeurs abstraites est noté par $\mathbb{E}^\sharp$. Un intervalle $x^\sharp$ est approché avec un intervalle défini par l'équation (3) qu'on note $\uparrow^\sharp (x^\sharp)$.

$$\uparrow^\sharp ([\underline{x},\overline{x}]) = [\uparrow (\underline{x}), \uparrow (\overline{x})] \ . \tag{3}$$

La fonction d'abstraction $\downarrow^\sharp$, quant à elle, abstrait la fonction concrète $\downarrow$, autrement dit, elle permet de sur-approcher l'ensemble des valeurs exactes d'erreur, $\downarrow (x) = x - \uparrow (x)$ de sorte que chaque erreur associée à l'intervalle $x \in [\underline{x},\overline{x}]$ est incluse dans $\downarrow^\sharp ([\underline{x},\overline{x}])$. Pour un mode d'arrondi au plus proche, la fonction d'abstraction est donnée par l'équation (4).

$$\downarrow^\sharp ([\underline{x},\overline{x}]) = [-y, y] \quad \text{avec} \quad y = \frac{1}{2}\text{ulp}\Big(\max(|\underline{x}|, |\overline{x}|)\Big) \ . \tag{4}$$

En pratique l'ulp(x), qui est une abréviation de *unit in the last place*, représente la valeur du dernier chiffre significatif d'un nombre à virgule flottante x. Formellement, la somme de deux flottants revient à additionner les erreurs générées par l'opérateur avec l'erreur causée par l'arrondi du résultat (voir équation (5)). Similairement pour la soustraction de deux flottants, on soustrait les erreurs sur les opérateurs et on les ajoute aux erreurs apparues au moment de l'arrondi. Quant à la multiplication de deux nombres à virgule flottante, la nouvelle erreur est obtenue par développement de la formule $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$ (voir équation (6)).

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)) \ , \tag{5}$$

$$(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)) \ . \tag{6}$$

Notons qu'il existe d'autres domaines abstraits plus efficaces, à titre d'exemple [4, 14, 15], et aussi des techniques complémentaires comme [2, 3, 11, 23]. De plus, on peut faire référence à des méthodes qui transforment, synthétisent ou réparent les expressions arithmétiques basées sur des entiers ou sur la virgule fixe [13]. On citera également [4, 12, 19, 20, 23] qui s'intéressent à améliorer l'intervalle de valeurs des variables à virgule flottante.

3 L'outil Salsa

Dans cette section, nous rappelons brièvement le fonctionnement de l'outil Salsa. Basé sur les méthodes d'analyse statique par interprétation abstraite, Salsa transforme automatiquement des programmes issus de l'arithmétique des flottants pour améliorer leur précision de calculs numériques. Cette transformation concerne les morceaux de code tels que les affectations, conditionnelles, les boucles, les fonctions, etc. Pour réaliser cette transformation, un ensemble de règles de transformations formelles permettant de réécrire les programmes en des programmes plus précis a été défini et implémenté dans Salsa. À titre d'exemples de nos règles de transformations, prenons la conditionnelle if_Φ e then $c1$ else $c2$. Un premier cas de règle stipule que si on connaît statiquement l'évaluation de la condition e, en d'autre termes, si la condition est toujours vraie (respectivement est toujours fausse), on transforme uniquement la branche `then` soit $c1$ (respectivement la branche `else` soit $c2$), sinon, on transforme les deux branches de la conditionnelle. Salsa prend en entrée un programme initialement écrit dans un langage impératif (langage C) ainsi que des intervalles sur les valeurs des variables de ce programme, et retourne en sortie un programme numériquement plus précis écrit dans le même langage avec des intervalles sur les valeurs des bornes d'erreurs avant et après la transformation de ce programme. En d'autre termes, Salsa minimise l'erreur de calcul de programmes. Il est à noter que pour chaque programme, il faut spécifier la variable de référence à être optimiser par Salsa. Cette variable correspond à la valeur retournée par le programme. Aussi, il est à rappeler que le programme initial et le programme transformé n'ont pas forcément la même sémantique mais mathématiquement sont équivalents. De plus, Salsa réécrits grâce à son analyseur statique les programmes donnés en entrée sous forme SSA pour *Static Single Assignment* afin que chaque variable soit écrite uniquement une seule fois dans le code source, ce qui évite les

confusions causées par la lecture et l'écriture d'une même variable dans le programme. Les différentes règles de transformation sont utilisées dans un ordre déterministe, c'est-à-dire qu'elles sont appliquées l'une après l'autre. La transformation est répétée jusqu'à ce que le programme final ne change plus. Un programme transformé est plus précis que le programme initial si et seulement si :

1. La variable retournée correspond mathématiquement à la même expression mathématique dans les deux programmes.
2. L'erreur de la valeur abstraite de cette variable de référence est plus petite que l'erreur dans la deuxième valeur abstraite.

Pour plus de détail sur le fonctionnement de Salsa, son architecture ainsi les différentes règles de transformations, nous redirigeons le lecteur vers la référence [7].

4 Dépliage de Boucles

La principale contribution de cet article est d'étudier et d'analyser l'impact de dépliages de boucles sur la précision numérique des calculs. En d'autre termes, l'idée consiste à comparer la précision numérique des programmes initiaux dépliés plusieurs fois avec celle correspondante aux programmes transformés avec Salsa pour le même nombre de dépliages. Lors du dépliage de boucles, on ré-écrit le corps des boucles plusieurs fois tout en tenant compte des différentes dépendances. Notons que le plus qu'il en existe des dépendances dans un programme, le plus qu'on a la possibilité d'en construire de larges expressions que nous parserons de différentes manières afin d'en trouver le meilleur programme en terme de précision lors de la transformation. La figure 1 donne un exemple de dépliage de corps de boucle du programme PID. En pratique, déplier le corps de boucles plusieurs fois revient à créer beaucoup de calculs au sein d'un même programme. Lorsqu'on donne ce programme déplié à Salsa, ce dernier le transforme (réécrit) de façon à trouver un meilleur programme en terme de parenthésage minimisant ainsi les erreurs de calculs.

Dans notre cas d'étude, on compare la précision numérique de chaque programme avant et après transformation et ce pour différent nombre de dépliages. Pour ce faire, on a :

— Déplié le corps de boucle de chaque programme plusieurs fois,
— Transformé le programme déplié avec Salsa,
— Déplié par la suite le programme transformé,
— Calculé l'erreur relative pour les deux programmes,
— Mesuré le temps de calcul pour les deux programmes,
— Calculé le gain en terme de précision numérique,
— Comparé les valeurs obtenues.

```
m0 = [0.0,8.0]
%salsa%
double main() {
invdt = 5.0; kp = 9.4514; ki = 0.69006;
kd = 2.8454; eold = 0.0; dt = 0.2;
c = 5.0; i0 = 0.0; t = 0.0;
  while (t < 200.0) {
    e = c - m0;
    p = kp * e;
    i = i0 + ki * dt * e;
    d = kd * invdt * (e - eold);
    r = p + i + d;
    m = m0 + 0.01 * r;
    eold = e;
    i0 = i ;
    t = t + dt ;
  }
 return m;
}
```

(a)

```
m0 = [0.0,8.0]
%salsa%
double main() {
invdt = 5.0; kp = 9.4514; ki = 0.69006; kd = 2.8454;
eold = 0.0; dt = 0.2; c = 5.0; i0 = 0.0; t = 0.0;
  while (t < 100.0) {
    e = c - m0 ;
    p = kp * e ;
    i = i0 + ki * dt * e ;
    d = kd * invdt * (e - eold);
    r = p + i + d;
    m = m0 + 0.01 * r ;
    eold = e ;
    e1 = c - m ;
    p1 = kp * e1 ;
    i1 = i + ki * dt * e1 ;
    d1 = kd * invdt * (e1 - eold);
    r1 = p1 + i1 + d1;
    m1 = m + 0.01 * r1 ;
    eold = e1 ;
    t = t + 2.0 * dt ;
    m0 = m1 ; i0 = i1 ;
  }
 return m1 ;
}
```

(b)

FIGURE 1 – (a) Le programme PID initial. (b) Le programme PID avec un seul dépliage.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[0.18982e^{-13} , **0.52832e^{-13}**]	[0.23766e^{-13} , **0.44858e^{-13}**]	0.185s	15,09
2	[0.19081e^{-13} , **0.53116e^{-13}**]	[0.23912e^{-13} , **0.45065e^{-13}**]	0.419s	15,15
3	[0.19219e^{-13} , **0.53438e^{-13}**]	[0.24096e^{-13} , **0.45310e^{-13}**]	0.844s	15,21
4	[0.19395e^{-13} , **0.53798e^{-13}**]	[0.24318e^{-13} , **0.45594e^{-13}**]	1.428s	15,24
5	[0.19610e^{-13} , **0.54196e^{-13}**]	[0.24578e^{-13} , **0.45915e^{-13}**]	2.121s	15,27

FIGURE 2 – Mesure d'erreur relative avant et après transformation pour un dépliage allant de 1 à 5, gain de précision et temps de transformation du programme PID.

Il est à noter que le dépliage de corps de boucles de chaque programme est effectué à la main et qu'il est effectué de un à 9. Pour des raisons de simplification, les tableaux de la Section 5 montrent les résultats obtenus pour un dépliage allant jusqu'à 5.

5 Résultats expérimentaux

De nombreux tests ont été menés sur plusieurs exemples provenant des systèmes embarqués et des méthodes d'analyse numérique afin d'évaluer l'efficacité de Salsa pour les entrées (intervalles) considérées. Les résultats obtenus, sur une suite de programmes provenant de l'avionique (PID), de robotique (Odometry) ainsi que les méthodes numériques (Runge-Kutta d'ordre 2), sont concluants. Notons que la taille initiale de programme PID et Runge-Kutta d'ordre 4 est sur une vingtaine de lignes de code alors que le programme d'Odometry est sur une soixantaine de lignes de code. Dans la suite de cette section, on calcule le gain en terme de précision numérique pour chacun de ces programmes ainsi que le temps de transformation nécessaire pour chaque dépliage sur chaque programme considéré.

5.1 Contrôleur PID

Le contrôleur PID est un programme très utilisé dans l'avionique. Il permet de maintenir une mesure physique `m` à une certaine valeur qu'on appelle consigne `c`. Le programme du PID initial ainsi celui déplié une seule fois sont donnés par la Figure 1.

Les différentes mesures effectuées sur le programme PID sont données par la figure 2. La première colonne illustre le nombre de dépliages réalisé. Les colonnes 2 et 3 donnent respectivement la valeur de l'erreur relative du programme avant et après transformation avec Salsa. La colonne 4 illustre le temps de calcul nécessaire pour transformer les programmes. La dernière colonne calcule le gain (en pourcentage) obtenu en terme de précision numérique. Les résultats obtenus montrent qu'en dépliant le corps de la boucle de programme PID, la précision est améliorée en moyenne (géométrique) de **15.23%**. Et si on

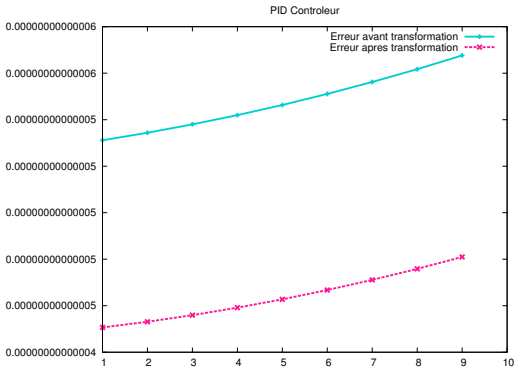


FIGURE 3 – Représentation graphique de l'erreur absolue avant et après transformation du Contrôleur PID pour un dépliage allant de 1 à 9.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[+0.53137e^{-14} , **+0.53843e^{-14}**]	[+0.43109e^{-14} , **+0.43676e^{-14}**]	0m26.065s	18,88
2	[+0.12737e^{-13} , **+0.13009e^{-13}**]	[+0.98486e^{-14} , **+0.10038e^{-13}**]	1m53.330s	22,83
3	[+0.21921e^{-13} , **+0.22527e^{-13}**]	[+0.16526e^{-13} , **+0.16899e^{-13}**]	4m53.642s	24,98
4	[+0.28899e^{-13} , **+0.29575e^{-13}**]	[+0.22501e^{-13} , **+0.22930e^{-13}**]	9m34.918s	22,46
5	[+0.40266e^{-13} , **+0.41518e^{-13}**]	[+0.31046e^{-13} , **+0.31675e^{-13}**]	16m49.551s	23,70

FIGURE 4 – Mesure d'erreur relative avant et après transformation, gain de précision et temps de transformation du programme Odometry.

observe le temps de transformation nécessaire pour chaque dépliage, on remarque que le programme PID nécessite 2.121 secondes lorsqu'on le déplie 5 fois. La figure 3 montre que nous améliorons de manière significative la précision numérique en minimisant l'erreur de calcul du programme PID pour un dépliage allant de 1 à 9 fois.

5.2 Odometry

Le deuxième exemple considéré est l'odometry. Ce programme calcule la position instantanée d'un robot à deux roue avec la méthode d'odométrie. Les résultats donné par la Figure 4 obtenus sur le programme Odometry montrent que la précision a été amélioré en moyenne (géométrique) de **23.26%**. Par exemple, l'erreur de calcul pour ce programme passe de **+0.41518e^{-13}** à **+0.31675e^{-13}** pour un dépliage de 5. Ces résultats illustrent l'efficacité de la transformation du programme déplié. Figure 5 illustre l'amélioration de la précision numérique du programme Odometry pour un dépliage allant de 1 à 6 fois.

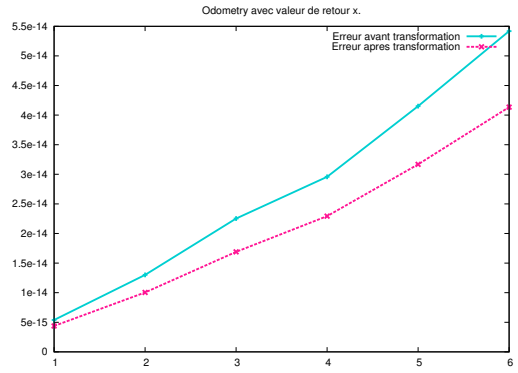


FIGURE 5 – Représentation graphique de l'erreur absolue avant et après transformation du programme Odometry pour un dépliage allant de 1 à 6.

5.3 Runge-Kutta d'ordre 2

La méthode de Runge-Kutta d'ordre 2 (RK2), est une méthode numérique couramment utilisée pour résoudre les équations différentielles ordinaires (EDO).

La figure 6 illustre les résultats obtenus en terme d'erreur relative avant et après chaque transformation pour les différents dépliages ainsi temps de transformation correspondant dans chaque cas pour le programme RK2. Ces résultats montrent que la précision numérique a été améliorée en moyenne (géométrique) de **63.50%**. Si nous observons aussi le temps de transformation nécessaire, nous remarquons que l'erreur de calcul pour ce programme passe de **+0.94748e^{-14}** à **+0.27507e^{-14}** pour un dépliage de 5. Ces résultats illustrent l'efficacité de la transformation du programme déplié encore une fois. Figure 7 illustre l'amélioration de la précision numérique du programme Runge-Kutta d'ordre 2 pour un dépliage allant de 1 à 9 fois.

Nbr de dépliages	Erreur avant transformation de programme	Erreur après transformation de programme	Temps de transformation	Gain %
1	[+0.53734e^{-15} , **+0.53734e^{-15}**]	[+0.24091e^{-15} , **+0.24091e^{-15}**]	0.079s	55,16
2	[+0.14903e^{-14} , **+0.14903e^{-14}**]	[+0.59159e^{-15} , **+0.59159e^{-15}**]	0.193s	60,30
3	[+0.30638e^{-14} , **+0.30638e^{-14}**]	[+0.10899e^{-14} , **+0.10899e^{-14}**]	0.402s	64,42
4	[+0.55726e^{-14} , **+0.55726e^{-14}**]	[+0.17867e^{-14} , **+0.17867e^{-14}**]	0.772s	67,93
5	[+0.94748e^{-14} , **+0.94748e^{-14}**]	[+0.27507e^{-14} , **+0.27507e^{-14}**]	1.384s	70,96

FIGURE 6 – Mesure d'erreur relative avant et après transformation, gain de précision et temps de transformation du programme Runge-Kutta d'ordre 2.

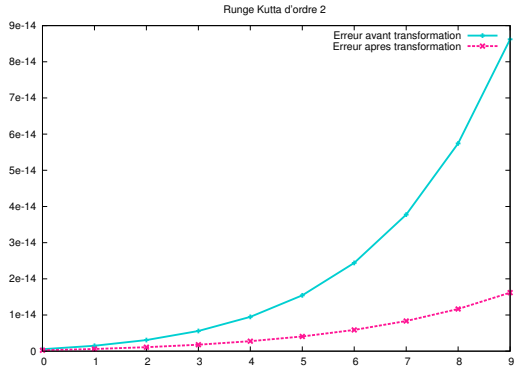


FIGURE 7 – Représentation graphique de l'erreur absolue avant et après transformation du programme Runge-Kutta d'ordre 2 pour un dépliage allant de 1 à 9.

6 Conclusion

L'objectif principal de notre travail est d'étudier l'impact de dépliage de boucles sur la précision numérique des calculs ainsi sur le temps de transformation correspondant. Les résultats obtenus montrent qu'en dépliant le corps de boucles plusieurs fois, la précision numérique des programmes est améliorée. Cette technique est très efficace pour améliorer la précision numérique lorsqu'il existe une forte dépendance entre les instructions de calcul dans le programme.

Une perspective consiste à étendre nos méthodes de transformation automatique de programmes pour améliorer la précision numérique de calculs en dépliant le corps de boucles dans un programme. Autrement dit, nous souhaiterons définir des règles de dépliage de boucles pour transformer les programmes. Un point très ambitieux porte sur les problèmes de reproductibilité des résultats, plus précisément, plusieurs exécutions d'un même programme donne des résultats différents et ce à cause de la variabilité de l'ordre d'exécution des expressions mathématiques.

Références

[1] ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.

[2] E-T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 549–560. ACM, 2013.

[3] F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.

[4] J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT*, 36(1) :1–8, 2011.

[5] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages, POPL*, pages 238–252, 1977.

[6] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astreé analyzer. In S. Sagiv, editor, *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Proceedings*, volume 3444 of *LNCS*, pages 21–30. Springer, 2005.

[7] N. Damouche and M. Martel. Salsa : An automatic tool to improve the numerical accuracy of programs. In B. Dutertre and N. Shankar, editors, *Automated Formal Methods, AFM@NFM 2017, Moffett Field, CA, USA, May 19-20, 2017.*, volume 5 of *Kalpa Publications in Computing*, pages 63–76. EasyChair, 2017.

[8] N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *Formal Methods for Industrial Critical Systems - 20th International Workshop, FMICS 2015, Oslo, Norway, June 22-23, 2015 Proceedings*, volume 9128 of *LNCS*, pages 31–46. Springer, 2015.

[9] N. Damouche, M. Martel, and A. Chapoutot. Improving the numerical accuracy of programs by automatic transformation. *STTT*, 19(4) :427–448, 2017.

[10] N. Damouche, M. Martel, and A. Chapoutot. Numerical accuracy improvement by interprocedural program transformation. In S. Stuijk, editor, *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems, SCOPES*, pages 1–10. ACM, 2017.

[11] E. Darulova and V. Kuncak. Sound compilation of reals. In S. Jagannathan and P. Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.

[12] J. Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 33–48. Springer, 2004.

[13] X. Gao, S. Bayliss, and G-A. Constantinides. SOAP : structural optimization of arithmetic expressions for high-level synthesis. In *Field-Programmable Technology, FPT*, pages 112–119. IEEE, 2013.

[14] E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *LNCS*, pages 1–3. Springer, 2013.

[15] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.

[16] A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In Antoine Miné and David Schmidt, editors, *Static Analysis - 19th International Symposium, SAS*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.

[17] M. Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1) :7–30, 2006.

[18] M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *ENTCS*, 287 :3–16, 2012.

[19] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004.

[20] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[21] J-M. Muller, N. Brisebarre, F. De Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.

[22] J. R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.

[23] A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.

Contribution à la formalisation des propriétés graphiques des systèmes interactifs pour la validation automatique

Pascal Béger[1], Valentin Becquet[1], Sébastien Leriche[1], et Daniel Prun[1]

[1]ENAC, Université de Toulouse, France

Résumé

Dans la plupart des secteurs, les systèmes d'aujourd'hui sont interactifs et disposent d'interfaces graphiques sophistiquées. A notre connaissance, il existe peu d'études sur la vérification des propriétés spécifiques de la scène graphique des interfaces homme-machine et en particulier sur celles liées à la visibilité des composants graphiques par l'utilisateur humain. Dans ce papier, nous introduisons notre formalisme permettant de spécifier certaines propriétés des composants graphiques, dans l'objectif de réaliser à terme un outillage de validation automatique de ces propriétés. Nous illustrons l'usage de ce formalisme pour des propriétés extraites d'une norme décrivant un instrument critique présent dans les cockpits d'avions commerciaux. Nous avons développé une version complètement fonctionnelle de cet instrument au moyen de Smala, un langage interactif de haut niveau produit par notre équipe. Cela nous permet de montrer la manière dont nous envisageons les opérations de validation formelle automatique de propriétés graphiques de tels systèmes.

1 Introduction

Les systèmes interactifs sont des systèmes informatiques réactifs qui traitent des informations (clics de souris, entrées de données, etc.) provenant de leur environnement (autres systèmes ou humains) et produisent une représentation (notifications sonores, représentations visuelles, etc.) de leur état interne. Ils sont devenus largement répandus dans différents secteurs tels que l'aéronautique, le spatial, le médical ou les applications mobiles. Ces systèmes prennent de plus en plus en compte l'utilisateur humain par le biais de nouvelles interactions et proposent de nouvelles interfaces riches en composants graphiques et en interactions sophistiquées.

Pour valider ou certifier ces systèmes, les outils issus des méthodes formelles ne sont pas toujours adaptés pour prendre en compte les nouvelles interactions et considérations de l'humain.

En analysant les travaux sur les méthodes formelles appliquées aux systèmes interactifs tels que [13, 19, 5], nous avons constaté que les propriétés liées à la scène graphique et en particulier la visibilité des composants graphiques par l'utilisateur humain étaient peu étudiées. Cela peut s'expliquer par le fait qu'historiquement les programmes informatiques ne possédaient pas d'interface graphique (ou que celle-ci restait peu développée) tandis qu'aujourd'hui nous trouvons des systèmes avec des interfaces où la scène graphique est riche et dynamique. De plus, la conception de la scène graphique suit souvent une norme ou des règles de conception imposées [26, 27]. De ce fait, l'étude formelle de ces propriétés reste considérée comme peu pertinente, le processus de validation reposant généralement sur une étude manuelle du système, en suivant des checklists afin d'assurer que pour tout scénario d'utilisation du système, la scène graphique respecte les exigences imposées.

Ce papier suit les travaux que nous avons présentés dans Béger *et al.* [6] afin de contribuer à la certification des systèmes interactifs en utilisant le langage **Smala** [1] et son environnement d'exécution **Djnn** [2]. Nous pensons que la formalisation de ces propriétés graphiques permettra d'automatiser le processus de validation de la scène graphique en fonction des exigences et en particulier celles liées à la visibilité des composants graphiques qui nous serviront d'exemple

1. `http://smala.io`
2. `http://djnn.net`

dans cet article. Bien que ces propriétés graphiques pourraient paraître simples, nous estimons que le gain, en temps et en réduction d'erreurs humaines, apporté par ce processus de validation automatique serait important.

Après avoir réalisé un état de l'art sur les propriétés étudiées pour les systèmes interactifs et sur les variables décrivant les éléments graphiques, nous présentons notre formalisme permettant de définir des propriétés graphiques. Nous introduisons ensuite notre cas d'étude, le système d'alerte de trafic et d'évitement de collision TCAS (*Traffic alert and Collision Avoidance System*), ainsi que les propriétés issues de la norme associée à l'instrument IVSI (*Instantaneous Vertical Speed Indicator*) qui produit la visualisation. Nous illustrons l'usage de notre formalisme en définissant formellement des propriétés graphiques issues de la norme puis, à partir de l'implémentation de l'instrument dans le langage Smala, nous montrons comment nous envisageons la validation automatique de ces propriétés.

