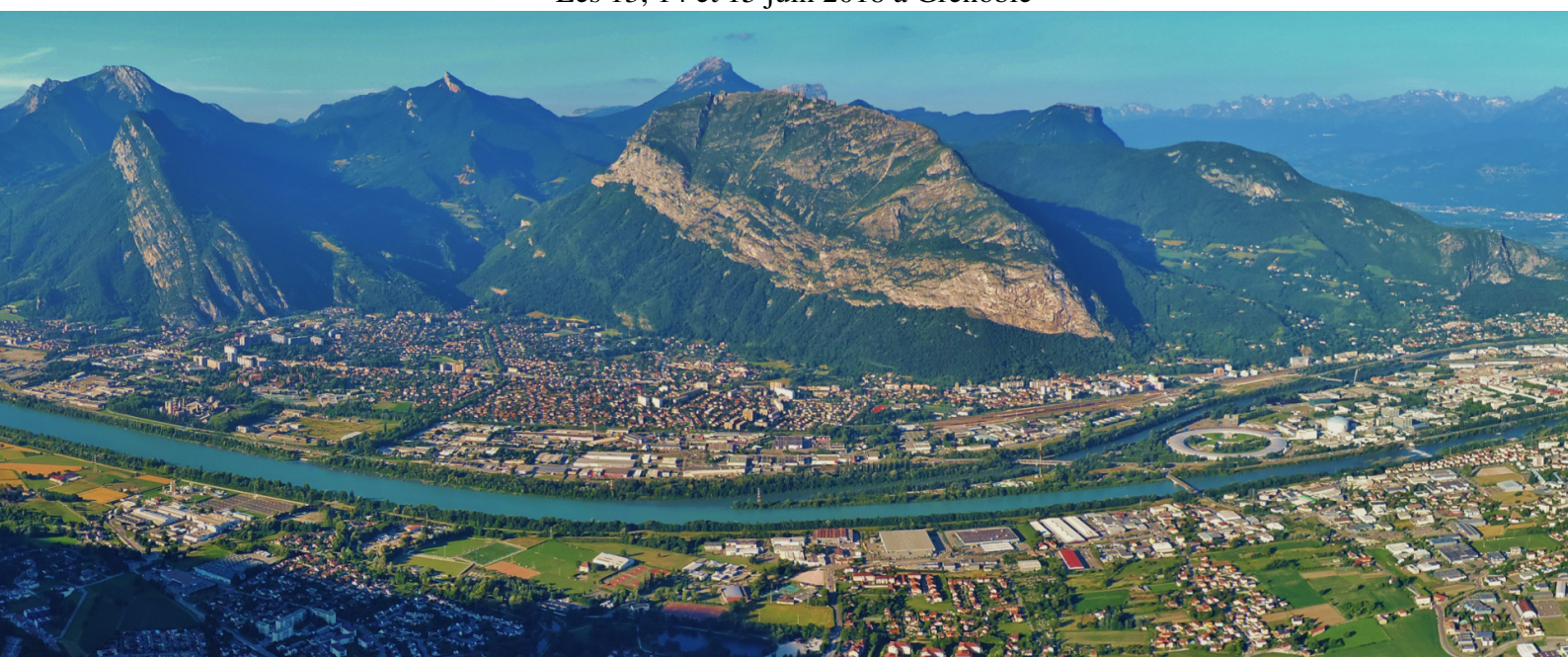
Actes des 17èmes journées sur les

Approches Formelles dans l'Assistance au Développement de Logiciels

Édités par Antoine Rollet et Arnaud Lanoix

Les 13, 14 et 15 juin 2018 à Grenoble



Préface

L'atelier francophone AFADL sur les Approches Formelles dans l'Assistance au Développement de Logiciels, se tiendra pour sa 17ème édition les 13, 14 et 15 juin 2018 à Grenoble. Cette année encore, cet atelier est organisé conjointement avec les journées du GDR-GPL, la Conférence en IngénieriE du Logiciel (CIEL) et la Conférence francophone sur les Architectures Logicielles (CAL).

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... ou toute autre approche formelle permettant de développer des applications sûres, allant de la spécification 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, vingt 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
— 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 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. Sans toutes ces personnes, AFADL ne pourrait pas exister.

Nous remercions les membres du comité d'organisation des journées du GDR-GPL 2018 qui ont pris en charge tous les aspects logistiques.

Le 15 Mai 2018,

Antoine Rollet et Arnaud Lanoix
Présidents du comité de programme AFADL 2018.

Comité de programme

Présidents du comité de programme

— Antoine Rollet (LaBRI, Bordeaux INP)
— Arnaud Lanoix (LS2N, Nantes)

Organisateurs locaux

— Yves Ledru (LIG, Grenoble)
— Sophie Dupuy-Chessa (LIG, Grenoble)

Membres du comité de programme

— Yamine Ait Ameur (IRIT - INPT-ENSEEIHT)
— Sandrine Blazy (University of Rennes 1 - IRISA)
— Pierre Casteran (Université de Bordeaux)
— Frédéric Dadeau (FEMTO-ST)
— David Deharbe (ClearSy System Engineering)
— Lydie Du Bousquet (LIG - Grenoble)
— Catherine Dubois (ENSIIE-Samovar)
— Christele Faure (SafeRiver)
— Alain Giorgetti (FEMTO-ST)
— Aurélie Hurault (IRIT - ENSEEIHT)
— Akram Idani (LIG - Grenoble)
— Florent Kirchner (CEA LIST)
— Nikolai Kosmatov (CEA List)
— Régine Laleau (Paris Est Creteil University)
— Pascale Le Gall (CentraleSupelec)
— Yves Ledru (LIG - 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)
— Vlad Rusu (INRIA)
— Nicolas Stouls (INSA Lyon, CITI)
— Safouan Taha (CentraleSupelec)
— Sylvie Vignes (ENST)
— Laurent Voisin (Systerel)
— Virginie Wiels (ONERA / DTIM)
— Fatiha Zaidi (Univ. Paris-Sud)

Relecteurs additionnels

— Oscar Carrillo (CPE)
— Sylvain Conchon (Univ. Paris-Sud, LRI)

Sommaire

Sessions AFADL

Session MTV2 et AFADL

Session MFDL et AFADL

Session LTP et AFADL

Une méthode dirigée par les modèles pour la construction rapide de services web

David Sferruzza[1,3], Jérôme Rocheteau[1,2], Christian Attiogbé[1], et Arnaud Lanoix[1]

[1]LS2N - UMR CNRS 6004 / F-44322 Nantes Cedex 3, France
[2]ICAM / 35, avenue du Champ de Manœuvres, 44470 Carquefou, France
[3]Startup Palace / 18, rue Scribe, 44000 Nantes, France

Mots-clés : ingénierie logicielle, applications web, services web, ingénierie dirigée par les modèles, vérification formelle.

Dans l'article *« A Model-Driven Method for Fast Building Consistent Web Services in Practice »* [3] publié à la conférence internationale *MODELSWARD 2018* nous présentons une méthode outillée de construction de services web.

De nombreuses sociétés œuvrant dans le logiciel dépendent des technologies web pour tester des hypothèses de marché et développer des entreprises viables. Elles ont souvent besoin de construire rapidement des services web qui sont au cœur de leurs « Minimum Viable Products » (MVPs). Les services web sont des applications réparties conçues pour permettre des interactions de machine à machine au sein d'un réseau, via le protocole HTTP. Les MVPs doivent être fiables et sont basés sur des spécifications et des hypothèses qui ont de fortes chances de changer. Des approches basées sur l'Ingénierie Dirigée par les Modèles (IDM) ont déjà été proposées et utilisées pour développer et faire évoluer des services web [1]. Cependant, ces approches ne sont pas appropriées pour *(i)* le prototypage rapide, *(ii)* la vérification de modèle et *(iii)* la compatibilité avec les langages de programmation classiques.

Nous introduisons un méta-modèle de services web, volontairement simple et poursuivant deux objectifs : *(i)* donner aux développeurs des abstractions pour écrire du code ré-utilisable tout en leur offrant une bonne flexibilité et *(ii)* permettre l'utilisation d'outils offrant du support aux développeurs, comme de la vérification de cohérence par exemple. Ce méta-modèle comprend trois principales parties : des *entités* représentant des structures de données, des *composants* représentant des unités de traitement et des *services* faisant l'interface entre les composants et le protocole HTTP.

Les composants peuvent être *atomiques* ou *composites*. Dans le premier cas, ils sont notamment définis par leur contrat c'est à dire des préconditions sur leur contexte d'exécution (qui indiquent les variables devant être présentes pour que le composant puisse s'exécuter) et la description des effets que l'exécution du composant aura sur ce même contexte (ajout ou suppression de variables). Ils doivent

également être accompagnés d'une implémentation du comportement du composant dans un langage de programmation. Dans le second cas, ils sont définis en terme d'autres composants ; ces sous-composants seront exécutés séquentiellement lorsque le composant composite est exécuté. Tous les composants peuvent déclarer des paramètres dont les valeurs (toutes résolvables à l'étape de la conception) sont accessibles par les implémentations des composants atomiques.

Nous proposons également des règles de vérification de la cohérence des modèles. Le processus de vérification est rigoureux mais superficiel, pour permettre le prototypage rapide par des développeurs qui ne sont pas des experts des méthodes formelles, tout en offrant des garanties au moment de la conception qui permettent d'améliorer la qualité du produit et l'efficacité du développement. Par exemple, il est utile de pouvoir vérifier que les différents assemblages de composants présents dans un modèle sont bien construits et que chaque composant est instancié dans un contexte qui vérifie ses préconditions.

Pour mettre en œuvre ces apports théoriques, nous présentons une méthode de développement et SWSG [2], un outil qui automatise la vérification d'un modèle et génère du code pour obtenir une implémentation fonctionnelle des services web. De part sa nature de prototype, SWSG impose l'utilisation du langage PHP et du framework Laravel pour l'écriture des implémentations de composants atomiques et aussi comme cible de génération ; l'approche n'est cependant pas restreinte à cet environment.

Enfin, nous illustrons notre approche en l'utilisant sur un cas d'étude de faible envergure. L'objectif est de montrer la simplicité et la flexibilité du méta-modèle, ainsi que la capacité de SWSG à repérer les incohérences et erreurs qui survienent dans tous les projets réels, même minimalistes. Comme ces erreurs sont mises en évidence à la conception (avant que des utilisateurs puissent manipuler les services web), notre approche permet aux développeurs d'aller vite sur les opérations à faible valeur ajoutée et de ré-utiliser du code, tout garantissant un certain niveau de confiance dans l'application produite.

Références

[1] Mario Luca BERNARDI et al. « M3D : A Tool for the Model Driven Development of Web Applications ». In : *Proceedings of the Twelfth International Workshop on Web Information and Data Management, WIDM 2012, Maui, HI, USA, November 02, 2012*. 2012, p. 73–80.

[2] David SFERRUZZA. *Safe Web Services Generator*. 2017. URL : `https://gitlab.startup-palace.com/research/swsg` (visité le 26/01/2018).

[3] David SFERRUZZA et al. « A Model-Driven Method for Fast Building Consistent Web Services in Practice ». In : 6th International Conference on Model-Driven Engineering and Software Development. Funchal, Madeira, Portugal, 23 jan. 2018. URL : `https://hal.archives-ouvertes.fr/hal-01654287`.

COnfECt : Une Méthode Pour Inférer Les Modèles De Composants D'un Système.

Elliott Blot, Patrice Laurençot, Sébastien Salva
LIMOS, Université Clermont Auvergne, France
Email: eblot@isima.fr, laurencot@isima.fr,sebastien.salva@uca.fr

Résumé

Cet article présente la méthode COnfECt qui complète les méthodes passives d'inférence de modèle pour extraire et représenter les comportements de composants d'un système vu comme une boite noire. COnfECt utilise les notions d'analyse de traces, de corrélation et de similarité de modèles. Nous montrons également comment intégrer COnfECt à la méthode GK-tail.
Mots clés : Inférence de modèle ; Inférence passive ; Composants

1 Introduction

L'inférence de modèles formels, aussi appelé *model learning*, regroupe un ensemble de méthodes permettant de retrouver un modèle comportemental d'un système, soit en interagissant avec ce dernier (méthodes actives, e.g., [1]), soit en analysant un ensemble de traces d'exécution obtenu en monitorant le système (méthode passive, e.g., [3]). Un modèle inféré peut ensuite être employé pour analyser le fonctionnement du système ou pour générer des tests. Bien qu'il soit aujourd'hui possible d'inférer des modèles à partir de certains systèmes réels, plusieurs points restent à étudier avant de pouvoir passer dans une phase industrielle. Parmi ceux-ci, nous avons relevé que les méthodes proposées considèrent un système comme une boite noire globale, qui prend des événements d'entrées depuis un environnement et produisent des événements de sorties. Pourtant, la grande majorité des systèmes actuels sont constitués de composants ré-utilisables, qui interagissent entre eux. La représentation de ces composants et de leurs compositions à travers plusieurs modèles permettrait une plus grande lisibilité du fonctionnement du système, voire une plus grande précision.

A travers cet article, nous nous penchons sur cette problématique et introduisons la méthode COnfECt (COrrelate Extract Compose) qui vise à compléter les méthodes d'inférence passives pour produire des systèmes de CEFSMs (Callable Extended Finite State Machine). Une CEFSM est une EFSM spécialisée équipée d'un événement interne spécial exprimant l'appel d'une autre CEFSM. L'idée fondamentale utilisée par COnfECt est qu'un composant dans un système peut être identifié par son comportement. D'une manière simplifiée, COnfECt analyse les

traces, détecte les blocs de comportements, extrait ces comportements dans de nouveaux ensembles de traces et aide à produire un ensemble de CEFSMs. COnfECt utilise pour cela les notions d'exploration d'événements (event mining), de corrélation d'événements et de clustering d'ensembles de traces.

Dans cet article, nous présentons sommairement les deux étapes de COnfECt appelées *Analyse de Trace et Extraction* et *Composition de CEFSMs*. Nous montrons enfin un exemple d'intégration de COnfECt avec la méthode GK-tail [3].

La suite de cet article est organisée de la manière suivante : la Section 2 présente les modèles générés par notre méthode, les CEFSMs. La description de l'approche est donnée dans la Section 3. Nous montrons un exemple d'intégration avec la méthode GK-tail, dans la Section 4 et nous concluons et donnons quelques perspectives en Section 5.

2 Callable Extended Finite State Machine

Une CEFSM permet de représenter l'appel d'un composant à partir d'un autre composant. Par manque de place, seul un résumé de la définition est donnée ici.

Une CEFSM est un tuple $\langle S, S_0, \Sigma, P, T \rangle$ avec S l'ensemble des états, S_0 l'état initial, $\Sigma = \Sigma_I \cup \Sigma_O \cup \{call\}$ l'ensemble des symboles avec $call$ un symbole spécial, et P l'ensemble des paramètres. T est l'ensemble des transitions de type $s_1 \xrightarrow{e(p),G} s_2$ avec $e(p)$ un événement composé de paramètres $p \subseteq P$ et G une garde sur P qui doit être satisfaite pour autoriser le franchissement de la transition. Comme dit précédemment, une transition d'une CEFSM C_1 peut être annotée par un événement interne dénoté *call(CEFSM)* exprimant le début d'exécution d'une autre CEFSM. Cet événement non observable signifie que la CEFSM appelante C_1 est mise en pause tandis que la CEFSM appelée C_2 démarre son exécution à partir de son état initial. Quand C_2 a atteint un état final, C_1 reprend son exécution après l'événement *call(CEFSM)*. Nous ne considérons pas qu'un composant retourne des résultats à un autre composant. Un système de CEFSMs est supposé inclure au moins une CEFSM qui appelle d'autres CEFSMs.

Nous disons également qu'une CEFSM C est callable-complete sur un système de CEFSMs SC, si toutes les CEFSMs de SC peuvent être appelées à partir de tous les états de C.

3 Approche

COnfECt est une approche permettant de générer un système de CEFSMs à partir d'un système en boite noire, afin de compléter les méthodes d'inférence passive. Pour cela, COnfECt analyse les traces obtenues à partir du système à inférer et essaie de détecter les différents composants pour pouvoir les modéliser avec des CEFSMs. L'outil se base sur les traces obtenues du système pour en extraire le

comportement des composants. Par conséquent, plus le nombre de traces à disposition est importante, plus l'analyse de traces et l'inférence du modèle sera correcte.

Nous supposons que l'ensemble de traces est collecté de façon synchrone (environnement synchrone). Celles-ci peuvent être obtenues par des outils de monitoring. Dans ce papier, nous ne considérons pas le formatage des traces, et nous considérons que nous avons un "mapper" qui nous fournit des traces comme une séquence d'événements sous la forme $e(p_1 := d_1, ..., p_k := d_k)$ avec $p_1 := d_1, ..., p_k := d_k$ des affectations de paramètres.

COnfECt est composé de deux étapes principales appelées *Analyse de Traces et Extraction*, et *Composition de CEFSMs*, que nous allons décrire ci-dessous.

3.1 Analyse de Traces et Extraction

Après l'étude de plusieurs systèmes à base de composants, nous avons observé qu'un composant produit un comportement souvent identifiable du reste des événements dans les différentes traces du fait de ses mots clés. De plus, certains composants, notamment dans des systèmes embarqués, produisent des événements non contrôlables (sorties non précédées par des entrées). A partir de ces observations, cette étape a pour but d'identifier les composants dans les traces à l'aide d'un *Coefficient de corrélation*, grâce à la détection des problèmes de contrôlabilité.

Le *coefficient de corrélation* nous permet d'évaluer le lien, ou relation entre deux événements successifs dans des traces. Nous définissons le *coefficient de corrélation* entre deux événements par une fonction, dépendant de facteurs, qui sont définis par les préférences ou les besoins de l'utilisateur. Un exemple de facteur est la fréquence d'apparition d'événements successifs dans les traces, ou encore la similitude entre les paramètres des événements. Ce coefficient est une valeur comprise entre 0 et 1, qui nous permet de déterminer les séquences d'événements dans une trace ayant une forte corrélation s'il est supérieur a un seuil X, ou s'ils ont une *faible corrélation* s'il est inférieur a un seuil Y. Les seuils X et Y dépendent du contexte, et doivent être déterminés par un expert.

L'étape *Analyse de Trace & Extraction*, dont l'algorithme est donné dans [2], essaie de segmenter chaque trace en séquences d'événements et de reconnaître les séquences d'événements produites par différents composants. Ces dernières séquences sont extraites et placées dans de nouveaux ensembles de traces. Chaque ensemble sera ensuite transformé en CEFSM.

De façon résumée, une trace est segmentée en séquences $\sigma_1 \dots \sigma_k$ de telle sorte qu'une séquence est composée d'événements ayant une forte corrélation et que 2 séquences successives ont une faible corrélation. Nous considérons que ces séquences correspondent à l'appel de composants ayant des comportements différents. L'étape *Analyse de Trace & Extraction* détecte l'appel de ces composants en analysant la corrélation des séquences $\sigma_1 \dots \sigma_k$. Lorsqu'un appel de composant est détecté avec la séquence $\sigma_i \dots \sigma_j$, cette dernière est remplacée par l'événement $call(CEFSM := C)$ avec C un nouveau composant. La séquence $\sigma_i \dots \sigma_j$ est considérée comme une nouvelle trace qui est placée dans un nouvel ensemble de

traces. Cette nouvelle trace est elle même récursivement analysée pour détecter l'appel d'autres composants.

Prenons l'exemple de trace σ donné en Figure 1 pour illustrer le fonctionnement de l'étape d'extraction de traces effectuée par la procédure Extraire. Celle-ci prend une trace déjà segmentée et un ensemble de traces T_1 où sera stockée la séquence d'événements résultat. Nous appelons Extraire($\sigma_1\sigma_2\sigma_3\sigma_4\sigma_5\sigma_6, T_1$).

A) L'algorithme commence avec σ_1. Supposons que la première séquence qui ait une forte corrélation avec σ_1 est σ_5. σ est transformé et devient $\sigma_1 call(CEFSM := C_2)\sigma_5\sigma_6$. $Extraire(\sigma_2\sigma_3\sigma_4, T_2)$ est récursivement appelé pour analyser $\sigma' = \sigma_2\sigma_3\sigma_4$.

B) $\sigma_2\sigma_4$ ayant une forte corrélation, σ' est modifié et devient $\sigma' = \sigma_2 call(CEFSM := C_3)\sigma_4$. La séquence σ_3 est une nouvelle trace du nouvel ensemble T_3. Maintenant que σ' est totalement parcourue, elle est ajoutée dans un ensemble de traces T_2.

C) Nous revenons à la trace σ, au niveau de la séquence σ_5 et réappliquons le même procédé. Comme il n'y a plus de séquence qui est fortement corrélée avec σ_5, la fin de la trace correspond à un appel de composant. σ_6, est extraite et placée dans un nouvel ensemble de traces T_4. La trace σ devient $\sigma = \sigma_1 call(CEFSM := C_2)\sigma_5 call(CEFSM := C_4)$. Elle est placée dans l'ensemble de traces T_1.

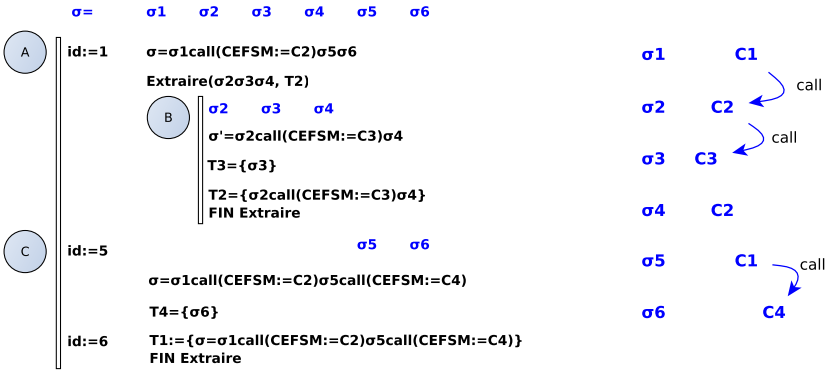


FIGURE 1 – Exemple d'execution de Extraire(σ, T).

A chaque fois que nous avons trouvé des comportements différents dans la trace, nous créons un nouvel ensemble de traces, qui sera transformé en CEFSM.

3.2 Composition de CEFSMs

Le but de cette étape est d'analyser les compositions de CEFSMs obtenues. Nous proposons trois stratégies de composition dans ce papier.

Composition stricte : dans cette stratégie, nous cherchons à ne pas sur-généraliser le système de CEFSMs. Dans ce cas, une CEFSM ne peut être appelée qu'une et une seule fois dans le système de CEFSMs. La CEFSM appelée est alors composée d'un seul chemin. Cette stratégie de composition est déjà accomplie par l'étape

Analyse de Traces et Extraction.

Composition faible : l'intuition de cette stratégie consiste à regrouper les CEFSMs ayant une forte similarité. En effet, les CEFSMs similaires doivent provenir du même composant. La notion de similarité est définie et évaluée par un coefficient de similarité. Ce coefficient est défini par le recouvrement (Overlap) des symboles et des paramètres entre 2 CEFSMs. Nous construisons une matrice de similarité à partir des coefficients, ce qui nous permet d'utiliser des algorithmes de clustering (ex : Agglomerative Hierarchical Clustering) pour trouver les classes de CEFSMs similaires, qui sont assemblées dans le même cluster. Les CEFSMs d'un même cluster sont fusionnées via une union disjointe. Toutes les transitions $s_1 \xrightarrow{call(CEFSM:=C_i)} s_2$ sont mises à jour pour que la bonne CEFSM soit appelée, en remplaçant le C_i par la CEFSM correspondant au cluster. De plus, toutes les transitions $s_1 \xrightarrow{call(CEFSM:=C_i)} s_2$ sont remplacées par une boucle $(s_1, s_2) \xrightarrow{call(CEFSM:=C_i)} (s_1, s_2)$, en fusionnant les états s_1 et s_2.

Composition forte : cette stratégie est basée sur la précédente, à laquelle nous rajoutons en plus pour tout état s, une transition $s \xrightarrow{call(CEFSM:=C_i)} s$ pour chaque CEFSMs C_i. Les CEFSMs deviennent maintenant callable-complete.

Dans la section suivante, nous montrons un exemple d'intégration avec la méthode passive GK-tail [3].

4 Intégration avec GK-tail

GK-tail est une méthode d'inférence de modèle passive proposée par Lorenzoli et al. [3], inférant une EFSM à partir de plusieurs traces d'exécution d'un système, et ceci en quatre étapes : la fusion de traces, la génération de prédicat, la construction d'une première EFSM, et enfin la fusion d'états équivalents.

Notre approche COnfECt peut s'intégrer à GK-tail comme décrit en Figure 2. L'étape *Analyse de Traces et Extraction* de COnfECt permet de segmenter les traces et d'identifier des composants. Elle est réalisée après la fusion des traces (étape 1 GK-tail), pour accélérer le calcul du coefficient, car moins de traces sont à analyser. L'étape *Composition de CEFSMs* permet de grouper certains CEFSMs similaires, si la stratégie composition faible ou forte est employée. Cette étape est faite avant que GK-tail ne fusionne les états. Cet ordre peut favoriser le groupement d'états équivalents qui seront fusionnés ensuite.

La Figure 3 illustre un exemple de ce que nous pouvons obtenir en utilisant notre méthode avec GK-tail sur un petit échantillon de traces d'un objet connecté. L'objet est composé d'une interface WEB, d'un capteur de température, ainsi qu'un capteur de mouvement. Notre méthode permet de séparer d'un coté l'interface WEB, à gauche dans la Figure 3, et les capteurs à droite. Les états obtenus dans cette figure ne peuvent plus fusionner car leurs 2futurs sont tous différents.

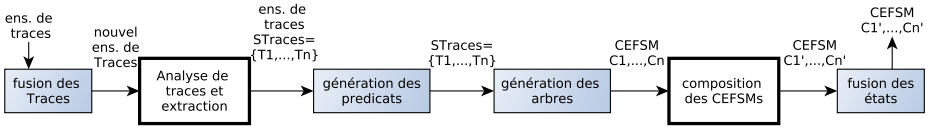


FIGURE 2 – Intégration des étapes avec GK-tail.

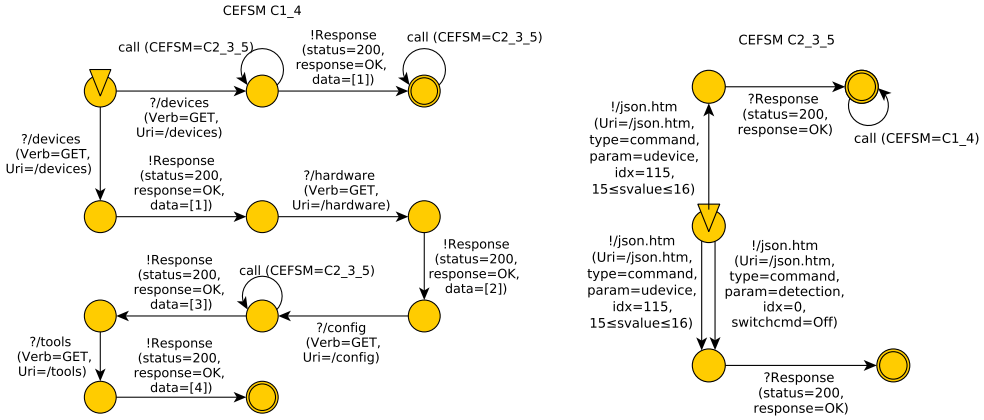


FIGURE 3 – Exemple de CEFSMs obtenus.

5 Conclusion

Nous avons introduit la méthode COnfECt, qui permet d'inférer un système de CEFSMs à partir de traces d'exécution d'une implémentation. Des composants sont identifiés dans les traces, et sont modélisés par des CEFSMs, qui peuvent être appelées à partir d'autres CEFSMs. Les algorithmes et définitions sont disponibles dans [2]. En termes de perspectives, il faudrait adapter la méthode afin de l'utiliser sur des systèmes aux communications asynchrones, ou encore considérer des composants dont les exécutions se font en parallèle. Nous prévoyons l'implémentation de cette méthode et son intégration avec GK-tail pour effectuer des évaluations.

Références

[1] D. ANGLUIN : Learning regular sets from queries and counterexamples. *Information and Computation*, 75(2):87 – 106, 1987.

[2] E. BLOT, S. SALVA et P. LAURENÇOT : COnfECt : Une Méthode Pour Inférer Les Modèles De Composants D'un Système. Research report, fév. 2018. http ://sebastien.salva.free.fr/RR-18-02.pdf.

[3] D. LORENZOLI, L. MARIANI et M. PEZZÈ : Automatic generation of software behavioral models. *In Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, p. 501–510, New York, NY, USA, 2008. ACM.

Algorithme rapide de calcul des dépendances de contrôle sur des graphes arbitraires *

Jean-Christophe Léchenet[1,2] Nikolai Kosmatov[1] Pascale Le Gall[2]

[1] CEA, LIST, Laboratoire de Sûreté des Logiciels, PC 174, 91191 Gif-sur-Yvette
prenom.nom@cea.fr
[2] Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 91190 Gif-sur-Yvette
firstname.lastname@centralesupelec.fr

Contexte et motivations. Les dépendances de contrôle sont une notion importante dans le domaine de l'analyse logicielle. Elles sont utilisées notamment dans les compilateurs, les outils de vérification, les générateurs de tests, les outils de transformation de programmes, les simulateurs et les débogueurs. Elles forment aussi l'un des deux piliers du *slicing*, avec les dépendances de données. Intuitivement, à l'intérieur d'un programme, une instruction i_2 a une dépendance de contrôle vis-à-vis de l'instruction i_1 si l'exécution de i_1 décide si i_2 sera exécutée ou non. Par exemple, dans une instruction conditionnelle `if (c) then p1 else p2`, l'évaluation de la condition `c` décide du choix de la branche à exécuter. Toutes les instructions dans `p1` et toutes les instructions dans `p2` sont donc dépendantes de la condition `c`.

Traditionnellement, les dépendances de contrôle sont définies sur des programmes représentés par leur graphe de flot de contrôle, et on exige de ces graphes qu'ils n'aient qu'un unique nœud de sortie. Cette restriction permet en effet de définir les dépendances de contrôle de manière classique, en termes de post-dominateurs. Mais cette condition est trop restrictive lorsqu'on souhaite manipuler des programmes n'ayant aucune sortie ou en ayant plusieurs. C'est notamment le cas des programmes réactifs, qui lorsqu'ils s'exécutent normalement, ne sont pas censés terminer. En 2011, Danicic et al. [1] proposent une formalisation élégante des dépendances de contrôle sur des graphes orientés finis. Ils montrent que leur formalisation est une généralisation de toutes les formalisations précédentes de dépendances de contrôle. Ils définissent aussi un algorithme calculant à partir d'un sous-ensemble de sommets du graphe sa clôture, c'est-à-dire le plus petit sur-

*Cet article est un résumé étendu de l'article *Fast Computation of Arbitrary Control Dependencies* [2] accepté à la conférence *FASE 2018*.

ensemble contenant tous les sommets vis-à-vis desquels au moins un des ses éléments a une dépendance de contrôle. Ils démontrent sur papier que cet algorithme est correct. Mais alors qu'il existe pour la définition classique en termes de post-dominateurs des algorithmes assez performants, l'algorithme proposé par Danicic et al. ne semble pas très efficace. En particulier, il ne tire pas pleinement profit de sa structure itérative : les calculs intermédiaires effectués lors d'une itération ne sont pas réutilisés lors des itérations suivantes. De plus, les raisonnements effectués manipulent des chemins de graphe de manière non-triviale. On souhaite donc renforcer notre confiance dans les résultats en faisant appel à la preuve mécanisée.

Approche et contributions. Nous avons d'abord choisi de formaliser une partie des résultats de Danicic et al. dans l'assistant de preuve Coq. Cette formalisation contient les principaux concepts, mais également l'algorithme ainsi que sa preuve de correction. Une version certifiée de l'algorithme peut être extraite de ce développement. Au cours de cette formalisation, nous avons découvert un lemme [1, Lemme 53] dont la preuve était inexacte, bien que son énoncé fût correct.

Nous avons ensuite conçu un algorithme optimisant celui de Danicic. L'idée principale de cet algorithme est d'enregistrer une partie de l'information obtenue en parcourant les chemins du graphe lors d'une itération sous la forme d'étiquettes attachées aux sommets du graphe. Cet étiquetage pourra être réutilisé lors des itérations suivantes et permettra à l'algorithme d'être plus rapide. La principale difficulté de cette approche est de gérer précisément la mise à jour de l'étiquetage. Pour prouver la correction de cet algorithme, nous avons eu besoin d'invariants plus compliqués que ceux de Danicic et al. Pour diminuer l'effort de preuve, nous avons utilisé Why3 de manière à pouvoir faire appel à des prouveurs automatiques. L'algorithme, sa preuve de correction, ainsi que les concepts sous-jacents sont formalisés dans Why3.

Pour s'assurer de la pertinence de notre optimisation, nous avons comparé le temps d'exécution de l'algorithme de Danicic et al. et du nôtre, implantés en OCaml grâce à OCamlgraph, sur des graphes générés aléatoirement. Sur chaque graphe, nous avons vérifié que les résultats des deux algorithmes étaient identiques. Nous les avons également comparés sur des petits graphes avec l'implantation extraite de Coq. Ces expériences confirment que notre algorithme est significativement plus rapide que celui de Danicic et al.

Les développements en Coq, Why3 et OCaml sont disponibles sur `http://perso.ecp.fr/~lechenetjc/control/`.

Références

[1] S. Danicic, R. W. Barraclough, M. Harman, J. Howroyd, Á. Kiss, and M. R. Laurence. A unifying theory of control dependence and its application to arbitrary program structures. *Theor. Comput. Sci.*, 412(49) :6809–6842, 2011.

[2] J.-C. Léchenet, N. Kosmatov, and P. Le Gall. Fast computation of arbitrary control dependencies. In *FASE'18 (Part of ETAPS'18)*, pages 207–224, 2018.

Vérifier des fonctions d'ordre supérieur à l'aide d'automates d'arbre*

Thomas Genet Timothée Haudebourg Thomas Jensen

Univ Rennes, Inria, CNRS, IRISA

14 mai 2018

Les fonctions d'ordre supérieur font partie intégrante des langages de programmation modernes comme Haskell, Caml, mais aussi Java, Scala ou même JavaScript. Là où leur utilité n'est plus à démontrer, leur utilisation pose problème dès lors qu'il s'agit de prouver la correction des programmes qui les utilisent. Un ingénieur soucieux de prouver la correction de son travail pourra se tourner vers des assistants de preuves interactifs (ex. Coq [1] ou Isabelle/HOL [8]), ou écrire une spécification dans un formalisme adapté à la preuve semi-automatique (ex. Liquid Types [9] ou Bounded Refinement Types [11, 10]). Cependant ces approches requièrent une expertise des méthodes formelles utilisées, ainsi que du temps que ce soit pour l'annotation du programme ou sa preuve. D'autres approches visent à vérifier le programme de manière complètement automatique : la preuve est réalisée sans annotation ni lemme intermédiaire. Ces approches, comme le model-checking d'ordre supérieur [7, 4, 5, 6], sont accessibles à un public plus important, mais sont applicables à un ensemble de propriétés plus restreint.

Dans ce travail [2], nous poursuivons cette ligne de recherche en proposant une approche basée sur les systèmes de réécriture de termes (TRS). Les TRS permettent une représentation formelle simple des programme fonctionnels. Dans ce formalisme, la fonction $\mathit{filtrer}$ est définie par le TRS $\mathcal{R}$ suivant :

$$\begin{aligned} @(@(\mathit{filtrer}, \underline{p}), \mathit{cons}(\underline{x}, \underline{l})) \to\ & \textbf{si } @(p, x) \\ & \textbf{alors } \mathit{cons}(\underline{x}, @(@(\mathit{filtrer}, \underline{p}), \underline{l})) \\ & \textbf{sinon } @(@(\mathit{filtrer}, \underline{p}), \underline{l}) \\ @(@(\mathit{filtrer}, \underline{p}), \mathit{vide}) \to\ & \mathit{vide} \end{aligned}$$

On remarquera que $\mathit{filtrer}$ est une fonction d'ordre supérieur : elle prend en premier paramètre p qui est lui même une fonction (prédicat). Le symbole $@$ est utilisé ici pour représenter l'application. Ainsi $@(\underline{p}, \underline{x})$ signifie « x appliqué à p ». Les termes

*Cet article est un résumé long de l'article *Verifying Higher-Order Functions with Tree Automata* [2] accepté à la conférence FoSSaCS 2018

soulignés représentent les variables. Un TRS est donc un ensemble de règles régissant l'évaluation des expressions/termes d'un programme. Évaluer une expression, un terme, reviens à le réécrire en appliquant les règles définies dans $\mathcal{R}$. Par exemple $@(@(\mathit{filtrer}, \mathit{pair}), \mathit{cons}(0, \mathit{cons}(s(0), \mathit{vide})))$ se réécrit (en plusieurs étapes) en $\mathit{cons}(0, \mathit{vide})$. Ainsi vérifier qu'une fonction est correcte consiste à vérifier que l'ensemble des termes atteignables, en appliquant $\mathcal{R}$ à l'ensemble des entrées, est lui-même « correct ». Soit $\mathcal{L}$ le *langage* (l'ensemble) des termes initiaux. On définit un ensemble Err de termes « interdits », en théorie inatteignable si la fonction considérée est correcte. L'objectif ne notre méthode est de vérifier qu'aucun terme de Err n'est effectivement atteignable en réécrivant un terme $\mathcal{L}$ avec $\mathcal{R}$. Autrement dit que $\mathcal{R}^*(\mathcal{L}) \cap Err = \emptyset$, où $\mathcal{R}^*(\mathcal{L})$ est l'ensemble des termes atteignables à partir de $\mathcal{L}$ avec $\mathcal{R}$.

Dans notre exemple, $\mathcal{L}$ est l'ensemble des termes de la forme $@(@(\mathit{filtrer}, \mathit{pair}), l)$ où l est une liste quelconque d'entier naturels et Err est l'ensemble des liste contenant au moins un entier impair (on souhaite vérifier que seuls les entiers pairs sont conservés).

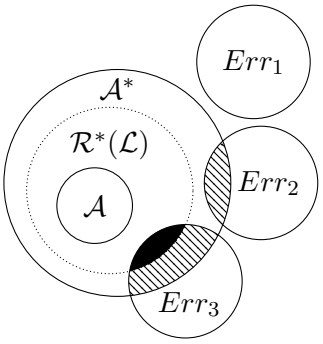


FIGURE 1 – Vérification à l'aide d'automates d'arbres. L'automate $\mathcal{A}$ représente le langage initial $\mathcal{L}$. $\mathcal{A}^*$ est calculé à partir de $\mathcal{A}$ comme une sur-approximation de $\mathcal{R}^*(\mathcal{L})$ (qui n'est jamais explicitement calculé). La fonction n'est vérifiée que si aucun terme de Err n'est dans $\mathcal{A}^*$.

En pratique les langages $\mathcal{L}$ et Err sont représentés à l'aide d'*automates d'arbres*. Si l'on nomme $\mathcal{A}$ l'automate représentant $\mathcal{L}$, notre méthode consiste ensuite à « compléter » [3] $\mathcal{A}$ à l'aide de $\mathcal{R}$ afin d'obtenir une sur-approximation, la plus précise possible, de $\mathcal{R}^*(\mathcal{L})$ en un automate $\mathcal{A}^*$. Il est ensuite possible de tester l'intersection de $\mathcal{A}^*$ avec Err. Si celle-ci est vide, la fonction est prouvée correcte. Sinon il est *possible* d'avoir $\mathcal{R}^*(\mathcal{L}) \cap Err \neq \emptyset$ (cas Err_2 ou Err_3 de la figure 1). Dans cet article, nous avons défini un mécanisme automatique de calcul de $\mathcal{A}^*$ pour une classe de programmes fonctionnels relativement large. Il s'agit des programmes d'ordre supérieur terminants, complets et ne construisant pas de « tours d'applications partielles », c'est à dire des structures arbitrairement grandes contenant des fonctions partiellement appliquées. L'implantation de cette technique a permis de prouver automatiquement des propriétés non triviales sur des programmes d'ordre supérieur classiques (`http://people.irisa.fr/Thomas.Genet/timbuk/funExperiments`).

Références

[1] Coq. The coq proof assistant reference manual : Version 8.6. 2016.

[2] Thomas Genet, Timothée Haudebourg, and Thomas Jensen. Verifying Higher-Order Functions with Tree Automata. In *21st International Conference on Foundations of Software Science and Computation Structures*, 2018.

[3] Thomas Genet and Vlad Rusu. Equational approximations for tree automata completion. *Journal of Symbolic Computation*, 45(5) :574–597, 2010.

[4] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 416–428, 2009.

[5] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 222–233, 2011.

[6] Yuma Matsumoto, Naoki Kobayashi, and Hiroshi Unno. Automata-based abstraction for automated verification of higher-order tree-processing programs. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, pages 295–312, 2015.

[7] C.-H. Luke Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90. IEEE, 2006.

[8] Lawrence C Paulson et al. The isabelle reference manual. Technical report, University of Cambridge, Computer Laboratory, 1993.

[9] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 159–169, 2008.

[10] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 48–61, 2015.

[11] Niki Vazou, Patrick Maxim Rondon, and Ranjit Jhala. Abstract refinement types. In *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 209–228, 2013.

Preuve de programmes d'énumération avec Why3

Alain Giorgetti et Rémi Lazarini
FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France

Résumé

L'énumération est une technique élémentaire de génération automatique de données pour le test du logiciel. Cet article présente une spécification formelle de programmes d'énumération de données stockées dans des tableaux d'entiers, pour leur vérification déductive avec la plateforme Why3. L'approche est illustrée par l'exemple de l'énumération de toutes les permutations d'une taille donnée, dans l'ordre lexicographique.

1 Introduction

Les outils de vérification formelle de programmes ont désormais atteint une maturité qui permet de les utiliser dans l'industrie du logiciel critique. Ces implémentations complexes laissent cependant planer des doutes sur leur propre correction. Plusieurs travaux contribuent à rétablir la confiance en ces outils, en certifiant certains de leurs composants. Par exemple, en tant que composant d'un environnement de test, un solveur de contraintes a été certifié avec l'assistant de preuve Coq [CDG12]. Le test automatique fondé sur les propriétés a été formalisé en Coq [PHD+15] pour certifier des générateurs aléatoires de données de test.

Dans cette lignée, ce travail est une contribution à la certification d'outils de test exhaustif borné (BET, pour *Bounded-Exhaustive Testing*). Le BET [SYC+04] automatise le test unitaire en générant toutes les données possibles d'entrée de la fonction sous test, jusqu'à une taille donnée. Quoique cette méthode soit limitée aux données de petite taille, sa pertinence est reconnue : "*a large portion of faults is likely to be revealed already by testing all inputs up to a small scope*" [JD96]. Fournissant toujours des contre-exemples de taille minimale, le BET facilite le déboguage. Il est complémentaire de méthodes adaptées aux données de plus grande taille, comme la génération aléatoire.

Le BET s'applique aux données munies d'une taille telle qu'il existe un nombre fini de données différentes de chaque taille. Ces données sont appelées *structures combinatoires*. Nous souhaitons spécifier et vérifier formellement diverses propriétés des programmes qui implémentent des algorithmes d'énumération de ces structures combinatoires, lorsqu'elles sont stockées dans un tableau d'entiers bornés soumis à des contraintes structurelles, telles que l'unicité des valeurs. Nous traitons ici l'exemple significatif d'un tableau de longueur n qui stocke les valeurs d'une permutation des n premiers entiers naturels.

La combinatoire énumérative propose de nombreux algorithmes de génération exhaustive bornée, plus ou moins difficiles à spécifier et vérifier formellement. Nous étudions ici les algorithmes dits de *génération lexicographique*, qui énumèrent tous les tableaux de même taille selon l'ordre lexicographique, défini comme suit :

Définition 1 (Ordre lexicographique) *Soit A un ensemble muni d'un ordre strict $<$ (relation binaire irréflexive et transitive). On appelle* ordre lexicographique (strict, induit par $<$) *la relation binaire sur*

les tableaux à valeurs dans A, notée $\prec$, telle que, pour tout entier $n \geq 0$ et pour tous les tableaux a et b de longueur n dont les éléments sont dans A, on a $a \prec b$ si et seulement s'il existe un indice i ($0 \leq i \leq n-1$) tel que $a[j] = b[j]$ pour tout j entre 0 et $i-1$ inclus et $a[i] < b[i]$.

La relation binaire $\prec$ ainsi définie est un ordre strict, qui est total si l'ordre sur A est total.

L'énumération est une forme particulière du concept général d'itération en programmation. Filliâtre et Pereira [FP16a, FP16b] ont spéficié formellement le processus d'itération lorsqu'il parcourt des données stockées en mémoire. Nous complétons ce travail en spécifiant une autre forme d'itération, appelée *énumération* ou *génération exhaustive*, qui consiste à produire chaque donnée au fur et à mesure, à la volée, avec pas ou peu de stockage en mémoire des précédentes données produites.

Par analogie avec la classification des itérateurs à grands pas et à petits pas [FP16a], nous considérons deux classes de générateurs exhaustifs. Un *générateur à grands pas* garde le contrôle de l'itération, en produisant lui-même toutes les structures combinatoires d'une même famille, tout en appliquant le même traitement à chacune d'elles. Il est souvent implémenté à partir d'un *générateur à petits pas* dont chaque appel construit une nouvelle structure de la même famille, à partir d'une quantité d'information limitée, typiquement la précédente structure construite par ce générateur. Un intérêt d'un générateur à petits pas est qu'il laisse le contrôle au code client, lui permettant par exemple de paralléliser l'énumération.

Nous souhaitons prouver formellement quatre propriétés de ces générateurs. La *correction* est la propriété que chaque structure générée satisfait la contrainte structurelle attendue (tableau borné, permutation, ...). La *complétude* exige que le générateur produise toutes les structures d'une même taille donnée. La *terminaison* exige l'arrêt de l'énumération. L'*absence de doublons* exige que chaque donnée ne soit générée qu'une seule fois. Nous considérons ici que ces deux dernières propriétés sont des conséquences de la propriété de *progression*, selon laquelle la donnée produite par le générateur à petits pas est strictement supérieure à sa donnée d'entrée, pour l'ordre lexicographique de la définition 1.

Plusieurs générateurs séquentiels de tableaux structurés ont été implémentés en C et spécifiés en ACSL [GGP15]. Leur correction et leur progression ont été prouvées automatiquement avec le greffon WP de la plateforme Frama-C, mais pas leur complétude, dont les conditions de vérification sont plus complexes. Pour les simplifier, nous adaptons ces générateurs au langage WhyML de la plateforme Why3 [BFM+18] et à sa structure de tableaux mutables. Why3 est une plateforme de vérification déductive qui permet d'utiliser des prouveurs automatiques (comme Alt-Ergo [Alt18]), mais aussi des assistants de preuve (comme Coq). Son mécanisme d'extraction de générateurs OCaml corrects par construction permet par exemple d'utiliser ces générateurs dans un outil de test exhaustif borné de conjectures Coq [DG18].