2 État de l'art

Historiquement, les premières propriétés étudiées pour les logiciels et systèmes informatiques concernaient la sûreté (e.g. absence d'événement indésirable, bornitude) ainsi que la vivacité des programmes (e.g. retour à un état donné, absence d'interblocage) [23]. Les principales méthodes utilisées pour vérifier et valider formellement des systèmes sont la vérification de modèle par modèle checking [2], par preuve mathématique [4], l'analyse statique [15] ainsi que les processus de tests. Cependant, l'évolution de ces systèmes et l'apparition des IHM (Interfaces Homme-Machine) modernes font émerger de nouvelles propriétés qui challengent ces méthodes.

Une partie de ces nouvelles propriétés concernent le comportement de l'utilisateur : objectifs de l'utilisateur [7], attention de l'utilisateur [25], expérience et apprentissage de l'utilisateur [8]. Cerone et Elbegbayan [9] ont par exemple modélisé le comportement de l'utilisateur au cours des interactions avec une interface web représentant un forum de discussion. Ce modèle de comportement consiste à définir les objectifs de l'utilisateur, par exemple l'envoi de message sur le forum, pour ensuite restreindre les actions possibles de l'utilisateur en implémentant des contraintes dans l'interface par le biais de privilèges en fonction du niveau de l'utilisateur (e.g. connecté ou non).

Nous identifions également la classe des propriétés relatives aux IHM adressant par exemple la latence [20] ou les propriétés CARE [11]. Masci *et al.* [22] ont étudié la prédictabilité de l'IHM d'une pompe à perfusion, i.e. l'utilisateur est conscient de l'état interne du système par le biais des informations données par l'IHM et peut prédire son état interne futur en fonction des saisies de données possibles.

Une autre classe de propriétés concerne la sécurité [18]. Rukšėnas *et al.* [24] ont utilisé une architecture cognitive du comportement de l'utilisateur modélisée en SAL [14] afin d'étudier les risques de failles de sécurité dans le cadre de l'utilisation d'un système d'authentification.

Au niveau de la représentation graphique des données ou de l'état du système, Hjelmslev [16] donne deux plans permettant de définir un élément graphique et l'information qu'il contient. Cette information appartient au plan du contenu, aussi appelé plan des signifiés. L'élément graphique appartient au plan de l'expression ou plan des signifiants. Le signifiant peut être caractérisé par des paramètres associés à des éléments graphiques tels que les variables visuelles définies par Bertin et Barbut [3] (coordonnées, taille, valeur, grain, couleur, orientation et forme) et les attributs esthétiques de Wilkinson [28] (transparence). Jacques Bertin, qui était cartographe, portait son intérêt principalement sur les éléments graphiques placés sur une feuille de papier. De ce fait, les coordonnées étudiées se limitaient au plan (x, y). Dans le cas des systèmes informatiques où les IHM ont un affichage dynamique, il faut également ajouter la variable l correspondant à l'ordre ou couche d'affichage des composants graphiques. La couche d'affichage et le dynamisme de l'affichage induisent des propriétés liées à la superposition.

3 Cas d'étude : TCAS

Le TCAS (*Traffic alert and Collision Avoidance System*) est un système aéronautique ayant pour objectif d'améliorer la sécurité aérienne en prévenant les collisions entre aéronefs (avions, hélicoptères, etc.). Il est obligatoire sur tous les avions commerciaux (transportant plus de 19 passagers ou pesant plus de 5.7 tonnes en Europe). Il est défini par la norme ED-143 [1] et on peut trouver dans Williams [29] des informations détaillées de différents scénarios d'utilisation, la logique de fonctionnement ou les différents affichages du système.

a. IVSI - Compteur de vitesse verticale (pieds×1000)
b. IVSI - Aiguille indiquant la vitesse verticale
c. TA - Position de l'avion (*own aircraft*)
d. TA - Portée de 2 NM autour de *own aircraft*
e. TA - Echelle de l'affichage du trafic environnant
f. TA - Trafic de niveau de menace *threat aircraft*
g. TA - Trafic de niveau de menace *intruder aircraft*
h. TA - Trafic de niveau de menace *proximate aircraft*
i. RA - Vitesse verticale à atteindre
j. RA - Vitesse verticale à éviter

FIGURE 1 – TCAS implémenté sur l'IVSI et réalisé en Smala

Par le biais de calculs effectués sur les informations reçues du trafic environnant, le TCAS fournit deux services aux pilotes : les TAs (*traffic alerts*) et les RAs (*resolution advisories*). Les TAs permettent d'informer les pilotes du type de trafic environnant (*threat aircraft*, *intruder aircraft*, *proximate aircraft*, *other aircraft*) en y associant une symbolique (forme et couleur). Les RAs permettent de notifier les pilotes des manœuvres à effectuer afin d'éviter tout risque de collision (atteindre une vitesse verticale donnée, conserver la vitesse verticale actuelle). Ces deux services sont fournis sous forme de notifications sonores et visuelles, ce qui fait du TCAS un **système critique multimodal**.

Un des éléments du TCAS est l'instrument intégré au cockpit fournissant la visualisation. Il en existe plusieurs. Nous présentons l'IVSI (*Instantaneous Vertical Speed Indicator*, figure 1) qui indique aux pilotes la vitesse verticale actuelle de leur aéronef, c'est-à-dire la vitesse instantanée de montée ou de descente. Cet instrument accueille aussi la visualisation des informations élaborées par le TCAS. Ainsi, un RA est visualisé par l'affichage conjoint d'un arc rouge, indiquant au pilote la plage de vitesses verticales à éviter, et d'un arc vert désignant la vitesse cible qu'il doit atteindre.

Nous avons extrait de la norme ED-143 les exigences relatives à la visualisation du TCAS, applicables à l'IVSI. Ces exigences, représentatives de la complexité de la norme graphique, sont présentées dans le tableau 1. Nous avons mis en évidence (cases sur fond gris) certaines des exigences qui serviront d'exemples pour la formalisation dans la section suivante.

4 Contribution

Notre objectif est de pouvoir vérifier automatiquement que l'implémentation de l'IVSI respecte les éléments issus de la norme ED-143. Pour cela, nous proposons ici une première formalisation des propriétés graphiques issues de cette norme. Nous appliquons ensuite ce formalisme à notre exemple d'instrument IVSI. Nous montrons enfin comment pourra se faire la vérification de ces propriétés à partir du graphe de scène extrait du code Smala, le langage qui nous a servi pour l'implémentation.

Ex.	Information	Forme	Couleur	Position
E_1	Propre position relative (*own aircraft*)	Avion (3 traits)	Blanc ou cyan ($\neq$ couleur *proximate* / *other aircraft*)	Centré horizontalement, 1/3 hauteur affichage
E_2	Portée de 2 NM autour de *own aircraft*	8 traits en cercle	Couleur *own aircraft*	Centré sur *own aircraft*, rayon en fonction de la portée
E_3	TA - Trafic de type *threat aircraft*	Carré plein	Rouge	Zone délimitée par le compteur de vitesse
E_4	TA - Trafic de type *intruder aircraft*	Cercle plein	Jaune ou ambre	Zone délimitée par le compteur de vitesse
E_5	TA - Trafic de type *proximate aircraft*	Losange plein	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_6	TA - Trafic de type *other aircraft*	Losange vide	Blanc ou cyan ($\neq$ couleur *own aircraft*)	Zone délimitée par le compteur de vitesse
E_7	TA - Altitude relative (en centaine de pieds)	"+" / "-" et deux chiffres	Couleur trafic	Au-dessus / en dessous du symbole du trafic
E_8	TA - Sens vertical	Flèche verticale (haut ou bas)	Couleur trafic	A droite du symbole du trafic
E_9	RA - Vitesse à atteindre	Arc de cercle	Vert	Compteur de vitesse
E_{10}	RA - Vitesse à éviter	Arc de cercle	Rouge	Compteur de vitesse

TABLE 1 – Liste des exigences relatives au graphique du TCAS implémenté sur l'IVSI

4.1 Formalisation des propriétés graphiques

Afin de définir formellement les propriétés liées à la scène graphique, nous considérons les variables visuelles suivantes : **coordonnées**, **couche (l)**, **taille**, **couleur**, **orientation**, **forme** et **transparence**. Il est à noter que nous proposons pour le moment une définition mathématique générale que nous restreindrons dans la suite de nos travaux en fonction des techniques de validation formelle que nous utiliserons.

Les éléments graphiques considérés peuvent être définis complètement à l'aide d'un point spécifique (origine) et d'un ensemble de grandeurs caractérisitiques gc. Par exemple, un rectangle peut être défini à l'aide de son origine (coin supérieur gauche), sa hauteur h et sa largeur w. Les coordonnées x et y représentent des pixels et sont des entiers naturels. La couche d'affichage l est différente pour chaque élément graphique. $l = 0$ indique la couche la plus profonde et donc la première affichée.

Définitions

Afin d'aider à la compréhension des propriétés, voici un ensemble de définitions relatives aux termes et symboles employés. Nous précisons entre des parenthèses les variables visuelles concernées.

- $e_i = \langle (x_0, y_0), l, gc, c, o \rangle$: élément graphique (**coordonnées, couche, taille, forme, couleur, transparence**)
- $(x_0(e_i), y_0(e_i)) \in \mathbb{N} \times \mathbb{N}$: coordonnées de l'origine de e_i (**coordonnées, forme**),
- $l(e_i) \in \mathbb{N}$: ordre ou couche d'affichage de e_i (**couche**) et $(e_1 \neq e_2) \Leftrightarrow (l(e_1) \neq l(e_2))$,
- $c(e_i) = \langle c_r(e_i), c_g(e_i), c_b(e_i) \rangle, c_j(e_i) \in [0, 255]$: couleur de e_i en RGB (**couleur**),
- $o(e_i) \in [0, 100]$: coefficient d'opacité de e_i (**transparence**),
- $\mathcal{D}(e_i)$: domaine de e_i (**coordonnées, forme, taille**).
  Le domaine est calculé grâce aux coordonnées de l'origine et des grandeurs caractéristiques de e_i. Pour un rectangle dont la largeur est parallèle à l'axe des abscisses : $\mathcal{D}(e_i) = \{(x, y) \mid x \in [x_0(e_i), x_0(e_i) + w(e_i)], y \in [y_0(e_i), y_0(e_i) + h(e_i)]\}$

Propriétés

Nous avons identifié un premier ensemble de propriétés de base qui nous serviront à exprimer les propriétés plus complexes relatives à la visibilité des éléments graphiques. Le tableau 2 présente chacune de ces propriétés avec une définition dans notre formalisme et une illustration.

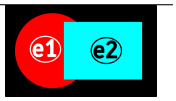
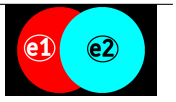
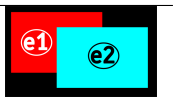
Propriété	Formalisation	Illustration
ordre d'affichage de e_1 inférieur à celui de e_2	$e_1 <_d e_2 \Leftrightarrow l(e_1) < l(e_2)$	
intersection de e_1 et e_2	$e_1 \bigcap_? e_2 \Leftrightarrow (\mathcal{D}(e_1) \cap \mathcal{D}(e_2)) \neq \emptyset$	
égalité des couleurs de e_1 et e_2	$e_1 =_c e_2 \Leftrightarrow c_i(e_1) = c_i(e_2), \forall i \in [r,g,b]$ $e_1 \neq_c e_2 \Leftrightarrow \exists i \in [r,g,b] \mid c_i(e_1) \neq c_i(e_2)$	
superposition de e_1 par e_2	$e_1 <_{sup} e_2 \Leftrightarrow e_1 \bigcap_? e_2 \wedge e_1 <_d e_2$	
masquage (tout ou partie) de e_1 par e_2	$e_1 <_{mask} e_2 \Leftrightarrow e_1 <_{sup} e_2 \wedge o(e_2) > 0$	

TABLE 2 – Formalisation des propriétés graphiques de base

4.2 Application au TCAS

Afin d'utiliser notre formalisme pour exprimer les exigences du TCAS, la notation ta_o représentera l'élément graphique associé à l'information TA *other aircraft*. De manière analogue, nous aurons : ta_p (TA *proximate aircraft*), ta_i (TA *intruder aircraft*), ta_t pour (TA *threat aircraft*) et own (position relative *own aircraft*). Nous utiliserons également vsi pour représenter l'élément graphique correspondant au compteur de vitesse verticale, ext pour le fond hors de la zone délimitée par le compteur de vitesse ainsi que red, $yellow$, $amber$, $cyan$, $white$ représentant respectivement les couleurs rouge, jaune, ambre, cyan et blanc.

Nous illustrons les propriétés d'intersection et d'égalité des couleurs avec les propriétés extraites de la norme du TCAS.

La propriété d'intersection $e_1 \bigcap_? e_2$ nous permet de définir les exigences suivantes du tableau 1 ($E_{i,j}$ désignant la caractéristique de la colonne j pour l'exigence E_i) :

— $E_{3,position} = \neg(ta_t \bigcap_? vsi) \wedge \neg(ta_t \bigcap_? ext)$
— $E_{4,position} = \neg(ta_i \bigcap_? vsi) \wedge \neg(ta_i \bigcap_? ext)$
— $E_{5,position} = \neg(ta_p \bigcap_? vsi) \wedge \neg(ta_p \bigcap_? ext)$
— $E_{6,position} = \neg(ta_o \bigcap_? vsi) \wedge \neg(ta_o \bigcap_? ext)$

L'exigence $E_{3,position}$ est à comprendre comme suit : l'intersection entre ta_t (élément graphique d'un trafic de type *threat aircraft*) et vsi (élément graphique du compteur de vitesse verticale) doit être vide et de même pour l'intersection entre ta_t et la zone hors du compteur de vitesse.

Avec la propriété d'égalité des couleurs $e_1 =_c e_2$ nous définissons ces exigences :

— $E_{1,couleur} = (own =_c white \vee own =_c cyan) \wedge ta_p \neq_c own \wedge ta_o \neq_c own$,
— $E_{3,couleur} = ta_t =_c red$
— $E_{4,couleur} = ta_i =_c yellow \vee ta_i =_c amber$
— $E_{5,couleur} = (ta_p =_c white \vee ta_p =_c cyan) \wedge ta_p \neq_c own$
— $E_{6,couleur} = (ta_o =_c white \vee ta_o =_c cyan) \wedge ta_o \neq_c own$

L'exigence $E_{5,couleur}$ est à comprendre comme suit : la couleur de ta_p (élément graphique d'un trafic de type *proximate aircraft*) doit être blanc ou cyan et ne doit pas être la même que celle de own (élément graphique de la propre position relative).

4.3 Validation dans Smala

Le langage Smala [21], associé à son environnement d'exécution Djnn est notre point d'entrée afin d'étudier concrètement l'expression et la validation formelle des propriétés relatives à la scène graphique des IHM. Des précédents travaux de l'équipe (Chatty *et al.* [10]) ont montré comment exploiter le graphe d'activation que l'on peut produire automatiquement depuis le code des applications Smala. Le graphe d'activation est une structure contenant l'ensemble des composants (graphiques ou non) du programme et qui permet de visualiser la propagation des évènements et l'ordre d'exécution de chaque composant (tous les éléments du graphe d'activation sont ordonnés ce qui détermine l'ordre d'exécution, selon un parcours en profondeur de gauche à droite). En l'état, ce graphe d'activation déduit du code Smala nous permet de valider par exemple la propriété d'accessibilité suivante pour le TCAS : l'événement *"type de trafic == threat_aircraft"* sera toujours suivi de l'événement *"affichage symbolique threat_aircraft"*.

Pour l'étude de la scène graphique des applications Smala, nous réalisons une simplification de ce graphe d'activation afin de n'en garder que les composants graphiques et donc obtenir le graphe de scène de l'application. La sémantique du langage et l'ordre de parcours du graphe d'activation étant conservés, nous pouvons raisonner sur ce graphe de scène comme sur le graphe de d'activation. Dans un premier temps, nous considérons un graphe de scène statique. Nous connaissons donc l'ordre d'affichage des composants graphiques et nous pouvons ainsi prendre en compte la première de nos variables visuelles pour les composants : **couche**. Les instructions graphiques de Smala sont calquées sur la norme SVG (Scalable Vector Graphics), un format de données ASCII conçu pour décrire des ensembles de graphiques vectoriels [3]. Pour chaque objet défini dans la norme, il existe un composant Smala qui le représente. Ainsi, en plus d'une API de dessin simple et consistante pour le programmeur, le chargement d'un fichier SVG permet de réifier chaque élément graphique sous la forme d'un composant Smala. Cela permet de concevoir une partie de l'interface graphique dans un logiciel dédié (Illustrator, Inkscape…) et par la suite de programmer en Smala des interactions avec (et entre) ces différents composants. Nous avons utilisé ce processus dans notre implémentation de l'IVSI, la partie graphique est définie dans un fichier SVG, les interactions sont décrites en Smala. Le graphe de scène contient donc toutes les propriétés propres à la norme SVG et notamment celles en rapport avec nos variables visuelles non encore décrites : **coordonnées**, **taille**, **valeur**, **grain**, **couleur**, **orientation**, **forme** et **transparence**.

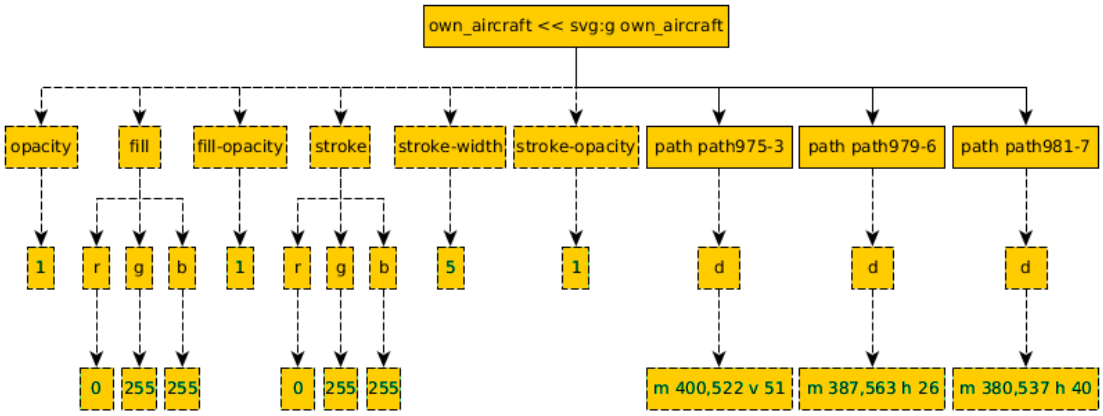


FIGURE 2 – Graphe de scène du composant graphique *own aircraft* du TCAS

La figure 2 présente un exemple de graphe de scène que nous avons réalisé manuellement à partir du code Smala et du fichier SVG pour le composant *own aircraft* qui peut être vu sur la figure 1 (symbole c). Ce graphe nous permet de valider, dans un premier temps par lecture sans outillage, les exigences graphiques suivantes :

3. https://www.w3.org/TR/SVG11

— $E_{1,symbole}$: le composant $own_aircraft$ est composé de trois *path* (un vertical et deux horizontaux) correspondant aux trois traits représentant l'avion,
— $E_{1,couleur}$: à l'état initial, les valeurs *r*, *g* et *b* de *fill* (remplissage) et *stroke* (contour) sont respectivement à 0, 255 et 255 ce qui correspond à la couleur cyan.

En plus du respect de ces exigences, la sémantique opérationnelle du langage Smala nous permet d'avoir des informations supplémentaires sur l'ordre d'affichage des trois traits. Au cours du processus d'affichage de la scène graphique, le graphe est parcouru en profondeur, de gauche à droite. Ainsi, le composant *path975-3* est affiché avant *path979-6*, lui-même affiché avant *path981-7*. Dans notre formalisme, cela revient à écrire $path$975-3 $<_d$ $path$979-6 $<_d$ $path$981-7.

5 Conclusion et perspectives

Les propriétés liées à la scène graphique des IHM sont peu étudiées par les méthodes formelles. Nous pensons que formaliser ces propriétés permettrait d'une part de mieux les étudier (en particulier celles portant sur la visibilité des composants graphiques), et d'autre part de contribuer à l'automatisation du processus de validation des interfaces. À partir de cette problématique, nous avons commencé à formaliser certaines propriétés de base des composants graphiques telles que l'**intersection**, la **superposition**, le **masquage** (tout ou partie) et l'**égalité de couleur**. Nous avons utilisé ces propriétés afin de définir formellement certaines exigences graphiques de la visualisation d'un système critique, le TCAS, et utilisé le graphe de scène issu de l'implémentation en Smala afin d'étudier les liens entre nos propriétés et le graphe.

Pour l'instant, nous générons manuellement les graphes de scène des applications Smala complétés avec les informations nous permettant d'avoir une vue sur les différentes variables visuelles qui nous intéressent. Nous pouvons valider certaines propriétés graphiques à partir de ce graphe et des variables accessibles en faisant le lien avec notre formalisme. Nous travaillons actuellement sur un outil permettant de générer automatiquement ces graphes de scène à partir du code Smala et des fichiers SVG.

De plus, nous ne considérons actuellement que le graphe de scène de manière statique. Nous augmenterons les définitions afin de pouvoir prendre en compte la dynamicité de la scène graphique.

À partir des catégories de modèles formels que nous avons présentées dans Béger *et al.* [6], nous avons réfléchi aux modèles utilisables pour étudier ces propriétés. La logique de Hoare [17] permet de vérifier la véracité d'une assertion donnée après exécution (de tout ou partie) d'un programme, en fonction de pré-conditions. Le plugin WP de Frama-C [12] utilise notamment la logique de Hoare pour vérifier des propriétés par annotation de code. C'est en nous basant sur cette logique que nous prévoyons de vérifier automatiquement les propriétés graphiques par le biais d'annotation du code Smala par analyse statique.

Enfin, afin de couvrir davantage d'exigences graphiques, nous réfléchissons à étendre notre formalisme à de nouvelles propriétés telles que la distance entre deux composants graphiques, la position relative entre deux composants graphiques et la confusion des couleurs résultant d'une proximité sur les composantes.

6 Remerciements

Ce travail est en partie financée par le projet ANR FORMEDICIS, ANR-16-CE25-0007.

Références

[1] Ed 143 - minimum operational performance standards for traffic alert and collision avoidance system ii (tcas ii), April 2013.

[2] B. BERARD, M. BIDOIT, A. FINKEL, F. LAROUSSINIE, A. PETIT, L. PETRUCCI et P. SCHNOEBELEN : *Systems and Software Verification : Model-Checking Techniques and*

Tools. Springer Publishing Company, Incorporated, 1st édn, 2010. ISBN 3642074782, 9783642074783.

[3] J. BERTIN et M. BARBUT : *Sémiologie graphique : les diagrammes, les réseaux, les cartes*. Gauthier Villars, 1973.

[4] S. BOLDO, C. LELAY et G. MELQUIOND : Formalization of Real Analysis : A Survey of Proof Assistants and Libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, oct. 2016.

[5] J. BOUCHET, L. MADANI, L. NIGAY, C. ORIAT et I. PARISSIS : Formal testing of multimodal interactive systems. *In* J. GULLIKSEN, M. B. HARNING, P. PALANQUE, G. C. van der VEER et J. WESSON, éds : *Engineering Interactive Systems*, p. 36–52, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-92698-6.

[6] P. BÉGER, S. LERICHE et D. PRUN : Vers la certification de programmes interactifs Djnn. *In Afadl 2018, 17èmes journées Approches Formelles dans l'Assistance au Developpement de Logiciels* , Grenoble, France, juin 2018.

[7] A. CERONE : Closure and attention activation in human automatic behaviour : A framework for the formal analysis of interactive systems. 45, 01 2011.

[8] A. CERONE : Towards a cognitive architecture for the formal analysis of human behaviour and learning. *In Software Technologies : Applications and Foundations*, p. 216–232, Cham, 2018. Springer International Publishing. ISBN 978-3-030-04771-9.

[9] A. CERONE et N. ELBEGBAYAN : Model-checking driven design of interactive systems. *Electronic Notes in Theoretical Computer Science*, 183:3 – 20, 2007. ISSN 1571-0661. Proceedings of the First International Workshop on Formal Methods for Interactive Systems.

[10] S. CHATTY, M. MAGNAUDET et D. PRUN : Verification of properties of interactive components from their executable code. *In Proceedings of the 7th ACM SIGCHI*, EICS '15, p. 276–285, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3646-8.