Les contributions de cet article sont une spécification formelle de la notion de générateur lexicographique en WhyML (partie 2) et une discussion sur la vérification déductive de leurs propriétés (partie 3). La partie 4 conclut cette étude. Une version étendue de cet article et le code présenté sont téléchargeables ici : `http://members.femto-st.fr/alain-giorgetti/fr/ressources`.

2 Spécification des générateurs lexicographiques

Un *générateur lexicographique* à grands pas énumère toutes les structures combinatoires d'une même famille (comme les permutations) dans l'ordre lexicographique $\prec$ (définition 1). Nous étudions le cas où il procède par itération d'un générateur lexicographique à petits pas qui construit la structure b à partir de la structure a qui la précède immédiatement selon l'ordre $\prec$. Nous spécifions successivement en WhyML le générateur à grands pas, la propriété de correction et les fonctions d'initialisation

et de passage au suivant du générateur à petits pas. Nous illustrons ces spécifications avec l'exemple fil-rouge d'un générateur lexicographique de permutations. Pour tout entier naturel n, la permutation p sur $[0..n-1]$ est représentée en WhyML par un tableau d'entiers mutable `a`, de type `(array int)` et de longueur `a.length` $= n$, tel que `a[`i`]` $= p(i)$ pour $0 \leq i < n$.

Génération de toutes les structures. Les générateurs sont spécifiés dans un style impératif, avec un état modifié et des boucles pour itérer. L'état des générateurs est stocké dans une structure de données appelée *curseur*, définie en WhyML par le type suivant :

```
type cursor = { current: array int; mutable new: bool; }
```

Cette structure est une variante de la structure de curseur proposée par Filliâtre et Pereira pour l'itération [FP16b]. Le curseur contient un champ `current` qui stocke la dernière structure générée et un champ `new` qui prend la valeur `true` si la structure stockée dans le champ `current` est nouvelle, c'est-à-dire qu'elle n'a pas encore été traitée. Le type de la structure `current` est ici un tableau d'entiers mais la spécification peut être adaptée à d'autres types.

Le générateur à grands pas

```
let gen (c: cursor) = let f = ... in while c.new do f c.current; next c done
```

doit être appelé avec un curseur `c` dont le champ `current` stocke une première structure. Le corps de la boucle applique un traitement (une fonction `f`) à chaque nouvelle structure, puis génère la structure suivante en appelant le générateur à petits pas `next` (présentée plus loin) qui modifie le curseur par effets de bord. La boucle se répète tant qu'une structure différente est générée par la fonction `next`, ce qui est caractérisé par l'égalité `c.new = true`.

Propriété de correction. La correction est spécifiée par le prédicat

```
predicate sound (c: cursor) = is_X c.current
```

à partir d'un prédicat `is_X` caractéristique de la famille `X` de structures générée. Par exemple, la famille `permut` des permutations est définie comme une endofonction injective, par le prédicat

```
predicate is_permut (a:array int) = (range a) ∧ (injective a)
```

où `(range a)` spécifie que le tableau `a` est à valeurs dans $[0..$`a.length` $- 1]$ et `(injective a)` spécifie l'injectivité de la fonction représentée par le tableau `a`. Nous avons plus généralement défini l'injectivité pour le type polymorphe `(array 'a)`, en spécialisant un prédicat analogue défini dans la librairie standard de Why3 pour le type polymorphe `(map 'a 'b)` des tableaux associatifs.

Création et initialisation du curseur. Les générateurs `gen` et `next` prennent en paramètre d'entrée un curseur `c` qui doit être construit et initialisé par une fonction

```
val create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures  { result.new → sound result }
```

qui prend en paramètre la taille n des structures à générer. Sa précondition impose une taille positive ou nulle. Sa postcondition exige que le champ `current` du curseur construit contienne une structure de la famille `X`, si son champ `new` contient la valeur `true` (ce champ doit valoir `false` s'il n'existe aucune structure de taille n).

Par exemple, la fonction `create_cursor` du listing 1 construit un curseur contenant la permutation identité ($p(i) \leftarrow i$), à l'aide d'une boucle `for` spécifiée par un invariant `(is_id` p i`)` qui exige que la partie $p[0..i-1]$ du tableau p représente l'identité. Ceci permet de démontrer la postcondition de correction de ce curseur. La seconde postcondition indique que cette permutation identité existe et qu'elle est nouvelle pour toute taille $n \geq 0$. Ces postconditions sont plus précises que dans le cas général car il existe une permutation pour toute taille $n \geq 0$.

```
predicate sound (c: cursor) = is_permut c.current
predicate is_id (a:array int) (n:int) =
  ∀ i:int. 0 ≤ i < n → a[i] = i

let create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures  { sound result }
  ensures  { result.new }
= let p = make n 0 in
  for i = 0 to p.length - 1 do
    invariant { 0 ≤ i ≤ p.length }
    invariant { is_id p i }
    p[i] ← i
  done;
  { current = p; new = true }

let next_permutation (c: cursor) : unit
= let p = c.current in
  let n = p.length in
  if n ≤ 1 then c.new ← false
  else
  let r = ref (n-2) in       (* 1. *)
  while !r ≥ 0 && p[!r] > p[!r+1] do
   invariant { -1 ≤ !r ≤ n-2 }
   variant   { !r + 1 }
   r := !r - 1
  done;
  if !r < 0 then (* last array reached. *)
   c.new ← false
  else                              (* 2. *)
   let j = ref (n-1) in
   while !j > !r + 1 && p[!r] ≥ p[!j] do
    invariant { !r + 1 ≤ !j ≤ n-1 }
    variant   { !j }
    j := !j - 1
   done;
   swap p !r !j;                    (* 3. *)
   reverse p (!r+1) n;              (* 4. *)
   c.new ← true
```

Listing 1 – Création du curseur et passage à la permutation suivante.

Génération de la structure suivante. Le contrat de la fonction `next` est

```
val next (c: cursor) : unit
  requires { sound c }
  ensures  { sound c }
  ensures  { c.current.length = (old c).current.length }
```

La précondition et la première postcondition spécifient la propriété de correction à l'aide du prédicat `sound`. La seconde postcondition assure que les structures d'entrée et de sortie ont la même taille.

Par exemple, une fonction `next_permutation` de passage à la permutation suivante est détaillée dans le listing 1. Elle utilise deux fonctions auxiliaires `swap` et `reverse` non reproduites ici. La fonction `swap` vient de la librairie standard de Why3. L'instruction `(swap` a i j`)` échange les éléments du tableau a aux indices i et j. La fonction `reverse` est telle que `(reverse` a l u`)` inverse la partie $a[l..u-1]$ du tableau a.

Afin de faciliter la lecture du code, les variables p et n représentent respectivement la permutation courante et sa taille. Si cette taille est 0 ou 1, la permutation courante est la dernière permutation (`c.new ← false`). Sinon, le programme procède par révision du suffixe du tableau p, comme détaillé dans l'exemple d'exécution suivant. Soit p le tableau d'entiers

i	0	1	2	3	4	5
$p[i]$	4	1	2	5	3	0

qui stocke les valeurs d'une permutation sur $[0..5]$, également notée p. Ainsi $p[i] = p(i)$ pour $i = 0, \ldots, 5$. Le programme transforme le tableau p en place, pour qu'il devienne le plus petit tableau p' strictement supérieur à p (selon l'ordre lexicographique $\prec$) qui représente une permutation p'. L'étape (1) cherche l'*indice de révision* $!r$ tel que p et p' aient le plus grand préfixe commun $p[0..!r-1] = p'[0..!r-1]$ (r est une référence et ! est l'opérateur de déréférencement). Si p est une permutation, cet indice est le plus grand indice i (le plus à droite) tel que $p[i]$ est inférieur à $p[i+1]$. Dans notre exemple de permutation p, l'indice de révision est $!r = 2$. Le *suffixe* est la fin du tableau $p[!r..5]$, à partir de l'indice de révision. L'étape (2) détermine la nouvelle valeur de $p[!r]$, telle que le tableau p' soit supérieur à p et le plus petit possible. Dans le cas d'une permutation, cette nouvelle valeur de $p[!r]$ est la plus petite valeur $p[!j]$ supérieure à $p[!r]$ et présente dans la partie $p[!r+1..5]$ du tableau après l'indice de révision. Dans notre exemple, c'est la valeur $p[4] = 3$, pour $!j = 4$. L'étape (3)

échange les valeurs de $p[!r]$ et $p[!j]$, grâce à la fonction `swap`. On obtient alors le tableau p_1 suivant :

$p_1[i]$	4	1	3	5	2	0

. L'étape (4) calcule la partie $p'[!r+1..5]$ la plus petite possible. Pour que p' soit une permutation, cette partie doit être la partie $p_1[!r+1..5]$ triée dans l'ordre croissant. Puisque cette partie $p_1[!r+1..5]$ est triée dans l'ordre décroissant, il suffit de l'inverser avec la fonction `reverse`, ce qui produit le tableau

$p'[i]$	4	1	3	0	2	5

. Si un indice de révision n'a pas été trouvé durant l'étape (1), alors $!r$ vaut -1 et p est la dernière permutation, ce qui est indiqué en attribuant la valeur `false` au champ `new` du curseur.

3 Propriétés

Correction. Dans la fonction `next_permutation`, les boucles `while` des étapes (1) et (2) sont spécifiées par un invariant qui fixe des bornes pour les indices de tableau modifiés par la boucle, et par un variant pour justifier leur terminaison. Avec ces annotations, la correction et la terminaison de cette fonction sont prouvées automatiquement avec Alt-Ergo 1.30.

Progression. Quand la fonction `next` de passage au suivant génère une nouvelle structure, elle doit satisfaire la propriété de progression

```
ensures { c.new → lt_lex (old c.current) c.current }
```

où le prédicat `lt_lex` formalise sur les tableaux d'entiers l'ordre lexicographique strict $\prec$ de la définition 1. Cet ordre est induit par l'ordre strict $<$ prédéfini sur les entiers du langage WhyML. Quand la structure courante est la dernière à générer, la fonction `next` ne doit pas la modifier, comme le stipule la postcondition

```
ensures { not c.new → array_eq (old c.current) c.current }
```

où le prédicat `array_eq` de la librairie standard de Why3 formalise l'égalité entre deux tableaux. Nous spécifions formellement l'ordre lexicographique grâce aux deux prédicats suivants :

```
predicate eq_prefix (a1 a2: array int) (u: int) = array_eq_sub a1 a2 0 u
predicate lt_lex (a1 a2: array int) = a1.length = a2.length ∧
  ∃ i:int. 0 ≤ i < a1.length ∧ eq_prefix a1 a2 i ∧ a1[i] < a2[i]
```

Le prédicat `eq_prefix` spécialise le prédicat `array_eq_sub` de la bibliothèque standard de Why3, qui est tel que `(array_eq_sub` a_1 a_2 l u`)` formalise l'égalité des sous-tableaux $a_1[l..u-1]$ et $a_2[l..u-1]$. Le prédicat `lt_lex` formalise la définition 1.

Après ajout de la postcondition

```
ensures { eq_prefix a (old a) l }
```

pour la fonction `reverse`, qui assure qu'elle ne modifie pas le préfixe $a[0..l-1]$ du tableau a, la preuve de la propriété de progression est automatique, avec Alt-Ergo 1.30.

Complétude. Dans cet article, nous réduisons la propriété de complétude aux trois propriétés suivantes : (1) La propriété *min_lex* impose que la première structure générée soit la plus petite structure selon l'ordre lexicographique. (2) La propriété *max_lex* exige que la dernière structure générée soit la plus grande selon l'ordre lexicographique. (3) La propriété d'*incrémentation* spécifie que la structure a_2 générée par la fonction `next` à partir de la structure a_1 est toujours la plus petite structure strictement supérieure à la structure a_1, selon l'ordre lexicographique. Autrement dit, il n'existe aucune structure a_3 tel que $a_1 < a_3 < a_2$.

Ces propriétés sont plus difficiles à démontrer que la correction et la progression, car elles incluent une quantification sur un tableau. Nous introduisons le lemme $(\forall a_1 a_2.\ a_1 \not\prec a_2 \rightarrow a_2 \preceq a_1)$ de totalité de l'ordre lexicographique. Ce lemme permet de valider automatiquement les propriétés *min_lex*

et *max_lex*. Cependant, les solveurs automatiques ne démontrent pas la propriété d'*incrémentation*. Nous envisageons de prouver interactivement sa traduction en Coq, produite par Why3 à partir du code WhyML. Nous commençons par le cas plus simple d'un générateur lexicographique de tableaux bornés (un tableau a est dit *borné* (par n) si $a[i] < n$ pour tous les indices i du tableau). Pour les tableaux bornés, les propriétés *min_lex* et *max_lex* sont également démontrées automatiquement, tandis que la propriété d'*incrémentation* l'est interactivement en Coq.

4 Conclusion

Nous avons présenté une formalisation générale des générateurs lexicographiques de structures combinatoires dans des tableaux d'entiers. Cette formalisation peut être généralisée à d'autres structures de données, comme les arbres. Nous avons spécifié et implémenté un générateur de permutations en WhyML, puis prouvé sa correction et sa progression avec Why3. Toutes les preuves sont effectuées sur une machine équipée d'un processeur Intel Core i3-2125 à 3.30 GHz $\times$ 4 sous Linux Ubuntu 16.04, en 1.2 secondes pour la fonction `next_permutation`, 0.3 secondes pour la fonction `reverse` et 0.06 secondes pour la fonction `create_cursor`.

Ce travail en cours doit se poursuivre avec une preuve de complétude pour le générateur de permutations, mais aussi avec la spécification, l'implémentation et la vérification déductive d'autres générateurs.

Références

[Alt18] The Alt-Ergo SMT solver. `http://alt-ergo.lri.fr`, 2018.

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

[CDG12] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *FM'12*, volume 7436 of *LNCS*, pages 116–131. Springer, 2012.

[DG18] C. Dubois and A. Giorgetti. Test and proofs for custom data generators. 2018. À paraître.

[FP16a] J.-C. Filliâtre and M. Pereira. Itérer avec confiance. In *JFLA'16*, 2016. `https://hal.inria.fr/hal-01240891`.

[FP16b] J.-C. Filliâtre and M. Pereira. A modular way to reason about iteration. In *NFM'16*, volume 9690 of *LNCS*, pages 322–336. Springer, 2016.

[GGP15] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP'15*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[JD96] D. Jackson and C. Damon. Elements of style : Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7) :484–495, 1996.

[PHD+15] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP'15*, volume 9236 of *LNCS*, pages 325–343. Springer, 2015.

[SYC+04] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. *SIGSOFT Softw. Eng. Notes*, 29(4) :133–142, July 2004.

Des listes et leurs fantômes : vérification d'un module critique de Contiki avec FRAMA-C* (résumé étendu)

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

Les appareils et services connectés, aussi appelés internet des objets (*Internet of Things : IoT*) sont de plus en plus populaires dans des domaines où la sécurité est critique. Ces questions de sécurité sont une cible de choix pour l'usage de méthodes formelles.

Ce travail se concentre sur Contiki, un système d'exploitation *open-source* populaire pour les objets connectés, qui propose notamment une pile IPv6 pour environnements contraints. Il est implémenté en C avec une attention particulière pour l'optimisation de la consommation mémoire et énergétique, et est disponible pour de nombreuses plateformes physiques. Au début du développement de Contiki en 2002, la sécurité n'était pas un objectif central. La sécurité des communications a été ajoutée plus tard, mais la vérification formelle du code n'a commencé que très récemment.

Nous présentons la vérification formelle du module de listes chaînées de Contiki. Ce module est critique pour le système d'exploitation, il est utilisé par 32 modules de Contiki et appelé plus de 250 fois dans la partie cœur du système. Son API est riche, il est possible d'ajouter ou enlever des éléments à n'importe quelle position dans la liste. Elle garantit l'unicité des éléments au sein des listes : avant toute insertion, le module s'assure de supprimer l'élément de la liste avant de l'ajouter à la position voulue. Finalement, Contiki ne permet pas l'allocation dynamique, contrairement aux implémentations classiques des listes, le module de Contiki ne fait donc pas de telle opération.

La vérification formelle du module est effectuée grâce au langage de spécification ACSL et au greffon WP de la plate-forme FRAMA-C. Notre approche se base sur des tableaux fantômes (*ghost*) utilisés comme représentation des listes chaînées

*Cette soumission est un résume étendu de l'article [1] accepté à NFM 2018.

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

manipulées. Cette représentation permet de raisonner à propos des éléments de la liste automatiquement mais demande de maintenir une relation forte entre les listes et leurs tableaux compagnons.

La relation entre un tableau fantôme et une liste chaînée est formulée à l'aide d'un prédicat inductif qui à partir d'un $index$ donné du tableau, et parallèlement à partir de l'adresse du premier élément de la liste, assure que l'on trouve l'adresse de chaque cellule i à la position $index + i - 1$ du tableau. Maintenir la relation entre la liste et le tableau fantôme nécessite de mettre à jour le tableau chaque fois que la liste est mise à jour, ce qui est réalisé à l'aide de fonctions fantômes.

Cette relation étant inductive, elle ne peut pas être manipulée directement par les prouveurs SMT qui ne sont pas capables de réaliser des preuves par induction. Cette formalisation inductive est donc accompagnée d'un ensemble de lemmes qui expriment des relations utiles à partir du prédicat inductif, et qui peuvent être directement manipulés en preuve automatique.

Ces lemmes permettent notamment de couper une liste et la plage du tableau auquel elle est liée en plusieurs sous-listes associées à des sous-plages du tableau (et inversement de fusionner des sous-listes en une liste complète). Ces relations sont par exemple nécessaires dans le cas d'une suppression d'un élément : on doit d'abord prouver que l'on peut couper la liste en deux sous-listes séparées par l'élément à supprimer, puis prouver que l'on peut « recoller » les deux sous-listes sans l'élément à supprimer.

A partir du code original du module (176 lignes de C), nous avons produit environ 1400 lignes d'annotations, dont 500 pour les contrats et 240 pour les définitions et lemmes. Sur les 798 obligations de preuves générées par WP, 24 correspondent aux lemmes et sont prouvées à l'aide de l'assistant de preuve COQ, par induction sur le prédicat de liaison. 770 obligations sont prouvées automatiquement. Sur les 4 restantes, 2 sont prouvées avec COQ et 2 avec le prouveur interactif de WP (TIP).

En travaux futurs, nous prévoyons d'effectuer la vérification de modules client du module de listes. Pour permettre l'utilisation du test, nous avons reformulé nos spécifications pour les rendre exécutables. Par ailleurs, nous prévoyons de comparer la formalisation actuelle avec des formalisations plus abstraites : listes logiques et fonctions partielles contiguës.

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 leurs discussions et conseils.

Références

[1] 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.

Vers la certification de programmes interactifs Djnn

Pascal Béger[1], Sébastien Leriche[1], and Daniel Prun[1]

[1]ENAC, Université de Toulouse - France

Résumé

Les systèmes critiques, particulièrement aéronautiques, contiennent de nouveaux dispositifs hautement interactifs. Pour certifier les logiciels qui les exploitent, les approches de vérification existantes ne sont pas toujours adaptées. Dans ce papier court, nous introduisons notre approche de construction d'applications interactives au moyen de Djnn (le modèle et l'API) et Smala (le langage réactif de haut niveau) puis nous discutons de la pertinence des outils et méthodes de vérifications dans ce contexte. Nous présentons enfin nos premiers résultats et perspectives de vérification formelle de programmes interactifs Djnn.

1 Introduction

Les systèmes critiques, particulièrement aéronautiques, contiennent de nouveaux dispositifs hautement interactifs : par exemple les cockpits de nouvelle génération font appel à une électronique sophistiquée pilotée par des applications logicielles complexes. Dans ce contexte, les processus de certification décrits dans la DO-178C/ED-12C offrent une place importante à la vérification formelle. Même si, depuis ces 3 dernières décennies, de nombreuses méthodes outillées ont été proposées pour le développement de logiciels sûrs et corrects d'un point de vue fonctionnel, les logiciels interactifs n'ont cependant pas bénéficié de la même attention : leur vérification repose encore principalement sur des évaluations de prototypes au cours de tests avec les utilisateurs finaux. Lorsque des techniques de vérification formelle sont utilisées, celles-ci ne proposent très souvent qu'une couverture partielle de la problématique des applications interactives. Par exemple les éléments issus de l'interface concrète sont peu pris en compte. La nature réactive, synchrone, mixant flot de contrôle et flot de données est difficile à appréhender à travers une unique approche et bien souvent plusieurs techniques doivent être utilisées de manière conjointes, ce qui complexifie la tâche.

Nous pensons qu'une part importante de ces difficultés est due à l'absence d'un langage spécifique permettant, par le biais d'une sémantique adaptée, aux concepteurs d'itérer sur la conception et aux développeurs de vérifier que leur produit est conforme à une spécification de référence.

Après avoir présenté le contexte de notre travail qui se concrétise par le développement du framework Djnn et du langage Smala, nous discutons du positionnement des principaux paradigmes de programmation ainsi que des approches de vérification formelles associées. Cela nous permet de positionner et justifier notre approche : Djnn repose sur un ensemble de concepts réduits et unifiés, ce qui nous semble pertinent pour décrire efficacement les applications interactives et facilite la création de passerelles vers les techniques usuelles de vérification formelle. Nous présenterons alors nos premiers résultats et les pistes envisagées pour nos travaux futurs.

2 Contexte

L'équipe « Informatique Interactive » de l'École Nationale de l'Aviation Civile travaille depuis plusieurs années sur une approche autour d'un modèle conceptuel d'exécution unifié dédié aux

applications interactives. Une implémentation sous la forme d'une API nommée **Djnn**[1] et des applications significatives sont venues valider notre approche [4]. Nous avons récemment montré comment ce modèle permettait la vérification et la validation de certaines propriétés inhérentes aux systèmes interactifs [10] et Djnn sera un des environnements d'exécution cible du projet FORMEDICIS[2].

Pour simplifier l'utilisation de Djnn, nous développons le langage **Smala**[3]. Smala permet de décrire de manière déclarative une application interactive selon le paradigme réactif. Une application en Smala contient un ensemble de composants où le contrôle est réduit à la transmission d'activation entre ces composants. A la base Smala contient 4 types de composants primitifs :

- La *property* est un composant permettant de stocker une valeur. La mise à jour de la valeur provoque l'activation du composant.
- Le *binding* (noté ->) permet la création d'un couplage entre 2 composants : quand le composant source est activé, alors le composant destination est immédiatement activé.
- L'*assignment* (noté =:) permet la transmission de valeur entre 2 *properties*.
- Le *comparator* (noté =) permet le test d'égalité de valeurs entre 2 *properties*.

A partir de ces composants primitifs, des composants de plus haut niveau sont définis par composition. Par exemple le *connector* contenant un *binding* et un *assignment* permet de transmettre simultanément une valeur et une activation ; le *if* constitué d'un *comparator* et d'un *binding* permet de transmettre une activation sous condition. En continuant ainsi, de nombreux autres composants plus complexes sont construits : le *switch*, la *fsm* (machine à état), le conteneur, les composants arithmétiques et logiques.... L'interface avec l'environnement est assurée par des composants dédiés fournis par Djnn. Par exemple les composants graphiques permettent de décrire les éléments à afficher (forme géométrique, position, couleur, opacité...) ainsi que les éventuels clics souris les concernant (position du clic, type de clic : press ou release...) ; les composants *clock* permettent de déclencher périodiquement une activation. L'exécution d'une telle application consiste à gérer l'activation des différents composants résultant soit de l'occurrence d'événements provenant de l'environnement (un clic souris, la modification de la valeur d'une propriété...), soit de la propagation explicite de l'activation décrite par le programmeur via les composants de *binding*.

Le compilateur Smala génère actuellement du code C ou Java utilisant l'API Djnn. Une illustration de Smala est proposée par la Figure 1.

Figure 1: Code Smala, extrait du code C/Djnn et résultat de l'exécution d'un chronomètre

Dans cet exemple, nous pourrions souhaiter valider les propriétés suivantes : l'aiguille des secondes est toujours visible (ie. elle n'est à aucun moment masquée par un autre élément graphique),

[1] `http://djnn.net`

[2] FORMEDICIS est un projet ANR en cours (ANR-16-CE25-0007) à l'origine du financement de ces travaux.

[3] `http://smala.io`

la valeur affichée par l'aiguille des secondes (liée à la perception humaine de la lecture d'une horloge) correspond à la valeur du chronomètre ou encore qu'un clic de souris sur le bouton gauche du chronomètre déclenche son départ. La spécificité de ce type de propriétés est d'être liée à la représentation concrète des éléments et aux interactions permises à l'utilisateur. De ce fait, leur vérification se fait traditionnellement au moyen de tests et nécessite un observateur humain.

3 Positionnement des principaux paradigmes de programmation

Les applications informatiques peuvent être développées en utilisant différents paradigmes de programmation. Même si nous nous intéressons à des applications réactives, cette section permet un positionnement qui nous permettra de discuter de l'impact de ces différences sur la vérification.

3.1 Analyse

Le paradigme impératif consiste en l'écriture d'une séquence de commandes devant être réalisées par l'application et modifiant son état [15]. Des structures de contrôle (conditionnelle, boucle...) permettent de faire évoluer le flot d'exécution par des changements d'états (mémoire/variables et pointeur sur l'instruction courante). Cet état permet l'interruption ou l'exécution pas à pas [23]. Dans ce paradigme, la vérification peut se focaliser sur l'état de la mémoire, le respect des bornes des variables ainsi qu'à l'absence de boucles infinies.

Un programme fonctionnel correspond à une fonction mathématique transformant son entrée en une sortie [24]. Il repose sur la conversion, consistant en un renommage des paramètres (pouvant être également des fonctions) et de réduction, représentant les appels de fonctions. Le flot d'exécution est décrit par les entrées et sorties des fonctions. En fonctionnel pur, les effets de bord sont interdits, l'affectation de valeur n'existant pas [24]. Le paradigme fonctionnel est donc sans état, ce qui simplifie la mise au point du programme puisqu'il est possible de tester et valider les fonctions indépendamment les unes des autres. Dans ce paradigme, les vérifications portent souvent sur la propriété de terminaison, les empilements des appels de fonction (déterminant l'utilisation de la pile).

La programmation orientée objet permet d'abstraire les données [13] de façon à obtenir une représentation proche des objets manipulés dans la réalité. Les concepts clés [17] sont une structuration du code et des données au sein de classes isolant ces données par encapsulation. La réutilisation est favorisée par l'héritage, le polymorphisme permet de manipuler des objets dont les types sont compatibles (issus d'une hiérarchie de classes). Les attributs d'un objet définissant son état interne, la programmation objet est de type avec état. Le flot d'exécution d'une application orientée objet suit les appels de méthodes entre objets [20]. Les propriétés intéressant la vérification sont par exemple le respect du principe d'encapsulation, des relations entre objets et de la causalité des appels de méthodes.

Dans une application réactive, des événements provenant de sources internes (horloge...) ou extérieures (clic de souris, appui sur le clavier...) déclenchent un comportement programmé. Ces événements se propagent en mettant à jour leurs dépendances par le concept de suite d'activation. Ainsi, la modification d'une variable peut provoquer la mise à jour d'autres variables et l'exécution de comportements qui en dépendent. L'exécution d'un programme réactif suit le flot de données, les événements extérieurs et la propagation automatique des changements se produisant en fonction des dépendances entre les composants [5]. Un programme réactif est donc sans état. Le respect des dépendances, le suivi du flot de données et de l'activation sont des propriétés pertinentes à vérifier pour les applications réactives.

3.2 Synthèse

Nous avons volontairement ciblé notre analyse (tableau 1) de manière à extraire les caractéristiques pertinentes à chaque paradigme, dans un objectif de vérification formelle. Toutefois, les langages récents offrent la possibilité de manipuler différents paradigmes. Par exemple, Scala.React est une couche réactive de Scala qui est une extension fonctionnelle de Java. Certains langages comme

Paradigme	Impératif	Fonctionnel	Orienté objet	Réactif
Flot d'exécution	Itératif, structure de contrôle	Suivi I/O fonctions	Contrôle, séquences appels méthodes	Données, arbre de dépendance
État	Avec état	Sans état	Avec état	Sans état
Définition de l'état	Variables en mémoire, locus de contrôle	*N/A*	État interne des objets (attributs) en mémoire	*N/A*
Propriétés pertinentes	État mémoire, bornes variables, boucles finies	Terminaison	Encapsulation, causalité appels méthodes	Dépendances, suivi flot de données et activation
Exemples de langages	Fortran, Pascal, C	Caml, Haskell,	Java, C#	FlapJax, Scala.React

Table 1: Synthèse des paradigmes de programmation

Oz ont été même inventés pour être multi-paradigme. Nous pensons que si de tels langages sont pertinents pour offrir au programmeur une forte expressivité, leur complexité augmente la difficulté des vérifications. Notre approche, ciblant les programmes interactifs et basée sur un modèle conceptuel simple devrait permettre de réduire la complexité des vérifications pour aboutir à de la certification.

4 Méthodes et outils pour la vérification

La vérification formelle consiste à s'assurer que l'application est conforme à une spécification de référence. Elle se base sur l'élaboration d'un modèle dont la syntaxe et la sémantique reposent sur des bases mathématiques permettant ainsi des raisonnements objectifs.

4.1 Moyens de modélisation

On distingue différentes catégories de modèles formels. Nous présentons ci-dessous les principales.

La logique de Hoare [16] repose sur la notion du triplet de Hoare $\{P\}Q\{R\}$ dans lequel P et R sont des formules logiques ou assertions représentant respectivement une précondition et une postcondition et Q le programme typiquement impératif. Cette modélisation permet de prouver qu'un programme vérifie une spécification, en garantissant la véracité de R à la fin de Q si P était vraie à l'initialisation de Q.

Un réseau de Petri est un triplet $\langle P, T, F\rangle$ avec P un ensemble de places p, T un ensemble de transitions t et F la relation de flot [22]. Les places peuvent contenir des jetons qui conditionnent l'exécution (on parle de tirage) des transitions : un tirage se concrétise par la consommation et la production de jetons. Ce modèle permet de vérifier des propriétés génériques d'atteignabilité d'état (vivacité, non-blocage) et de bornes.

Un automate [7] est défini par des ensembles finis d'états, de transitions associées à des labels, d'un état initial et d'une fonction de correspondance associant des propriétés à chaque état. Un tel automate avec une relation de transition et une fonction d'étiquetage est appelé structure de Kripke. Cette modélisation permet de mettre en oeuvre les techniques de *model checking*, où des propriétés exprimées sur les séquences de transitions franchies sont vérifiées.

L'interprétation abstraite [12] permet de décrire la sémantique opérationnelle d'un programme en ne conservant que les informations pertinentes pour le type de propriété à vérifier. Cette sémantique opérationnelle est représentée par un ensemble de traces sur lequel des vérifications par preuve sont possibles.

Le B au même titre que VDM ou Z, est une technique de raffinements successifs qui permet d'obtenir, à partir d'une abstraction haut niveau, une abstraction de plus en plus concrète. Cette méthode a évolué vers une approche plus générale, B-événementiel [2], prenant en compte les événements. A chaque raffinement, de nouvelles propriétés du système sont introduites. La vérification consiste à assurer (par preuve) que les propriétés exprimées dans les raffinements précédents

restent vraies pour le raffinement en cours. La plateforme Rodin permet de mettre en oeuvre cette approche.

Les propriétés à vérifier sont souvent exprimées sous la forme de formules de logique temporelle (CTL*, CTL et LTL). Elles reposent sur des combinateurs booléens, temporels ainsi que des quantificateurs de chemin [7]. CTL* et CTL s'intéressent à l'arbre de l'ensemble des exécutions tandis que LTL s'intéresse à une seule exécution. Ces propriétés peuvent être encodées par des observateurs qui permettent de reconnaitre des traces les respectant. De part leur grande expressivité, elles sont largement utilisées dans de nombreuses approches reposant sur la preuve ou le *model checking*.

4.2 Techniques de vérification

On distingue généralement 2 grandes techniques de vérification : le *model checking* et la preuve.

Le **model checking** est une technique permettant de vérifier quels états d'une structure de Kripke vérifient une formule de logique temporelle donnée [11]. Elle repose sur l'élaboration par simulation de l'ensemble des états possible du système, ainsi que des transitions entre ces états. Elle n'est utilisable que pour des applications dont le nombre d'états possibles est fini, pour des propriétés de sûreté. UPPAAL, SMV (Symbolic Model Verifier) ou encore SPIN sont des outils représentatifs de cette approche. TINA, Design/CPN ou GreatSPN supportent cette approche avec les réseaux de Petri.

La **preuve** est une approche où un ensemble de constructions et de règles mathématiques sont appliquées pour démontrer un résultat à atteindre. La théorie des graphes, la théorie des ensembles, les différentes logiques sont les principaux domaines utilisés où des techniques de déduction sont appliquées. Cette approche est utilisée pour des applications dont le nombre d'états possibles est potentiellement infini et permet de vérifier des propriétés de vivacité. PVS-Studio, Coq, HOL (a Higher Order Logic Theorem Prover) ou encore Isabelle sont les outils principaux utilisés pour cette approche.

4.3 Synthèse

Le paradigme des systèmes interactifs étant relativement récent, peu de travaux ont été réalisés concernant leur vérification formelle. Par exemple, concernant l'utilisation du *model checking*, [1] vérifie des interfaces utilisateur avec SMV (Symbolic Model Verifier) en utilisant CTL pour exprimer les propriétés à vérifier. [19] modélise le système avec un réseau de Petri orienté objet appelé ICO et mène des vérifications formelles limitées à un sous-ensemble du réseau. [14] a utilisé un langage à flots de données pour la validation automatique de systèmes interactifs. Dans le domaine des techniques de preuve, VDM, et Z ont été utilisées pour la définition de structures atomiques d'interaction [6]. HOL a été utilisée pour la vérification de spécifications d'interfaces utilisateur [8] et Event-B pour la modélisation et la validation de propriétés sur des systèmes multi-modaux [3]. Plus récemment, pour des systèmes médicaux, [18] ont utilisé un modèle en VDM sur lequel des propriétés de cohérence et de feedback ont été prouvées.

Historiquement, les techniques classiques d'analyse ont été développées pour être appliquées sur des langages impératifs ou fonctionnels. Celles-ci restent peu utilisées pour la prise en compte des spécificités des systèmes interactifs ainsi que celles des propriétés qui s'y rapportent. Ainsi, la modélisation concerne les parties abstraites de l'interaction (tâches, interface abstraite) et n'aborde pas les aspects concrets (objets graphiques, dispositif d'entrée, etc.). Démontrer des propriétés impliquant ce niveau de description (visibilité d'information par exemple) reste difficile à prendre en compte.

Dans ce contexte, notre objectif à moyen terme est d'étudier comment les techniques classiques de vérifications formelles peuvent être adaptées aux systèmes interactifs. Nous pensons que Djnn est un environnement pertinent pour réduire la complexité des activités de vérification formelle, grâce à sa capacité à exprimer de manière uniforme et avec peu de concepts, des éléments de l'interface aussi bien abstraite (contrôle...) que concrète (graphisme...).

La section suivante présente l'état de nos recherches ainsi que les perspectives que nous envisageons.

5 Travail réalisé et perspectives

Bien que Smala soit encore en cours de développement, nous avons déjà pu développer des techniques de vérification formelle de propriétés de systèmes interactifs, toutes s'appliquant au niveau du code Smala. Dans [9] nous avons exploité les caractéristiques du **graphe d'activation**. Ce graphe, déduit de la description en Smala, décrit toutes les relations d'activation possibles suite à l'occurrence d'un événement. Sur la base de son analyse, il est possible de vérifier formellement des propriétés d'atteignabilité (par exemple qu'une entrée finit toujours par générer une sortie attendue ou qu'une alarme est toujours déclenchée dans une configuration) ou encore relatives à la relation causale d'activation (par exemple qu'un message d'erreur affiché ne sera jamais recouvert par un autre).

Sur l'exemple introduit précédemment en figure 1, la propriété stipulant "qu'un clic de souris sur le bouton gauche du chronomètre déclenche son départ" est formellement vérifiée en s'assurant qu'il existe un chemin dans le graphe d'activation reliant l'événement "clic de souris" du composant "bouton gauche" à l'événement "départ" du composant "chronomètre". La propriété stipulant que "l'aiguille des secondes doit être toujours visible" est vérifiée en exploitant l'ordre de parcours des éléments du graphe suivi à l'exécution : en profondeur, de gauche à droite. Les éléments graphiques étant affichés selon cet ordre, vérifier la propriété consiste à s'assurer qu'aucun composant graphique est présent dans le graphe après le composant de l'aiguille des secondes, ou dans le cas contraire que le composant graphique ne recouvre pas l'aiguille[4].

Nous travaillons actuellement sur l'implantation d'**observateurs synchrones** en Smala : il s'agit de composants pouvant observer les activations au sein d'une application Smala, et reconnaitre l'occurrence de patterns pré-définis. D'autre part, nous étudions [21] la transformation de code Smala vers des **réseaux de Petri**, avec l'idée d'exprimer précisément une sémantique opérationnelle pour Smala et de profiter des possibilités de vérification qui sont disponibles sur ces réseaux. Nos perspectives à moyen terme concernent, d'une part, la prolongation des études précédentes (analyse du graphe d'activation et utilisation des réseaux de Petri) et, d'autre part, l'étude de l'utilisation de techniques de preuve formelle à partir de code Smala (traduction en Caml et utilisation de COQ, traduction en event-B).

References

[1] G. D. Abowd et al. A formal technique for automated dialogue development. In *Proceedings of the 1st Conference on Designing Interactive Systems: Processes, Practices, Methods, & Techniques*, DIS '95, pages 219–226, New York, NY, USA, 1995. ACM.

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

[3] Y. Aït-Ameur et al. Vérification et validation formelles de systèmes interactifs fondées sur la preuve : application aux systèmes Multi-Modaux. *JIPS*, 1(1):1–30, September 2010.

[4] P. Antoine et al. Volta: the first all-electric conventional helicopter. In *MEA 2017, More Electric Aircraft* , Bordeaux, France, February 2017.

[5] E. Bainomugisha et al. A survey on reactive programming. *ACM Comput. Surv.*, 45(4):52:1–52:34, August 2013.

[6] J. Bowen and S. Reeves. Modelling safety properties of interactive medical systems. In *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, pages 91–100, New York, NY, USA, 2013. ACM.

[4] En toute rigueur, d'autres vérifications complémentaires sont effectuées : vérifier que l'aiguille des secondes n'est pas affichée de manière transparente, c'est à dire que ce composant graphique n'est pas précédé dans le graphe d'un composant d'opacité ayant une valeur faible ; vérifier que la couleur d'affichage de l'aiguille est différente de celle des composants affichés précédemment.

[7] B. Bérard et al. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer Publishing Company, Incorporated, 1st edition, 2010.

[8] P. Bumbulis et al. Validating properties of component-based graphical user interfaces. In Francois Bodart and Jean Vanderdonckt, editors, *Design, Specification and Verification of Interactive Systems '96*, pages 347–365, Vienna, 1996. Springer Vienna.

[9] S. Chatty et al. Verification of properties of interactive components from their executable code. In *Proceedings of the 7th ACM SIGCHI*, EICS '15, pages 276–285, New York, NY, USA, 2015. ACM.

[10] S. Chatty et al. Designing, developing and verifying interactive components iteratively with djnn. In *ERTS 2016*, TOULOUSE, France, January 2016.

[11] E. M. Clarke, Jr. et al. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[12] P. Cousot. Interprétation abstraite. *Technique et science informatiques*, 19(1):155–164, 2000.

[13] B. J. Cox et al. *Object-Oriented Programming; An Evolutionary Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1991.

[14] B. d'Ausbourg. Using model checking for the automatic validation of user interface systems., 01 1998.

[15] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[16] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[17] M. Hristakeva et al. A survey of object oriented programming languages. Technical report, University of California, Santa Cruz, 2009.

[18] P. Masci et al. Formal verification of medical device user interfaces using pvs. In *Fundamental Approaches to Software Engineering*, pages 200–214, Berlin, Heidelberg, 2014.

[19] D. Navarre et al. A formal description of multimodal interaction techniques for immersive virtual reality applications. In *Proceedings of the 2005 IFIP TC13*, INTERACT'05, pages 170–183, Berlin, Heidelberg, 2005. Springer-Verlag.

[20] O. Nierstrasz. Object-oriented concepts, databases, and applications. chapter A Survey of Object-oriented Concepts, pages 3–21. ACM, New York, NY, USA, 1989.

[21] D. Prun, M. Magnaudet, and S. Chatty. Towards support for verification of adaptative systems with djnn. In *Proceedings of Cognitive 2015*, pages 191–194. IARIA, 03 2015.

[22] W. Reisig. *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*. Springer Publishing Company, Incorporated, 2013.

[23] G. Salvaneschi et al. Debugging for reactive programming. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 796–807, New York, NY, USA, 2016. ACM.

[24] P. Van Roy et al. *Concepts, Techniques, and Models of Computer Programming*. The MIT Press, 1st edition, 2004.

RPP : Preuve automatique de propriétés relationnelles par Self-Composition *

Lionel Blatter[1], Nikolai Kosmatov[1], Pascale Le Gall[2] and Virgile Prevosto[1]

[1] CEA, LIST, Software Reliability and Security Laboratory, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2]Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupelec, Université Paris-Saclay, 91190 Gif-sur-Yvette France
firstname.lastname@centralesupelec.fr

Contexte. Les méthodes de vérification déductive basées sur la logique de Hoare [7] fournissent une approche puissante pour prouver qu'une fonction respecte certaines propriétés. Elles sont généralement couplées à un langage permettant de spécifier formellement les propriétés attendues, en particulier sous forme de contrats de fonction. Un tel contrat comprend des pré- et des post-conditions décrivant respectivement les entrées attendues et le comportement exigé de la fonction.

Problème. Cependant, toutes les propriétés qu'on peut vouloir établir sur un programme donné ne s'expriment pas facilement sous cette forme. En effet, un contrat de fonction décrit le déroulement d'une exécution d'une fonction donnée. Cependant, il est fréquent qu'on veuille parler d'une propriété relationnelle mettant en jeu l'exécution de plusieurs fonctions, ou comparer les résultats d'une même fonction sur différents paramètres. En particulier, des groupes de fonctions sont fréquemment liés par des spécifications algébriques [8] précisant leurs relations. Les contrats de fonction et les méthodes déductives classiques se prêtent mal à la spécification et à la vérification de telles propriétés. Des exemples de telles propriétés incluent :

$$\forall x, y; x <= y => f(x) <= f(y)$$

$$Decrypt(Crypt(Msg, PrivKey), PubKey) = Msg$$

On retrouve un sous ensemble des propriétés relationnelles dans le domaine du test, appelées *propriétés métamorphique* [6], liant plusieurs appels de la même fonction.

Contributions. Afin de répondre à ce manque de support pour les propriétés relationnelles lors de la vérification à base de contrats, nous proposons le plugin FRAMA-C/RPP présenté dans [4]. L'outil réalise un ensemble de transformations automatiques à partir d'un langage de spécification capable d'exprimer des propriétés relationnelles. Le résultat des transformations permet de prouver des propriétés relationnelles et de les utiliser comme hypothèse dans d'autres preuves en utilisant des outils de vérification déductive standard.

*Cet article est un résumé étendu de l'article [4] paru à TACAS 2017.

Approche. Notre langage de spécification est basé sur une extension du langage de spécification ACSL [3] afin de pouvoir exprimer ces propriétés relationnelles et ainsi pouvoir implémenter notre méthode de vérification dans un plugin FRAMA-C. Pour la vérification de propriétés relationnelles, RPP construit une fonction enveloppe, inspirée de la méthode de *Self-composition* [1], qui effectue les appels mis en jeu par la propriété dans un contexte adéquat. On est alors ramené à l'utilisation d'un calcul de plus faible précondition classique, effectué dans notre cas par le greffon FRAMA-C/WP [2].

En outre, afin de pouvoir utiliser les propriétés relationnelles, nous proposons une réécriture des propriétés sous forme axiomatique. WP ajoutera ces définitions dans le contexte des obligations de preuves suivantes.

Dans sa version actuelle [5], RPP supporte des fonctions avec effet de bord et des fonctions récursives. L'outil permet de traiter un large ensemble de propriétés relationnelles de manière automatique et de réutiliser les propriétés dans le contexte d'autres preuves. Il a également été testé avec succès sur différents *benchmarks*, ce qui a permis de valider notre approche.

Travaux futurs. Différents axes de recherches restent à explorer. Premièrement, la gestion des invariants de boucle sous forme d'invariant relationnel permettra de vérifier plus facilement des propriétés sur des fonctions avec boucles. Par ailleurs, la spécification de propriétés sur des appels de fonctions effectués le long d'une même trace d'exécution reste difficile, et pourrait être améliorée. Enfin, une utilisation directe du plugin WP par RPP, sans passer par une transformation syntaxique de code, permettra des gains de performance et une simplification de l'approche.

Références

[1] Barthe, G., D'Argenio, P.R., Rezk, T. : Secure information flow by self-composition. J. Mathematical Structures in Computer Science 21(6), 1207–1252 (2011)

[2] Baudin, P., Bobot, F., Correnson, L., Dargaye, Z. : WP Plugin Manual v1.0 (2017), `http://frama-c.com/download/frama-c-wp-manual.pdf`

[3] Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V. : ACSL : ANSI/ISO C Specification Language, `http://frama-c.com/acsl.html`

[4] Blatter, L., Kosmatov, N., Gall, P.L., Prevosto, V. : RPP : automatic proof of relational properties by self-composition. In : Proceedings of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017). vol. 10205, pp. 391–397. Springer (2017)

[5] Blatter, L., Kosmatov, N., Le Gall, P., Prevosto, V., Petiot, G. : Static and Dynamic Verification of Relational Properties on Self-Composed C Code. In : Proceedings of the 12th Conference on Tests and Proofs (TAP 2018). To appear.

[6] Gotlieb, A., Botella, B. : Automated metamorphic testing. In : Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC 2003). pp. 34–40. IEEE

[7] Hoare, C.A.R. : An axiomatic basis for computer programming. Communications of the ACM 12(10) (1969)

[8] Sannella, D., Tarlecki, A. : Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. An EATCS Series, Springer (2012)

Une analyse précise de caches *LRU* *

Valentin Touzeau[1], Claire Maïza[1], and David Monniaux[1]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP†, VERIMAG, 38000 Grenoble, France

1 Introduction

La latence de la mémoire principale étant généralement bien supérieure à celle du CPU, la plupart des processeurs modernes possèdent des caches matériels. Ces mémoires de petite capacité mais de faible latence permettent en effet d'éviter de coûteux accès à la mémoire principale. Leur présence rend toutefois certaines tâches plus difficiles, comme calculer le pire temps d'exécution du programme ou assurer l'absence de fuites d'informations via des canaux cachés.