[11] J. COUTAZ, L. NIGAY, D. SALBER, A. BLANDFORD, J. MAY et R. M. YOUNG : *Four Easy Pieces for Assessing the Usability of Multimodal Interaction : The Care Properties*, p. 115–120. Springer US, Boston, MA, 1995. ISBN 978-1-5041-2896-4.

[12] P. CUOQ, F. KIRCHNER, N. KOSMATOV, V. PREVOSTO, J. SIGNOLES et B. YAKOBOWSKI : Frama-c. *In Software Engineering and Formal Methods*, p. 233–247, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] B. D'AUSBOURG : Using model checking for the automatic validation of user interfaces systems. *In* P. MARKOPOULOS et P. JOHNSON, éds : *Design, Specification and Verification of Interactive Systems '98*, p. 242–260, Vienna, 1998. Springer Vienna. ISBN 978-3-7091-3693-5.

[14] L. M. de MOURA, S. OWRE et N. SHANKAR : The sal language manual. 2003.

[15] A. GOSAIN et G. SHARMA : Static analysis : A survey of techniques and tools. *In Intelligent Computing and Applications*, p. 581–591, New Delhi, 2015. Springer India. ISBN 978-81-322-2268-2.

[16] L. HJELMSLEV : *Prolegomena to a theory of language*. Num. 7 de Prolegomena to a Theory of Language. University of Wisconsin Press, 1961.

[17] C. A. R. HOARE : An axiomatic basis for computer programming. *Commun. ACM*, 12 (10):576–580, oct. 1969. ISSN 0001-0782.

[18] C. W. JOHNSON : Using assurance cases and boolean logic driven markov processes to formalise cyber security concerns for safety-critical interaction with global navigation satellite systems. 45, 01 2011.

[19] N. KAMEL et Y. AIT-AMEUR : Modèle formel général pour le traitement d'interactions multimodales. *In IHM 04*, p. 219–222, Namure, Belgique, 03 2004.

[20] S. LERICHE, S. CONVERSY, C. PICARD, D. PRUN et M. MAGNAUDET : Towards Handling Latency in Interactive Software. *In FMIS 2018, 7th International Workshop on Formal Methods for Interactive Systems*, vol. 11176, p. pp 233–239 /ISBN : 978–3–030–04770–2, Toulouse, France, juin 2018. Springer.

[21] M. MAGNAUDET, S. CHATTY, S. CONVERSY, S. LERICHE, C. PICARD et D. PRUN :

Djnn/Smala : A Conceptual Framework and a Language for Interaction-Oriented Programming. *Proceedings of the ACM on Human-Computer Interaction* , 2(EICS):1 – 27, juin 2018. URL https://hal-enac.archives-ouvertes.fr/hal-01815222.

[22] P. MASCI, R. RUKŠĖNAS, P. OLADIMEJI, A. CAUCHI, A. GIMBLETT, Y. LI, P. CURZON et H. THIMBLEBY : On formalising interactive number entry on infusion pumps. 45, 01 2011.

[23] S. OWICKI et L. LAMPORT : Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, juil. 1982. ISSN 0164-0925.

[24] R. RUKŠĖNAS, P. CURZON et A. BLANDFORD : Detecting cognitive causes of confidentiality leaks. *Electronic Notes in Theoretical Computer Science*, 183:21 – 38, 2007. ISSN 1571-0661. Proceedings of the First International Workshop on Formal Methods for Interactive Systems.

[25] L. SU, H. BOWMAN et P. BARNARD : Performance of reactive interfaces in stimulus rich environments, applying formal methods and cognitive frameworks. *Electronic Notes in Theoretical Computer Science*, 208:95 – 111, 2008. ISSN 1571-0661. Proceedings of the 2nd International Workshop on Formal Methods for Interactive Systems.

[26] E. R. TUFTE : *The Visual Display of Quantitative Information*. Graphics Press, Cheshire, CT, USA, 1986. ISBN 0-9613921-0-X.

[27] C. WARE : *Information Visualization : Perception for Design*. Morgan Kaufmann Series in Interactive Technologies. Morgan Kaufmann, Amsterdam, 3 édn, 2012. ISBN 978-0-12-381464-7.

[28] L. WILKINSON : *The Grammar of Graphics (Statistics and Computing)*. Springer-Verlag, Berlin, Heidelberg, 2005. ISBN 0387245448.

[29] E. WILLIAMS : Airborne collision avoidance system. *In Proceedings of the 9th Australian Workshop on Safety Critical Systems and Software - Volume 47*, SCS '04, p. 97–110, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc. ISBN 1-920-68229-5.

Formalisation, vérification et évaluation de stratégies d'élasticité dans le Cloud

Khaled Khebbeb[1], Nabil Hameurlain[1] et Faiza Belala[2]

[1]LIUPPA - Université de Pau et des Pays de l'Adour
[2]LIRE - Université Constantine 2

[1]{*khaled.khebbeb, nabil.hameurlain*}*@univ-pau.fr*
[2]*faiza.belala@univ-constantine2.dz*

Cet article est un résumé long de "*Formal Modelling and Verifying Elasticity Strategies in Cloud Systems*" publié dans *IET Software. 13(1), pp. 25-35* (Février 2019) [1].

Mots-clés : cloud computing, élasticité, stratégies, formalisation, systèmes réactifs bigraphiques, logique de réécriture, évaluation, théorie des files d'attente.

Contexte et problématique. L'élasticité est une propriété qui permet aux systèmes Cloud de s'adapter à leur charge de travail en entrée, en provisionnant et en libérant des ressources informatiques lorsque la demande augmente et diminue [2], dans le but de maintenir une qualité de service optimale tout en minimisant les coûts de fonctionnement. Un tel comportement, dit "élastique", est généralement assuré par un contrôleur d'élasticité : une entité autonomique qui régit l'élasticité d'un système Cloud contrôlé, lui conférant des capacités d'auto-adaptation en termes de gestion dynamique des ressources (cf. Figure 1).

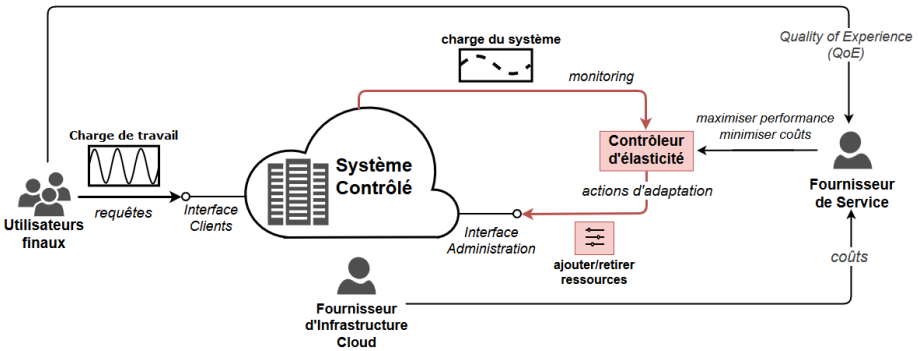


FIGURE 1 – Comportement élastique autonomique d'un système Cloud

Le comportement élastique d'un système Cloud dépend de plusieurs facteurs qui se chevauchent tels que la quantité de ressources disponibles, la charge de travail en entrée, la logique gouvernant le comportement du contrôleur d'élasticité ou

encore les différentes politiques de haut niveau (coût, performances, etc.) et propriétés (sûreté, vivacité, etc.) à satisfaire [3]. La complexité de ces dépendances, ainsi que la maîtrise des potentiels effets de bords néfastes sur l'état global du système (instabilité dans l'allocation des ressources, dégradation de la qualité de service et de la fiabilité du système, etc.), font de la conception des systèmes Cloud élastiques une tâche particulièrement difficile à assurer.

Objectif et contributions. L'objectif est de développer une méthodologie générique, exhaustive et non-ambigüe visant à réduire la complexité de modélisation et d'analyse des systèmes Cloud et de leurs comportements élastiques. Nous proposons une solution à fondements formels pour la spécification des systèmes Cloud, la vérification du bon fonctionnement de leurs comportements élastiques ainsi que l'évaluation de leurs performances. Notre approche de modélisation repose sur l'association complémentaire, à base de logique de réécriture, du formalisme des Systèmes Réactifs Bigraphiques (BRS) [4] et du langage de spécification formelle Maude [5]. D'un côté, les BRS permettent de décrire des aspects structurels et comportementaux pour les systèmes Cloud élastiques. D'un autre côté, le langage Maude permet d'encoder les spécifications basées sur les BRS de manière à préserver leur sémantique fonctionnelle. Maude offre un support d'exécutabilité pour les comportements définis en plus de permettre de vérifier leur bon fonctionnement.

Précisément, nous adoptons les bigraphes comme cadre formel pour la spécification de la structure des systèmes Cloud élastiques. Cette modélisation fournit une sémantique robuste et détaillée pour la représentation des architectures Cloud en (1) identifiant, à travers une discipline de typage (*sorting*), toutes les entités les composant (serveurs physiques, machines virtuelles, instances de service, requêtes) et en (2) restreignant les possibilités de modélisation, à travers des règles de construction bigraphiques, afin d'assurer l'expression de configurations correctes. À partir de cette spécification, nous proposons une sémantique basée sur les BRS pour modéliser des actions de reconfiguration dynamique des systèmes Cloud. Nous définissons un ensemble de règles de réaction bigraphiques décrivant les restructurations possibles. Ces règles assurent une cohérence structurelle correcte par définition, conformément à la discipline de typage et aux règles de construction définies. Notons que les bigraphes se distinguent particulièrement d'autres formalismes (notamment le π-calcul ou les réseaux de Petri) par la capacité de représenter des systèmes graphiquement et algébriquement sous forme de forêts de noeuds typés (via la notion de sorting). Cela permet de définir une sémantique structurelle intuitivement compréhensible et de raisonner sur sa reconfiguration (via les règles de réaction), de manière à en décrire un comportement donné. Dans nos travaux, le typage des noeuds permet de distinguer (selon le type, la localisation, etc.) les différentes ressources d'un système Cloud, et les règles de réaction permettent d'en décrire des comportements élastiques à des granularités différentes.

Afin de fournir une logique gouvernant le comportement du contrôleur d'élasticité, nous définissons un ensemble de stratégies d'élasticité horizontale (i.e., pour

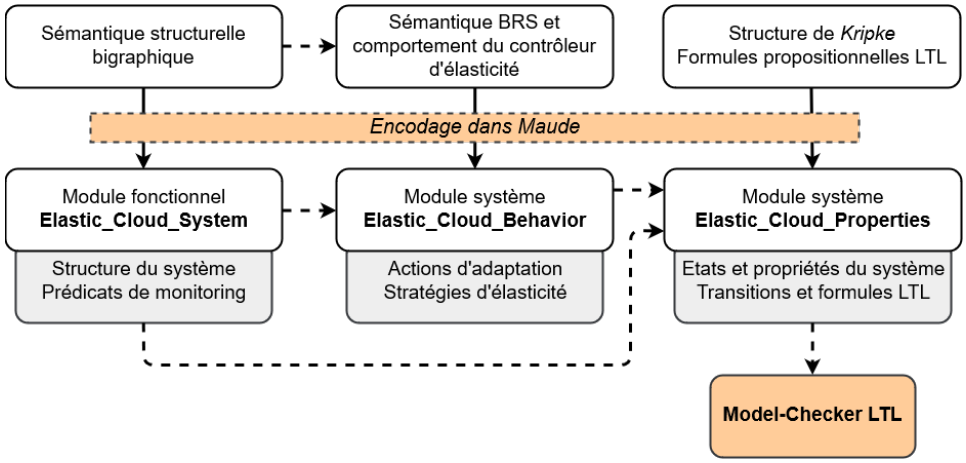


FIGURE 2 – Vue d'ensemble de la solution de spécification, d'exécution et de vérification des systèmes Cloud élastiques

ajouter /retirer des ressources) en multi-couches, càd. aux niveaux infrastructure (serveurs physiques, machines vituelles) et application (instances de service) d'un système Cloud. Ces stratégies permettent de gérer l'allocation des ressources du système Cloud contrôlé, tout en préservant sa cohérence architecturale. Elles sont de nature réactive, càd. de la forme `Si` *`condition`* `Alors` *`action`*. Les actions sont les règles de réaction bigraphiques définies et les conditions représentent des prédicats de surveillance (*monitoring*) du système. Ces conditions sont exprimées dans la logique du premier ordre et permettent d'identifier un système, en termes de consommation de ressources (e.g., $\forall$ les instances de service sont surchargées, $\exists$ une machine virtuelle non-utilisée, etc.). Elles servent à guider les décisions d'actions (ajout/retrait de ressources) du contrôleur qui gère son élasticité.

La Figure 2 donne une vision d'ensemble de la solution que nous proposons pour la spécification, l'exécution et la vérification de l'élasticité des systèmes Cloud. Le langage Maude permet d'encoder les spécifications des systèmes Cloud jusqu'à lors décrites (structure, comportement, stratégies, prédicats, états) sous une approche modulaire. Précisément, la spécification structurelle bigraphique, la sémantique comportementale à base de BRS et la définition des propriétés du système via une structure de *Kirpke* sont respectivement encodées dans le langage Maude sous forme de modules. Cela résulte en une spécification construite de manière incrémentale. Notons qu'une structure de *Kripke* est un modèle de calcul qui permet d'identifier les états et les comportements désirés [6], notamment en termes de gestion autonomique de l'élasticité d'un système Cloud dans nos travaux. Les modules construits permettent une exécution de manière générique et autonomique, des comportements définis, sur un support de système de réécriture. Les tâches manuelles d'encodage sont représentées par des flèches continues et la relation de dépendance entre spécifications et modules définis, par des flèches discontinues.

En outre, nous procédons à la vérification formelle du bon fonctionnement

du comportement du contrôleur d'élasticité, grâce aux outils fournis par Maude. Cette vérification repose sur une technique de model-checking à base d'états, supportée par la logique temporelle linéaire (LTL), et décrite par la structure de *Kripke* définie.

Enfin, nous proposons une étude de cas expérimentale des stratégies d'élasticité introduites, afin de les évaluer et les valider en termes de coûts et de performance. Cela consiste à étudier la quantité de ressources déployées et le délai d'attente de service observés respectivement. Nous proposons une approche à base de la théorie des files d'attente [7] afin de simuler le fonctionnement d'un système Cloud étudié, opérant selon les comportements élastiques introduits. À ces fins, nous avons conçu un outil pour la simulation et le monitoring des comportements élastiques d'un système Cloud. Cet outil permet de simuler un système Cloud quelconque en le confrontant à une infinité de scénarios possibles. Les scénarios proposés se basent sur l'intensité de la charge de travail (taux λ d'arrivée des requêtes) et l'intensité de service (taux μ de requêtes traitées). Ils permettent d'observer la gestion de l'allocation dynamique des ressources d'un système, à partir d'une configuration initiale quelconque, selon des scénarios décrivant des situations de demande oscillante (hausse et baisse de la charge de travail). Les résultats obtenus sont validés par comparaison à ceux donnés par la formule *Erlang-C* qui calcule, pour un couple (λ, μ), le nombre minimal de ressources requises pour assurer un niveau de service donné.

Références

[1] K. Khebbeb, N. Hameurlain, F. Belala, and H. Sahli, "Formal modelling and verifying elasticity strategies in cloud systems," *The Institution of Engineering and Technology - IET Software*, vol. 13, no. 1, pp. 25–35(10), 2019.

[2] N. R. Herbst, S. Kounev, and R. H. Reussner, "Elasticity in Cloud Computing : What It Is, and What It Is Not.," in *ICAC*, vol. 13, pp. 23–27, 2013.

[3] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić, "Towards the formalization of properties of cloud-based elastic systems," in *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, pp. 38–47, ACM, 2014.

[4] R. Milner, "Bigraphical reactive systems," in *International Conference on Concurrency Theory*, pp. 16–35, Springer, 2001.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about maude-a high-performance logical framework : how to specify, program and verify systems in rewriting logic*. Springer-Verlag, 2007.

[6] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[7] B. Baynat, "Théorie des files d'attente," *Hermès, Paris*, 2000.

Unification de la Vérification et de l'Exécution Embarquée de Modèles

Valentin Besnard

ERIS, ESEO-TECH, Angers, France, *valentin.besnard@eseo.fr*

Résumé

Pour lutter contre la complexité croissante des systèmes embarqués, les modèles de ces systèmes peuvent être vérifiés dès la phase de conception en utilisant diverses techniques de vérification et de validation (V&V). L'application de ces techniques requiert généralement deux transformations à partir du modèle de conception : une première pour obtenir le code exécutable du système et une seconde vers un modèle formel sur lequel pourront être appliqués les outils de V&V. Cependant, ces transformations créent des fossés sémantiques et nécessitent d'établir une relation d'équivalence entre le code exécutable et le modèle formel obtenu. Pour aborder ces problèmes, cet article présente une approche permettant d'exécuter et de vérifier des modèles avec une seule implémentation de la sémantique du langage de modélisation. L'exécution et la vérification de ces modèles est assurée par un interpréteur de modèles pilotable par des outils de V&V. Cette approche a été appliquée au langage UML avec l'interpréteur de modèles embarqué EMI et le model-checker OBP2.

Mots clés— Interprétation de modèles, Model-checking, Systèmes embarqués, UML

1 Introduction

Les systèmes embarqués deviennent de plus en plus complexes, ce qui rend leur conception plus difficile et les exposent davantage aux défaillances logicielles. Dans le contexte de l'ingénierie dirigée par les modèles, ces systèmes peuvent être représentés sous forme de modèles et analysés dès la phase de conception en appliquant des techniques de vérification et de validation (V&V).

Cependant, trois inconvénients majeurs subsistent généralement. D'abord, la transformation du modèle de conception en code exécutable crée un premier fossé sémantique entre ce modèle et le code généré. Ensuite, l'application des outils de V&V requiert généralement une seconde transformation vers un langage formel. Cette transformation crée un second fossé sémantique qui complexifie la compréhension des résultats de vérification. Enfin, une relation d'équivalence entre le modèle formel et le code exécutable doit être établie pour prouver que ce qui est exécuté est bien ce qui a été vérifié. Ces problèmes sont principalement dus aux transformations qui capturent la sémantique du langage de modélisation vers différents formalismes.

Pour lutter contre ces problèmes, cet article présente une approche permettant d'utiliser une seule implémentation de la sémantique du langage de modélisation pour l'exécution et la vérification de modèles. Cette sémantique est encodée dans un interpréteur de modèles pouvant être piloté par des outils de V&V. Cette approche permet ainsi d'appliquer la vérification directement sur le modèle interprété tout en réutilisant la même sémantique opérationnelle que celle utilisée pour l'exécution sur le système réel.

Notre approche a été mise en oeuvre avec le langage UML [18]. Un interpréteur de modèles embarqué, appelé EMI (Embedded Model Interpreter) [4, 5, 6], permet de simuler et vérifier des modèles UML avec le model-checker OBP2 (Observer-Based Prover 2) [20, 21] (`https://plug-obp.github.io/`), puis de les exécuter sur une cible embarquée STM32 discovery.

2 Approche

Pour mieux comprendre les problématiques et les enjeux abordés dans ce projet, notre approche est comparée avec l'approche classique de vérification et d'exécution embarquée de modèles. Cette comparaison permet de mettre en relief les spécificités et les atouts de l'approche proposée dans le contexte de l'exécution de modèles.

L'approche classique est illustrée sur la Figure 1. Le système à l'étude est modélisé sous la forme d'un *Modèle de Conception* dans le langage de modélisation choisi par l'équipe de développement. Le modèle doit donc se conformer au *Métamodèle* de ce langage. À partir du *Modèle de Conception*, l'approche classique utilise généralement deux transformations pour pouvoir exécuter et vérifier ce modèle. La première permet de transformer le *Modèle de Conception* en *Code* exécutable via de la génération de code automatique, semi-automatique, ou manuelle. Le *Code* produit peut ensuite être exécuté sur une *Cible Embarquée* et interagir avec l'environnement du système via les *Entrées/Sorties* de celle-ci. La seconde utilise des techniques de transformation de modèles pour produire un *Modèle d'Analyse* (ou modèle formel si exprimé dans un langage formel) à partir du *Modèle de Conception*. Ce *Modèle d'Analyse* peut ensuite être exploité par des *Outils de V&V Haut Niveau* afin de vérifier que le modèle satisfait aux exigences du système à un haut niveau d'abstraction.

Dans cette approche classique de développement embarqué, trois problèmes principaux subsistent. *(1)* La génération de code crée un premier fossé sémantique entre le modèle de conception et le code exécutable. Il devient ainsi plus difficile d'établir des liens entre les concepts du langage de modélisation et les fragments de code générés. Ce fossé sémantique a aussi un impact sur les outils de V&V puisque le modèle exprimé sous forme de code exécutable ne peut plus être analysé à un haut niveau d'abstraction et requiert l'utilisation d'*Outils de V&V Bas Niveau. (2)* Un second fossé sémantique est créé entre le modèle de conception et le modèle d'analyse. Il complexifie la compréhension des résultats de V&V en particulier pour des ingénieurs non-experts en méthodes formelles. Les activités de vérification peuvent aussi nécessiter de charger ce modèle d'analyse dans différents outils. Ce besoin induit un risque d'erreur supplémentaire si les différents processus de chargement ne sont pas exactement équivalents. *(3)* Une relation d'équivalence entre le modèle d'analyse et le code exécutable doit également est établie, prouvée, et maintenue afin d'assurer que les propriétés vérifiées en phase de V&V le soient aussi lors de l'exécution.

Ces trois problèmes sont principalement dus à l'utilisation de multiple implémentations de la sémantique du langage de modélisation. Ces multiples définitions proviennent

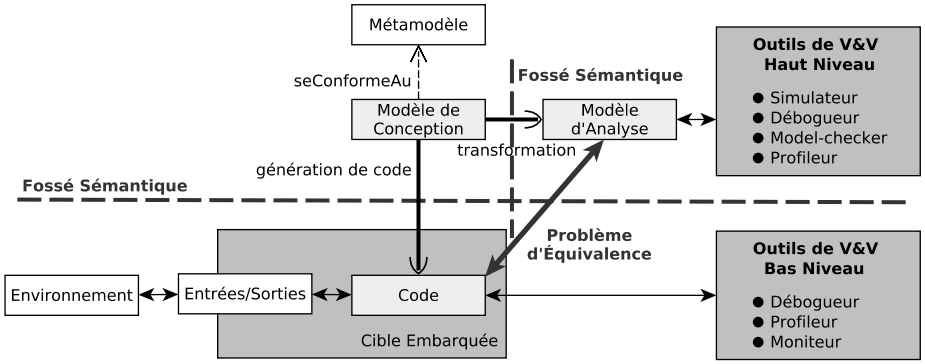


FIGURE 1 – Approche classique

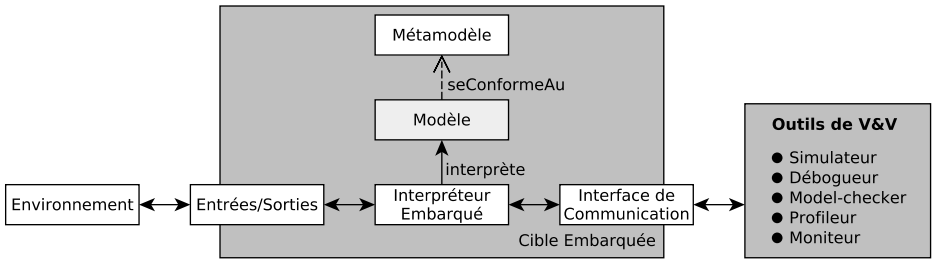


FIGURE 2 – Approche proposée

de l'utilisation de transformations qui capturent la sémantique du langage de modélisation en termes de la sémantique cible.

Pour traiter ces problèmes, la Figure 2 présente une nouvelle approche basée sur l'utilisation d'un interpréteur de modèles pilotable à distance par des outils de V&V. Dans cette approche, le *Modèle de Conception* et son *Métamodèle* sont directement chargés sur la *Cible Embarquée* afin d'éviter les deux transformations mentionnées précédemment. Le modèle est exécuté avec un *Interpréteur Embarqué* qui peut interagir avec l'environnement via les *Entrées/Sorties* de la cible embarquée comme dans l'approche classique. La spécificité de cet *Interpréteur Embarqué* est qu'une seule implémentation de la sémantique du langage est utilisée à la fois pour l'exécution et la vérification des modèles. Cette unique implémentation correspond à la sémantique opérationnelle de l'*Interpréteur Embarqué*. Pour la vérification, les *Outils de V&V* peuvent se connecter à l'*Interpréteur Embarqué* via une *Interface de Communication*. Les activités de V&V sont ainsi directement appliquées sur le modèle exécutable chargé par l'interpréteur tout en réutilisant la même implémentation de la sémantique du langage que celle utilisée pour l'exécution du système. Cet interpréteur de modèles doit pouvoir être déployé sur une cible embarquée mais doit aussi pouvoir être exécuté sur un ordinateur pour la phase de V&V afin d'obtenir des performances de vérification acceptables.

L'utilisation d'une seule implémentation de la sémantique du langage possède plusieurs avantages. Elle permet d'éviter les deux fossés sémantiques identifiés sur l'approche classique. Les résultats de vérification sont ainsi directement exprimés en termes du langage de modélisation ce qui permet de faciliter leur compréhension. Cette approche apporte

aussi une solution au problème d'équivalence et permet d'assurer que ce qui est exécuté est bien ce qui a été vérifié. Même en cas de bogue dans la sémantique opérationnelle de l'interpréteur, si le comportement du modèle satisfait aux exigences système alors celles-ci seront aussi satisfaites lors de l'exécution sur le système embarqué réel. En effet, le déploiement sur la cible embarquée utilise le même modèle et le même interpréteur que ceux utilisés pendant la phase de vérification. Cette approche tend donc à améliorer la qualité de développement et de vérification des systèmes embarqués dans le contexte de l'ingénierie dirigée par les modèles.

3 Application au Langage UML

L'approche proposée a été appliquée au langage UML et un interpréteur de modèles implémentant la sémantique de ce langage a été développé. Cet interpréteur de modèles, appelé EMI, est dédié à l'exécution et à la vérification de systèmes embarqués spécifiés sous la forme de modèles UML.

Bien qu'UML soit un langage semi-formel, l'approche reste valide car c'est la sémantique opérationnelle de l'interpréteur qui sert de référence. Par exemple, pour les points de variation sémantique d'UML, les choix d'implémentation faits dans l'interpréteur permettent de déterminer le comportement du système. Ces choix ont certes un impact sur la sémantique du langage mais cette même définition de la sémantique sera ensuite employée pour la vérification et l'exécution du système, préservant ainsi la relation d'équivalence entre le modèle d'analyse et le code exécutable.

Avant de pouvoir vérifier un modèle UML, plusieurs étapes sont nécessaires afin de pouvoir exécuter ce modèle sur l'interpréteur EMI. La première étape consiste à modéliser le système à concevoir en UML. Le modèle doit se conformer au sous-ensemble d'UML supporté par EMI. Ce sous-ensemble peut se représenter avec les diagrammes de classes, de structure composite, et d'états-transitions. Il permet de représenter à la fois la partie structurelle et la partie comportementale du modèle tout en exploitant partiellement les aspects orientés objet d'UML (e.g., héritage simple). Le modèle utilise aussi un langage d'action pour décrire les gardes et les effets des machines à états. Ce langage d'action est le langage C enrichi avec des macros C pour accéder aux éléments du modèle (e.g., les objets du modèle et leurs attributs). Une fois le modèle UML du système élaboré, il est sérialisé en langage C, le langage natif de l'interpréteur EMI. La sérialisation permet d'associer à chaque élément du modèle un initialiseur de structure C. Afin d'optimiser au mieux l'espace mémoire et les performances d'exécution, seuls les éléments du sous-ensemble considéré et nécessaires pour l'exécution du modèle sont pris en compte lors de la sérialisation. Contrairement à la génération de code, cette opération ne génère que des données mais pas de fonctions (sauf pour les expressions du langage d'action). Elle ne capture donc pas la sémantique du langage UML. Le modèle sérialisé en C et le code de l'interpréteur sont ensuite compilés et linkés avec un compilateur C pour produire le binaire exécutable de EMI. Cette opération peut être vue comme un chargement du modèle à la compilation. L'exécution de ce binaire exécutable permet d'interpréter le modèle UML chargé dans EMI. Cet interpréteur de modèles peut être exécuté sur un ordinateur équipé avec un OS Linux, ou en bare-metal (i.e., sans OS) sur une carte embarquée STM32 discovery.

Cet interpréteur de modèles UML peut ensuite être utilisé pour vérifier et analyser le comportement du modèle UML. Différentes activités de V&V peuvent être mises en oeuvre en connectant le model-checker OBP2 à l'interface de communication de EMI. Cette interface de communication permet de récupérer la configuration courante de l'interpréteur (i.e., la partie dynamique du modèle), de remettre l'interpréteur dans une confi-

guration donnée, de calculer l'ensemble des transitions tirables des machines à états du système, de tirer des transitions, et d'évaluer des prédicats. Cette interface permet aux outils de V&V de piloter l'interpréteur de modèles pour la vérification.

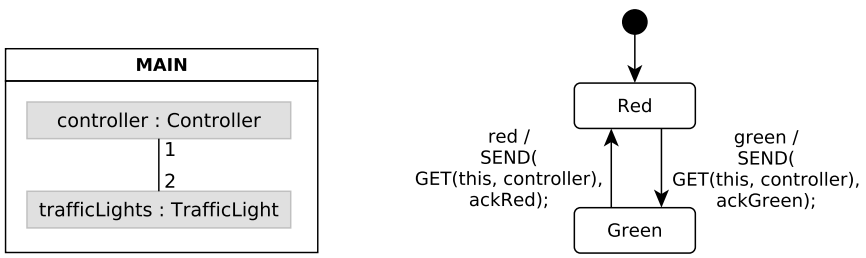
Le model-checker OBP2 permet d'appliquer plusieurs activités de V&V sur les modèles UML interprétés par EMI :
Simulation : OBP2 offre une interface de simulation permettant de visualiser et d'explorer différentes traces d'exécution. L'interface utilisateur de ce simulateur permet notamment de visualiser la configuration courante de l'interpréteur, de visualiser les transitions tirables, de tirer des transitions, et de visualiser les traces d'exécution déjà explorées.
Model-checking de propriétés LTL : Le model-checker OBP2 permet aussi de vérifier des propriétés LTL. Pour y parvenir, OBP2 explore l'espace d'états du système et utilise des techniques classiques de model-checking basées sur la composition d'automates de Büchi. Si une propriété est violée, le model-checker retourne un contre-exemple qui peut être visualisé sous la forme d'une trace dans l'interface de simulation.
Model-checking et monitoring avec des automates observateurs : Notre approche permet également d'exprimer des propriétés de sûreté en UML sous la forme d'automates observateurs. Ces automates sont composés de façon synchrone avec le système par l'interpréteur EMI et le model-checker n'a plus qu'à vérifier si les états de rejets de ces automates sont atteignables. Ces mêmes automates observateurs peuvent également être déployés avec EMI sur une cible embarquée pour faire du monitoring. Surveiller le comportement du système dans son environnement réel permet notamment de détecter des composants matériels défectueux, de réagir en cas de défaillances, et de faciliter l'analyse post-mortem.

La mise en oeuvre de ces activités de V&V avec EMI a ainsi permis de démontrer l'applicabilité de cette approche pour l'exécution et la vérification de modèles UML.

4 Exemple

Pour illustrer notre approche, cette section présente sa mise en oeuvre sur un exemple très simple. Il s'agit d'un modèle de gestion de deux feux de signalisation bicolores (vert et rouge) qui a pour objectif de garantir la sécurité des usagers et la fluidité de la circulation. Ce système a été modélisé en UML comme le montre le diagramme de composite structure et la machine à états de la classe *TrafficLight* en Figure 3. Ce modèle a ainsi pu être chargé dans EMI afin de vérifier les deux propriétés suivantes :

1. Le premier feu est au vert infiniment souvent.
2. Les deux feux ne sont jamais verts tous les deux simultanément.



(a) Diagramme de composite structure (b) Machine à états de *TrafficLight*

FIGURE 3 – Diagrammes d'un modèle de gestion de feux de signalisation.

Ces propriétés ont été traduites en LTL afin d'être vérifiées avec le model-checker OBP2 :

1. `"[] <> firstIsGreen"`
2. `"[] !(firstIsGreen and secondIsGreen)"`

Les proposition atomiques `firstIsGreen` et `secondIsGreen`, exprimées à l'aide du langage d'action C, permettent de savoir si respectivement le premier feu et le second feu sont verts à un instant donné. Ces deux propriétés ont été vérifiées avec succès sur le modèle UML qui a été conçu. Notre outil a également été validé sur des modèles plus complexes comme un contrôleur de passage à niveau [6].

5 État de l'Art

Ce projet se focalise sur l'exécution et la vérification de modèles UML dans le contexte des systèmes embarqués. D'autres outils permettent également d'exécuter et de vérifier des modèles UML comme le présente l'étude en [10].

Rational Software Architect [14] et Rhapsody [11] sont des outils de modélisation permettant de modéliser des modèles UML dans un environnement graphique, et de générer du code pour simuler et déboguer ces modèles. Les interpréteurs Moka [19] et Moliz [16] permettent d'interpréter des modèles conformes au standard fUML [17]. Ils peuvent être intégrés à l'outil de modélisation Papyrus [13] et être utilisés pour simuler, déboguer et tester des modèles fUML. GUML [8] et UniComp [9] sont des compilateurs de modèles permettant de compiler directement des modèles UML en code exécutable performant sans avoir besoin de passer par un formalisme intermédiaire. GEMOC Studio [7] est un framework permettant de concevoir un environnement de modélisation générique avec différents moteurs d'exécution et des outils de V&V (e.g., débogueur omniscient, animateur graphique). En comparaison avec notre interpréteur EMI, tous ces outils ne permettent pas d'appliquer des techniques formelles, comme le model-checking, sur des modèles de conception en utilisant leurs propres implémentations de la sémantique du langage.

D'autres outils permettent également d'analyser l'exécution des systèmes embarqués à un haut niveau d'abstraction comme les débogueurs [2] et [12]. Ces outils permettent de résoudre le problème du fossé sémantique entre le modèle de conception et le code exécutable mais ne permettent pas d'appliquer des méthodes formelles sur ces modèles en assurant que ce qui est vérifié est bien ce qui sera exécuté. Les compilateurs certifiés (e.g., CompCert [15]) permettent également de résoudre ce fossé sémantique en prouvant formellement que le code exécutable généré se comporte exactement comme le programme source.

Des approches alternatives permettent également de garantir la sûreté de fonctionnement du code exécutable. Event-B [1] est une méthode formelle permettant la modélisation et l'analyse de systèmes. Cette méthode utilise à la fois des raffinements successifs pour représenter le système à différents niveaux d'abstraction et des preuves mathématiques pour garantir la cohérence entre ces différents niveaux. SCADE [3] permet également de modéliser, vérifier, et exécuter des modèles de systèmes embarqués. Cette méthode applique des techniques de vérification formelle directement sur les modèles SCADE et un générateur de code certifié permet d'obtenir le code exécutable de l'application.

6 Conclusion

L'ingénierie dirigée par les modèles facilite le développement des systèmes embarqués complexes en permettant leur vérification et leur exécution sous forme de modèles. L'ap-

proche présentée dans cet article permet d'unifier la vérification et l'exécution embarquée de ces modèles en utilisant une seule définition de la sémantique du langage de modélisation. Cette définition de la sémantique est implémentée dans un interpréteur de modèles dédié à l'exécution de modèles pour l'embarqué. Le modèle peut également être vérifié en connectant des outils de V&V à cet interpréteur et en réutilisant la même sémantique opérationnelle que celle utilisée pour l'exécution.

Cette approche permet d'améliorer la qualité de développement des systèmes embarqués. Elle permet notamment d'assurer que les propriétés vérifiées pendant la phase de vérification le restent lors de l'exécution. Elle facilite également la compréhension des résultats de vérification et reste applicable dans le cas des langages semi-formels comme UML. L'utilisation de l'interpréteur de modèles EMI couplé au model-checker OBP2 a en effet permis de simuler, model-checker, monitorer, et exécuter des modèles UML. L'interpréteur de modèles EMI doit néanmoins encore être amélioré pour évaluer et mettre en oeuvre cette approche sur des systèmes embarqués industriels.

Remerciements Ce projet est partiellement financé par Davidson Consulting. L'auteur remercie particulièrement David Olivier, Matthias Brun, Ciprian Teodorov, Frédéric Jouault, et Philippe Dhaussy pour leurs conseils et commentaires avisés sur le projet.

Références

[1] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2013.

[2] Mojtaba Bagherzadeh, Nicolas Hili, and Juergen Dingel. Model-level, Platform-independent Debugging in the Context of the Model-driven Development of Real-time Systems. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 419–430, New York, USA, 2017. ACM.

[3] Gérard Berry. SCADE : Synchronous Design and Validation of Embedded Control Software. In S. Ramesh and Prahladavaradan Sampath, editors, *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, pages 19–33, Dordrecht, 2007. Springer Netherlands.

[4] Valentin Besnard, Matthias Brun, Philippe Dhaussy, Frédéric Jouault, David Olivier, and Ciprian Teodorov. Towards one Model Interpreter for Both Design and Deployment. In *3rd International Workshop on Executable Modeling (EXE)*, Austin, United States, September 2017.

[5] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Embedded UML Model Execution to Bridge the Gap Between Design and Runtime. In *MDE@DeRun 2018 : First International Workshop on Model-Driven Engineering for Design-Runtime Interaction in Complex Systems*, Toulouse, France, June 2018.

[6] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, Copenhagen, Denmark, October 2018.

[7] Erwan Bousse, Thomas Degueule, Didier Vojtisek, Tanja Mayerhofer, Julien Deantoni, and Benoit Combemale. Execution Framework of the GEMOC Studio (Tool

Demo). In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2016, pages 84–89, New York, NY, USA, 2016. ACM.

[8] Asma Charfi Smaoui, Chokri Mraidha, and Pierre Boulet. An Optimized Compilation of UML State Machines. In *ISORC - 15th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, Shenzhen, China, April 2012.

[9] Federico Ciccozzi. Unicomp : A Semantics-aware Model Compiler for Optimised Predictable Software. In *Proceedings of the 40th International Conference on Software Engineering : New Ideas and Emerging Results*, ICSE-NIER '18, pages 41–44, New York, NY, USA, 2018. ACM.

[10] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models : a systematic review of research and practice. *Software & Systems Modeling*, April 2018.

[11] Eran Gery, David Harel, and Eldad Palachi. Rhapsody : A Complete Life-Cycle Model-Based Development System. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, pages 1–10, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[12] Padma Iyenghar, Elke Pulvermueller, Clemens Westerkamp, Juergen Wuebbelmann, and Michael Uelschen. *Model-Based Debugging of Embedded Software Systems*, pages 107–132. Springer New York, New York, NY, 2017.

[13] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML : an open source toolset for MDA. In *Proceedings of the Fifth European Conference on Model-Driven Architecture Foundations and Applications*, pages 1–4, 2009.

[14] Daniel Leroux, Martin Nally, and Kenneth Hussey. Rational Software Architect : A tool for domain-specific modeling. *IBM systems journal*, 45(3) :555–568, 2006.

[15] Xavier Leroy. The CompCert C verified compiler : Documentation and user's manual. Intern report, Inria, June 2017.

[16] Tanja Mayerhofer and Philip Langer. Moliz : A Model Execution Framework for UML Models. In *Proceedings of the 2nd International Master Class on Model-Driven Engineering : Modeling Wizards*, MW '12, pages 3 :1–3 :2, New York, NY, USA, 2012. ACM.

[17] OMG. Semantics of a Foundational Subset for Executable UML Models, October 2017. `https://www.omg.org/spec/FUML/1.3/PDF`.

[18] OMG. Unified Modeling Language, December 2017. `https://www.omg.org/spec/UML/2.5.1/PDF`.

[19] Sebastien Revol, Géry Delog, Arnaud Cuccurru, and Jérémie Tatibouët. Papyrus : Moka overview, 2018. `https://wiki.eclipse.org/Papyrus/UserGuide/ModelExecution`.

[20] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, 19(2) :229–245, April 2017.

[21] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-Free[ze] reachability analysis : reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7) :516–542, 2016.

EMI : Un Interpréteur de Modèles Embarqué pour l'Exécution et la Vérification de Modèles UML

Valentin Besnard[1], Matthias Brun[1], Philippe Dhaussy[2], Frédéric Jouault[1], et Ciprian Teodorov[2]

[1]ERIS, ESEO-TECH, Angers, France
prenom.nom@eseo.fr
[2]Lab-STICC UMR CNRS 6285, ENSTA Bretagne, Brest, France
prenom.nom@ensta-bretagne.fr

Résumé

Pour faire face à la complexité croissante des systèmes embarqués, les activités de vérification et de validation, et notamment le model-checking, sont de plus en plus sollicités. Les model-checkers permettent de vérifier des propriétés par rapport au modèle formel fourni en entrée mais deux problèmes subsistent généralement. D'une part, les outils de model-checking n'apportent pas l'assurance que les propriétés sont aussi vérifiées sur le code exécutable du système. D'autre part, le modèle formel utilisé par les model-checkers est souvent le résultat d'une transformation de modèle dont l'exactitude n'est pas prouvée. Pour y remédier, cet article présente EMI, un interpréteur de modèles UML visant l'exécution et la vérification de systèmes embarqués à l'aide d'une seule implémentation de la sémantique du langage. En connectant cet outil au model-checker OBP2, diverses activités de vérification peuvent ainsi être menées sur des modèles semi-formels en UML.

Mots clés— UML, Interprétation de modèles, Model-checking, Systèmes embarqués

1 Introduction

Les systèmes embarqués sont de plus en plus complexes à concevoir et s'exposent à de multiples défaillances logicielles (e.g., erreurs de conception, bogues, failles de sécurité). Pour garantir la sûreté de fonctionnement de ces systèmes, leurs développements requièrent des besoins croissants en vérification et validation (V&V). Grâce à l'ingénierie dirigée par les modèles, ces systèmes peuvent être modélisés sous la forme de modèles et vérifiés à un plus haut niveau d'abstraction en utilisant des techniques comme le model-checking. Les model-checkers sont des outils de vérification formelle permettant de vérifier des propriétés par rapport au modèle formel qui leur est fourni. Cependant, l'utilisation classique d'un model-checker pose généralement deux problèmes susceptibles d'impacter la vérification du modèle. (1) Une relation d'équivalence entre le modèle formel et le code exécutable doit être établie et prouvé pour assurer que ce qui est exécuté est bien ce qui a été vérifié.

(2) Le modèle formel est souvent le résultat d'une transformation de modèle complexe qui n'apporte pas la preuve que le modèle obtenu est fidèle au modèle de conception.

Afin d'apporter une réponse à ces deux problématiques, nous présentons dans cette publication l'outil EMI (Embedded Model Interpreter) [1], un interpréteur de modèles UML dédié à l'exécution et à la vérification des systèmes embarqués. EMI interprète directement le modèle UML de conception et sa sémantique opérationnelle est la seule définition de la sémantique du langage qui est utilisée. Cet interpréteur de modèles peut en effet être piloté par des outils de vérification afin de vérifier directement le modèle exécutable du système tout en réutilisant la même implémentation de la sémantique du langage que celle utilisée lors de l'exécution. Notre approche permet d'éviter les transformations de modèle difficiles à prouver et permet d'améliorer la qualité des activités de V&V. Elle offre également l'avantage d'être applicable même si le langage utilisé n'est que semi-formel comme UML. Au stade de prototype, EMI peut être utilisé pour diverses activités de vérification en se connectant au model-checker OBP2 (Observer-Based Prover 2) [3, 4] (`https://plug-obp.github.io/`).

2 Exécution de Modèles UML

L'interpréteur de modèles EMI est l'outil central de cette approche permettant d'exécuter des systèmes embarqués spécifiés sous la forme de modèles UML. Le modèle UML du système est directement chargé dans EMI et exécuté avec une seule implémentation de la sémantique d'UML. Cette implémentation correspond à la sémantique opérationnelle de notre outil qui sera utilisée à la fois pour l'exécution du modèle sur un microcontrôleur embarqué et pour la phase de vérification via des outils de V&V. De sa conception jusqu'à son exécution par EMI, différentes étapes sont nécessaires pour pouvoir interpréter un modèle.

Modélisation. La première étape consiste à modéliser en UML le système embarqué à concevoir. Notre outil supporte un sous-ensemble d'UML auquel le modèle de conception doit se conformer. Ce sous-ensemble permet de décrire la partie structurelle et la partie comportementale du modèle et peut se représenter à l'aide des diagrammes de classes, de structure composite, et d'états-transitions. Le comportement à grains fins est spécifié à l'aide d'un langage d'action permettant de définir précisément les gardes et effets des transitions des machines à états du système. Le langage d'action d'EMI est le langage C enrichi avec des macros C permettant d'accéder aux objets du modèle et à leurs attributs (e.g., état courant de la machine à états, valeurs des attributs).

Sérialisation. Le modèle de conception est ensuite sérialisé en langage C, le langage natif de notre interpréteur de modèles. La sérialisation est une transposition des éléments du modèle en termes d'initialiseurs de structures C. Contrairement à la génération de code, cette étape ne capture pas la sémantique du langage puisqu'elle ne génère que des données et pas de fonctions (sauf pour les expressions du langage d'action définies de façon opaque dans le modèle).

Chargement du modèle. Le modèle sérialisé et le code source de l'interpréteur sont ensuite compilés à l'aide d'un compilateur C afin d'obtenir le binaire exécutable du système. Cette opération peut être vue comme le chargement du modèle UML dans EMI lors de la compilation. Afin de rendre déterministe l'exécution du modèle, la sémantique opérationnelle encodée dans EMI ne doit pas admettre de comportements indéfinis. Pour

chaque point de variation sémantique d'UML, l'interpréteur doit choisir un comportement satisfaisant aux contraintes imposées par UML. Dans EMI, le comportement choisi est soit directement encodé dans l'interpréteur, soit configurable par l'utilisateur au moment de la compilation lorsque plusieurs alternatives sont disponibles.

Exécution. A partir du binaire exécutable généré, le modèle UML peut être exécuté par EMI. L'exécution est pilotée soit par la boucle d'exécution principale de l'interpréteur soit par l'outil de V&V connecté à EMI. Cet interpréteur de modèles peut être exécuté sur un ordinateur équipé d'un système d'exploitation Linux ou en bare-metal (i.e., sans OS) par exemple sur une carte embarquée STM32 discovery.

3 Vérification de Modèles UML

Pour vérifier un modèle UML, des outils de V&V peuvent être connectés à EMI pour piloter l'exécution du modèle. Les activités de V&V portent ainsi directement sur le modèle UML exécutable chargé dans EMI tout en réutilisant la même implémentation de la sémantique opérationnelle que celle utilisée lors de l'exécution du système. Cette approche permet ainsi d'assurer que les propriétés vérifiées par les outils de vérification formelle le sont aussi sur le modèle exécutable puisque le même couple (modèle UML + interpréteur de modèles) est utilisé pour l'exécution et l'analyse du modèle. Cette approche permet également d'assurer la validité des résultats de vérification même pour des langages semi-formels comme UML puisque tous les choix d'implémentation sont capturés dans une seule définition de la sémantique d'exécution.

Interface de communication. L'interpréteur EMI fournit une interface de communication permettant de connecter des outils de V&V. Cette interface de communication permet de récupérer la configuration courante de l'interpréteur (i.e., la partie dynamique du modèle), de mettre l'interpréteur dans une configuration donnée, de collecter l'ensemble des transitions tirables, et de tirer une transition à partir de sa configuration courante. Pour le model-checking, une requête supplémentaire est nécessaire afin de pouvoir évaluer des prédicats sur EMI et ainsi mieux découpler les opérations spécifiques à la vérification de celles spécifiques au langage de modélisation.

Activités de Vérification et Validation. Notre outil, EMI, peut s'interfacer avec le model-checker OBP2 pour effectuer différentes activités de V&V sur des modèles UML :

Model-checking de propriétés LTL OBP2 permet de vérifier des propriétés LTL sur les modèles exécutés par EMI. La vérification de ces propriétés LTL se base sur la composition d'automates de Büchi afin d'analyser des traces infinies. Cette technique nécessite notamment d'évaluer des propositions atomiques (i.e., des prédicats dépendants des objets du modèle) sur EMI. Si une propriété est violée, un contre-exemple sous la forme d'une trace est retourné par le model-checker.

Model-checking avec des automates observateurs Des propriétés de sûreté peuvent être spécifiées dans le langage de modélisation (ici UML) sous la forme d'automates observateurs. La vérification de ces propriétés est ensuite réalisée par composition synchrone avec le système et l'utilisation d'un algorithme d'atteignabilité.