Dans ces conditions, il est intéressant d'effectuer des analyses de caches qui calculent statiquement quelles instructions ou données sont dans le cache à chaque point du programme. Cet article est un résumé de [2], où une méthode précise d'analyse de cache est présentée. Celle-ci combine interprétation abstraite et *model-checking* pour classifier les accès mémoire en plusieurs catégories : les accès *Always-Hit* pour lesquels le bloc accédé est toujours garanti d'être dans le cache, les accès *Always-Miss* pour lesquels le bloc n'est assurément pas dans le cache, et les accès *Definitely Unknown* pour lesquels il existe (au moins) un chemin d'exécution menant à un *hit* et un autre menant à un *miss*.

Les analyses existantes pour la politique de remplacement qui nous intéresse (*LRU - Least Recently Used*), appelées analyses *May* et *Must* [1], permettent de prédire avec certitude que certains accès sont *Always-Miss* ou *Always-Hit* respectivement. Toutefois, ces analyses fonctionnent par sous/sur-approximations et ne permettent pas de conclure pour certains blocs. Un bloc peut par exemple être *Always-Hit* sans que l'analyse l'ait classifié comme tel, à cause d'une sur-approximation effectuée.

Pour éliminer cette incertitude, nous procédons en deux étapes. Nous avons d'abord élaboré une analyse similaire aux analyses *May* et *Must*, permettant d'assurer l'existence de chemins menant à un *hit* et à un *miss*. Les blocs restant sont finalement classifiés un par un en utilisant un *model-checker* et un modèle adapté à chacun.

2 Analyse *Definitely Unknown*

Notre première contribution consiste à réduire le nombre d'appels au *model-checker*. De même que les analyses *May* et *Must* permettent de classifier certains blocs

*This work was partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

†Institute of Engineering Univ. Grenoble Alpes

comme *Always-Hit* ou *Always-Miss*, il est possible de classifier certains blocs de façon certaine en *Definitely Unknown* par interprétation abstraite. L'idée est de maintenir pour chaque bloc une borne inférieure et une borne supérieure de sa position dans le cache à la manière des analyses *May* et *Must*. Ces bornes sont garanties d'être valides sur au moins un chemin d'exécution (et non sur l'ensemble des chemins d'exécutions comme pour les analyses *May* et *Must*) et permettent ainsi d'assurer pour certains blocs l'existence d'au moins un chemin menant à un *hit* et d'un autre menant à un *miss*. Ces blocs ne nécessitent donc pas d'appeler le *model-checker* pour être classifiés.

3 Analyse de cache par *Model Checking*

Finalement, notre approche raffine la classification établie par les analyses précédentes en utilisant un *model-checker* pour examiner les accès restants sans avoir recours à ces sous/sur-approximations. Pour cela, on fournit au *model-checker* un modèle du programme et un modèle du cache, et on vérifie un invariant comme « Pour l'accès mémoire donné, le bloc mémoire accédé est toujours dans le cache au moment de l'accès ».

Le modèle du programme consiste en un système de transitions étiquetées par des accès mémoire, construit à partir du graphe de flot de contrôle. Chaque bloc de base du graphe est divisé en sous-blocs correspondant aux frontières entre blocs mémoires.

Pour modéliser le cache efficacement, nous tirons parti d'une propriété intéressante de la politique *LRU* : la position d'un bloc donné dans le cache ne dépend que des blocs qui ont été accédés depuis sa mise en cache. Par exemple, pour prévoir l'éviction du bloc a, il suffit de connaître l'ensemble des blocs plus « jeunes » que a dans le cache : il n'est pas nécessaire de connaître l'ordre dans lesquels ces blocs ont été mis en cache, ni quels blocs étaient dans le cache avant l'accès à a. On construit de cette façon un modèle de cache spécifique à un bloc donné, de taille réduite et permettant de classifier exactement le bloc souhaité.

4 Conclusion

La combinaison d'analyses par interprétation abstraite et par *model-checking* permet d'améliorer la précision des analyses de cache *LRU* pour un coût raisonnable. Les expériences réalisées montrent que cette approche permet d'analyser précisément de gros programmes.

Références

[1] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, Dec. 1999.

[2] V. Touzeau, C. Maïza, D. Monniaux, and J. Reineke. Ascertaining uncertainty for efficient exact cache analysis. In *CAV 2017*, 2017.

Génération automatique des procédures de test

César Ochoa Escudero[1,2], Rémi Delmas[1], Thomas Bochot[2], Matthieu David[2], Virginie Wiels[1]
[1]ONERA, Toulouse, France.
[2]Liebherr-Aerospace Toulouse SAS, Toulouse, France.

Cet article constitue un bref résumé d'un article accepté et présenté au *NASA Formal Methods Symposium 2018*, intitulé *Automatic generation of D0-178 test procedures* [3]

Contexte industriel et problématique

Liebherr-Aerospace Toulouse (LTS) est un fournisseur de *Système de Gestion d'Air* (AMS) pour des avions civils et militaires. Les principales fonctions d'un AMS sont l'alimentation en air, le contrôle de l'environnement et la pressurisation de la cabine. Ainsi, un AMS met en œuvre plusieurs sous-fonctions telles que l'acquisition de données, la consolidation, le contrôle, etc. L'aspect critique pour la sécurité de l'AMS exige que son processus de développement logiciel soit conforme aux directives DO-178 DAL-B. Pour ce faire, le processus de test du logiciel LTS, décrit dans la Fig. 1, est construit sur les notions suivantes:

(1) *High-Level Requirements* (HLRs) sont les exigences, rédigées en langage naturel, qui doivent être mises en œuvre et testées;

(2) *Low-Level Requirements* (LLRs) sont des spécifications détaillées du logiciel, dont la plupart sont formalisés sous forme de modèles SCADE [1] (flot de données). Ces LLRs forment ensemble le *modèle SCADE* du logiciel applicatif, à partir duquel le code objet exécutable est automatiquement généré. Un HLR peut être raffiné dans un ou plusieurs LLRs, et un LLR peut supporter plus d'un HLR;

(3) *Test Cases* (TCs) sont des *specifications de test* déclaratives rédigées en langage naturel. Les conditions de test et les résultats prévus sont exprimés en termes d'identificateurs du HLR. Tout HLR doit être couvert par un ou plusieurs TCs. Les critères de couverture sont spécifiés dans le *Software Test Standards* (STS), contenant une procédure opérationnelle assurant la conformité au DO-178;

(4) *Test Procedures* (TPs) sont des *implémentations exécutables* de TCs, étant destinés à tester l'exécution du logiciel applicatif sur l'ordinateur cible. Un TP implémente plusieurs TCs en séquence, en pilotant les variables d'entrée d'interface du modèle du logiciel en vérifiant les résultats spécifiés dans les TCs.

La difficulté de la mise en œuvre des TPs est due au fait qu'ils sont limités à piloter que les variables d'entrée d'interface du logiciel. Afin d'établir les conditions de test requises sur les flots de données internes, il est nécessaire d'effectuer un raisonnement de rétro-propagation à travers la structure de flots de données pour identifier les variables d'interface d'intérêt et leurs instanciations pour un TP donné. Cette tâche est très consommatrice de ressources et est aujourd'hui réalisée par des ingénieurs. Par ailleurs, cette dépendance de la structure de flots de données *fragilise* les TPs: un TP pour un HLR donné peut être invalidé par des évolutions de HLRs/LLRs apparemment non liées, ou par une évolution non fonctionnelle du modèle, par exemple par sous-échantillonnage de ses sous-fonctions.

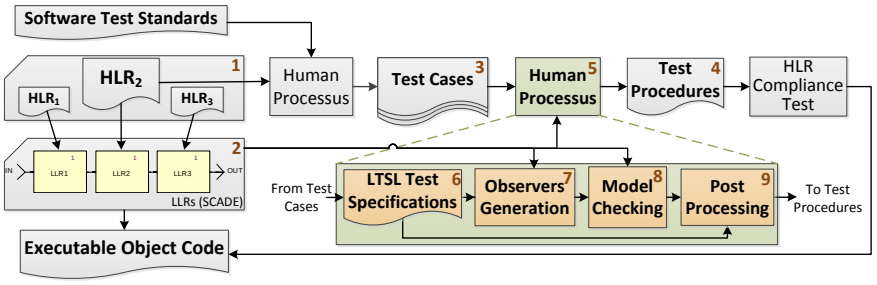


Figure 1: Processus de test du logiciel embarqué chez LTS

Approche présentée

L'approche présentée dans [3] vise à réduire l'effort humain nécessaire à la production et au maintien des TPs dans le cadre des évolutions des HLRs/LLRs correspondants. Pour ce faire, le processus de production de TP – (5) dans la Fig. 1 – est remplacé pour les sous processus suivants:

(6) Rédaction des TCs en *Liebherr Test Specification Language* (LTSL), qui est un langage formel et déclaratif, utilisé comme une étape de formalisation syntaxique et sémantique des spécifications de TCs. Néanmoins, les conditions de test et les résultats prévus sont exprimés en termes d'identificateurs du LLR. Une sémantique de trace formelle pour LTSL est définie, c'est-à-dire les conditions formelles selon lesquelles une trace d'exécution du logiciel applicatif satisfait une spécification LTSL, y compris les latences entre TCs consécutifs;

(7) Observers Generation, est la génération automatique d'observateurs SCADE synchrones à partir des spécifications LTSL. L'observateur généré partage les entrées du LLR sous test ainsi qu'un flot de sortie `Incomplete` dont la valeur par défaut est `vrai`. Un observateur se présente sous la forme d'un automate avec un état initial, un état final et un nombre d'états intermédiaires dépendant de la quantité des TCs. Il est aussi fourni d'un compteur permettant de calculer le nombre de cycles étant passé dans chaque état, sur la base de l'horloge du LLR. Une trace est acceptée pour l'automate si et seulement si l'état final est atteint en instanciant `Incomplete` à `false`. Autrement dit, si elle réalise toutes les conditions spécifiées dans la spécification LTSL en séquence;

(8) Model-Checking (Systerel S3 [2]), est utilisé pour réfuter le flot `Incomplete` dans un modèle SCADE instrumenté avec un observateur synchrone, donnant des traces pour les variables d'entrée d'interface du logiciel qui satisfont aux conditions de test exprimées dans la spécification LTSL;

(9) Post-Processing, est l'enrichissement de la trace produite par le model-checker avec des conditions de checks, c'est-à-dire des résultats attendus des tests – ils sont issus de la spécification LTSL – en obtenant les implémentations de TP.

Références

[1] SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite`.

[2] Systerel Smart Solver. `http://www.systerel.fr/innovation/produits/systerel-smart-solver`.

[3] César Ochoa Escudero, Rémi Delmas, Thomas Bochot, Matthieu David, and Virginie Wiels. Automatic generation of DO-178 test procedures. In *NASA Formal Methods - 10th International Symposium, NFM 2018, Newport News, VA, USA, April 17-19, 2018, Proceedings*, pages 399–415, 2018.

Validation Formelle d'Architectures Logicielles Basée sur des Patrons de Sécurité

Fadi Obeid Philippe Dhaussy
Lab-STICC CNRS, UMR 6285 Ensta Bretagne
fadi.obeid@ensta-bretagne.org
philippe.dhaussy@ensta-bretagne.fr

May 4, 2018

Abstract

Les modèles de patrons de sécurité ont été proposés comme des solutions méthodologiques permettant de modéliser des mécanismes qui répondent à des problèmes de sécurité récurrents. Ceux-ci sont décrits dans la littérature et peuvent être exploités dans différents contextes de modélisation. Lors de leur intégration au sein d'un modèle d'architecture, ces modèles de patrons sont à adapter à ses spécificités. Une fois les modèles de patrons intégrés, il est nécessaire de valider formellement le résultat de cette intégration au regard des propriétés fonctionnelles de l'architecture initiale qui doivent être préservées, et au regard des propriétés formelles de sécurité associées aux patrons. Dans notre travail, nous exploitons une technique de model-checking pour la vérification des propriétés. Nous cherchons à exploiter notre approche dans le cadre de la modélisation des architectures SCADA.

Mots Clés: Patron de Sécurité, Model Checking, SCADA

1 Introduction

Un modèle de patron de sécurité est une solution méthodologique réutilisable pour un problème de sécurité récurrent. Divers patrons ont été décrits dans la littérature [1]. Associés à ces modèles, certains auteurs ont proposé des méthodologies pour leur mise en œuvre et leur intégration dans des modèles d'architectures logicielles. La description d'un patron inclut une description de ses propriétés qui doivent être respectées lors de leur intégration dans l'architecture à sécuriser. Il offre au développeur un guide sur la façon d'utiliser et de mettre en œuvre une solution de sécurité en fonction du contexte précis d'un problème de sécurité et d'un modèle d'architecture qu'il souhaite sécuriser. De ce fait, les patrons facilitent la communication entre experts et non-experts du domaine de la sécurité. L'intégration d'un modèle de patron dans un modèle d'architecture impose des adaptations lors de la conception de l'architecture à sécuriser. Dans ce travail, nous étudions une méthodologie et les concepts pour intégrer des modèles de patrons, conformes à des politiques et des exigences de sécurité. Les modèles d'architectures générés doivent pouvoir être validés formellement au regard des exigences attendues. Pour l'instant nous raisonnons sur des modèles d'architectures génériques dont nous décrivons les caractéristiques section 2. Sur la base de ces types de modèle, des

Étude financée par la DGA-Maîtrise de l'Information et Brest Métropôle Océanne(BMO).

bibliothèques de patrons de la littérature et d'une politique de sécurité, nous cherchons à générer des modèles sécurisés. Ceux-ci doivent pouvoir être exploités pour des vérifications formelles des propriétés fonctionnelles du modèle et des propriétés de sécurité. Bien que nous raisonnons aujourd'hui sur des modèles abstraits d'architectures, nous pensons que cette approche peut s'adapter à des cas de modèles de systèmes de supervision et d'acquisition de données (SCADA). Ces systèmes contrôlent de nombreuses infrastructures et procédés critiques, telles que les installations nucléaires et les usines chimiques. Dans un contexte de cyber sécurité, des efforts conséquents doivent être portés sur la conception des mécanismes de protection. Les approches de modélisation et de validation formelles ont tout à fait leur place dans ce contexte. Dans la suite de ce papier, nous donnons, section 2, quelques références de l'état de l'art et nous présentons les principes de formalisation des patrons étudiés. Nous illustrons en section 3, un patron particulier de type *Single Acces Point*. Nous précisons les types de modèles d'architectures sur lesquels portent nos travaux actuels. Ensuite, section 4, nous décrivons notre approche pour l'intégration des modèles de patrons et les principes de validation de propriétés. Nous concluons, section 5, par un bilan et les perspectives de ce travail.

2 Les patrons de sécurité

Un état de l'art sur les modèles de patrons de sécurité peut être trouvé entre autre dans [2, 3]. De nombreux efforts de recherche ont été faits sur la vérification des modèles de conception intégrant des combinaisons de patrons de sécurité. D'un point de vue modélisation, les travaux décrits dans [4] nous ont inspirés dans le sens où des schémas de modélisation UML sont décrits, intégrant des mécanismes de sécurité dans des modèles applicatifs simples. Nous souhaitons étendre ce travail pour des modèles d'architectures et des politiques de sécurité plus complexes et en prenant en compte non seulement les propriétés de sécurité associées aux patrons mais également les propriétés fonctionnelles initiales des architectures. De nombreuses études ont traité de la sécurité dans les SCADA. Ces études fournissent des voies de recherche pour différentes approches pour améliorer la sécurité dans ces systèmes.

La sécurisation d'une architecture implique en général l'intégration de plusieurs mécanismes de sécurité. La gestion de l'ensemble d'une politique de sécurité est donc partagée entre plusieurs dispositifs. Dans une approche orientée modèles, plusieurs modèles de patrons sont impliqués dans la construction d'un modèle d'architecture. Nous précisons ici que les types de modèle que nous considérons dans notre travail sont organisés comme un ensemble d'entités (ou composants), communiquant par des liens de communication ($fifo$ ou $ports\ synchrones$). Chaque entité a un comportement décrit par un automate. Sur les liens de communication, sont véhiculés des messages comportant des données (requêtes ou réponses) impliquant des accès à des ressources (mémoires) détenues par les entités. L'architecture interagit avec un environnement dans lequel des entités clientes sollicitent les services rendus par l'architecture. Le principe de la sécurisation

d'une telle architecture est de déployer les mécanismes de sécurité au sein de ce réseau d'entités.

Compte tenu de l'avancée de nos travaux, nous nous sommes focalisés sur quatre patrons que nous pourrons étendre par la suite sur les mêmes bases méthodologiques. Le patron $SingleAccessPoint(SAP)$ concentre les ports d'accès permettant de gérer les contrôles de sécurité au travers d'un seul point d'accès. Le patron $CheckPoint(CHP)$ est dédié à réaliser certaines vérifications relatives à la *politique* de sécurité qui est souhaitée être implantée. Le patron $Authorization$ complète les contrôles de sécurité consacrée à des droits d'accès. Le patron $FireWall$ implante des mesures de sécurité consacrées aux restrictions et filtrages des messages. Ces patrons intègrent les entités et concourent ensemble aux différents aspects liés aux exigences de sécurité devant être implantées. Dans la figure 1.a, nous illustrons la cohabitation entre deux patrons SAP et CHP au sein d'un composant.

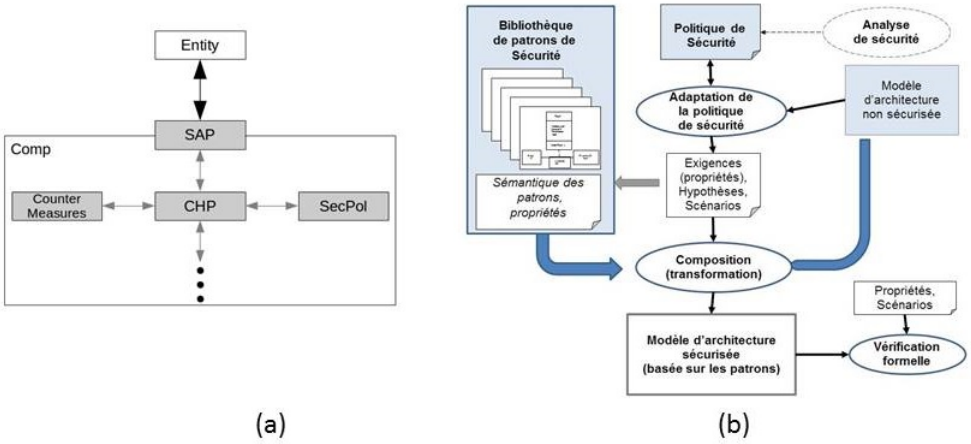


Figure 1: Processus d'intégration de patrons au sein d'une architecture.

3 Formalisation du patron Single Access Point

En guise d'illustration, nous présentons ici plus en détail le mécanisme de type SAP. Le patron SAP, introduit par [5], a pour objectif d'implanter un point d'accès unique afin d'optimiser le contrôle et la surveillance des messages d'entrées à un composant ou un ensemble de composants. Toute requête entrante est acheminée via un canal unique, où la surveillance peut être effectuée aisément. Le point d'accès unique est un endroit approprié pour éventuellement capturer un historique (Log) des accès. Ces données peuvent être utiles pour vérifier le séquencement des accès en fonction de leurs droits. SAP peut aussi avoir pour rôle de vérifier et d'apposer des signatures sur des requêtes. Il peut également avoir pour mission le contrôle d'accès à un ensemble de ressources localisées au sein d'un composant ou le contrôle d'accès à un ensemble de composants. Lors d'une demande d'accès à une ressource res détenue par un composant c, SAP implanté dans un composant

c vérifie si res est accessible au sein de c. Dans le cas d'une demande de transfert d'une requête à un autre composant c', le contrôle vérifie si c' est accessible depuis c. La mise en œuvre de SAP peut impliquer d'autres patrons, tel que CHP (figure 1.a), pour réaliser des contrôles sur les données transitant par ce point.

Une politique de sécurité est décrite par des objectifs de sécurité qui se déclinent en un ensemble de propriétés de sécurité qui peuvent être définies sous une forme littéraire ou formelle. Ces propriétés sont regroupées en trois grandes classes: confidentialité, intégrité et disponibilité. Chaque propriété représente les conditions que le système doit respecter. Une définition incorrecte ou l'application partielle d'une politique, peut entraîner le système dans un état non sécurisé. Parmi les exigences de SAP, voici deux exemples. La communication entre les composants internes n'est pas divulguée aux entités externes (*Confidentialité*). Les requêtes internes ne peuvent pas être modifiées par des entités externes (*Intégrité*). Dans notre travail, nous avons cherché à formaliser l'ensemble des exigences caractérisant chaque patron. Un exemple de propriété formelle est illustrée ci-dessous par l'invariant (1). Elle exprime que tout accès à une ressource (res) a été contrôlée auparavant ($checked$).

$$\begin{array}{l} \forall\, c \in \mathcal{S}ap,\, \forall\, e \in \mathcal{E}nt,\, \forall\, res \in \mathcal{R}es, \\ \Box\,(accessed\,(c,\, e,\, res) \Rightarrow checked_c\,(c,\, e,\, res)) \end{array} \quad (1)$$

Dans (1), $\mathcal{S}ap$ est l'ensemble des composants c détenant le mécanisme de type SAP, $\mathcal{E}nt$ est l'ensemble des entités demandant l'accès à la ressource res. $accessed\ (c,\ e,\ res)$ est le prédicat signifiant que e a accédé à la ressource res localisée sur c. $checked_c\ (c,\ e,\ res)$ est le prédicat signifiant que la requête de demande d'accès par e à la ressource res a été contrôlée. Lors de notre étude, nous avons ainsi formalisé l'ensemble des propriétés pour les quatre patrons étudiés, sous la forme de propriétés invariantes, comme (1), ou de propriétés de type vivacité. Nous indiquons dans la suite du papier quelques principes que nous proposons, d'une part, pour intégrer les patrons au sein d'un modèle d'architecture et, d'autre part, pour vérifier les propriétés sur le modèle suite à cette intégration.

4 Processus d'intégration des patrons

Dans notre approche, nous proposons un processus d'intégration des patrons dans un modèle d'architecture. Le processus invoque des éléments (en grisé sur la figure 1.b) pris en entrée de ce processus. Tout d'abord, nous considérons un modèle d'architecture non sécurisée du type décrit précédemment. Ensuite, une politique de sécurité est choisie. Elle est décrite sous la forme d'exigences de sécurité. La mise en œuvre de cette politique implique le choix de patrons de sécurité. Le choix des patrons est effectué pour l'instant par le concepteur au regard des exigences de sécurité qu'il souhaite implanter dans son modèle d'architecture. Les exigences de sécurité sont ensuite spécifiées sous la forme d'un ensemble de propriétés formelles adaptées au modèle de l'architecture à sécuriser. La réécriture des exigences constituant la politique de sécurité en propriétés formelles est basée

sur l'ensemble des propriétés formalisées pour chaque patron. En complément des propriétés, un ensemble de scénarios sont identifiés. Ils décrivent les scénarios nominaux d'interaction entre l'architecture et son environnement ainsi que les scénarios d'attaque contre lesquels l'architecture est supposée se protéger. Pour intégrer les mécanismes de sécurité sur chaque entité de l'architecture, nous avons proposé des principes de transformation des automates de l'architecture initiale. Les transformations sont basées sur l'ajout d'états et de transitions dans les automates et des appels de fonctions associées aux différents patrons.