Monitoring Les automates observateurs utilisés pendant la phase de vérification peuvent être déployés sur la cible embarquée avec EMI afin de surveiller le système lors de

son exécution. Malgré l'overhead engendré, le monitoring permet de vérifier les propriétés du système dans son environnement réel, de réagir en cas de défaillance, et d'analyser le problème à posteriori.

Simulation L'interface de simulation permet aux utilisateurs de visualiser la configuration courante d'EMI, de tirer des transitions, et d'explorer différents chemins d'exécution pour mieux comprendre le comportement du système.

Exploration de l'espace d'états OBP2 permet d'explorer l'espace d'états du modèle exécuté par EMI en utilisant un algorithme d'exploration à la volée.

Détection de deadlocks Le model-checker peut également détecter des deadlocks dans le modèle UML et retourner une trace des chemins ayant conduit à ces deadlocks.

Pour toutes ces activités, notre approche permet d'exprimer les résultats de vérification directement avec les concepts du langage de modélisation. Cela offre l'avantage de faciliter l'analyse et la compréhension des résultats par les utilisateurs.

4 Conclusion

Notre interpréteur de modèles UML permet d'améliorer la vérification des systèmes embarqués en appliquant directement les activités de V&V sur le modèle de conception et en utilisant la même sémantique opérationnelle que celle utilisée à l'exécution. De nombreux outils d'exécution de modèles existent, comme le montre l'étude systématique en [2], mais aucun ne se base sur une seule définition de la sémantique d'exécution pour vérifier et exécuter des modèles UML. Afin d'évaluer notre approche, EMI a notamment été mis en oeuvre sur un modèle de contrôleur de passage à niveau [1]. Une analyse plus approfondie reste à mener pour évaluer le passage à l'échelle et les performances de cet outil sur des modèles industriels. Pour enrichir cet outil, plusieurs perspectives sont à l'étude dont l'amélioration du déploiement qui permet de lier le modèle UML aux périphériques de la cible embarquée.

Remerciements Ce projet est partiellement financé par Davidson Consulting. Les auteurs remercient particulièrement David Olivier pour ses conseils et ses commentaires avisés sur le projet.

Références

[1] Valentin Besnard, Matthias Brun, Frédéric Jouault, Ciprian Teodorov, and Philippe Dhaussy. Unified LTL Verification and Embedded Execution of UML Models. In *ACM/IEEE 21th International Conference on Model Driven Engineering Languages and Systems (MODELS '18)*, Copenhagen, Denmark, October 2018.

[2] Federico Ciccozzi, Ivano Malavolta, and Bran Selic. Execution of UML models : a systematic review of research and practice. *Software & Systems Modeling*, April 2018.

[3] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems. *International Journal on Software Tools for Technology Transfer*, 19(2) :229–245, April 2017.

[4] Ciprian Teodorov, Luka Le Roux, Zoé Drey, and Philippe Dhaussy. Past-Free[ze] reachability analysis : reaching further with DAG-directed exhaustive state-space analysis. *Software Testing, Verification and Reliability*, 26(7) :516–542, 2016.

Spécification incrémentale d'un système d'aide au diagnostic de l'épuisement professionnel : problématique et revue de littérature

Marion Kissous[1], Anne-Lise Courbis[1], Thomas Lambolais[1], Gérard Dray[1], Sophie Martin[2]

[1]LGI2P, IMT Mines Alès, prenom.nom@mines-ales.fr
[2]Epsylon, Université Montpellier 3, prenom.nom@univ-montp3.fr
Plateforme COGITHON, MSH Sud, Montpellier
Financée par la Région Occitanie

avril 2019

Résumé

Cet article présente la problématique d'une thèse initiée en novembre 2018, ainsi qu'une synthèse de la revue de littérature sur le syndrome d'épuisement professionnel. Nous cherchons à étudier l'apport d'une approche formelle de développement incrémental dans le domaine de la santé et des applications mobiles, pour spécifier, concevoir, évaluer et développer une solution logicielle d'aide au diagnostic de ce syndrome. Nous expliquons les choix faits à ce jour aussi bien pour le versant psychologique que pour le versant informatique. Le cas particulier de l'épuisement professionnel nous semble approprié à ce type de recherche. Les approches formelles permettent de détecter au plus tôt des erreurs de spécification et de conception. Ces approches formelles seront associées à une technique de développement incrémental maintenant les propriétés de vivacité et de sûreté du système tout au long du développement. Nous utiliserons également l'approche Agile afin de favoriser les échanges entre utilisateurs et concepteurs. Le système réalisé vise ainsi à répondre aux exigences non fonctionnelles de fiabilité, robustesse, performance et utilisabilité.

Mots-clés : épuisement professionnel, *burnout*, charge mentale, stress, approches formelles, approche Agile, développement incrémental.

1 Introduction

Nos travaux visent à étudier la faisabilité et l'intérêt d'une approche formelle et incrémentale [19, 20] pour le développement d'un système de détection de situations à risque d'épuisement professionnel. Ce système sera composé d'une application à destination des professionnels de soins et de santé et d'une autre application à destination des individus. À travers cet objet d'étude, nous cherchons à savoir *en quoi l'approche incrémentale et formelle [19] permet de rendre compte de l'aspect multi-factoriel des comportements humains présents dans le cas du syndrome de l'épuisement professionnel.* Cet objectif se justifie par le fait que d'une part les applications de santé exigent un haut degré de fiabilité, d'où l'intérêt pour des approches formelles, d'autre part la multiplicité des facteurs et des données en entrée nous incite à suivre une démarche en spirale, visant à ne considérer que quelques facteurs initialement.

Pour aborder cette problématique, nous devons avant tout nous familiariser avec le travail mené par les chercheurs en sciences humaines et sociales ainsi que les professionnels de soins et de santé pour définir le syndrome d'épuisement professionnel. Plusieurs angles d'approches ont été formulés. En effet, l'épuisement professionnel est vu comme l'état final de l'individu en rupture ainsi que comme un

processus menant à l'état final de rupture. Nous avons pris la décision de considérer l'épuisement professionnel comme un processus. L'épuisement professionnel est un phénomène multi-factoriel complexe nécessitant de prendre en compte la situation professionnelle de l'individu mais également ses attributs psychologiques. Nous allons donc développer une application à destination des professionnels de soins et de santé leur permettant d'évaluer la situation professionnelle et personnelle de l'individu. Une seconde application sera également développée permettant d'obtenir des données physiologiques, physiques et comportementales sur l'individu. C'est la combinaison de ces deux sources de données qui permettra au professionnel de soins et de santé de poser son diagnostic.

L'application ne prétend pas se substituer aux professionnels de soins et de santé, seuls aptes à poser un diagnostic et formuler des recommandations. Nous souhaitons leur fournir une application « boîte à outils » leur permettant de combiner les données nécessaires à poser leur diagnostic.

Les approches formelles permettent de s'inscrire dans une démarche de formulation et de spécification des exigences (qui, quoi, pourquoi, avec quelles qualités) mais également de conception, d'évaluation et de développement au plus près des besoins utilisateur (pour quoi, pour qui, par qui, par quoi et comment).

Dans cet article, une synthèse des travaux menés par les psychologues sur l'épuisement professionnel est donnée, ainsi qu'un panel de marqueurs pouvant être utilisés pour repérer les états de stress, d'anxiété ou de charge mentale. Nous vous présentons ensuite un aperçu de état de l'art des applications mobiles existantes et utilisées. Puis, nous décrivons la démarche que nous adopterons pour répondre à notre problématique. Enfin, nous présentons les perspectives envisagées à ce jour.

2 Épuisement professionnel et indicateurs de stress

2.1 Principales approches

Le syndrome d'épuisement professionnel (ou *burnout*) est un phénomène sociétal grandissant [1], traduisant un mal-être de certains salariés dans leur entreprise ou institution [15, 13, 3, 2, 5]. Les causes de cet état prennent leurs racines dans la sphère professionnelle. L'épuisement professionnel bénéficie de plus de cinquante définitions à travers la littérature scientifique [17, 34, 31]. Il n'est pas reconnu comme une maladie et n'est donc pas inscrit au tableau des maladies professionnelles [14].

Les premiers travaux sur le concept d'épuisement professionnel ont été menés par Freudenberger [7], à partir de 1974, et continuent de nos jours avec, par exemple, les travaux de Gil-Monte [9] en 2012. Au vu du nombre de définitions depuis 1970, des différentes approches théoriques et empiriques utilisées, ou encore des différents outils développés, la définition de l'épuisement professionnel évolue au fur-et-à-mesure des nombreux travaux menés.

Dans les années 70, pour Freudenberger, c'est la « maladie du battant » [7] et pour Maslach et Jackson, moins de dix ans plus tard, c'est « un syndrome tri-dimensionnel » [27].

Il y a plusieurs points communs à ces théories. Le premier est d'identifier qu'il y a une forte rupture de sens chez l'individu. Rupture entre le sens porté par l'individu et les tâches qu'il effectue quotidiennement. Un autre point commun concerne celui de l'organisation du travail (planning, nombre de tâches, niveau de responsabilité, place à l'autonomie). Il joue un rôle déterminant dans l'apparition du syndrome. Les causes de l'épuisement professionnel prennent leur racine dans la sphère professionnelle. Un autre point commun identifié se rapporte à l'individu lui-même *i.e.* sa personnalité. Deux individus vivant l'exacte même situation n'auront pas le même ressenti, n'appliquant ainsi pas la même « stratégie de défense » appelée *coping*. Ce qui nous amène au dernier point commun identifié, à savoir le fait que l'épuisement professionnel est une stratégie de défense mise en place par l'individu. Il correspond au dernier rempart de protection de l'individu pour se sortir d'une situation perçue comme dangereuse.

2.2 Approche de Maslach

Les travaux de Maslach [27, 24] dominent la littérature. Aussi bien sa théorie que les outils qu'elle a développés sont utilisés par les chercheurs pour comprendre et évaluer le syndrome d'épuisement professionnel. Maslach s'intéresse particulièrement aux professions dites d'aidants où la relation avec l'usager est quasi-permanente. Elle supposera d'ailleurs au début que ce syndrome ne peut s'observer que parmi ces types de métiers : personnel hospitalier, forces de l'ordre, enseignant.

Les trois dimensions qu'elle identifie pour décrire ce syndrome sont : 1) *l'épuisement émotionnel*, défini comme le sentiment d'être envahi ou épuisé par son travail, 2) *la dépersonnalisation*, qui se manifeste par des sentiments d'insensibilité et des réponses impersonnelles aux clients et 3) *l'accomplissement personnel au travail*, traduit par des sentiments de compétence et de réalisation de soi au travail. L'épuisement professionnel se caractérise par un niveau élevé des dimensions 1) et 2) et un niveau faible de la dimension 3). Un outil appelé MBI — *Maslach Burnout Inventory* — a été développé en ce sens, tel que nous le verrons au paragraphe 3. Nous envisageons d'intégrer le MBI dans notre boîte à outils pour le professionnel de santé.

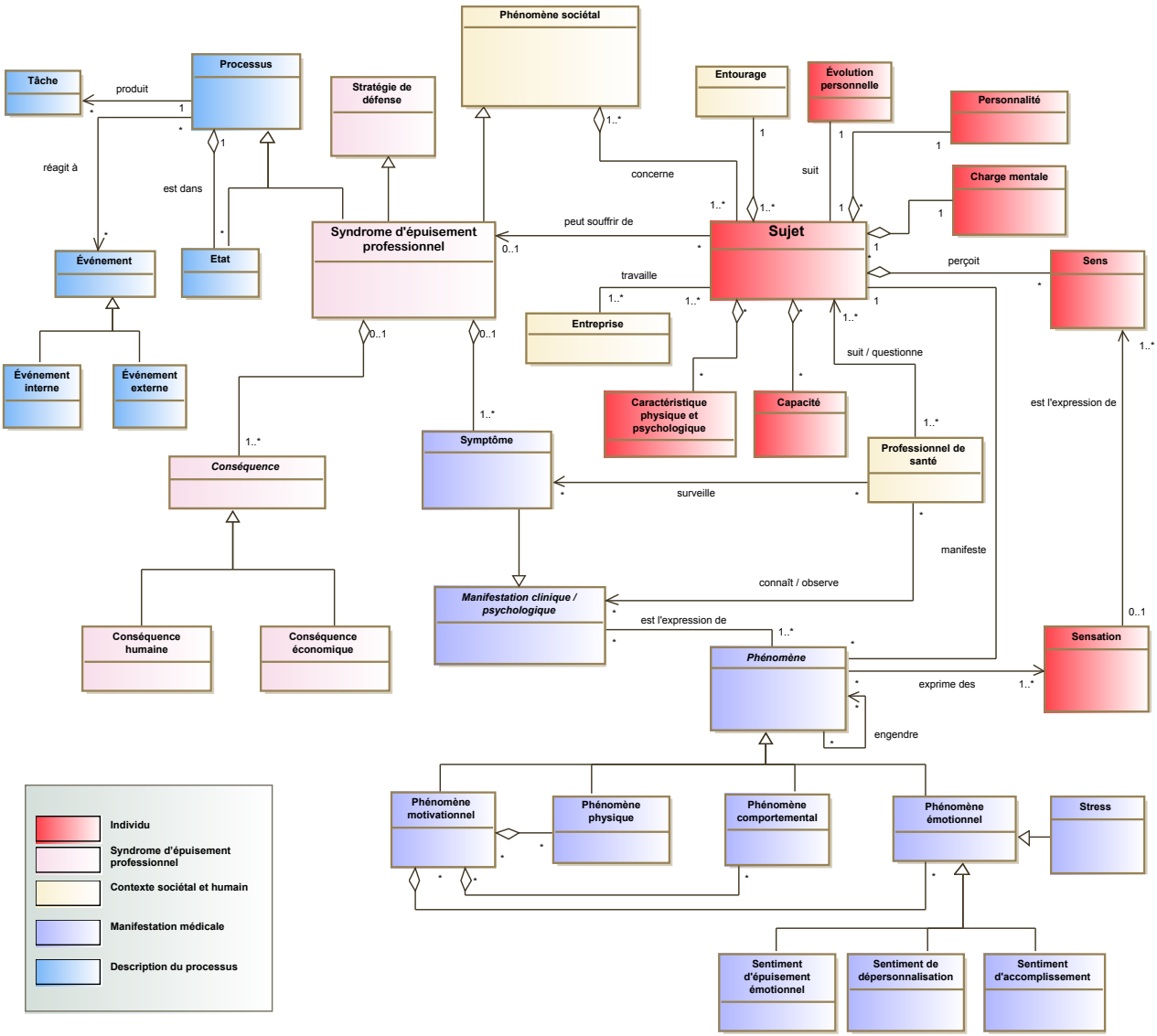


FIGURE 1 – Diagramme de classe du domaine : principaux concepts de l'épuisement professionnel.

2.3 Notre compréhension

Nous n'abordons pas l'épuisement professionnel comme l'état de rupture mais comme le processus conduisant à cet état de rupture. En effet, l'épuisement professionnel est le résultat d'une charge mentale [1] [12] répétée et élevée, dont les effets se cumulent avec ceux d'une organisation mal-adaptée : les ressources tendent à baisser, particulièrement au regard de la charge de travail qui, elle, augmente, où l'autonomie n'est pas ou peu valorisée, où une perte de sens s'opère pour l'individu entre ce qu'il pensait réaliser dans l'entreprise *vs* ce qu'il y fait réellement.

Les principales notions que nous retenons sont présentées dans le diagramme de classe du domaine (fig. 1). Nous distinguons cinq catégories de classes :

— ce qui concerne le sujet, à savoir l'individu dont on veut surveiller les risques d'épuisement professionnel : sa personnalité, ses caractéristiques physiques et psychologiques, sa charge mentale, ses perceptions sensorielles, ses capacités physiques et cognitives et enfin son évolution personnelle ;
— ce qui concerne le contexte humain et organisationnel de l'individu : l'entreprise dans laquelle il travaille, son entourage personnel et social. C'est aussi dans cette catégorie que nous plaçons le professionnel de santé qui le suit ;
— le syndrome d'épuisement professionnel, dont on souligne ici une cause ou une « utilité » (la stratégie de défense) et des conséquences ;
— l'ensemble des symptômes qui caractérisent l'épuisement professionnel. Nous soulignons ici les trois dimensions émotionnelles de Maslach : sentiments d'épuisement émotionnel, de dépersonnalisation et d'accomplissement ;
— enfin, le fait de voir l'épuisement professionnel comme un processus dont on veut surveiller l'évolution : ce processus réagit à des événements endogènes ou exogènes, et produit des actions (tâches).

Le syndrome d'épuisement professionnel a été largement étudié parmi les mêmes types de professions (*cf.* paragraphe 2.2). Les travaux de recherche pour ces professions dominent la littérature. Des travaux menés sur d'autres professions émergent depuis dix ans environ, à propos des agriculteurs par exemple [23, 22].

Des outils pour identifier et prévenir l'épuisement professionnel sont développés à partir de théories qui postulent de l'existence d'une situation multi-factorielle déclenchant une situation à risque pour l'individu. Les outils élaborés au regard des théories sont des questionnaires dont la passation se fait par auto-évaluation et dont l'objectif est de déterminer la situation professionnelle et personnelle du sujet. Cette méthode rencontre des limites dont la principale nous semble être celle de la sincérité et donc la fiabilité des réponses apportées. Nous nous devons de nous interroger sur la sincérité et donc la fiabilité des réponses fournies en auto-évaluation. Le sujet peut-il répondre librement ? Est-il dans le déni (ou la dénégation) de ce qu'il vit ? Nous pensons que l'état de déni est un état transitoire commun aux individus. Il est alors important de garantir que l'individu est sincère vis-à-vis de lui-même pendant la passation du questionnaire. Le soutien par un professionnel de santé dans la passation du questionnaire permet d'avoir un regard extérieur objectif pour « contrôler » la sincérité des réponses fournies.

Il est nécessaire d'avoir recours à ce type de questionnaires, que nous détaillerons dans le paragraphe 3.1, puisqu'ils permettent de connaître le profil psychologique, comportemental de l'individu en situation dite normale. Le questionnement réside plutôt dans la pertinence ou non de les utiliser, d'en utiliser d'autres ou d'en combiner plusieurs. En effet, nous pourrions utiliser le MMPI-2 [11] ou le Big Five [10] qui sont les deux questionnaires utilisés en psychologie et appartenant chacun

1. La charge mentale est un terme qui s'est récemment popularisé pour désigner la charge cognitive, invisible, que représente l'organisation de tout ce qui se situe dans la sphère domestique : tâches ménagères, rendez-vous, achats, soins aux enfants, etc. Ce terme est apparu en premier lieu dans la sphère privée et comme incombant aux femmes. Monique Haicault est la première sociologue, française, à faire émerger ce terme.

à un courant de pensée différent. Ce point est encore en discussion. En revanche, nous savons que nous devons combiner ce type d'outils à des observations de paramètres physiques, physiologiques et environnementaux. *Nous formulons l'hypothèse que, quel que soit le métier exercé par l'individu, l'épuisement professionnel peut être décrit comme un processus.* Ce processus comporte des étapes, des événements internes et externes, des manifestations d'actions, peut-être des cycles, voire des états éventuellement concurrents.

Sans s'arrêter à une seule définition, il nous est possible de dire que l'épuisement professionnel est le résultat d'une période de stress intense et répété, vécu comme une charge mentale, dont les facteurs déclencheurs surviennent en premier lieu dans le milieu professionnel avant d'envahir la sphère personnelle. L'épuisement professionnel prendrait sa source spécifiquement dans un contexte de travail et serait défini comme une réponse à un stress excessif et/ou continu au travail.

Mesurer des données physiques, physiologiques et comportementales est nécessaire pour identifier le seuil de passage entre les différents états transitoires inhérents au processus de l'épuisement professionnel. En relevant ce type de données, nous pourrons évaluer la pertinence de s'y limiter ou non.

2.4 Indicateurs de stress

Des facteurs de stress ont été mesurés et validés grâce à plusieurs études, nous pouvons en trouver une synthèse dans [4]. Le stress est identifié à partir des symptômes qui en découlent. Par exemple, certains symptômes reconnus à ce jour sont d'ordre physiologiques comme les maux de tête ou les douleurs musculaires, les troubles du sommeil, de l'appétit, les infections à répétition. D'autres symptômes relèvent plutôt du comportement de l'individu, comme des émotions aux intensités accrues, le recours à des produits « calmants » (tabac, alcool), l'inhibition, le repli sur soi, les oublis, les erreurs (à répétition), et ceci sur une période longue [34].

Certains indicateurs paraissent, à ce jour, difficiles à observer via les capteurs présents sur un téléphone portable (consommation d'alcool ou tabac, les infections à répétition). D'autres semblent observables (troubles éventuels du sommeil ainsi que la période concernée et sa durée par exemple). Les indicateurs retenus seront implémentés au fur-et-à-mesure du développement.

3 Outils et applications numériques existants

3.1 Questionnaires

Grâce aux recherches de Freudenberger, Maslach, Lieter, Schauffeli, Cherniss, Farber, Siegrist, Karasek et d'autres, de nombreux questionnaires existent et servent d'outil d'évaluation de l'épuisement professionnel. Il n'est pas possible de les citer tous. Les plus connus sont [21] : *Burnout Questionary* de Freudenberger et Richelson (1980), *Individual Burnout Symptomatic Questionary* d'Appelbaum (1980), *Staff Burnout Scale* de Jones (1980), *Tedium Measure* de Pines et Aronson (1981), *Emener-Luck Burnout Scale* d'Emener, Luck et Gohs (1982), *Job Burnout Inventory* de Ford, Murphy et Edwards (1983), *Meier Burnout Assessment* de Meier (1984), *Energy Depletion Index* de Garden (1985), *Cherniss Burnout Measure* de Burke et Deszca (1986), *Teacher Burnout Scale* de Seidman et Zager (1986–1987), *Copenhagen Burnout Inventory* (CBI, 2007), *Spanish Burnout Inventory* (SBI, 2011), etc.

L'outil le plus utilisé, le *Maslach Burnout Inventory* (1980) [25], est resté longtemps sans traduction et donc uniquement utilisable pour les anglo-américains. Une traduction française est disponible. Elle a été élaborée par des universitaires canadiens [6] et convient moins à la culture française. Ayant fait ses preuves dans la culture canadienne, le MBI a également donné de bons résultats sur les populations observées dites d'aidants. Il faut garder à l'esprit qu'il a été élaboré pour eux.

Un autre outil commence à émerger dans les études : le CBI — *Copenhagen Burnout Inventory* [18]. Des études ont déjà été menées pour tester sa validité, comme par exemple en Nouvelle-Zélande sur les enseignants de second degré [28]. Il a donné de bons résultats permettant ainsi de le valider sur la population observée. Il a également été testé dans sa version chinoise auprès d'employés de deux compagnies privées à Taïwan [33] avec des résultats plus mitigés puisque certains concepts se confondaient. Des améliorations ont été proposées.

Une limite de ces outils est que le diagnostic repose sur des résultats produits par des outils d'auto-évaluation. Nous ne remettons pas en question toute l'utilité d'un questionnaire par auto-évaluation car parler de ce qui va « mal » est déjà un pas dans le processus d'acceptation et de donc de « guérison ». Toutefois, le caractère auto-évalué en fait un outil posant question sur la sincérité des réponses apportées. En effet, si l'épuisement professionnel est bien un syndrome des employés les plus attachés à leur travail avec des valeurs assumées, ils peuvent souvent traverser une phase de déni (ou dénégation) de leur état. Peut-on donc réellement se contenter d'un questionnaire de ce type ? Quelle serait la plus-value d'une « boîte à outils » permettant de prévenir ces situations de stress intense et/ou répété conduisant à l'épuisement professionnel ?

Enfin, ces questionnaires ont été validés sur des populations spécifiques (*cf.* paragraphe 2.2), à chaque fois dans un contexte national, et donc culturel, également spécifique. Nous n'avons pas trouvé de travaux observant les aides-soignants à Paris et les aides-soignants à Madrid par exemple. La question de la profession se pose alors. Est-elle un facteur clé déterminant dans la compréhension du processus aboutissant à l'épuisement professionnel ? Ou peut-on s'affranchir de ce paramètre et détecter des similitudes situationnelles conduisant à cet épuisement ?

3.2 Applications numériques

Nous nous intéressons ici aux applications développées sur tablettes et/ou téléphones portables, en évaluant les questions suivantes : Est-ce un marché ? Qui sont les utilisateurs ? Intègrent-elles d'autres outils que les questionnaires auto-administrés ? Sont-elles « populaires » ? Nous ne parlerons ici que d'environ la moitié des applications, traitant du stress, déjà identifiées, une vingtaine environ.

Ces applications sont en lien avec le stress et/ou l'anxiété, parfois même la dépression. Une seule application porte le nom de *burnout*. Les autres ne parlent jamais directement d'épuisement professionnel mais plutôt de stress ou d'anxiété. La majorité de ces applications est disponible en langue anglaise. Elles s'adressent directement à l'individu et non à un chef d'entreprise ou à un responsable des relations humaines par exemple, et encore moins à un professionnel de santé type médecin du travail ou psychologue de l'entreprise. Ces applications peuvent se répartir en quatre familles :

1. Les premières relèvent de la **gestion du stress ou de l'anxiété** vécus, perçus par l'individu. Ces applications partent du principe que l'individu a connaissance à un instant donné d'être en état de stress ou d'anxiété. L'application propose alors de guider l'utilisateur vers un état d'apaisement, de détente. La plupart contienne très peu de questions, par exemple une simple évaluation par une échelle de type Likert [2] de son niveau d'humeur, les échelles variant suivant l'application. Toute une série de « trucs et astuces » autour de la méditation, de la respiration ou de la relaxation est ainsi proposée à l'individu pour retrouver rapidement et facilement un niveau de stress motivant ou d'apaisement en cas de stress sévère.
2. La seconde famille d'applications permet de **détecter un éventuel état de stress ou d'anxiété** puis de fournir une aide à la compréhension de cet état. Le principe est le même que précédemment avec une auto-évaluation de son humeur mais également une série de questions permettant « d'analyser » la journée d'un point de vue facteurs positifs *vs* facteurs négatifs. Un « bilan » est alors proposé et un accès à des outils de type développement personnel pour gérer l'état de stress vécu ou perçu.

2. Une échelle de Likert est un outil psychométrique permettant de mesurer une attitude chez des individus.

3. La troisième famille est celle des applications indirectement liées à un « diagnostic » de stress. Elles se basent sur un **suivi de mesures de facteurs physiologiques** tel que le rythme cardiaque, la pression sanguine ou encore la qualité du sommeil pour observer s'il y a un état de déséquilibre et donc de stress. Ces applications ne fournissent aucune analyse de ce suivi, et c'est à l'utilisateur de comprendre si son rythme cardiaque est bon ou sa pression sanguine élevée.
4. La quatrième famille concerne **les applications pour les (hauts-)responsables de l'entreprise** et développées par des autorités de santé ou organismes reconnus mais non-gouvernementaux. Cette famille ne contient qu'une application pour le moment. Il nous parait important de la distinguer des autres applications pour plusieurs raisons. Tout d'abord, nous devons mentionner que les applications décrites ci-dessus sont développées en collaboration avec des développeurs et des psychologues et/ou psychiatres, ou une personne ayant vécu un épuisement professionnel. *A contrario*, cette application est développée par *The International Labour Office* (ILO). L'application du ILO est totalement gratuite et libre. Cette application, appelée « Checkpoints ou La prévention du stress au travail », s'adresse non pas à l'individu mais bien à l'entité « entreprise ». Cela pourrait se comparer à un guide complet où sont identifiés dix dimensions, telles que le « Leadership and justice at work », « Job demands », « Social support », etc., pour lesquelles cinq items construisent une dimension, soit cinquante items au total. Cette liste exhaustive des points de contrôle permet ainsi à l'entreprise de choisir les siens, qui lui sont propres. Ceci permet de construire un outil personnalisé selon le type d'organisation, à partir de l'outil complet de ILO. ILO indique également dans ce guide comment utiliser leur liste de points de contrôle. Par exemple, en organisant des workshops ou des journées de formation pour les employés. Cet outil est intéressant. Il est diffusé via les canaux pour les outils numériques à usage personnel et intègre le stress au travail comme réel, digne d'intérêt et pour lequel l'entreprise a un rôle à assumer [29]. Enfin, bien que les autres applications soient gratuites d'installation, elles comprennent toutes une partie payante si l'on veut « upgrader » ou accéder à l'ensemble des options — certaines applications de la troisième famille échappent à cette règle.

Toutes ces applications sont apparues au début des années 2010. Elles viennent du Nord de l'Amérique, de France, d'Espagne, d'Allemagne, d'Angleterre, d'Australie, d'Inde, etc. Le nombre de téléchargements dépasse les 500 000 pour certaines et comptent plusieurs dizaines de milliers de notes et commentaires.

Les limites que nous percevons à ce type d'application sont le caractère auto-évalué — les mêmes limites que pour les questionnaires —, l'absence de lien avec un professionnel de santé — seul apte à décider de la prise en charge éventuelle de l'individu —, ou encore l'accès à des fonctionnalités payantes pour bénéficier de l'ensemble de l'application.

Notre application se distingue de celles existantes dans la mesure où nous souhaitons fournir un système (application mobile et application web) où les professionnels de soins et de santé bénéficieront d'une boîte à outils pour les aider à prévenir les situations à risque d'épuisement professionnel. L'individu ne doit pas être livré à lui-même pour faire son propre bilan et l'interpréter. Cette vue est présentée à la figure 2, où l'on voit apparaître que le système d'aide au diagnostic compare un syndrome d'épuisement professionnel « de référence » avec l'évolution effective du sujet. La partie web du système peut être en relation avec plusieurs professionnels de soins et de santé pour les aider à poser leur éventuel diagnostic.

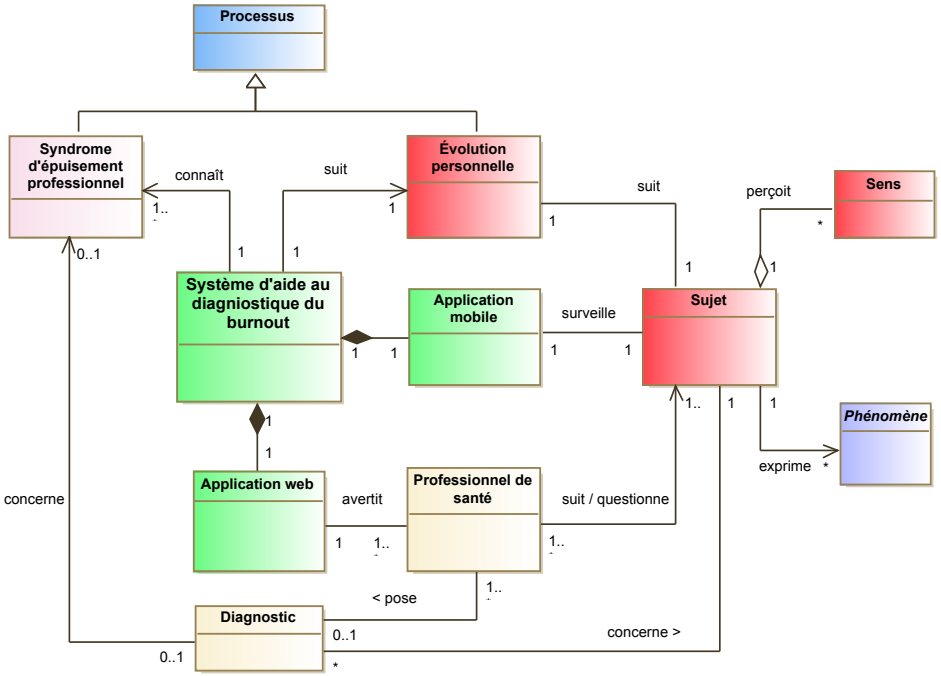


FIGURE 2 – Diagramme de classe décrivant l'interaction du système avec son environnement.

4 Quelle démarche suivre ?

4.1 Questionnaires et enquêtes

Nous avons vu que nous disposons de plusieurs outils d'évaluation psychologique utilisés pour « dresser un portrait » du sujet. Nous avons également vu que la passation peut être soit en mode auto-évaluation soit réalisée par le professionnel de santé. Dans tous les cas, il ne serait pas judicieux de nous soustraire à la passation d'un questionnaire. Comme mentionné plus haut (*cf.* paragraphe 3.1), le fait de répondre à des questions sur l'organisation de son entreprise, la marge d'autonomie effective, les rapports entre collègues et avec ses supérieurs, etc, permet à l'individu de formuler son éventuel mal-être professionnel, premier pas vers le mieux-être. Il est donc important de ne pas s'en priver. Cela permet aussi de récolter des données comportementales, de vécu et de ressenti sur le travail. Par exemple, le travail est-il associé (uniquement) à des contraintes ou plutôt à un épanouissement personnel ? Nous faisons le choix d'utiliser plusieurs outils d'évaluation psychologique reconnus pour leur robustesse. Les questionnaires choisis devront être validés statistiquement.

La plateforme d'enquêtes Epsyline® du laboratoire Epsylon [3] permet de faire des passations de questionnaires de façon simple, à grande échelle puiqu'elle donne un accès direct à l'ensemble des étudiants de l'université. Il est envisageable de compter sur la participation d'environ 300 étudiants, population connue pour « souffrir » de stress intense et/ou continu.

4.2 Interviews et développement itératif

En parallèle, nous mènerons une série d'interviews auprès de professionnels de soins et de santé (médecins du travail, psychologue du travail, kinésithérapeutes, médecins généralistes) et d'individus lambda. Le but de ces interviews est multiple : constater la part des professionnels de soins et de santé qui pensent que l'épuisement professionnel existe, puis parmi eux ceux qui y ont déjà été confrontés. Identifier l'aide dont voudraient bénéficier les professionnels de soins et de santé fait également partie des objectifs des interviews. Quant aux individus non-experts, nous voulons faire remonter des besoins ou des facteurs de l'épuisement professionnel qui ne seraient pas ressortis lors des interviews avec les

3. https ://www.epsylab.fr/

« experts ». Nous cherchons ainsi à confronter les avis experts *vs* novices. Un autre objectif est de faire adhérer des professionnels de soins et de santé à notre démarche afin de les intégrer dans notre processus de développement. Des techniques telles que celles utilisées dans l'approche Agile permettent de centrer l'utilisateur dans la conception et le développement de l'outil. Ce type de démarche permet de clarifier les liens entre besoins identifiés et fonctionnalités nécessaires pour y répondre. Dans notre démarche, l'utilisateur principal est le professionnel de soins ou de santé. Le professionnel de soins ou de santé sera en charge de faire passer les questionnaires, évitant les limites identifiées par l'auto-évaluation. Le professionnel de santé, surtout s'il est en lien avec l'entreprise, sera plus à même de faire preuve d'objectivité et de détecter les situations de déni ou de dénégation de l'individu si elles se présentent. Par ailleurs, c'est le professionnel de santé qui recommandera à l'individu d'installer et d'utiliser notre système embarqué. En effet, si le médecin détecte une situation potentiellement « déséquilibrante » pour l'individu, il pourra établir un suivi à partir de données physiques et physiologiques. Ainsi, au rendez-vous suivant, et en repassant éventuellement le questionnaire, le professionnel de santé disposera de critères physiques, physiologiques et comportementaux pour évaluer la situation dans laquelle se trouve l'individu. Le but est de donner les moyens au professionnel de santé d'anticiper une situation qui deviendrait trop critique et finirait par un probable épuisement de l'individu.

Les phases de développement du système embarqué se dérouleront de façon itérative afin de permettre un aller-retour régulier avec les utilisateurs pour valider ou infirmer les développements au fur-et-à-mesure. Quant à la vérification de la pertinence des fonctionnalités développées au regard des besoins, elle découlera des tests avec les professionnels de soins et de santé.

4.3 Méthode IDF

La méthode IDF [19] d'aide à la spécification comportementale est un cadre de construction incrémental formé :

— d'un ensemble de techniques d'évaluation entre deux étapes de spécification : utilisation de relations de comparaison (équivalences et préordres) entre deux spécifications comportementales, cherchant à détecter si la nouvelle spécification n'autorise pas de comportements qu'elle ne devrait pas autoriser (propriétés de sûreté), ainsi que si la nouvelle spécification n'interdit pas de comportements qu'elle devrait autoriser (propriétés de vivacité) ;

— d'un ensemble de techniques de construction. Ces techniques s'appuient sur des opérateurs de construction permettant de suivre des démarche horizontales, par extension ou restriction de fonctionnalités, ainsi que des démarches verticales, par raffinement ou abstraction de comportement. Les spécifications sont données, soit comme des composants composites UML dont les comportements sont définis par des machines d'états UML, soit comme des processus décrit dans une algèbre de processus.

Il s'agit donc d'une application des méthodes formelles (algèbres de processus, systèmes de transitions étiquetées et techniques de vérification, de raffinement et construction incrémentale associées) aux méthodes Agile.

Nous souhaitons développer un outil qui ne soit ni intrusif pour l'individu, ni chronophage. C'est pourquoi nous avons décidé d'utiliser uniquement les composants disponibles sur téléphones portables. Un programme de détection des composants présents sur le téléphone de l'individu devra être intégré à notre système afin d'anticiper les mesures pouvant être relevées. En l'absence d'un composant donné, nous devrons trouver un moyen pour substituer l'information recherchée soit via un autre composant soit via la combinaison/le croisement de plusieurs composants. Une évaluation de la précision des mesures relevées devra être faite afin d'obtenir et/ou de construire des indicateurs fiables, complémentaires aux outils psychologiques existants.

5 Perspectives. Conclusion.

Les travaux menés par les chercheurs en psychologie fournissent un socle de travail essentiel pour la détection du syndrome d'épuisement professionnel. Plusieurs courants existent. Le caractère psychologique de l'épuisement professionnel doit être mesuré afin d'appréhender les aspects multi-factoriels de ce syndrome. À notre charge ensuite d'évaluer si ces critères multi-factoriels peuvent être détectés uniquement pas des mesures physiologiques, physiques et comportementales sans la nécessité d'utiliser des outils d'évaluation psychologique. L'originalité de notre approche consiste à appréhender l'épuisement professionnel comme un processus transverse aux métiers, et notamment nous visons à le modéliser grâce aux techniques de description formelle de processus. Dans le cadre de la validation de nos recherches, deux options sont envisagées : la première étant d'utiliser les questionnaires et la seconde de s'inspirer des travaux menés à l'Université de Louvain en Belgique sur « Le burnout parental » [30]. Ces chercheurs utilisent le cortisol, hormone du stress, contenu dans les cheveux pour détecter le niveau de stress des parents. L'intérêt est d'obtenir un historique du niveau de cortisol chez le sujet, historique qui ne peut pas être obtenu par la salive. Cette mesure pourrait nous servir de critère de validité externe de nos propres mesures.

La santé est un domaine sensible du fait des enjeux de (sur)vie pour les individus. Le risque d'avoir des faux positifs et des faux négatifs doit être le plus faible possible. C'est pourquoi nous avons fait le choix d'utiliser conjointement la méthode IDF (approche formelle et développement incrémental), l'approche Agile et les outils exploités en sciences humaines et sociales.

Références

[1] La Commission des AFFAIRES SOCIALES. *Rapport d'information - en conclusion des travaux de la mission d'information relative au syndrome d'épuisement professionnel.* `http://www2.assemblee-nationale.fr/documents/notice/14/rap-info/i4487`. 2017.

[2] ANACT. *Les facteurs psychosociaux de risques au travail et la santé : une approche par genre des données statistiques nationales.* `https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=9&ved=2ahUKEwim17Xo6fzgAhVCThoKHVpwAXIQFjAIegQIBRAC&url`. 2014.

[3] DARES ANALYSES. *Quelles sont les évolutions récentes des conditions de travail et des risques psychosociaux ?* `https://dares.travail-emploi.gouv.fr/IMG/pdf/2017-082v3.pdf`. 2017.

[4] D. CHOUANIÈRE et al. « Expositions psychosociales et santé : état des connaissances épidémiologiques. » In : *Documents pour le Médecin du Travail.* Volume 127, TP 13.1 (2011), pp. 509-517.

[5] P. CRESEGUT. *Documentaire : Infirmières, à coeur ouvert.* `https://france3-regions.francetvinfo.fr/auvergne-rhone-alpes/emissions/qui-sommes-nous/documentaire-3-bonnes-raisons-regarder-infirmieres-coeur-ouvert-lundi-11-fevrier-22h45-france-3-1616815.html?fbclid=IwAR0YlQMEPtOiTivRlkmOQaUWnxm3auV7Ocl_kkJnOqIbvUWodlNhYqp1Lfs`. 2019.

[6] G. DION et R. TESSIER. « Validation de la traduction de l'inventaire d'épuisement professionnel de Maslach et Jackson. » In : *Revue canadienne des sciences du comportement.* Volume 26, Issue 2.1 (1994), pp. 210-227.

[7] H. J. FREUDENBERGER. « The staff burn-out syndrome in alternative institutions. » In : *Psychotherapy : Theory, Research & Practice.* Volume 12, Issue 1.1 (1975), pp. 73-82.

[8] H.J. FREUDENBERGER et G. RICHELSON. *Burn-out : The High Cost Of High Achievement.* 1re éd. Doubleday Anchor Press, 1980.

[9] P. R. GIL-MONTE et H. FIGUEIREDO-FERRAZ. « Psychometric properties of the 'Spanish Burnout Inventory' among employees working with people with intellectual disability. » In : *Journal of Intellectual Disability Research.* Volume 57, Issue 10.2 (2013), pp. 959-968.

[10] L. R. GOLDBERG. « An alternative "description of personality" : The Big-Five factor structure. » In : *Journal of Personality and Social Psychology.* Volume 59, Issue 6.1 (1990), pp. 1216-1229.

[11] R. L. GREENE. *The MMPI-2 : An interpretive manual.* 2e éd. Needham Heights, MA, US : Allyn Bacon, 2000.

[12] M. HAICAULT. « La gestion ordinaire de la vie en deux. » In : *Sociologie du Travail.* Volume 26, Issue 3 - Travail des femmes et famille.1 (1984), pp. 268-277.

[13] INRS. *Risques psycho-sociaux. Ce qu'il faut retenir.* http://www.inrs.fr/risques/psychosociaux/ce-qu-il-faut-retenir.html. 2018.

[14] INRS. *Tableaux des maladies professionnelles.* http://www.inrs.fr/publications/bdd/mp/listeTableaux.html. 2019.

[15] INSEE. *L'exposition des travailleurs aux risques psychosociaux a-t-elle augmenté pendant la crise économique de 2008 ?* https://www.insee.fr/fr/statistiques/2110921?sommaire=2110927. 2016.

[16] E. F. IWANICKI et R. L. SCHWAB. « A Cross Validation Study of the Maslach Burnout Inventory. » In : *Educational and Psychological Measurement.* Volume 41, Issue 4.1 (1981), pp. 1167-1174.

[17] V. KOVESS-MASFETY et L. SAUNDER. « Le burnout : historique, mesures et controverses. » In : *Archives des Maladies Professionnelles et de l'Environnement. Risques psychosociaux.* Volume 78, Issue 1.1 (2017), pp. 16-23.

[18] T. S. KRISTENSEN et al. « The Copenhagen Burnout Inventory : A new tool for the assessment of burnout. » In : *Work & Stress.* Volume 19, Issue 3.1 (2007), pp. 192-207.

[19] T. LAMBOLAIS et al. « IDF : A framework for the incremental development and conformance verification of UML active primitive components. » In : *Journal of Systems and Software.* Volume 113.1 (2016), pp. 275-295.

[20] T. LAMBOLAIS et al. *Incremental Development of Compliant Models.* https://idcm.wp.imt.fr/. 2016.

[21] V. LANGEVIN et al. « Les questionnaires dans la démarche de prévention du stress au travail. » In : *Documents pour le Médecin du Travail.* Volume 125, TC 134.1 (2011), pp. 23-35.

[22] F.-R. LENOIR et L. RAMBOARISON-LALAO. « Equilibre des sphères de vie et prévention des risques psychosociaux. Le cas des exploitants agricoles. » In : *Revue Interdisciplinaire Management, Homme & Entreprise - RIMHE.* Volume 12, N3.1 (2014), pp. 45-61.

[23] M. LOUREL et C. MABIRE. « Le déséquilibre efforts-récompenses et les débordements entre vie au travail, vie privée chez les éleveurs laitiers : leurs effets sur l'épuisement professionnel. » In : *Santé publique.* Volume 20, Hors-série.1 (2008), pp. 89-98.

[24] C. MASLACH. « Job Burnout : New Directions in Research and Intervention. » In : *Current Directions in Psychological Science.* Volume 12, Issue 5.1 (2003), pp. 189-192.

[25] C. MASLACH et S. E. JACKSON. « The measurement of experienced burnout. » In : *Journal of occupational behaviour.* Volume 2, Issue 2.1 (1981), pp. 99-113.

[26] C. MASLACH, S. E. JACKSON et M. P. LEITER. *Maslach Burnout Inventory.* 3e éd. Palo Alto, CA : Consulting Psychologists Press, 1986.

[27] C. MASLACH et S.E. JACKSON. « Burnout in health professions : A social psychological analysis. » In : *Social Psychology of Health and Illness.* Sous la dir. de Glenn S. SANDERS et Jerry SULS. Psychology Press, 1982, pp. 227-251.

[28] T. L. MILFONT et al. « Burnout and Wellbeing : Testing the Copenhagen Burnout Inventory in New Zealand Teachers. » In : *Social Indicators Research.* Volume 89, Issue 1.2 (2007), p. 169–177.

[29] International Labour OFFICE. *Stress Prevention at Work Checkpoints.* http://www.ilo.org/wcmsp5/groups/public/@ed_protect/@protrav/@safework/documents/instructionalmaterial/wcms_177108.pdf. 2012.

[30] I. ROSKAM, M. MIKOLAJCZAK et M. E. BRIANDA. *Le burnout parental.* https://uclouvain.be/fr/decouvrir/actualites/le-burnout-parental-parlons-en-sans-tabou.html. 2015.

[31] W. B. SCHAUFELI, M. P. LEITER et C. MASLACH. « Burnout : 35 years of research and practice. » In : *Career Development International.* Volume 14, Issue 3.1 (2009), pp. 204-220.

[32] W. B. SCHAUFELI et al. « On the clinical validity of the Maslach Burnout Inventory and the burnout measure. » In : *Psychology & Health.* Volume 16, Issue 5.1 (2001), pp. 565-582.

[33] W.-H. YEH et al. « Psychometric properties of the chinese version of copenhagen burnout inventory among employees in two companies in Taiwan. » In : *International Journal of Behavioral Medicine.* Volume 14, Issue 3.1 (2007), pp. 126-133.

[34] P. ZAWIEJA et F. GUARNIERI. « Epuisement professionnel : approches innovantes et pluridisciplinaires. » In : 1re éd. Paris : Armand Colin, 2013. Chap. 1.

Modélisation du Domaine au Sein d'une Méthode Formelle d'Ingénierie des Exigences

Steve Jeffrey Tueno Fotso[1,2]

[1]*LACL – Université Paris Est Créteil Val de Marne, France, steve.tuenofotso@univ-paris-est.fr*
[2]*GRIL – Université de Sherbrooke, Canada, Steve.Jeffrey.Tueno.Fotso@USherbrooke.ca*

Abstract

SysML/KAOS est une méthode formelle d'ingénierie des exigences adoptée dans le cadre du projet *FORMOSE* afin de supporter les étapes de spécification, vérification et validation formelles des exigences de systèmes critiques et complexes. Les exigences sont modélisées sous forme d'hiérarchies de buts, ce qui permet de générer une spécification formelle *B System*. Cet article présente un langage permettant de modéliser le domaine d'un système et compatible avec le langage de modélisation des exigences. Il s'intéresse non seulement à la définition du langage, mais aussi des mécanismes permettant d'exploiter la modélisation du domaine pour compléter la spécification *B System* issue de la modélisation des exigences. Par ailleurs, l'article propose et illustre l'utilisation des diagrammes d'états-transitions algébriques afin de représenter les changements d'états des variables du modèle de domaine et le flow de contrôle régissant le comportement du système, au fur et à mesure qu'il satisfait ses buts. La méthode SysML/KAOS, ainsi enrichie, est supportée par l'outil *Openflexo* et a été évaluée sur différents cas d'étude d'envergure.