Sur le nouveau modèle issu de la transformation, une validation formelle doit être réalisée. Nous choisissons, dans notre approche, de la réaliser à l'aide d'une technique de $model - checking$. Celle-ci permet de vérifier, lors d'une simulation exhaustive, toutes les propriétés formelles identifiées sur ce nouveau modèle. Les propriétés sont de deux ordres : les propriétés initiales de l'architecture non sécurisée et les propriétés ajoutées par les patrons qui correspondent aux contraintes de sécurité. Pour cette vérification, les scénarios d'emploi nominaux seront exploités pour valider le fonctionnement de l'architecture. Mais aussi, des scénarios simulant des attaques doivent être exploités pour pouvoir tester la robustesse de l'architecture. Actuellement ces derniers sont spécifiés pour décrire des interactions simples avec les fifos internes de l'architecture. Des travaux en cours seront exploités pour établir des politiques d'attaques plus sophistiquées.

Nous avons mené des expérimentations sur différents modèles de réseau d'automates. Pour mener les vérifications formelles de propriétés, nous générons les modèles au format FIACRE[1] qui permettent de spécifier les comportements de notre architecture ainsi que ses interactions avec son environnement. Le but est de prouver que le modèle d'architecture généré, et donc doté des mécanismes basés sur les patrons de sécurité, respecte les exigences décrites dans la politique de sécurité choisie au regard d'attaques. Dans notre travail, les vérifications formelles sont réalisées avec l'outillage OBP[2] qui a été développé dans l'équipe [6]. Nous formalisons les exigences de sécurité à vérifier avec le langage CDL associé à cet outil. Les propriétés formelles de sûreté sont des invariants ou des observateurs de rejet. Les propriétés de vivacité sont spécifiées par des formules logiques LTL. Celles-ci seront bientôt prises en charge par une extension d'OBP ($Plug$), en cours de développement. Des scénarios nominaux, ainsi que des scénarios d'attaques peuvent aussi être décrits en CDL, ce qui permet de profiter des travaux sur la réduction de la complexité lors des explorations des modèles [7], ce qui est un aspect problématique bien connu du $model - checking$.

5 Conclusion

Dans ce travail, nous étudions les principes de transformation de modèles d'architectures basés sur l'intégration de mécanismes ou patrons de sécurité. Notre objectif est de

[1] Langage défini dans le cadre du projet TopCased (http://www.topcased.org).

[2] OBP est accessible librement sur http://www.obpcdl.org ainsi que les codes (Fiacre et CDL) de quelques modèles d'expérimentation.

fournir des outils méthodologiques pour valider formellement un modèle d'architecture sécurisée devant respecter une politique de sécurité donnée. Nous avons expérimenté des choix d'implantation de patrons au sein des modèles qui peuvent être optimisés. Un travail complémentaire doit être conduit pour optimiser ces implantations et identifier les bons critères de comparaison entre les différentes stratégies.

Notre approche implique une formalisation complète des patrons de sécurité que nous souhaitons prendre en compte. Notre travail est à poursuivre en prenant en compte d'autres patrons de la littérature pour lesquels les formalisations se sont pas disponibles. Aussi, des combinaisons de patrons doivent être étudiées pour répondre à des exigences de sécurité plus sophistiquées. Nous constatons, sur ce point, que la modélisation des attaques est fondamentale car ce sont elles qui dirigent le processus de sécurisation. Dans notre travail, nous sommes donc confrontés à une problématique où beaucoup de paramètres sont à prendre en compte pour répondre au besoin de sécurisation des architectures. Les perspectives de ce travail concernent plusieurs axes à prendre en compte dans différentes directions. Ils concernent la prise en compte des politiques de sécurité qui peuvent être complexes, être définies de manière dynamique, porter sur des niveaux différents des architectures (accès aux ressources, aux données, aux canaux de communication). Ces politiques doivent reposer sur une formalisation indispensable à partir de laquelle, un processus de génération automatique de modèles d'architectures peut être envisagé.

References

[1] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.

[2] Nobukazu Yoshioka, Hironori Washizaki, and Katsuhisa Maruyama. A survey on security patterns. *Progress in informatics*, 5(5):35–47, 2008.

[3] Eduardo B Fernandez and Rouyi Pan. A pattern language for security models. In *proceedings of PLOP*, volume 1, 2001.

[4] Ronald Wassermann and Betty HC Cheng. Security patterns. In *Michigan State University, PLoP Conf.* Citeseer, 2003.

[5] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1997.

[6] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012.

[7] Luka Le Roux Ciprian Teodorov, Philippe Dhaussy. Environment-driven reachability for timed systems : Safety verification of an aircraft landing gear system. *Int. Software Tools for Technology Transfer (STTT)*, 2016.

THEMIS: A Tool for the Design, Development, and Analysis of Decentralized Monitoring Algorithms

Antoine El-Hokayem, Yliès Falcone
Univ. Grenoble Alpes, Inria, CNRS, Laboratoire d'Informatique de Grenoble,
F-38000 Grenoble, France
firstname.lastname@univ-grenoble-alpes.fr

Abstract

THEMIS is a tool to facilitate the design, development, and analysis of decentralized monitoring algorithms; developed using Java and AspectJ. It consists of a library and command-line tools. THEMIS provides an API, data structures and measures for decentralized monitoring. These building blocks can be reused or extended to modify existing algorithms, design new algorithms, and elaborate new approaches to assess existing algorithms.

1 Overview

Runtime Verification Runtime Verification (RV) [1, 10, 12] is a lightweight formal method which consists in verifying that a run of a system is correct with respect to a user-provided specification. The specification formalizes the expected behavior of the system. Typically, the system is considered as a blackbox that feeds events to a monitor. An *observation* is an association of an atomic proposition describing abstract operations or states in the system with the boolean $true$ or $false$. A set of observations is called an *event*, a sequence of events is referred to as a *trace*. A monitor receives a trace and emits a verdict indicating the compliance of the system to the specification. We focus on methods to monitor decentralized systems, that is, systems with multiple components having no central observation point. In decentralized systems, monitors have a partial view of the system and need to account for communication [4] and computation. A decentralized monitoring algorithm consists in defining the topology of monitors and specifying their behavior and communication.

Design Goals THEMIS [7, 9] is written in Java, uses AspectJ [11] and is provided as a library with a set of command-line tools. The main design goal of THEMIS is to provide a general API for decentralized monitoring by supplying an environment that accounts for changes at all levels: traces, specification, monitoring logic. By doing so, we allow for new approaches implementing the API to

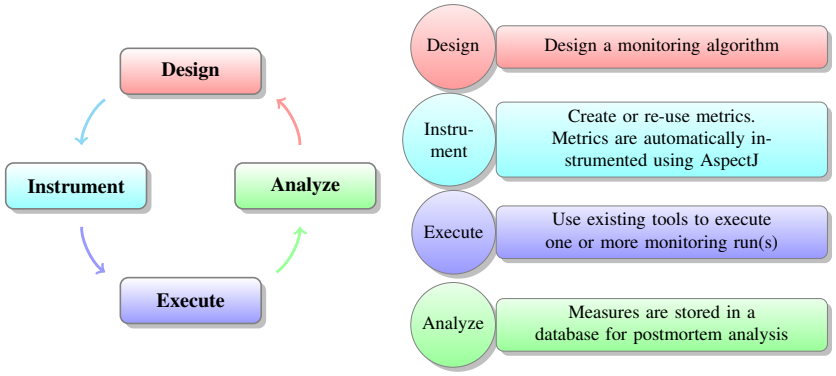


Figure 1: Using the THEMIS Framework

benefit from all existing metrics and analysis. This allows metrics to be collected at API level, for example the metric `messages sent` could be simply reused if new algorithms exchange messages. Following this goal, we also aimed that our measures be stored per run in a database allowing analysis and benchmarking to be reduced to querying the database. This effectively separates the analysis from the monitoring, and allows third-party tools to be used to explore and analyze the data. Another important design goal is reproducibility. We wanted to minimize the effort of re-running older simulations or comparing new approaches with older ones. This is reflected with the `Experiment` command-line tool which, in short, allows users to bundle all traces, specifications and algorithms. Since metrics are designed to work at the API level and data structures, any algorithm using the same building blocks can be measured similarly without added effort. New algorithms or variants of older algorithms can be easily compared against older ones with the same data and measures. By accomplishing these two primary goals, we minimize the overhead needed to design new algorithms and study them, and let researchers focus on the algorithm and the monitoring itself. Finally, THEMIS is designed to introduce decentralized specifications [6]. That is, having different specifications for different components in the system. THEMIS encompasses existing approaches [2, 4] that focus on presenting one global formula of the system from which they derive multiple specifications, and in addition supports any decentralized specification.

Methodology To assess the behavior of a monitoring algorithm, we identify four phases (Figure 1): design, instrument, execute, and analyze. The design phase consists in elaborating a monitoring algorithm. THEMIS generalizes the monitoring steps and provides an API to describe the operations. These operations are used as building-blocks to assemble an algorithm. The instrument phase consists in the definition of measures. Measures are instrumented into THEMIS and the algorithms at run-time and operate on the API and data structures. The execute phase consists in using the THEMIS set of tools to run simulations of the monitoring algorithms and record the measures. The analyze phase consists in using the recorded measures to study, compare, and refine the algorithm.

2 The THEMIS Framework

Monitoring A monitoring algorithm has two phases: setup and monitor. In the first phase, the algorithm creates and initializes the monitors, connects them to each other so they can communicate, and attaches them to components so they receive the observations generated by components. In the second phase, each monitor receives observations at a timestamp based on the component it is attached to. The monitor can then perform some computation, communicate with other monitors, abort monitoring or report a verdict. To accomplish this we provide the two interfaces `MonitoringAlgorithm` and `Monitor`. In the basic use case, `MonitoringAlgorithm` is expected to provide the `setup()` method, which does the setup phase of the algorithm, and returns a map specifying monitors and their ids. A monitor has to implement the `monitor()` method for the monitoring logic, and the `reset()` method to reset its state when executing multiple runs. The method `monitor()` provides the monitor with a timestamp and a memory of observations at that timestamp based on the monitored component. The provided flow of the base `MonitoringAlgorithm` is similar to the Bulk Synchronous Parallel (BSP) [13] model. In the BSP model, all processes execute a computation phase, then, they communicate and finally synchronize. The timestamp is associated with the round number. The monitoring phase begins by setting up the monitor network. Then, for each timestamp, the observations are gathered from the trace, then all monitors execute their `monitor()` method.

Measuring `THEMIS` uses AspectJ to record measures of a metric for a given algorithm. Writing a metric for an algorithm consists in using AspectJ's aspects to intercept the points in the execution. To simplify the task, `THEMIS` provides the base aspect `Instrumentation` and the classes `Measure` and `MeasureFunction`. The Instrumentation aspect already defines basic pointcuts and triggers simple methods upon reaching them.When executing the monitoring algorithms, metrics are instrumented into the code at load-time using AspectJ.

Traces The provided tools and algorithms use a simple format to represent components and the traces of events they receive. The components are named alphabetically starting with a (for example: `a`, `b`, `c`). The observations bound to the components are prefixed by the component and followed with a number starting from 0 (for example: `a0`, `a1`, `a2`, `b0`, `b1`). The trace consists of multiple files, prefixed by the trace ID and suffixed by the component name. A trace for two components a and b consists of two files: `1-a.trace` and `1-b.trace`. Each line in the file represents an event. Currently, we generate random traces for simulation. Moreover, traces can be provided by an external program using a stream interface.

Specifications A top level specification is by default a decentralized specification. A decentralized specification is a collection of specifications. Specifications

Listing 1 An LTL Specification

```
<specifications>
  <specification id="root"
             class="uga.corse.themis.monitoring.SpecLTL">
    <setLTL><![CDATA[XXXX(!a0 | (b1 U G(a0 & b0 & c0)))]]></setLTL>
  </specification>
</specifications>
```

are passed to a monitoring algorithm as a `Map`, where each specification is identified by a key. Each specification must provide two attributes: an id and a class name. The id is a string name for the specification, it is used by the algorithm during the `setup` phase. The class name is a string representing a full class name of the specification class. Listing 1 shows an example of LTL specification. It is given the name `root` and is loaded by the class `SpecLTL`. `THEMIS` handles both LTL and Automata specifications.

Execution History Encoding (EHE) The execution of the specification automaton, is in fact, the process of monitoring, upon running the trace, the reached state determines the verdict. In a decentralized system, a component receives only local observations, it generally does not have enough information to determine the state at a given timestamp. Typically, when sufficient information is shared between various components, it is possible to know the state reached in the automaton. The `EHE` is a data structure that encodes the execution of the automaton using boolean expressions and ensures strong eventual consistency in determining the state reached in the execution. Formal details are in [6].

Command-line Tools The `THEMIS` framework is bundled with several tools to execute monitoring. The `Run` tool takes as input the name of a monitoring algorithm class, a specification file, the number of components, the length of a trace (in order to timeout), a traces directory, and one or more traces to read and simulate the run of the algorithm on the given trace and specification. Upon finishing the execution, the measures will be printed. The `Experiment` tool is designed to execute a set of runs packaged as an experiment. An experiment is a folder containing all necessary files to define sets of parameters, traces and specifications.

Future Development We plan to extend the `THEMIS` framework with additional metrics to analyze more aspects of decentralized monitoring algorithms. Furthermore, we develop a fully distributed and parallel version of THEMIS, so we can extend functionality from simulation to monitoring.

Resources This paper presents an excerpt from [7] to highlight the `THEMIS` tool. Additional resources can be found in [6] for the theory, the tool repository [9], and the demo repository [8]. An example of using the tool to compare variants of an algorithm is provided in Appendix A.

References

[1] Bartocci, E., Falcone, Y., Bonakdarpour, B., Colombo, C., Decker, N., Havelund, K., Joshi, Y., Klaedtke, F., Milewicz, R., Reger, G., Rosu, G., Signoles, J., Thoma, D., Zalinescu, E., Zhang, Y.: First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. International Journal on Software Tools for Technology Transfer pp. 1–40 (2017)

[2] Basin, D.A., Klaedtke, F., Zalinescu, E.: Failure-aware runtime verification of distributed systems. In: FSTTCS 2015. LIPIcs, vol. 45, pp. 590–603. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2015)

[3] Bauer, A., Leucker, M., Schallhart, C.: Runtime verification for LTL and TLTL. ACM Trans. Softw. Eng. Methodol. 20(4), 14 (2011)

[4] Colombo, C., Falcone, Y.: Organising LTL monitors over distributed systems with a global clock. Formal Methods in System Design 49(1-2), 109–158 (2016)

[5] Duret-Lutz, A.: Manipulating LTL formulas using Spot 1.0. In: Proceedings of the 11th International Symposium on Automated Technology for Verification and Analysis (ATVA'13). Lecture Notes in Computer Science, vol. 8172, pp. 442–445. Springer, Hanoi, Vietnam (Oct 2013)

[6] El-Hokayem, A., Falcone, Y.: Monitoring decentralized specifications. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 125–135. ISSTA 2017, ACM, New York, NY, USA (2017)

[7] El-Hokayem, A., Falcone, Y.: THEMIS: a tool for decentralized monitoring algorithms. In: Proceedings of 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17-DEMOS), Santa Barbara, CA, USA, July 2017 (2017)

[8] El-Hokayem, A., Falcone, Y.: THEMIS demonstration repository (2017), `https://gitlab.inria.fr/monitoring/themis-demo`

[9] El-Hokayem, A., Falcone, Y.: THEMIS website (2017), `https://gitlab.inria.fr/monitoring/themis`

[10] Falcone, Y., Havelund, K., Reger, G.: A tutorial on runtime verification. In: Engineering Dependable Software Systems, NATO science for peace and security series, d: information and communication security, vol. 34, pp. 141–175. ios press (2013)

[11] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: ECOOP 2001, Proceedings. Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer (2001)

[12] Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. 78(5), 293–303 (2009)

[13] Valiant, L.G.: A bridging model for parallel computation. Commun. ACM 33(8), 103–111 (Aug 1990)

A The Migration Algorithm

The migration algorithm is a decentralized monitoring algorithm where information is passed throughout the components to eventually verify the specification. In our setup, the migration algorithm will assign a monitor per component. These monitors are strongly connected; each monitor is connected to all other monitors. The monitors are either active or inactive. Active monitors are monitors that seek to find a verdict while inactive monitors are idle, waiting for other monitors to send them their `EHE`. Both active and inactive monitors store the observations they receive in their memory. However, only active monitors will update the expressions in their `EHE`. After expressions are updated, active monitors will determine which other monitors should be sent the `EHE` to continue monitoring using a method `getNext`. In this demonstration, we use two different implementations of `getNext`. The first chooses the next monitor by cycling through all monitors in a round-robin fashion. The second chooses the monitor based on the earliest observation missing to evaluate the `EHE` [4].

Setup Listing 2 shows the setup phase. We first make sure to convert the main specification (identified by `root`) to an automaton specification (line 2-3). Next, we create the monitors map, and generate as many monitors as components, giving them ids starting from zero. Each monitor is then attached to a component (line 8) to receive observations on that component. We note that in the default implementation of communication, all monitors are connected to each other, therefore there is no need to connect monitors to each other.

Monitor The monitoring logic of the monitor is shown in Listing 3. First, the monitor updates its memory by adding the new observations (line 3). Then, the monitor checks if it received anything and merges the received `EHE`. If the monitor receives anything then they become active. Upon receiving observations the monitor then updates its `EHE` and checks for a verdict. If a verdict has changed (line 9) and the verdict reached is a final verdict (line 11), then we report it and remove unnecessary entries in the `EHE` (line 13). We then determine the id of the new monitor to send the `EHE` to (line 15). The two implementations of the method `getNext()` determine the variants of Migration. If it is a different monitor id,

Listing 2 Migration Setup Phase

```
1  protected Map<Integer, ? extends Monitor> setup() {
2    config.getSpec().put("root",
3        Convert.makeAutomataSpec(config.getSpec().get("root")));
4    Map<Integer, Monitor> mons = new HashMap<Integer, Monitor>();
5    Integer i = 0;
6    for(Component comp : config.getComponents()) {
7      MonMigrate mon = new MonMigrate(i);
8      attachMonitor(comp, mon);
9      mons.put(i, mon);
10     i++;
11   }
12   return mons;
13 }
```

then we must migrate, the `EHE` is sent to the next monitor (line 18) and the current monitor is rendered inactive (line 19).

Measures To evaluate the behavior of the migration algorithm we use the communication as an example metric. We measure the number of messages sent and the size of the messages. The number of messages indicates the number of migrations performed, while the size of the messages indicates how big the `EHE` is. The number of messages is shown in Listing 4. We begin by adding the measure and initializing with zero (line 2). We intercept the message sending using AspectJ (line 4) and update our measure (line 5).

Analyze We aim to compare the communication patterns of the two variants. To do so, we execute 2,934,400 runs to generate a database of the measures. We use 200 traces of 100 events per component, we associate with each component 2 observations. We vary the number of components between 3 and 5, and ensure that for each number we have at least 1,000 formulae that reference all components. Specifications are generated as random LTL formulae using `randltl` from `Spot` [5], then converted to automata using `ltl2mon` [3]. Figure 2 displays our example query on the database to retrieve the communication measures, where column alg (resp. comps, avg(msg_num), avg(msg_data), count) indicates the algorithm (resp. number of components, average number of messages, average size of messages, the number of runs). MigrationRR stands for Migration with round robin. We can see that the naive round-robin variant has both a higher number of messages and more communication. The smaller number of messages indicates that less migrations are performed overall.

Listing 3 Migration Monitor

```
public void monitor(int t, Memory<Atom> observations)
throws ReportVerdict, ExceptionStopMonitoring {
    m.merge(observations);
    if(receive()) isMonitoring = true;
    if(isMonitoring) {
      if(!observations.isEmpty())
        autRep.tick();
      boolean b = autRep.update(m, -1);
      if(b) {
        VerdictTimed v = autRep.scanVerdict();
        if(v.isFinal())
          throw new ReportVerdict(v.getVerdict(), t);
        autRep.dropResolved();
      }
      int next  = getNext();
      if(next != getID()) {
        Representation toSend = autRep.sliceLive();
        send(next, new RepresentationPacket(toSend));
        isMonitoring = false;
      }
    }
  }
```

Listing 4 Measuring Communication

```
protected void setupRun(MonitoringAlgorithm alg) {
 addMeasure(new
  ↪  Measure("msg_num","Msgs",0L,Measures.addLong));
}
after(Integer to, Message m) : Commons.sendMessage(to, m) {
 update("msg_num" , 1L);
}
```

```
SELECT alg, comps, avg(msg_num), avg(msg_data), count(*)
FROM bench WHERE alg in ('Migration', 'MigrationRR')
GROUP BY alg, comps
```

	alg	comps	avg(msg_num)	avg(msg_data)	count(*)
1	Migration	3	2.04226336011177	267.8458714635	572600
2	Migration	4	2.16402472527473	668.129401098901	364000
3	Migration	5	3.33806822465134	3954.09705050886	530600
4	MigrationRR	3	32.7222301781348	482.572275585051	572600
5	MigrationRR	4	31.8533351648352	932.708425824176	364000
6	MigrationRR	5	19.2345269506219	4361.30746324915	530600

Figure 2: Example Database Querying

Verde - a Tool for Interactive Runtime Verification

Raphaël Jakse, Yliès Falcone, Jean-François Méhaut
Univ. Grenoble Alpes, CNRS, Inria, Grenoble INP*, LIG, 38000 Grenoble France
FirstName.LastName@univ-grenoble-alpes.fr

Abstract

Runtime Verification consists in studying a system at runtime, looking for input and output events to discover, check or enforce behavioural properties. Interactive debugging consists in studying a system at runtime in order to discover and understand its bugs and fix them, inspecting interactively its internal state. Interactive Runtime Verification (i-RV) combines interactive debugging and runtime verification, aiming at linking bug detection with bug understanding and improving the experience of debugging by guiding classical interactive debugging techniques using runtime verification.

1 Introduction

When developing software, detecting and fixing bugs as early as possible is important. This can be difficult: an error does not systematically lead to a crash, it can remain undetected during the development. Besides, when detected, a bug can be hard to understand, especially if the method of detection does not provide methods to study the bug. *Interactive runtime verification* addresses the aforementioned challenges by gathering two verifications techniques, namely interactive debugging and runtime verification. We briefly recall the features of interactive debugging and runtime verification as well as their shortcomings.

A widespread way to fixing bugs consists in observing a bad behaviour and starting a debugging session to find the cause [3]. The program is seen as a white box and its execution as a sequence of program states that the developer inspects step by step using a debugger in order to understand the cause of a misbehaviour. The execution is seen at a low level (assembly code, often mapped to the source code) while one would ideally want it be abstracted. The program state can be modified at runtime: variables can be edited, functions can be called, the execution can be restored to a previous state. This lets the developer test hypotheses on a bug without having to modify the code, recompile and rerun the whole program, which would be time consuming. However, this process can be tedious and prone to a lot of trials and errors. Moreover, observing a bug does not guarantee that this bug will appear during the debugging session, especially if the misbehaviour is caused by a race condition or a special input that was not recorded when the bug was observed. Interactive debugging does not target bug discovery: usually, a developer already knows the bug existence and tries to understand it.

Runtime verification [1] aims at detecting bugs. The execution is abstracted into a sequence of events of program-state updates. RV aims at detecting misbehaviours of a black-box system: its internal behaviour is not accessible and its internal state generally cannot be altered. Information on the internal state can be retrieved by instrumenting the execution of the program. The execution trace can be analysed offline (i.e. after the termination of the program) as well as online (i.e. during the execution) and is a convenient abstraction on which it is possible to check runtime properties. While RV is a versatile and effective technique in providing formal guarantees on the correctness of program executions, it still remains a technique used by experts in formal methods and is not yet used by programmers.