Index terms— Ingénierie des exigences, Modélisation du domaine, *SysML/KAOS*, *B System*, *Event-B*, *ASTDs*

1 Introduction

L'ingénierie des exigences est la partie du génie logiciel qui s'intéresse aux activités d'élicitation, d'analyse, de spécification et de validation des exigences relatives au système à mettre en place. Elle désigne les activités qui constituent la pierre angulaire de tout projet de développement logiciel ou système. L'occurrence de défaillances au cours de l'une de ces étapes a souvent des conséquences extrêmement désastreuses [24]. Le projet *FORMOSE* [3], financé par l' Agence Nationale de la Recherche (ANR) française, vient en réponse à cette problématique et s'intéresse à l'élaboration d'une méthode outillée pour la modélisation, la vérification et la validation formelle des exigences de systèmes critiques et complexes. Dans le cadre de ce projet, la méthode *SysML/KAOS* [18, 21] est adoptée afin de supporter la modélisation des exigences fonctionnelles et non fonctionnelles d'un système sous la forme d'hiérarchies de buts. Afin de vérifier et valider formellement ces exigences, les travaux décrits dans [26] définissent une correspondance entre le langage SysML/KAOS de modélisation des exigences fonctionnelles et la méthode formelle *Event-B* [2] (sémantiquement équivalente à *B System*) permettant ainsi de produire des spécifications formelles à partir des modèles d'exigences. Cette spécification sert ensuite de base pour les tâches de vérification et de validation formelle afin de détecter et corriger les potentielles défaillances.

Les règles de correspondance définies dans [26] permettent d'obtenir l'ossature d'une spécification *Event-B* formalisant les exigences du système : chaque niveau de raffinement du modèle de buts se traduit par une machine *Event-B*, un squelette d'évènement est généré pour chaque but. Plusieurs obligations de preuve sont générées afin de traduire la sémantique des liens de

raffinement. Toutefois, il est nécessaire de fournir manuellement la *partie structurelle* du modèle *Event-B* (ensembles abstraits et énumérés, constantes et leurs propriétés, et variables et leur invariant) ainsi que les corps des évènements.

Étant donné que la partie structurelle d'une spécification *Event-B* constitue une caractérisation des propriétés du domaine d'application du système, nous nous intéressons à une approche permettant de modéliser le domaine du système et qui serait compatible avec le langage de modélisation d'exigences. Cet article présente le langage de modélisation défini à cet effet [10, 11] ainsi qu'un ensemble de règles [10] permettant d'obtenir automatiquement les éléments de la partie structurelle et l'initialisation des variables d'état en s'appuyant sur la modélisation du domaine. Notons que la correction de ces règles a été vérifiée formellement [14, 12]. Cette approche permet de distinguer la spécification du domaine de celle du comportement du système. De plus, elle accentue la réutilisabilité, la lisibilité et la maintenabilité des modèles. Il ne reste plus qu'à spécifier le corps des évènements pour compléter la spécification formelle des exigences. L'article propose et illustre en conséquence l'utilisation des diagrammes d'états-transitions algébriques (ASTDs) [15] afin de représenter les changements d'états des variables du modèle de domaine et le flow de contrôle régissant le comportement du système, au fur et à mesure qu'il satisfait ses buts. Ceci permet une semi-automatisation de la génération du corps des évènements et améliore la validation de ces derniers, notamment lorsque des parties prenantes, non-experts des méthodes formelles, sont impliquées.

La spécification formelle générée et complétée peut être vérifiée et validée grâce aux outils associés à la méthode *B* [1], largement utilisés sur des projets industriels depuis plus de 25 ans [23].

La méthode SysML/KAOS, ainsi enrichie, est supportée par l'outil *Openflexo* [30] et a été évaluée sur différents cas d'étude [9, 8, 13].

La suite de cet article est structurée de la manière suivante. la Section 2 introduit brièvement les méthodes *Event-B* et *B System* ainsi que SysML/KAOS et son langage de modélisation des buts. Par la suite, la Section 3 décrit et illustre le langage de modélisation du domaine et les règles de traduction. Finalement, la Section 4 discute le travail effectué au regard des travaux connexes pertinents et des cas d'étude réalisés. Elle conclut également le papier en énonçant des perspectives potentielles.

2 Préliminaires

2.1 Event-B et B System

Event-B [2] est une méthode formelle de modélisation de systèmes critiques. Elle a été utilisée dans de nombreux projets industriels pour la construction incrémentale de systèmes et la vérification de propriétés [23]. Un modèle *Event-B* comprend une partie statique appelée *contexte* et une partie dynamique appelée *machine*. Le contexte contient la définition des ensembles abstraits et énumérés, des constantes et des propriétés. La machine quant-à-elle contient la définition des variables contraintes par des invariants et des évènements agissant sur l'état des variables. L'état initial des variables est défini par un évènement spécial appelé *évènement d'initialisation*. Un lien de raffinement, défini entre une machine dite *abstraite* et une autre dite *concrète*, permet à la machine concrète d'accéder aux définitions de la machine abstraite afin d'enrichir ou de concretiser la dynamique du système. De la même manière, un lien d'extension peut être défini entre deux contextes afin de permettre à l'un d'accéder aux définitions de l'autre dans le but de les étendre. Il est enfin possible de préciser, au sein d'une machine, un ensemble de contextes afin de permettre à la machine d'accéder aux éléments qui y sont définis. Des invariants dits *de collage*, définis au sein d'une machine, permettent de caractériser la relation entre les variables introduites au sein de cette dernière et celles introduites dans la machine qu'elle raffine. La cohérence d'un modèle *Event-B* est assurée par des obligations de preuve qui doivent être démontrées en se basant sur la logique du premier ordre et la théorie des ensembles [2].

B System désigne une variante syntaxique d'*Event-B* proposée au sein de l'environnement de développement intégré *AtelierB* édité par *ClearSy* [5], un partenaire industriel au sein du projet FORMOSE [3]. Une spécification *B System* est constituée de composants. Un composant *B System* peut être un système ou un raffinement (s'il raffine un autre composant). De plus, chaque composant peut servir tant à définir des éléments de la partie statique que des éléments de la partie dynamique.

2.2 La méthode SysML/KAOS et son langage de modélisation d'exigences

SysML/KAOS [21, 25] est une méthode formelle d'ingénierie des exigences qui associe la traçabilité offerte par *SysML* [20] à l'expressivité du langage de modélisation d'exigences de *KAOS* [22]. Elle permet la représentation des exigences fonctionnelles et non fonctionnelles d'un système ainsi que des attentes vis-à-vis de l'environnement sous forme d'hiérarchies de buts. Une exigence fonctionnelle décrit un comportement attendu du système, à l'occurrence d'une condition précise. Une exigence non-fonctionnelle désigne une propriété ou une caractérisation du système [17]. Elle permet de définir des contraintes sur la façon avec laquelle le système atteint ses objectifs. Parmi les opérateurs intervenant dans la hiérarchisation des buts, on distingue principalement l'opérateur ***AND*** (*ET*) et l'opérateur ***OR*** (*OU*). L'opérateur *ET* apparaît lorsque la condition nécessaire et suffisante, pour la réalisation d'un but, est la réalisation de chacun de ses sous-buts. Lorsque la condition nécessaire et suffisante pour la réalisation d'un but se limite à la réalisation de l'un de ses sous-buts, alors c'est l'opérateur *OU* qui apparaît. Dans le cadre de ce travail, nous considérons un troisième type de raffinement : *le raffinement de données*. Ce type de raffinement intervient lorsque des buts sont réexprimés, au sein d'un modèle concret, du fait du raffinement de certains éléments de la partie structurelle, en l'occurrence des variables.

Dans le cadre de ce travail, nous considérons un cas d'étude portant sur un protocole de communication nommé *SATURN* proposé et mis en place par *ClearSy*. SATURN décrit les échanges de trames de communication entre différents agents connectés via un bus réseau [33]. Il est défini de façon à garantir une robustesse et une disponibilité élevées. SATURN considère trois principaux types d'agents : les agents d'entrée, les agents de sortie et un agent de contrôle. Les agents d'entrée fournissent périodiquement des données booléennes à l'agent de contrôle qui les transforme et met le résultat à la disposition des agents de sortie.

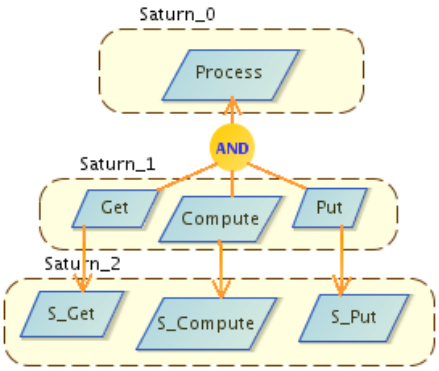


Figure 1: Extrait du diagramme des buts fonctionnels du protocole SATURN

La Figure 1 représente un extrait du diagramme des buts fonctionnels du protocole SATURN. L'objectif principal du protocole (but *Process*) est d'assurer la transformation des données fournies par les agents d'entrée (`in`) et de mettre le résultat (`out=FB(in)`) à la disposition des agents de sortie : `FB` désigne une fonction booléenne qui lie les données d'entrée aux données de sortie. Le but *Process* est raffiné en une conjonction de sous-buts : (i) le but *Get* pour l'acquisition des données d'entrée, (ii) le but *Compute* pour la transformation des données d'entrée en données de sortie et (iii) le but *Put* pour la mise à disposition des données de sortie.

Le deuxième niveau de raffinement présente un raffinement des données utilisées par les buts du premier niveau afin de tenir compte de la multiplicité des agents d'entrée et de sortie : chaque agent délivre ou récupère une donnée booléenne. Ainsi, l'acquisition des données d'entrée (but *S_Get*) consiste en une récupération asynchrone d'un tableau booléen (`s_in_l`) et la mise à disposition des résultats (but *S_Put*) consiste à délivrer un tableau booléen (`s_out_l`). La transformation des entrées en sorties (but *S_Compute*) consiste quant à elle à transformer un tableau booléen en un autre tableau booléen (`s_out_l = VFB(s_in_l)`).

2.3 Formalisation des modèles de buts SysML/KAOS

La formalisation des modèles de buts SysML/KAOS est décrite dans [26]. Les règles proposées permettent de générer un modèle *Event-B* dont la structure reflète la hiérarchie du modèle de buts : un composant est associé à chaque niveau de raffinement de la hiérarchie, ce composant définissant le squelette d'un évènement pour chaque but du niveau de raffinement. La sémantique des liens de raffinement entre buts est exprimée, au sein de la spécification formelle, à travers de nouvelles obligations de preuve, qui sont fonction des opérateurs de raffinement utilisés, et qui complémentent les obligations de preuve classiques de *préservation d'invariant* et de *faisabilité d'action* définies dans [2]. Par exemple, pour un but *G* se décomposant par l'opérateur *ET* en deux sous-buts G_1 et G_2, les obligations de preuve sont[1] :

- $G_1_Guard \Rightarrow G_Guard$: tout état où G_1 peut être déclenché doit être un état où G peut l'être également.
- $G_2_Guard \Rightarrow G_Guard$
- $(G_1_Post \wedge G_2_Post) \Rightarrow G_Post$: tout état où G_1 et G_2 sont satisfaits doit être un état où G l'est également.

Ainsi, en ce qui concerne le protocole SATURN, la spécification formelle issue du diagramme des buts de la Figure 1 définit les évènements :

Event *Process* $\widehat{=}$ **SELECT** ... **WHERE** $Process_Guard$ **THEN** $Process_Post$ **END**
Event *Get* ***ref_and*** *Process* $\widehat{=}$ **SELECT** ... **WHERE** Get_Guard **THEN** Get_Post **END**
Event *Compute* ***ref_and*** *Process* $\widehat{=}$ **SELECT** ... **WHERE** $Compute_Guard$ **THEN** $Compute_Post$ **END**
Event *Put* ***ref_and*** *Process* $\widehat{=}$ **SELECT** ... **WHERE** Put_Guard **THEN** Put_Post **END**

Le mot clé ***ref_and*** est utilisé afin de spécifier que les événements concrets *Get*, *Compute* et *Put* raffinent l'événement abstrait *Process* conformément aux raffinements SysML/KAOS, via l'opérateur *ET*. Ceci permet la génération automatique des obligations de preuve liées à l'utilisation de l'opérateur *ET*.

Néamoins, la spécification formelle obtenue est incomplète. Il est nécessaire de fournir manuellement la structure du système composée d'ensembles et constantes contraints par des propriétés, et de variables contraintes par des invariants. Il est aussi nécessaire de spécifier le corps de chaque évènement. L'objectif de notre étude est d'introduire un langage de modélisation de haut niveau qui complètera SysML/KAOS et permettra de représenter le domaine d'application du système afin de déduire automatiquement la spécification formelle correspondant à la structure du système et à l'initialisation des variables introduites.

3 Langage SysML/KAOS de modélisation du domaine

3.1 Présentation

Une **ontologie** peut se définir comme étant un modèle formel représentant des entités pouvant être regroupées en catégories à travers des relations de généralisation/spécialisation, leurs in-

[1] Pour un évènement *G*, *G_Guard* représente la garde de *G* et *G_Post* représente sa post-condition.

stances, leurs contraintes et propriétés ainsi que les relations existantes entre elles. Les ontologies permettent de représenter la connaissance d'un domaine, connaissance qui peut être exploitée par plusieurs systèmes.

De l'état de l'art réalisé sur les approches de modélisation du domaine en ingénierie des exigences, il ressort que les ontologies constituent une forme bien adaptée de représentation de la connaissance d'un domaine du fait principalement de leur expressivité, de leur aspect formel et de leur représentativité [6, 7, 29]. Ainsi, un langage de modélisation d'ontologies a été défini afin de permettre la représentation des modèles de domaine SysML/KAOS [10, 11]. Il est construit à partir d'*OWL* [32] et *PLIB* [31], deux formalismes de modélisation d'ontologies largement utilisés et complémentaires.

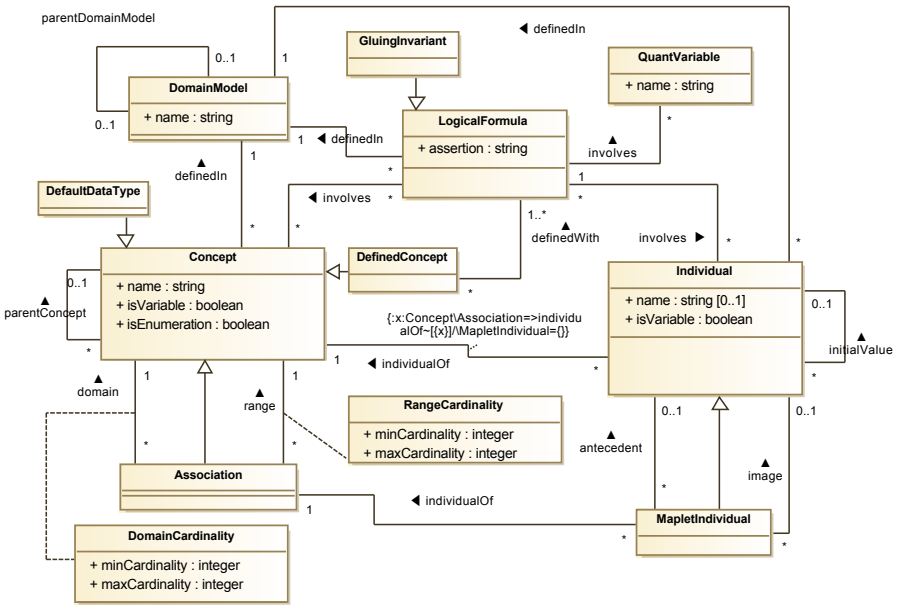


Figure 2: Extrait du métamodèle associé au langage SysML/KAOS de modélisation du domaine [10]

La Figure 2 présente un extrait du métamodèle associé au langage de modélisation du domaine [10]. Des contraintes additionnelles sont définies [10] afin de garantir que tout modèle de domaine puisse être traduit sans ambiguité en une spécification *B System*. Chaque modèle de domaine est donc défini formellement et correspond à un niveau de raffinement du modèle de buts. Un modèle de domaine *D1* est parent d'un modèle de domaine *D2* si le niveau du modèle de buts auquel *D2* est associé est constitué des buts qui raffinent ceux du niveau auquel *D1* est associé. Plusieurs éléments peuvent être définis au sein d'un modèle de domaine. Les *concepts* (instances de la classe Concept) représentent des ensembles d'individus ayant des caractéristiques communes. Un concept peut être déclaré *variable* lorsque l'ensemble de ses individus peut être mis à jour par ajout ou suppression d'individus. Dans le cas contraire, il est considéré *constant*. En outre, un concept peut être une *énumération* si tous ses individus sont définis au sein du modèle de domaine. De la même manière, les individus peuvent être variables ou constants. Un individu est dit variable s'il est introduit afin de représenter une variable d'état du système : son état peut changer sous l'action du système. Une *association* (instance de la classe Association) est un concept qui représente un lien d'un concept dit *domaine* vers un concept dit *range*. Un *couple* (instance de la classe MapletIndividual) représente quant à lui un lien d'un individu dit *antecedent* vers un autre dit *image*. Les *formules logiques* (instances de la classe LogicalFormula) définissent des contraintes sur des concepts ou individus à l'aide de la logique des prédicats et de la théorie des ensembles.

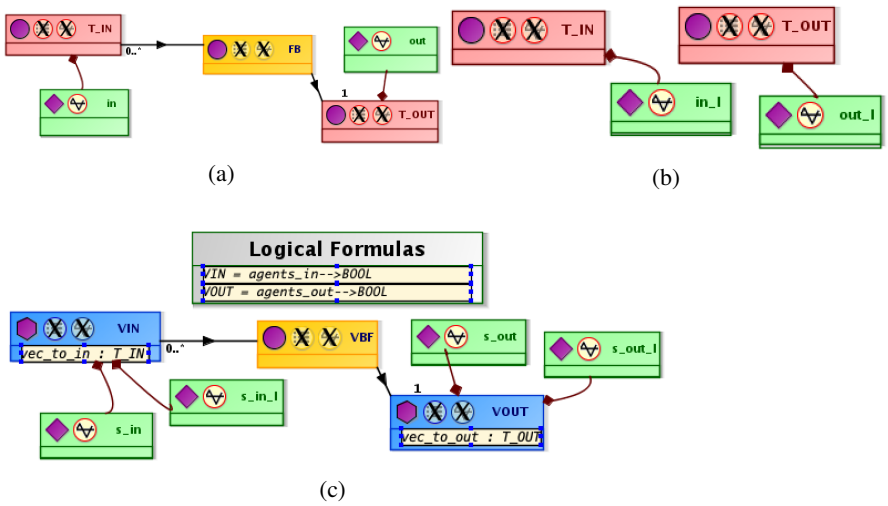


Figure 3: Modèles de domaine associés au diagramme des buts de la Figure 1

3.2 Illustration

La Figure 3 représente les modèles de domaine associés aux trois niveaux de raffinement du diagramme des buts de la Figure 1. La sous-figure (a), associée au niveau racine du diagramme des buts, définit les données d'entrée (`in`) et de sortie (`out`) et leurs types. Les données `in` et `out` sont définies comme des individus variables et leurs types comme des concepts. La fonction de transformation `FB` est quant à elle définie comme une association liant les entrées aux sorties. Le modèle de domaine associé au premier niveau de raffinement (sous-figure (b)) introduit les individus variables `in_l` et `out_l`. La donnée d'entrée `in` est transférée sur l'entrée `in_l` du contrôleur (but *Get*) avant d'être transformée (but *Compute*). Le résultat `out_l=FB(in_l)` est par la suite positionné sur la sortie `out` (but *Put*).

Au sein du modèle de la sous-figure (c), le concept `VIN`, défini comme un ensemble de tableaux booléens, représente les lectures des agents d'entrée. De même, le concept `VOUT` représente les écritures des agents de sortie. Ainsi, les variables d'entrée `in` et `in_l` sont raffinées comme des tableaux booléens `s_in` et `s_in_l`. Idem pour les variables de sortie. Les associations `vec_to_in` et `vec_to_out` assurent le collage entre variables abstraites et concrètes : $in = vec_to_in(s_in) \wedge in_l = vec_to_in(s_in_l) \wedge out = vec_to_out(s_out) \wedge out_l = vec_to_out(s_out_l)$ L'association `VBF` représente la fonction qui transforme l'entrée logique `s_in_l` en la sortie logique `s_out_l`.

3.3 Traduction des modèles de domaine en spécifications B System

La partie structurelle de la spécification *B System*, issue de la formalisation des modèles de buts SysML/KAOS, provient de la traduction des modèles de domaine. Les règles nécessaires à cette traduction sont décrites dans [10] et le principe de leur vérification formelle à travers *Event-B* est décrit dans [14]. Elles sont implémentées au sein de l'outil Openflexo qui permet la construction des modèles de domaine et de buts et la génération de la spécification *B System* correspondante [30]. Le tableau 1 décrit quelques règles de traduction choisies parmi les plus pertinentes. L'identification du niveau de raffinement du modèle de buts SysML/KAOS auquel est associé un modèle de domaine permet de déterminer le composant *B System* à considérer pour la définition des éléments issus de la traduction. Il est à noter que *b_x* désigne le résultat de la traduction d'un élément *x* du modèle de domaine en *B System*.

La spécification *B System* ci-dessous est issue de la traduction du modèle de domaine de la Figure 3 (a).

SYSTEM Saturn_0	**VARIABLES** b_in b_out
SETS b_T_IN b_T_OUT	**INVARIANT** $b_in \in b_T_IN \wedge b_out \in b_T_OUT$
CONSTANTS b_FB	**Event** *INITIALISATION* $\widehat{=}$ **then**
PROPERTIES $b_FB \in b_T_IN \rightarrow b_T_OUT$	$b_in :\in b_T_IN \mid\mid b_out :\in b_T_OUT$

Les concepts `T_IN` et `T_OUT` sont traduits comme des ensembles abstraits. Les individus `in` et `out` sont traduits comme des variables typées (clause **INVARIANT**) et initialisées. Finalement, l'association `FB` donne lieu à une fonction totale (clause **PROPERTIES**).

Table 1: Aperçu des règles de traduction [10]

		Modèle de domaine		B System	
		x	**Contrainte**	**b_x**	**Contrainte**
1	**Concept abstrait qui n'est pas une énumération**	CO	$CO \in$ Concept\(Association∩DefinedConcept∩ DefaultDataType) $CO \notin dom(parentConcept)$ $isEnumeration(CO) = FALSE$	b_CO	$b_CO \in$ AbstractSet
2	**Sous-concept constant d'un concept variable**	CO PCO PPCO	$\{CO, PCO, PPCO\} \subseteq$ Concept $Concept_isVariable(CO) = FALSE$ $parentConcept(CO) = PCO$ $b_PCO \in$ Variable $\wedge\, b_PPCO \in$ Set $\cup$ Constant $PPCO \in (closure1(parentConcept))[\{PCO\}]$[2]	b_CO	$b_CO \in$ Constant **Propriété :** $b_CO \subseteq b_PPCO$ **Invariant :** $b_CO \subseteq b_PCO$
3	**Individu constant d'un concept variable**	Ind CO PPCO	$Ind \in$ Individual \ MapletIndividual $Individual_isVariable(Ind) = FALSE$ $CO = individualOf(Ind) \wedge b_CO \in$ Variable $PPCO \in$ Concept $\wedge\, b_Ind \in$ Constant $PPCO \in (closure1(parentConcept))[\{CO\}]$ $b_PPCO \in$ Set $\cup$ Constant	b_Ind	**Property :** $b_Ind \in b_PPCO$ **Invariant :** $b_Ind \in b_CO$

3.4 Modélisation de la dynamique des variables du modèle de domaine

Une fois qu'un élément du domaine est identifié comme étant une variable, il est possible de décrire ses changements d'états, au fur et à mesure que le système satisfait ses buts, à l'aide des diagrammes d'états-transitions algébriques (ASTDs) [15] : un ASTD par niveau de raffinement du modèle des buts. Les ASTDs constituent une notation graphique formellement définie permettant de spécifier les traces d'actions acceptées par un système. Ils combinent l'expressivité graphique des diagrammes d'états [19] et le pouvoir d'abstraction des opérateurs de l'algèbre de processus [16]. Construits pour représenter les changements d'états des variables et le flow de contrôle régissant le comportement du système, les ASTDs peuvent être traduits en spécifications *B System* [28] afin de compléter la spécification formelle issue de la traduction des modèles de buts et de domaine SysML/KAOS : définition du corps des évènements.