Challenges Interactive Runtime Verification (i-RV) links *bug discovery* with *bug understanding*. i-RV gathers interactive debugging and RV by augmenting debuggers with RV techniques, making i-RV a practical debugging method backed with a strong formal background. Gathering interactive debugging and RV is challenging for several reasons. To the best of our knowledge, interactive debugging has not been formalised in previous work. We therefore provide an abstraction of the execution of a program being debugged that is compatible

*Institute of Engineering Univ. Grenoble Alpes

with formalisms used in RV. Moreover, instrumentation techniques traditionally used in RV, like aspects or dynamic binary instrumentation do not allow controlling the execution and the set of monitored events is often static. Therefore, an important aspect of interactive RV is the way the richness and expressiveness of the instrumentation as well as the control provided by the debugger are leveraged. Usage of verdicts issued by the monitor for guiding the interactive debugging session also has to be specified.

The key idea of interactive runtime verification is to use RV as a guide to interactive debugging. The program is run in an interactive debugger, allowing the developer to use traditional techniques to study the internal state of the program.

2 Overview of Interactive Runtime Verification

In i-RV, the developer provides a property to check against the execution trace of a program to debug. The property can be written according to its specification or the Application Programming Interface (API) of the libraries it uses. The program is run with a debugger which provides tools to instrument its execution, mainly breakpoints and watchpoints, and let us generate events to build the trace, including function calls and variable accesses. An extension of the debugger provides a monitor that checks this property at runtime. Breakpoints and watchpoints are automatically set at relevant locations as the evaluation of a property requires monitoring function calls and memory accesses. When an event stops influencing the evaluation of any property, the corresponding instrumentation (breakpoints, watchpoints) becomes useless and is therefore removed: the instrumentation is *dynamic*. We introduce the notion of scenario in i-RV. Scenarios are a way to guide and automate the interactive debugging session by reacting to verdicts produced by the monitor and modify the execution or the control flow of the program. They are a powerful way to explore the behaviour of programs by trying different execution paths. The user-provided scenario defines what actions should be taken during the execution according to the evaluation of the property. Examples of scenarios are: when the verdict given by the monitor becomes false (e.g. when the queue overflows), the execution is suspended to let the developer inspect and debug the program in the usual way, interactively; save the current state of the program (e.g. using a checkpoint, a feature provided by the debugger) while the property holds (e.g. while the queue has not overflown) and restore this state later, when the property does not hold anymore (e.g. at the moment the queue overflows). When an event is generated — when a breakpoint or a watchpoint is reached — at runtime, the monitor updates its state. Monitor updates are seen as input events for the scenario, e.g. "the monitor enters state X", "state X has been left", "an accepting state has been entered", "a non-accepting state has been left".

We conducted experiments to assess the usefulness of Interactive Runtime Verification and the performance of our tool Verde. Our results show that though breakpoint and watchpoint-based instrumentation incurs non-trivial performance costs, Interactive Runtime Verification is applicable performance-wise in a variety of cases and helps studying bugs.

This extended abstract is an excerpt from [2] which gives a detailed presentation of Interactive Runtime Verification with experiments. Verde, experiments and demos can be found at `https://gitlab.inria.fr/monitoring/verde`.

References

[1] Ezio Bartocci and Yliès Falcone, editors. *Lectures on Runtime Verification - Introductory and Advanced Topics*, volume 10457 of *Lecture Notes in Computer Science*. Springer, 2018.

[2] Raphaël Jakse, Yliès Falcone, Jean-François Méhaut, and Kevin Pouget. Interactive runtime verification - when interactive debugging meets runtime verification. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pages 182–193. IEEE Computer Society, 2017.

[3] Fábio Petrillo, Zéphyrin Soh, Foutse Khomh, Marcelo Pimenta, Carla M. D. S. Freitas, and Yann-Gaël Guéhéneuc. Towards understanding interactive debugging. In *2016 IEEE International Conference on Software Quality, Reliability and Security, QRS 2016*, pages 152–163. IEEE, 2016.

Mise en œuvre d'une approche formelle en ingénierie des modèles

Akram Idani

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

Akram.Idani@imag.fr

Résumé. L'ingénierie des modèles (IDM) prone l'utilisation de modèles tout au long du cycle de développement de logiciels dans le but d'augmenter le niveau d'abstraction et réduire ainsi la complexité du logiciel. L'intérêt de cette approche est qu'elle permet d'avoir une séparation claire des préoccupations allant de modèles d'exigences jusqu'à la mise en oeuvre dans des plateformes cibles, tout en passant par les divers modèles de conception. Cependant, une limitation majeure de l'IDM est le manque d'outils de raisonnement formel permettant de garantir la correction des modèles issus des différents niveaux d'abstraction. En effet, la plupart des activités de validation offertes par les outils IDM d'aujourd'hui (en l'occurrence EMF, Eclipse Modeling Framework) sont basées sur la vérification de contraintes OCL à partir d'instances de méta-modèles sans pour autant garantir que ces instances se comportent conformément aux propriétés attendues. Dans ce travail, nous présentons un outil dédié à réduire le fossé entre l'IDM et le monde rigoureux des méthodes formelles. L'outil repose sur la traduction de méta-modèles EMF en une spécification B équivalente et permet l'injection de modèles dans cette même spécification. Cette technique vise à tirer pleinement profil d'outils de raisonnements formels tels que les outils de preuve et de model-checking. Il devient en effet possible de spécifier le comportement du modèle EMF en B ainsi que ses propriétés invariantes. L'AtelierB sera utilisé pour prouver la correction de ce comportement, et le model-checker ProB sera utilisé pour animer des scénarios d'exécution sous-jacents et les remonter au niveau du modèle EMF de départ.

1 Introduction

L'Ingénierie Dirigée par les Modèles (IDM) devient de plus en plus adoptée dans un contexte industriel car elle suggère des solutions aux deux problèmes majeurs du développement logiciel : (1) la complexité du logiciel, et (2) le gap entre modèles conceptuels et activités de codage. En effet, d'une part, l'IDM prone l'utilisation de modèles tout au long du cycle de développement, et d'autre part, elle est assistée par des outils dédiés à la définition de modèles et à la production graduelle de code source au travers d'une succession de transformations. Dans [5] les auteurs font un tour d'horizon d'applications industrielles de l'IDM ainsi que des leçons qui en découlent. Les motivations de ces applications se résument par :

— Augmenter la productivité et réduire le temps de développement ;
— Disposer de formalismes standardisés permettant une automatisation accrue ;
— Améliorer la communication et le partage de l'information.

Bien que ces constatations montrent que l'IDM est un paradigme prometteur et efficace, elles n'abordent ni les questions liées à la correction des modèles ni le niveau de confiance qu'on peut avoir dans ces modèles. Aujourd'hui, les activités de validation en IDM font majoritairement usage de techniques de test et de simulation. Toutefois, le manque d'outils de validation formelle freine l'adoption de ce paradigme dans le domaine des systèmes critiques où l'absence de défaillances est une exigence forte. Nous proposons dans ce travail, de compléter ces activités de test et de simulation par des activités de preuve et de raffinement en introduisant l'usage d'une approche formelle. Cet article présente l'outil Meeduse [1] que nous avons développé à cet effet. L'outil repose sur la traduction de méta-modèles EMF [8] en une spécification B [1] équivalente et permet l'injection de modèles dans cette même spécification. Cette technique vise à tirer pleinement profil d'outils de raisonnement formel tels que les outils de preuve et de model-checking sans perdre la traçabilité avec le modèle EMF de départ.

2 Méta-modélisation

La notion de méta-modèle (*e.g.* figure 1) est centrale en IDM car d'une part elle définit la structure et la sémantique des modèles manipulés, et d'autre part, elle permet l'interopérabilité d'outils tels que des analyseurs de langages (*e.g.* XText) ou des générateurs de code (*e.g.* Acceleo), ou encore des outils de transformation de modèles (*e.g.* ATL).

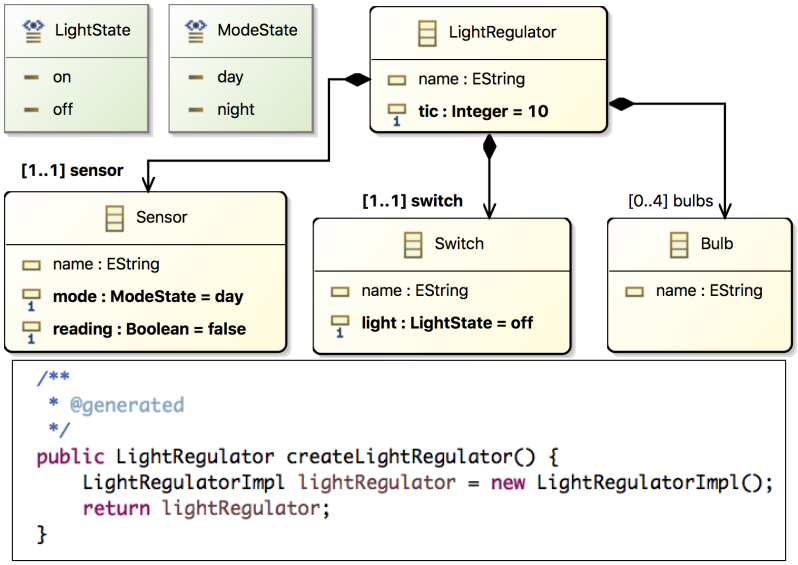


FIG. 1 – *Méta-modèle et constructeur d'instances de la classe LightRegulator*

La figure 1 donne le méta-modèle d'un régulateur automatique de lumière (class LightRegulator) qui, en fonction de la luminosité fournie par un capteur de luminosité (class Sensor),

1. Meeduse : Modeling Efficiently EnD USEr needs.

actionne un interrupteur (class Switch). Les modèles instances de ce méta-modèle doivent au moins respecter les contraintes structurelles. Par exemple, un régulateur de lumière contient nécessairement un switch et un sensor, tout en garantissant que la valeur initiale de l'attribut tic vaut 10. Toutefois, le mécanisme de validation en IDM part d'un modèle donné et permet d'indiquer si ce modèle est correct ou non vis à vis des contraintes du méta-modèle. En effet, les outils IDM, comme EMF, ne garantissent pas la correction des opérations de manipulation de modèles. On peut constater par exemple que la méthode Java createLightRegulator générée automatiquement par EMF et présentée dans la figure 1, se contente de créer de manière incontrôlée les instances de la classe LightRegulator. Par conséquent, il est plus facile de créer des modèles erronés que des modèles corrects surtout quand ces contraintes structurelles sont complétées par des contraintes contextuelles (le plus souvent exprimées en OCL). Il est néanmoins possible en EMF de réaliser la validation de ces contraintes une fois que le modèle ait été entièrement conçu. Pour remédier à cela, Meeduse traduit le méta-modèle en B, et génère toutes les opérations de manipulation de modèles (constructeurs, destructeurs, setters, et getters) en respectant les contraintes structurelles. Ces opérations sont ainsi correctes par construction vis à vis des invariants de typage et relatifs à la structure du méta-modèle. La figure 2 présente l'invariant généré automatiquement par Meeduse pour typer les variables ainsi que la spécification B du constructeur d'instances de LightRegulator.

INVARIANT
$Switch \in \mathcal{F}\ (SWITCH) \wedge$
$LightRegulator \in \mathcal{F}\ (LIGHTREGULATOR) \wedge$
$Sensor \in \mathcal{F}\ (SENSOR) \wedge$
$Bulb \in \mathcal{F}\ (BULB) \wedge$
$switch \in LightRegulator \rightarrowtail Switch \wedge$
$sensor \in LightRegulator \rightarrowtail Sensor \wedge$
$bulbs \in Bulb \nrightarrow LightRegulator \wedge$
$Switch\_light \in Switch \rightarrow LightState \wedge$
$LightRegulator\_tic \in LightRegulator \rightarrow \mathcal{Z} \wedge$
$Sensor\_mode \in Sensor \rightarrow ModeState \wedge$
$Sensor\_reading \in Sensor \rightarrow \mathbf{BOOL} \wedge$
$\forall\ (reg).(reg \in \mathbf{ran}(bulbs) \Rightarrow \mathbf{card}(bulbs^{-1}[\{reg\}]) \leq 4)$

LightRegulator_NEW$(aLightRegulator, aSwitch, aSensor) =$
PRE
$aLightRegulator \in LIGHTREGULATOR \wedge$
$aLightRegulator \notin LightRegulator \wedge$
$aSwitch \in Switch \wedge aSensor \in Sensor \wedge$
$switch^{-1}[\{aSwitch\}] = \emptyset \wedge sensor^{-1}[\{aSensor\}] = \emptyset$
THEN
$LightRegulator := LightRegulator \cup \{aLightRegulator\} \ ||$
$switch := switch \cup \{(aLightRegulator \mapsto aSwitch)\} \ ||$
$sensor := sensor \cup \{(aLightRegulator \mapsto aSensor)\} \ ||$
$LightRegulator\_tic := LightRegulator\_tic \cup \{(aLightRegulator \mapsto 10)\}$
END

FIG. 2 – *Spécification B du méta-modèle et du constructeur de la classe LightRegulator*

Cette traduction d'un méta-modèle vers B réalisée par Meeduse repose sur les traductions de UML vers B mises en œuvre dans la plateforme B4MSecure [3] :

— Les variables *LightRegulator*, *Switch*, *Sensor* et *Bulb* représentent les méta-classes.
— Les relations fonctionnelles correspondent à la traduction des attributs et des associations avec des spécialisations dépendant des multiplicités.
— Les ensembles *LightState* et *ModeState* sont des ensembles énumérés définis respectivement par $\{on, off\}$ et $\{day, night\}$.
— Les prédicats de la forme $aa \in \mathcal{F}\ (AA)$ font le lien entre l'ensemble des instances possibles *AA* et l'ensemble de instances effectives *aa*.

Contrairement au constructeur Java généré par EMF, l'opération LightRegulator_NEW ne permet la création d'une instance de LightRegulator que s'il existe un switch ($aSwitch \in Switch$) et un sensor ($aSensor \in Sensor$) qui ne sont pas déjà rattachés à d'autres régulateurs de lumière ($switch^{-1}[\{aSwitch\}] = \emptyset \wedge sensor^{-1}[\{aSensor\}] = \emptyset$). Ce constructeur impose ainsi au concepteur la création du capteur et de l'interrupteur avant la création du régulateur de lumière tout en interdisant le partage de ces composants par plusieurs régulateurs. Pour cet exemple, Meeduse a généré 26 opérations B, pour lesquelles l'AtelierB a produit 54 obligations de preuve dont 44 ont été prouvées automatiquement et 10 de manière interactive sans apporter aucune correction ou modification à la spécification B. Ce faisant, l'ajout de contraintes contextuelles nécessite la correction des opérations B pointées par l'AtelierB comme étant susceptibles de provoquer des violations d'invariants. En guise d'exemple, l'introduction de l'invariant $\forall\, tic \cdot (tic \in \text{ran}(LightRegulator\_tic) \Rightarrow tic \in 0..10)$ limitant le domaine des valeurs de l'attribut *tic* requiert la correction des opérations de modification de cet attribut.

3 Modélisation

Plusieurs outils Eclipse existent pour la création et la manipulation de modèles. Parmi ceux construits autour d'EMF, on peut citer : Sirius [2], EuGENia [3], GMF [4], ... Ces outils permettent d'associer des représentations graphiques aux concepts du méta-modèle et disposer ainsi de modèles aisément compréhensibles par l'utilisateur final. La figure 3 donne la palette d'éléments graphiques que nous utilisons pour représenter les capteurs, les ampoules et les interrupteurs. L'avantage principal de l'outil Sirius en comparaison avec les autres est qu'il permet l'utilisation d'expressions OCL (appelées styles conditionnels) pour conditionner l'usage de ces éléments graphiques. Par exemple, une instance de Switch aura deux représentations différentes en fonction de la valeur produite par les expressions OCL suivantes : `self.light = LightState::off` ou `self.light = LightState::on`. Notons que la méta-classe Bulb n'a pas d'attributs et n'a donc pas d'états. Ceci étant, la représentation d'une lampe allumée ou éteinte dépend de l'état de l'interrupteur. Quant à la représentation de Sensor, elle dépend de la valeur de l'attribut *reading*.

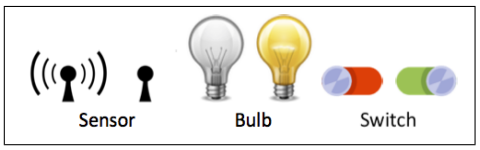


FIG. 3 – *Palette d'éléments graphiques*

2. https://www.obeo.fr/fr/produits/eclipse-sirius
3. https://www.eclipse.org/epsilon/doc/eugenia/
4. http://www.eclipse.org/modeling/gmp/

La figure 4 présente l'environnement de modélisation dans lequel nous avons créé au moyen de Sirius un régulateur de lumière (nommé `fus`) composé d'un capteur (`sen`), d'un interrupteur (`swi`) et de quatre lampes (`b1`, `b2`, `b3` et `b4`). Dans ce modèle le capteur est en mode reading, l'interrupteur est dans l'état on et les lampes sont, par conséquent, allumées.

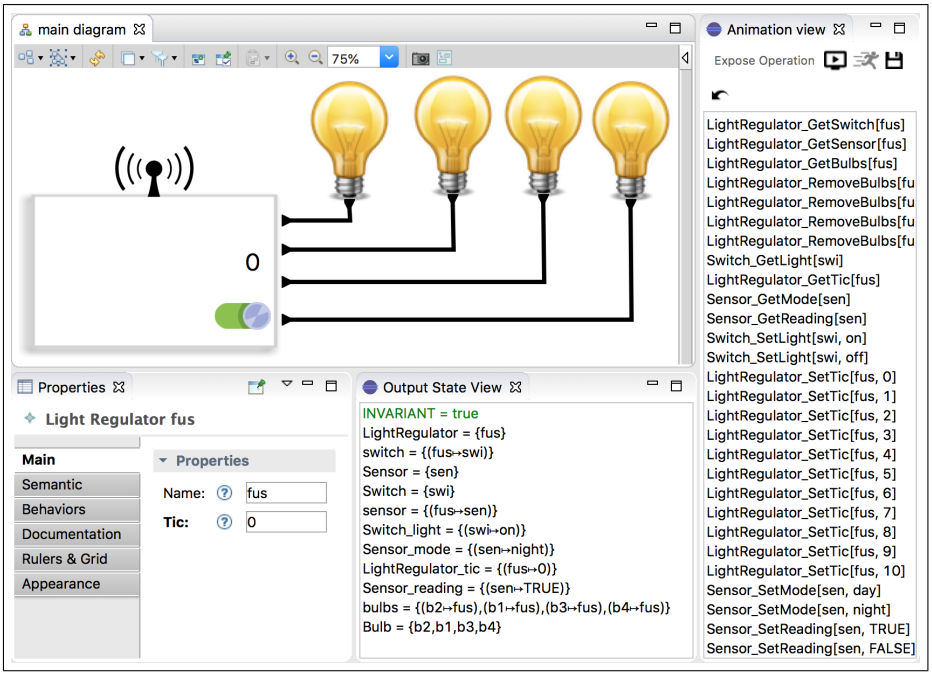


FIG. 4 – *Environnement de modélisation Meeduse*

La section précédente a montré que l'usage d'opérations B prouvées correctes, pour l'instanciation du méta-modèle, garantit le maintien de l'invariant et exclut donc tout modèle indésirable, contrairement à EMF qui est très permissif. Ceci est possible grâce à Meeduse qui permet d'intégrer dans cet environnement de modélisation les spécifications B issues de l'activité de méta-modélisation. Meeduse présente deux vues :

— Animation View : permet d'animer les opérations déclenchables à partir de l'état courant de la machine B.
— Output State View : représente l'état courant de la machine B ainsi que le respect ou non de l'invariant par cet état.

Remarquons que la vue "Properties" d'EMF permet d'associer n'importe quelle valeur de type integer à l'attribut tic, alors que la vue "Animation View" de Meeduse limite les possibilités uniquement aux valeurs respectant l'invariant, et ce au moyen des différentes instances de l'opération LightRegulator_SetTic. L'animation des opérations B est réalisée au moyen de ProB [4] dont l'API est disponible en open source. En effet, à chaque demande d'animation d'une opération, Meeduse invoque ProB pour réaliser l'action correspondante, et récupère l'état produit par ProB. Ensuite Meeduse interprète cet état dans le but d'établir les correspondances entre les informations de l'état courant et les éléments du modèle. Finalement, Meeduse remonte les informations au niveau EMF permettant ainsi de voir le modèle graphique évoluer suite à l'animation de l'opération. Cette remontée d'information permet en outre d'établir

l'équivalence entre l'état courant et le modèle garantissant ainsi le maintien d'un modèle correct tout au long de l'activité de modélisation.

4 Sémantique opérationnelle

Lors de l'activité de modélisation l'utilisateur a le choix entre l'usage d'un modeleur reposant sur EMF (tel que Sirius) ou l'usage de Meeduse. Dans le premier cas, la validation du modèle est réalisée après avoir conçu ce dernier, et dans le deuxième cas le modèle est maintenu correct tout au long de sa conception. L'utilisation de la méthode B permet de poursuivre formellement les activités de conception en associant une sémantique opérationnelle au modèle. Plusieurs travaux [2, 6, 7] se sont intéressés aux techniques et outils permettant de décrire comment un modèle s'exécute. Cependant, étant donné que l'exécution d'un modèle introduit souvent des modifications du modèle, alors la défintion d'une sémantique opérationnelle reposant sur EMF peut être lourd de conséquences. Nous avons en effet montré que les opérations fournies par EMF pour la manipulation de modèles ne donnent aucune garantie quant à la correction du modèle. Pour notre exemple, la sémantique d'exécution du modèle est spécifiée au moyen de trois opérations (setReadingSensor, decreaseTic et setMode) associées à des invariants indiquant que si le capteur est dans l'état reading alors tic vaut 0, et que l'interrupteur est à on ou off en fonction des modes day et night.

VARIABLES
regulator
INVARIANT
$regulator \in LightRegulator$
$\wedge$ ($Sensor\_reading(sensor(regulator)) = \mathbf{TRUE}$
$\Rightarrow LightRegulator\_tic(regulator) = 0$)
$\wedge$ ($Sensor\_mode(sensor(regulator)) = day$
$\Rightarrow Switch\_light(switch(regulator)) = off$)
$\wedge$ ($Sensor\_mode(sensor(regulator)) = night$
$\Rightarrow Switch\_light(switch(regulator)) = on$)
INITIALISATION
$regulator :\in LightRegulator$

setReadingSensor =
SELECT
$LightRegulator\_tic(regulator) = 0$
$\wedge\ Sensor\_reading(sensor(regulator)) = \mathbf{FALSE}$
THEN
Sensor_SetReading($sensor(regulator)$, **TRUE**)
END ;

setMode($mode$) =
PRE $mode \in 0 \,..\, 1$ **THEN**
SELECT
$Sensor\_reading(sensor(regulator)) = \mathbf{TRUE}$
THEN
IF $mode = 0$ **THEN**
Switch_SetLight($switch(regulator)$, *off*);
Sensor_SetMode($sensor(regulator)$, *day*)
ELSE
Switch_SetLight($switch(regulator)$, *on*) ;
Sensor_SetMode($sensor(regulator)$, *night*)
END ;
LightRegulator_SetTic($regulator$, 10) ;
Sensor_SetReading($sensor(regulator)$, **FALSE**)
END

decreaseTic =
SELECT
$LightRegulator\_tic(regulator) > 0$
THEN
LightRegulator_SetTic($regulator$, $LightRegulator\_tic(regulator)$ - 1)
END ;

FIG. 5 – *Sémantique opérationnelle du régulateur de lumière*

Pour cette spécification l'AtelierB a produit 49 obligations de preuve, qu'il a été en mesure de prouver automatiquement. Ces preuves ne portent pas sur les propriétés du modèle étant donnée que celles-ci ont été prouvées lors de l'activité de méta-modélisation, mais plutôt sur les propriétés définies pour ce comportement. Notons également que les opérations spécifiant le comportement du modèle font usage des opérations de base générées lors de l'activité de

méta-modélisation. Aussi, les modifications du modèle produites par cette sémantique sont-elles sûres vis-à-vis des contraintes structurelles et contextuelles intrinsèques au modèle.