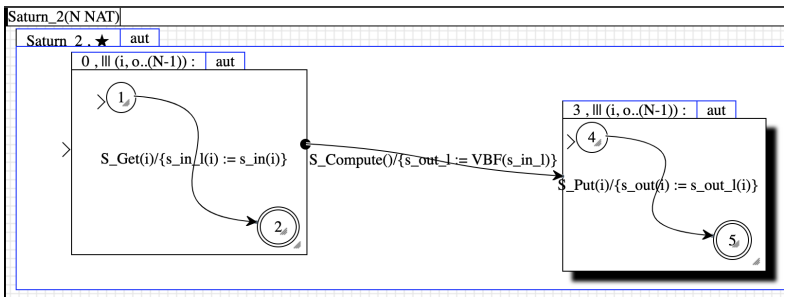


Figure 4: Aperçu de l'ASTD du deuxième niveau de raffinement du diagramme des buts de la Figure 1

[2] *closure1(parentConcept)* désigne la fermeture transitive de la relation parentConcept

La Figure 4 représente l'ASTD associé au deuxième niveau de raffinement du diagramme des buts de la Figure 1 auquel est associé le modèle de domaine de la Figure 3 (c). Le nombre d'agents d'entrée/sortie `N` est passé en paramètre. Un nombre indéterminé de fois (ASTD fermeture de Kleene [*Saturn_2*, ★]), les données d'entrée représentées par la variable `s_in` sont positionnées sur l'entrée `s_in_l` de l'agent contrôleur. Ce transfert, effectué par les agents d'entrée indépendamment les uns des autres, est exprimé par le sous-ASTD *0* qui est un entrelacement quantifié sur la variable `i` : chaque agent identifié par une certaine valeur de `i` effectue le transfert de manière à satisfaire le but *S_Get(i)*. Le but *S_Compute* est quant à lui satisfait lorsque chaque agent d'entrée `i` a satisfait le but *S_Get(i)* et lorsque la sortie du contrôleur `s_out_l` est positionnée au résultat de l'évaluation $VBF(s_in_l)$. L'écriture du résultat intervient alors en suivant la même logique que le positionnement de l'entrée (sous-ASTD *3*).

4 Discussion, conclusion et travaux futurs

La méthode KAOS [22] permet de modéliser le vocabulaire du domaine d'un système à travers un modèle objet représenté sous forme de diagrammes de classes UML. Toutefois, cette approche combine, au sein d'un même modèle, les éléments caractérisant le domaine (entités, associations) et ceux caractérisant la dynamique et les objectifs du système (agents, évènements). Dans [25], en plus des diagrammes de classes UML, la modélisation du domaine fait intervenir les diagrammes d'objets UML et les ontologies. Cette approche, de même que celle proposée pour KAOS dans [22], hérite de la sémi-formalité des diagrammes UML [27]. De plus, elles ne permettent pas, ou permettent à peine, de bénéficier de l'expressivité de la logique du premier ordre et de la théorie des ensembles pour la représentation des contraintes du domaine. Par ailleurs, elles ne permettent pas de représenter les entités et leurs instances au sein d'un même modèle, ce qui rend difficile l'expression et la vérification des contraintes. L'approche proposée dans [25] introduit une difficulté supplémentaire résidant dans l'utilisation de trois langages différents pour la caractérisation du domaine. Dans [4, 6], les modèles de domaine servent à définir un langage spécifique pour l'expression des exigences, ce qui rend les modèles interdépendants et difficilement maintenables. À l'opposé, notre approche assure une indépendance maximale entre modèles de domaine et modèles de buts SysML/KAOS, les deux modélisations devant juste suivre la même logique de raffinement.

Le langage présenté tout au long de cet article permet de représenter le domaine d'un système dont les exigences sont capturées à travers le langage de modélisation des buts de SysML/KAOS. Il s'agit, à notre connaissance, du premier langage de modélisation du domaine qui permet de distinguer les éléments statiques des éléments dynamiques, les changements d'états des éléments dynamiques pouvant être exprimés au travers d'ASTDs. Son métamodèle et les règles nécessaires pour la transformation de tout modèle de domaine en spécification *B System* ont été spécifiés et vérifiés formellement en utilisant la méthode Event-B. Couplé à la modélisation des buts SysML/KAOS, le langage a été évalué sur plusieurs études de cas : spécification du protocole de transport ferrovier *hybrid ERTMS/ETCS level 3* [9], spécification du système de contrôle d'une chaudière à vapeur [8], spécification d'un système de gestion du transport routier pour le compte de la *Ville de Montréal*, etc [13]. La méthode ainsi définie se révèle cohérente, robuste et scalable. Elle accentue la réutilisabilité, la lisibilité et la maintenabilité des modèles, grâce au raffinement et à la décomposition, et facilite la validation des exigences spécifiées formellement par des parties prenantes non spécialistes de méthodes formelles. Toutefois, les tâches de spécification des formules logiques et du corps des évènements et de vérification et validation formelles nécessitent non seulement du temps, mais surtout l'implication d'experts en méthodes formelles. Il s'agit là du prix à payer pour des exigences formellement correctes.

En perspective, nous nous intéressons au support de la propagation des mises à jour effectuées au sein d'une spécification *B System* vers les modèles SysML/KAOS correspondants.

Acknowledgment

Ce travail est supporté par le projet *FORMOSE* [3] financé par l'Agence Nationale de la Recherche française (ANR). Il est également partiellement supporté par le Conseil de Recherches en Sciences Naturelles et en Génie du Canada (CRSNG).

References

[1] Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (2005)

[2] Abrial, J.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)

[3] ANR-14-CE28-0009: Formose ANR project (2017), http://formose.lacl.fr/

[4] Carreira, P.J., Costa, M.E.: Automatically verifying an object oriented specification of the steam-boiler system. In: FMICS. pp. 345–360 (2000)

[5] ClearSy: Atelier B: B System (2014), http://clearsy.com/

[6] Crapo, A.W., Moitra, A., McMillan, C., Russell, D.: Requirements capture and analysis in ASSERT(TM). In: RE 2017. pp. 283–291. IEEE Computer Society (2017)

[7] Dermeval, D., Vilela, J., Bittencourt, I.I., Castro, J., Isotani, S., da S. Brito, P.H., Silva, A.: Applications of ontologies in requirements engineering: a systematic review of the literature. Requir. Eng. **21**(4), 405–437 (2016)

[8] Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Back propagating B system updates on SysML/KAOS domain models. In: ICECCS 2018. pp. 160–169. IEEE (2018)

[9] Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A.: Modeling the hybrid ERTMS/ETCS level 3 standard using a formal requirements engineering approach. In: ABZ. LNCS, vol. 10817, pp. 262–276. Springer (2018)

[10] Fotso, S.J.T., Frappier, M., Laleau, R., Mammar, A., Barradas, H.R.: The generic SysML/KAOS domain metamodel. ArXiv e-prints, cs.SE, 1811.04732 (2019), https://arxiv.org/pdf/1811.04732.pdf

[11] Fotso, S.J.T., Laleau, R., Mammar, A., Frappier, M.: Towards using ontologies for domain modeling within the SysML/KAOS approach. In: RE Workshops. pp. 1–5. IEEE Computer Society (2017)

[12] Fotso, S.J.T., Laleau, R., Frappier, M., Mammar, A.: Event-B specification of translation rules from ontology-based domain models to B System specifications (2019), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/blob/master/SysMLKAOSNewDomainModelRulesFormalisation_20190330.zip

[13] Fotso, S.J.T., Laleau, R., Mammar, A., Frappier, M.: The SysML/KAOS Domain Modeling Language (Tool and Case Studies) (2017), https://github.com/stuenofotso/SysML_KAOS_Domain_Model_Parser/tree/master

[14] Fotso, S.J.T., Mammar, A., Laleau, R., Frappier, M.: Event-B expression and verification of translation rules between SysML/KAOS domain models and B System specifications. In: ABZ. LNCS, vol. 10817, pp. 55–70. Springer (2018)

[15] Frappier, M., Gervais, F., Laleau, R., Fraikin, B., St.-Denis, R.: Extending statecharts with process algebra operators. ISSE **4**(3), 285–292 (2008)

[16] Frappier, M., St.-Denis, R.: EB 3: an entity-based black-box specification method for information systems. Software and System Modeling **2**(2), 134–149 (2003)

[17] Gnaho, C., Semmak, F.: Une extension SysML pour l'ingénierie des exigences non fonctionnelles orientée but. Ingénierie des Systèmes d'Information **16**(1), 9–32 (2011)

[18] Gnaho, C., Semmak, F., Laleau, R.: Modeling the impact of non-functional requirements on functional requirements. LNCS, vol. 8697, pp. 59–67. Springer (2014)

[19] Harel, D.: Statecharts: A visual formalism for complex systems. Sci. Comput. Program. **8**(3), 231–274 (1987)

[20] Hause, M., et al.: The SysML modelling language. In: Fifteenth European Systems Engineering Conference. vol. 9. Citeseer (2006)

[21] Laleau, R., Semmak, F., Matoussi, A., Petit, D., Hammad, A., Tatibouet, B.: A first attempt to combine SysML requirements diagrams and B. Innovations in Systems and Software Engineering **6**(1-2), 47–54 (2010)

[22] van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)

[23] Lecomte, T., Déharbe, D., Prun, É., Mottin, E.: Applying a formal method in industry: A 25-year trajectory. In: SBMF. LNCS, vol. 10623, pp. 70–87. Springer (2017)

[24] Lee, D.G., Suh, N.P.: Axiomatic design and fabrication of composite structures-applications in robots, machine tools, and automobiles. Oxford University Press p. 732 (2005)

[25] Mammar, A., Laleau, R.: On the use of domain and system knowledge modeling in goal-based Event-B specifications. In: ISoLA 2016, LNCS. pp. 325–339. Springer

[26] Matoussi, A., Gervais, F., Laleau, R.: A goal-based approach to guide the design of an abstract Event-B specification. In: ICECCS 2011. pp. 139–148. IEEE Computer Society

[27] McUmber, W.E., Cheng, B.H.C.: A general framework for formalizing UML with formal languages. In: ICSE 2001. pp. 433–442. IEEE Computer Society (2001)

[28] Milhau, J., Frappier, M., Gervais, F., Laleau, R.: Systematic translation rules from ASTD to Event-B. In: IFM. Lecture Notes in Computer Science, vol. 6396, pp. 245–259. Springer (2010)

[29] Nguyen, T.H., Vo, B.Q., Lumpe, M., Grundy, J.: KBRE: a framework for knowledge-based requirements engineering. Software Quality Journal **22**(1), 87–119 (2014)

[30] Openflexo: Openflexo project (2019), `https://www.openflexo.org/`

[31] Pierra, G.: The PLIB ontology-based approach to data integration. In: IFIP 18th World Computer Congress. IFIP, vol. 156, pp. 13–18. Kluwer/Springer (2004)

[32] Sengupta, K., Hitzler, P.: Web ontology language (OWL). In: Encyclopedia of Social Network Analysis and Mining, pp. 2374–2378 (2014)

[33] Technology, R.: SATURN: SIL2-Certified Fail-safe Remote I/O System Architecture for Trains (2015), `https://www.railway-technology.com/contractors/computer/leroy-automation/pressreleases/presssaturn-certified-fail-safe/`

Intégration d'outils tiers de preuve automatique dans Atelier B *

Lilian Burdy, David Déharbe, Ronan Saillard (CLEARSY, France)

1 Introduction

Atelier B est un environnement de développement de composants logiciels mettant en œuvre la méthode B [1]. Atelier B permet également de réaliser des modélisations système en B événementiel [2]. Atelier B est utilisé industriellement principalement pour le développement de composants pour des systèmes critiques, notamment dans le domaine ferroviaire.

Dans une perspective industrielle, l'utilisation de la méthode B dans un développement de logiciel sécuritaire permet notamment de remplacer le développement de tests unitaires par de la preuve. En effet, le développement au moyen de cette méthode garantit que le code généré pour un composant est une implémentation correcte de la spécification de ce composant, à la condition qu'un certain nombre d'obligations de preuve, produites lors de la conception de l'implémentation, soient vérifiées. Une obligation de preuve est une expression mathématique, exprimée en logique du premier ordre (avec arithmétique entière et ensembles), qu'il faut démontrer comme étant valide.

Atelier B est doté d'un outil de preuve qui lui est propre. Utilisable en mode automatique ou interactif, ce prouveur a été développé dans le « langage des théories », un langage de programmation logique conçu pour le développement d'Atelier B. Le moteur de preuve, qui est paramétrable par un jeu de règles de preuve, a été certifié, ainsi qu'un jeu de base de règles de preuve, pour un usage dans le domaine ferroviaire. Depuis cette certification, cet outillage est resté essentiellement figé. Les évolutions se font donc en apportant de nouvelles règles de preuve, en fonction des besoins de chaque projet industriel. Ces règles de preuve sont validées au cas par cas, de manière manuelle, ou outillée [8].

Néanmoins, ces contraintes opérationnelles ont pour l'essentiel figé le développement du moteur de preuve, qui ne peut donc pas profiter directement des progrès scientifiques et techniques réalisés dans le domaine de la démonstration automatique de théorèmes.

Pour remédier à cette situation, plusieurs approches ont récemment été mises en œuvre. Chronologiquement, celles-si sont :

*Ces travaux ont été financés en partie par les projets FUI-LCHIP et ANR-DISCONT.

1. Réalisation d'un générateur d'obligations de preuve ad hoc vers un langage d'entrée de preuve ;
2. Utilisation d'outils tiers pour générer des règles de preuve à la demande ;
3. Mise en place d'une plateforme générique d'intégration d'outils de preuve automatique.

Ces différentes approches sont présentées dans les sections qui suivent.

2 Génération ad hoc d'obligations de preuve

L'intérêt d'utiliser des outils de preuve généralistes pour traiter des obligations de preuve issues des méthodes formelles a déjà été largement démontrée (cf. [6], par exemple). Dans le cadre du projet BWare [5], a donc été mis en œuvre un nouveau générateur d'obligations de preuve dans Atelier B. La cible choisie fut le langage d'entrée de Why3, une plateforme de vérification de programmes multi-prouvers. En terme d'outillage, les résultats du projet sont les suivants:

bxml Un format structuré (XML), ouvert, pour représenter les composants du langage B, ainsi qu'un outil pour produire des fichiers bxml à partir des fichiers sources.

pog Un format structuré (XML), ouvert, pour représenter les obligations de preuve, ainsi qu'un outil pour produire des fichiers bxml à partir des fichiers bxml.

why Un convertisseur du format pog vers le format d'entrée de Why3 ainsi qu'un préambule Why3 formalisant les opérateurs d'expressions du langage B.

CLEARSY a mis à disposition des partenaires du projet BWare des ensembles d'obligations de preuve issus de divers projets. Sur cette base, les concepteurs des outils de preuve ont pu améliorer significativement le taux de preuve réussie et ainsi valider la démarche d'appliquer des outils généralistes dans le cadre de la méthode B [4].

Par la suite, CLEARSY a développé une interface graphique dédiée à l'interfaçage avec cette infrastructure de preuve appelée iapa [3] (interface d'appel des prouveurs automatiques). En plus des fonctionnalités de conversion et de gestion de statut des obligations de preuve, iapa offre des commandes permettant de définir des stratégies de sélection d'hypothèses qui sont pertinentes à la vérification du but courant. L'interface iapa est intégrée dans la version 4.5 d'Atelier B[1].

[1] https://www.atelierb.eu/outil-atelier-b/atelier-b-4/

3 Génération à la volée de règles de preuve

Cette démarche consiste à utiliser des outils tiers pour générer des règles de preuve à la demande, en cours de session de preuve interactive. En effet, l'utilisateur du prouveur interactif d'Atelier B doit souvent guider celui-ci pour réaliser des étapes de raisonnement qui lui paraissent simples (raisonnement purement booléen ou arithmétique, par exemple), alors qu'il s'attendrait à ne devoir intervenir que pour des étapes décisives du raisonnement (introduction d'une preuve par cas, choix d'une instance, etc.) Cette caractéristique se retrouve dans différents prouveurs interactifs, et l'utilisation systématique de prouveurs automatiques a été explorée par ailleurs (cf. Sledgehammer [10], par exemple).

Ce comportement du prouveur interactif est dû au fait que le moteur ne fait pas de distinction entre les différents niveaux de raisonnement et peut commencer à appliquer systématiquement des règles qui ne sont pas pertinentes pour la preuve du but courant, puis diverger.

Pour offrir une alternative, il a donc été mis en place une nouvelle commande dans l'interface graphique qui permet de guider le prouveur en lui fournissant une règle qui s'applique directement au but. Une telle règle peut être construite en fournissant le but courant et l'ensemble des hypothèses à un outil tiers explicatif, par exemple, un solveur SMT. On utilise alors la fonctionnalité explicative pour extraire uniquement les hypothèses pertinentes à la démonstration du but courant, et construire automatiquement une règle de preuve à partir de ce sous-ensemble d'hypothèses [9].

Une fois une telle règle créée, il est bien sûr nécessaire de vérifier qu'elle est valide. Atelier B est outillé pour gérer la vérification des règles de preuve. La vérification peut être conduite par un outil de vérification automatique, appelé `pp`, qui est fondé sur la preuve par tableaux pour la logique du premier ordre. Dans les cas où `pp` échoue (car le problème traité n'est évidemment pas décidable), cet outillage permet aussi d'associer à une règle de preuve la description textuelle d'une preuve manuelle. L'outillage exige une contre-vérification de cette description textuelle, laquelle doit être réalisée par une personne tierce.

Une première version de cette fonctionnalité est disponible dans la version 4.5 d'Atelier B, permettant de générer des règles valides en logique du premier ordre pur et en arithmétique entière. Elle a depuis été étendue pour produire des règles valides en arithmétique réelle.

4 Les mécanismes de preuve automatique

Ce troisième axe de développement a pour but de créer une infrastructure générique et ouverte d'extension de la preuve automatique dans Atelier B par ajout de greffons. Il s'agit donc de permettre, à toute partie intéressée, d'ajouter à Atelier B un nouvel outil de preuve automatique. La conception de cette fonctionnalité repose sur le concept que nous avons appelé « mécanisme de preuve ».

Un *mécanisme de preuve* est un moyen d'appliquer un ou plusieurs prouveurs automatiques aux obligations de preuve d'un projet et est une liste de pilotes de prouveurs. Un *pilote de prouveur* est un moyen d'appliquer un prouveur automatique à une ou plusieurs obligations de preuve. Un pilote est composé de trois outils:

writer traduit les fichiers au format pog (ensemble des obligations de preuve d'un composant) dans un format d'entrée du prouveur ciblé;

prover le prouveur ciblé;

reader interprète la sortie du prouveur et détermine le résultat.

Tout outil, que ce soit un prouveur, ou un traducteur, est bien sûr susceptible de produire des résultats faussés suite à une erreur introduite dans le développement. Si les outils de preuve d'Atelier B ont été qualifiés dans le cadre d'un projet industriel, peut-on vraiment considérer comme fiables les résultats d'outils tiers ?

La réponse est que, a priori, ces résultats ne sont pas fiables. Par contre, il est plus probable, que si une obligation de preuve a été démontrée valide par une chaîne d'outils tiers, elle le soit vraiment, que dans le cas contraire. Dans une démarche industrielle, ces résultats peuvent donc être utilisés pour établir une priorité parmi les obligations de preuve non prouvées devant être examinées et traitées. Pour permettre la mise en place ce processus, est introduit le statut *probablement vrai* (*unreliably proved*) parmi les statuts possibles d'une obligation de preuve.

Toutefois, l'infrastructure mise en place n'écarte pas la possibilité que les résultats d'un mécanisme de preuve soient considérés fiables, par exemple :

1. un mécanisme de preuve est compose de deux chaînes d'outils indépendantes, et que les deux chaînes d'outils concluent à la validité de l'obligation de preuve;

2. les outils du mécanisme de preuve ont été qualifiés dans le cadre du projet où ils sont utilisés;

3. le prouveur externe produit un résultat qui permet de reconstruire une preuve interprétable par les prouveurs de l'Atelier B (ce qui peut être vu comme un cas particulier de la situation précédente.

L'aspect de la qualification des logiciels externes est considérée de la manière suivante. Une obligation de preuve peut être considérée comme étant

- non démontrée (valeur initiale);
- vraie;
- fausse;
- probablement vraie.

Un mécanisme de preuve possède un attribut, dit de *fiabilité*, qui peut avoir trois valeurs:

- fiable (*trusty*);
- peu fiable (*untrusty*);
- fiable par redondance (*redundancy-based*).

Un mécanisme classé « fiable par redondance » doit posséder au moins deux pilotes pour se distinguer d'un mécanisme classé « peu fiable ». Lors de l'utilisation d'un mécanisme de preuve dans un projet industriel, les acteurs du projet peuvent classer la fiabilité du mécanisme en fonction des contraintes normatives sécuritaires pour ce projet.

On définit alors que le statut d'une obligation de preuve comme :

fausse si un pilote d'un mécanisme a prouvé sa non-validité ;

vraie si sa validité a été démontrée par un pilote d'un mécanisme « fiable », ou par deux pilotes d'un mécanisme « fiable par redondance ».

probablement vraie si sa validité a été démontrée par un seul pilote d'un mécanisme « par redondance » ou par un pilote d'un mécanisme « peu fiable ».

non prouvée dans les autres cas.

L'outillage des mécanismes de preuve est en cours de développement. Les composants suivants sont d'ores et déjà disponibles :

- Format structuré (XML) de description des mécanismes de preuve ;
- Traducteur, lecteur et pilote pour le prouveur Alt-Ergo [7].
- Outil d'interprétation d'un mécanisme de preuve
- Extension de l'interface en ligne de commandes d'Atelier B (programme bbatch) pour a) gérer les mécanismes de preuve d'un projet et b) appliquer des mécanismes de preuve à des composants B.

Sont en cours de développement l'extension de l'interface graphique d'Atelier B pour intégrer les nouvelles fonctionnalités liées aux mécanismes de preuve, ainsi que des pilotes vers les prouveurs utilisant le format SMT.

5 Conclusion et perspectives

Nous avons ici exposé les développements récents visant à intégrer à Atelier B des fonctionnalités pour bénéficier des progrès dans le domaine de la démonstration automatique de théorèmes. Une partie de ces développements est d'ores et déjà intégrée à la dernière version d'Atelier B.

Pour ouvrir à la communauté la possibilité de contribuer à cet effort, les différents formats de fichier seront publiés et un ensemble d'environ 200.000 obligations de preuve issues de différents projets internes sera disponibilisée. Ces ressources permettront d'une part aux auteurs d'outils de preuve de mesurer et de comparer leur performance sur des projets industriels, et d'autre part aux utilisateurs de la méthode B de bénéficier d'outils de preuve dans l'état de l'art.

References

[1] J.-R. Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 2005.

[2] J.-R. Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[3] L. Burdy, D. Déharbe, and É. Prun. Interfacing automatic proof agents in atelier B: introducing "iapa". In *Proc 3rd Workshop on Formal Integrated Development Environment.*, pages 82–90, 2016.

[4] S. Conchon and M. Iguernelala. Increasing proofs automation rate of Atelier-B thanks to Alt-Ergo. In T. Lecomte, R. Pinger, and A. B. Romanovsky, editors, *1st Int'l Conf. Reliability, Safety, and Security of Railway Systems (RSSRail'2016)*, volume 9707 of *LNCS*, pages 243–253. Springer, 2016.

[5] D. Delahaye, C. Dubois, C. Marché, and D. Mentré. The BWare project: Building a proof platform for the automated verification of B proof obligations. In *4th Int'l Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ'2014)*, 2014.

[6] D. Déharbe. Integration of SMT-solvers in B and Event-B development environments. *Science of Computer Programming*, 78(3), 2013.

[7] M. Iguernelala. *Strengthening the Heart of an SMT-Solver: Design and Implementation of Efficient Decision Procedures*. Thèse de doctorat, Université Paris-Sud, June 2013.

[8] M. Jacquel. *Automatisation des preuves pour la vérification des règles de l'Atelier B. (Proof Automation for Atelier B Rules Verification)*. PhD thesis, Conservatoire national des arts et métiers, Paris, France, 2013.

[9] T. Lecomte, D. Déharbe, É. Prun, and E. Mottin. Applying a formal method in industry: A 25-year trajectory. In *20th Brazilian Symp. Formal Methods: Foundations and Applications (SBMF'2017)*, 2017.

[10] L. C. Paulson and J. C. Blanchette. Three years of experience with sledgehammer, a practical link between automatic and interactive theorem provers. In *The 8th Int'l Work. Implementation of Logics, (IWIL'2010)*, 2010.