Tout comme dans le cadre de la figure 4, Meeduse permet le chargement de cette spécification B ainsi que l'animation des opérations qu'elle offre tout en gardant l'équivalence entre l'état de la machine B et le modèle EMF. Outre la preuve de correction du modèle et de son exécution, cet outillage rend possible la visualisation du comportement prévu à l'exécution et la validation de sa conformité par rapport aux besoins de l'utilisateur.

5 Conclusion

Dans cet article nous avons donné un aperçu de l'outil Meeduse dédié à introduire une approche formelle dans un outil IDM, en l'occurrence EMF. Le potentiel de la méthode B avec les outils de preuve et de model-checking devient applicable en IDM grâce à la traduction de méta-modèles en B d'une part, et au maintien de l'équivalence entre un modèle donné et l'état de la spécification B, d'autre part.

Références

[1] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, USA, 1996.

[2] Benoît Combemale, Xavier Crégut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE Topcased Toolkit. In *4th European Congress EMBEDDED REAL TIME SOFTWARE (ERTS)*, page http ://www.erts2008.org/, Toulouse, France, France, January 2008.

[3] Akram Idani and Yves Ledru. B for Modeling Secure Information Systems - The B4MSecure Platform. In *17th Int. Conference on Formal Engineering Methods*, volume 9407 of *LNCS*, pages 312–318. Springer, 2015.

[4] Michael Leuschel and Michael Butler. ProB : A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003 : Formal Methods*, LNCS 2805, pages 855–874. Springer-Verlag, 2003.

[5] Parastoo Mohagheghi and Vegard Dehlen. Where is the proof ? - a review of experiences from applying mde in industry. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, volume 5095 of *LNCS*, pages 432–443, Berlin, Heidelberg, 2008. Springer.

[6] Muller Pierre-Alain, Franck Fleurey, and Jean-Marc J&eacute ;z&eacute ;quel. Weaving executability into object-oriented meta-languages. In S. Kent L. Briand, editor, *Proceedings of MODELS/UML'2005*, LNCS, pages 264–278, Jamaica, 2005. Springer.

[7] Michael Soden and Hajo Eichler. Towards a model execution framework for eclipse. In *Proceedings of the 1st Workshop on Behaviour Modelling in Model-Driven Architecture*, BM-MDA '09, pages 4 :1–4 :7, New York, NY, USA, 2009. ACM.

[8] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF : Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

Génération de modèles pour les formules quantifiées : une approche basée sur la teinte*

Benjamin Farinier[1,2], Sébastien Bardin[1],
Richard Bonichon[1], and Marie-Laure Potet[2]

[1] CEA LIST, Université Paris-Saclay, France
prenom.nom@cea.fr
[2] Univ. Grenoble Alpes, Verimag, France
prenom.nom@univ-grenoble-alpes.fr

Contexte Les techniques de vérification formelle nécessitent de plus en plus de raisonner sur des formules logiques modulo théorie. La capacité à générer des modèles est particulièrement importante, typiquement dans un contexte de recherche de bogues ou de test intensif [7, 3]. Les formules du premier ordre sans quantificateur (QF) sur des théories adaptées permettant déjà de gérer de nombreux cas intéressants, la communauté de la Satisfiabilité Modulo Théorie (SMT) [2] s'est naturellement concentrée sur le développement de solveurs QF efficaces.

Cependant des quantificateurs universels s'avèrent parfois nécessaires, par exemple pour représenter des préconditions ou abstraire certains composants. Malheureusement, s'il existe des approches dédiées (coûteuses) pour quelques théories quantifiées décidables comme l'arithmétique de Presburger, il n'existe pas d'approche générique satisfaisante pour le problème de la génération de modèles de formules universellement quantifiées. En effet, les approches à base d'instanciations et réfutations sont orientées vers la preuve de l'absence de solution [5], tandis que les méthodes récentes par extensions locales de théories [1], instanciations finies [9] ou basées sur les modèles [8] sont soit de portée trop réduite, soit se restreignent aux modèles finis, soit peuvent se retrouver bloquée dans un processus de raffinement sans fin.

Objectif et challenge Notre but est de proposer une solution générique et efficace au problème de la génération de modèles pour des formules quantifiées avec support des théories fréquemment rencontrées en vérification logicielle. Du fait de la quantité impressionnante d'efforts accumulés par la communauté pour produire des solveurs QF à l'état de l'art, il est souhaitable que cette solution réutilise au maximum les technologies SMT existantes.

Proposition Notre approche transforme une formule quantifiée en une formule non quantifiée telle que tout modèle de la seconde contient un modèle de la première. Les bénéfices sont les suivants : la formule transformée est plus simple à résoudre, elle peut être envoyée à un solveur QF, et un modèle de la formule quantifiée est déductible d'un modèle de la formule non quantifiée. L'idée consiste à ignorer les quantificateurs de la formule tout en renforçant la partie non quantifiée par une *condition d'indépendance* contraignant les modèles à être indépendants des variables initialement quantifiées.

Contributions Cet article apporte les contributions suivantes :

Nous proposons un nouveau cadre générique pour la génération de modèles de formules quantifiées reposant sur l'inférence de *conditions suffisantes d'indépendance*. Nous prouvons sa *correction* et son *efficacité* sous des hypothèses raisonnables. En particulier notre approche induit un surcoût seulement linéaire en la taille de la formule. Nous étudions aussi brièvement sa *complétude*, liée à la notion de *plus faible condition d'indépendance*.

*Article accepté à CAV 2018 [6], disponible à l'adresse suivante : http://benjamin.farinier.org/cav2018/

TABLE 1 – Réponses et temps de résolution (en secondes, TIMEOUT de 1000s inclus)

		Boolector•	CVC4	CVC4•	$CVC4_E$	$CVC4_E$•	Z3	Z3•	$Z3_E$	$Z3_E$•
SMT-LIB	SAT	**399**	84	242	84	242	**261**	366	87	366
	# UNSAT	N/A	0	N/A	0	N/A	165	N/A	0	N/A
	UNKNOWN	870	1185	1027	1185	1027	843	903	1182	903
	total time	**349**	165	194 667	165	196 934	**270 150**	36 480	192	41 935
BINSEC	SAT	**1042**	951	954	951	954	**953**	1042	953	1042
	# UNSAT	N/A	62	N/A	62	N/A	319	N/A	62	N/A
	UNKNOWN	379	408	467	408	467	149	379	406	379
	total time	**1152**	64 761	76 811	64 772	77 009	**30 235**	11 415	135	11 604

solveur• : solveur amélioré avec notre méthode $Z3_E$, $CVC4_E$: essentiellement E-matching

Nous définissons une procédure inspirée de la teinte pour l'inférence de conditions d'indépendance, constituée d'un cœur générique raffiné en fonction des théories. Nous proposons de tels raffinements pour une large classe d'opérateurs.

Enfin, nous présentons une implémentation concrète de notre méthode spécialisée pour les bitvecteurs et les tableaux. La Table 1 présente les résultats de nos expérimentations portant sur des formules générées à partir de la SMT-LIB ou produites par l'outil d'exécution symbolique BINSEC [4], et montre que notre approche étend efficacement les solveurs QF au cas quantifié tout en étant complémentaire des dernières avancées du domaine [9, 8].

Conclusions Tandis que les techniques *à la* E-matching étendent les solveurs QF à la décision de l'insatisfiabilité de formules quantifiées, ce travail propose un mécanisme pour les étendre à la décision de la satisfiabilité et à la génération de modèles de formules quantifiées, résultant en une gestion plus symétrique des formules quantifiées. Cette nouvelle approche ouvre la voie à de futurs travaux tels que la définition de mécanismes d'inférence de conditions d'indépendance plus précises, l'identification de sous-classes pour lesquelles inférer une plus faible condition d'indépendance est faisable, ou la combinaison avec d'autres techniques d'instanciation.

Références

[1] K. Bansal, A. Reynolds, T. King, C. W. Barrett, and T. Wies. Deciding Local Theory Extensions via E-matching. In *CAV 2015, San Francisco, USA, 2015.*

[2] C. W. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, pages 825–885. 2009.

[3] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pages 457–481. 2009.

[4] R. David, S. Bardin, T. D. Ta, L. Mounier, J. Feist, M. Potet, and J. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In *SANER 2016, Osaka, Japan, 2016.*

[5] L. M. de Moura and N. Bjørner. Efficient E-Matching for SMT Solvers. In *CADE 2007, Bremen, Germany, July 17-20, 2007.*

[6] B. Farinier, S. Bardin, R. Bonichon, and M.-L. Potet. Model Generation for Quantified Formulas : A Taint-Based Approach. In *CAV 2018, Oxford, England, 2018.*

[7] P. Godefroid, M. Y. Levin, and D. A. Molnar. SAGE : whitebox fuzzing for security testing. *ACM Queue*, 10(1) :20, 2012.

[8] A. Reynolds, M. Deters, V. Kuncak, C. Tinelli, and C. W. Barrett. Counterexample-Guided Quantifier Instantiation for Synthesis in SMT. In *CAV 2015, San Francisco, USA, 2015.*

[9] A. Reynolds, C. Tinelli, A. Goel, and S. Krstic. Finite Model Finding in SMT. In *CAV 2013, Saint Petersburg, Russia, July 13-19, 2013.*

2

Extension des patrons de spécification pour la vérification de traces paramétriques

Yoann Blein Yves Ledru Lydie du-Bousquet Roland Groz

Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000
Grenoble, France
prénom.nom@imag.fr

Ceci est un résumé étendu de [1].

Bien que parfois critiques, certains domaines d'application du logiciel ne sont pas tenus d'utiliser des méthodes de vérification formelles. C'est le cas des dispositifs médicaux, et plus particulièrement ceux de dernière génération, dont les systèmes deviennent très complexes. Pour favoriser une transition vers des processus de développement plus rigoureux, la communauté des méthodes formelles a proposé l'utilisation de méthodes formelles légères [3]. Le projet MODMED, financé par l'Agence Nationale de la Recherche, rassemble le Laboratoire d'Informatique de Grenoble, un fabricant de dispositifs médicaux, BlueOrtho, et une entreprise spécialisée dans le développement logiciel pour ces systèmes, MinMax-Medical. Constatant qu'il est relativement facile d'instrumenter ces systèmes avec des points de traces, ce projet propose la vérification de traces de chirurgies comme une approche légère à la vérification de propriétés. Dans notre projet, les produits de BlueOrtho génèrent déjà des traces, dont les événements sont accompagnés de paramètres riches en données. L'entreprise dispose ainsi d'un corpus de 12 000 traces d'exécutions pour le produit phare.

Pour vérifier des propriétés sur ces traces paramétriques, nous proposons un nouveau langage : PARTRAP. Ce langage doit permettre à des ingénieurs logiciels sans expérience des méthodes formelles d'écrire des propriétés, principalement au travers d'une syntaxe verbeuse et de sa simplicité. Il étend les patrons de spécification initialement proposés par Dwyer et al. [2] de manière à les rendre composables et ainsi augmenter leur expressivité. Par ailleurs, PARTRAP fournit deux moyens complémentaires pour exploiter des événements paramétrés : la possibilité de nommer les occurrences d'un type d'événement et de contraindre leurs paramètres, et des quantificateurs permettant d'itérer sur des listes de valeurs, extraits de paramètres. PARTRAP supporte également des constructions temps-réel qui viennent s'ajouter aux patrons.

PARTRAP s'articule autour de quelques briques syntaxiques. Les événements sont désignés par leur types et peuvent être nommés et contraints comme dans

`Enter e where p(e)`, où `e` est un événement de type `Enter` dont les paramètres satisfont le prédicat `p`. Les patrons contraignent l'occurrence d'événements individuels (`absence_of Enter`), ou imposent des relations temporelles entre plusieurs événements. Par exemple, avec le patron `followed_by` permet d'exprimer qu'après être entré dans un certain état, il faut fatalement sortir de ce même état :

```
Enter enter followed_by Exit exit
    where exit.state == enter.state.
```

Les restrictions de portées `after` et `before` limitent les intervalles d'une trace où une propriété doit être vérifiée. Par exemple, on peut contraindre les températures autorisées après la mise en service d'une caméra comme suit :

```
after first CameraConnected,
    absence_of Temperature t where t.val < 40
```

Ces restrictions de portées sont emboitables et les événements nommés sont propagés à la propriété sous-jacente. Ces particularités permettent de capturer de nombreuses propriétés temporelles. En fait, le patron `followed_by` présenté ci-dessus est défini avec à une restriction `after` et une contrainte d'occurrence.

Une implémentation de PARTRAP réalisée en Haskell est disponible en ligne sous licence libre [1]. Elle a permis la vérification d'une douzaine de propriétés sur un corpus de 100 traces provenant de chirurgies réelles. Bien qu'aucun défaut critique n'a été identifié, ces expériences ont mis en exergue des comportement atypiques du système ayant produit ces les traces, et montré que PARTRAP permet de capturer des propriétés d'intérêt pratique.

Ce travail a été publié à la 6-ième édition de la conférence internationale FormaliSE [1]. Cet article décrit le langage, sa syntaxe et sa sémantique, et l'illustre sur l'étude de cas industrielle du projet MODMED.

Remerciements Ce travail est financé par le projet ANR MODMED (ANR-15-CE25-0010).

Références

[1] Blein, Yoann, Yves Ledru, Lydie du Bousquet et Roland Groz: *Extending Specification Patterns for Verification of Parametric Traces*. Dans *FormaliSE 2018*. ACM, 2018.

[2] Dwyer, Matthew B., George S. Avrunin et James C. Corbett: *Patterns in Property Specifications for Finite-State Verification*. Dans *ICSE*, pages 411–420. ACM, 1999. http://doi.acm.org/10.1145/302405.302672.

[3] Jackson, Daniel et Jeannette Wing: *Lightweight Formal Methods*. ACM Comput. Surv., 28(4) :121, 1996. http://doi.acm.org/10.1145/242224.242380.

1. https://gricad-gitlab.univ-grenoble-alpes.fr/modmed/partrap

Construction incrémentale de chorégraphies réalisables

(Travaux publiés dans les actes de la conférence NFM'2018 [1])

Sarah BENYAGOUB
Doctorante à
Université de Mostaganem - Algérie,
IRIT-Toulouse INP - FRANCE

Ce résumé présente une technique de conception de chorégraphies réalisables par construction. Les chorégraphies, appelées protocole de conversation (*Conversation Protocol CP*), sont modélisées par des systèmes états-transitions qui décrivent des échanges de messages entre différents pairs. La technique proposée consiste à produire, par le raffinement, une modélisation, par d'autres systèmes états-transitions, des pairs qui *réalisent* cette conversation i.e. cet échange de messages.

1 Problème étudié

Nous étudions le cas d'une conception descendante de systèmes distribués où l'interaction entre pairs est définie à l'aide d'une spécification globale décrite par un *CP*. Une problématique majeure, déjà abordée dans plusieurs travaux, concerne la propriété de *réalisabilité* d'un *CP*. Il s'agit de garantir l'existence d'un ensemble de pairs, communicant par envois et réceptions de messages, dont la composition génère les mêmes séquences d'échanges de messages que celles spécifiées par le *CP* décrivant ces échanges.

Dans le cas de la communication asynchrone, ce problème est en général indécidable. Néanmoins, plusieurs travaux ont proposé des cadres de communications pour lesquels cette propriété de réalisabilité est satisfaite. Des méthodes de vérification formelle ont été associées à ces propositions. Elles traitent les systèmes dans leur globalité, sans possibilité de décomposition et se heurtent au problème de la taille des files d'attente.

2 Notre proposition

Nous proposons une méthode correcte par construction qui garantit, a-priori, la réalisabilité d'un CP. Nous définissons un ensemble d'opérateurs de composition de CP. Pour chaque opérateur, des conditions suffisantes de préservation de la réalisabilité sont définies. Ainsi, chaque *CP* réalisable est construit, de manière incrémentale, par une séquence d'applications de ces opérateurs sur un CP initialement vide (et donc réalisable).

Les opérateurs que nous avons définis et formalisés sont : la séquence, le choix et l'itération. A ce stade de nos travaux, la composition parallèle est définie par entrelacement. Les conditions suffisantes garantissant la réalisabilité ont été également formalisées. La préservation de la réalisabilité par chaque opérateur lorsque les conditions suffisantes sont satisfaites a été prouvée par induction sur la structure des CPs.

Plusieurs études de cas issues de la littérature et utilisées pour valider les propriétés de réalisabilité ont été mises en œuvre. Chaque cas d'étude définit une instance particulière du modèle générique que nous avons développé. Ces instanciations ont permis de valider notre approche en identifiant des cas de réalisabilité ainsi que des cas de non réalisabilité. Nous avons également conduit un développement complètement formalisé avec Event-B en utilisant le raffinement pour décomposer la propriété de réalisabilité. Le raffinement a été mis en oeuvre pour décomposer la propriété de réalisabilité en plusieurs invariants introduits à différents niveaux du développement. Le modèle obtenu a été instancié pour les cas d'étude utilisés (Voir `https://www.irit.fr/~Sarah.Benyagoub/models`)

Références

[1] S. Benyagoub, M. Ouederni, Y. Aït-Ameur, A. Mashkoor, Incremental Construction of Realizable Choreographies, in : Proc. of 10th International Symposium, Nasa Formal Methods 2018, Newport News, VA, USA. DOI 10.1007/978-3-319-77935-5, Vol. 10811 of LNCS, Springer, 2018, pp. 1–19.

Une approche structurée de l'ornementation pour ML *

Thomas Williams et Didier Rémy, Inria

Les ornements permettent de décrire des changements de définitions de types algébriques, réorganisant, ajoutant ou supprimant certaines informations et certains invariants, de sorte que des fonctions opérant sur le type de base puissent être partiellement, et parfois totalement, relevées en des fonctions opérant sur le type ornementé. Le comportement des fonctions relevées est spécifié par une propriété de cohérence avec les fonctions de base : les fonctions relevées doivent se comporter de manière identique, aux modifications imposées par les ornements près.

Les ornements ont d'abord été introduits dans des langages avec types dépendants, profitant des encodages puissants qu'ils permettent. Ainsi la notion d'ornement et la propriété de cohérence sont programmées dans la théorie des types. De tels encodages ne sont pas disponibles en ML.

Nous proposons d'introduire les ornements comme une nouvelle construction en ML : nous proposons une extension de ML avec des ornements d'ordre supérieur, et une construction de relèvement par ornements. Nous illustrons l'expressivité de cette extension en présentant des exemples typiques d'utilisation, notamment la refactorisation de code et la spécialisation. Les illustrations données fonctionnent effectivement dans un prototype opérant sur un sous-ensemble d'OCaml.

Plutôt que de donner une présentation monolithique du processus d'élaboration, nous proposons une procédure de relèvement en deux étapes, s'appuyant sur une abstraction a posteriori du code de base. Dans un premier temps, le code de base est abstrait a posteriori, produisant un terme générique dans un métalangage étendant ML. Cette transformation dépend du terme de base, mais pas du relèvement demandé. Le terme générique décrit ainsi la famille de relèvements possibles d'un terme de base, et permet de présenter au programmeur les choix qu'il devra faire à l'instanciation. Le code relevé est ensuite obtenu en appliquant ce terme générique à des arguments bien choisis, déduits du relèvement demandé par le programmeur, décrivant les ornements utilisés et des éventuels patchs ajoutés au code.

Le langage intermédiaire utilisé étend ML avec des abstractions et des applications spécialement marquées, permettant de distinguer les indirections ajoutées par l'élaboration des constructions présentes dans le terme de base, et avec une forme restreinte de types dépendants permettant de tracer les branches sélectionnées par filtrage. Des restrictions de typage appropriées permettent de redescendre le code obtenu dans ML, par une réduction des abstractions marquées et l'application de quelques simplifications.

La structure de la transformation permet alors de voir le code de base comme une instance particulière du code générique. Le comportement du code relevé est alors précisément relié au comportement du code de base à l'aide d'un théorème de paramétricité sur le métalangage intermédiaire.

Références

[WR18] Thomas Williams and Didier Rémy. A principled approach to ornamentation in ML. In *Proceedings of the 45th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'18, pages 1–30. ACM, 2018.

*Cet article est un résumé étendu d'une communication à la conférence *Principles of Programming Languages* [WR18].

Une dépendance qui fait de l'effet

Pierre-Marie Pédrot et Nicolas Tabareau

Parmi la grande variété de systèmes de types, les types dépendants constituent un extrême à plusieurs égards. Du point de vue de l'expressivité, leur richesse est telle qu'elle permet de garantir des invariants arbitraires sur les programmes sous-jacents, à un point tel qu'ils peuvent être utilisés pour prouver la correction de programmes. En ce sens, les types dépendants concrétisent effectivement le dicton *well-typed programs can't go wrong*. En revanche, cette expressivité a un coût, et les systèmes de types dépendants font montre d'une complexité notoire, tant au niveau de leur implémentation que de leurs propriétés méta-théoriques. La présence de calcul dans les types a notamment longtemps empêché l'ajout d'effets de bords de première classe à ces langages, forçant l'utilisation d'un style monadique ou restreignant la forme des termes apparaissant dans les types.

Cette restriction malheureuse semble cependant en passe d'être levée, si l'on en croit les résultats que nous avons présentés à LICS 2017 et ESOP 2018 [PT17 ; PT18] et que nous synthétisons ici. Nous avons introduit une large classe d'effets en théorie des types dépendants grâce au sevrage, une famille de traductions de programmes ciblant le calcul des constructions inductives (CIC). Elle révèle une tension remarquable entre les effets et l'élimination dépendante. Cette dernière correspond au principe d'induction sur les types algébriques, et permet d'écrire un filtrage par motif dans lequel le type des branches dépend de la forme du constructeur de la branche correspondante. Ainsi, l'élimination dépendante sur les booléens est décrite par la règle

$$\frac{\Gamma \vdash M : \texttt{bool} \qquad \Gamma \vdash N_1 : P\{b := \texttt{true}\} \qquad \Gamma \vdash N_2 : P\{b := \texttt{false}\}}{\Gamma \vdash \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 : P\{b := M\}}$$

et est en général invalide en présence d'effets, du fait de l'existence de booléens impurs qui ne sont ni `true` ni `false`. Nous proposons une variante de la théorie des types qui ajoute à cette règle la condition de bord que le prédicat P doit évaluer son argument de manière stricte.

Il reste cependant possible de construire un système mêlant effets et élimination dépendante complète, à condition de sacrifier la cohérence logique. Notre théorie exceptionnelle, une variante du sevrage, constitue une instance de ce phénomène en rajoutant un mécanisme d'exceptions dynamiques aux types dépendants. Quoique de peu de valeur logiquement, ce système peut être compris comme un langage de programmation dépendant impur. Par ailleurs, il est possible d'utiliser la théorie hôte pour raisonner correctement sur de tels programmes impurs, ce qui ouvre la voie à une notion de typage dépendant sémantique.

Toutes ces extensions de la théorie des types sont justifiées par des traductions de programme purement syntaxiques ciblant CIC, qui est le langage implémenté par l'assistant à la preuve Coq. Cette propriété permet de réutiliser Coq comme langage hôte et de fournir à l'utilisateur une surcouche sous forme de greffon qui, étant donné un programme dans la théorie étendue, produit du code compilé à la volée vers Coq, donnant ainsi l'impression de travailler directement dans la théorie étendue.

Références

[PT17] Pierre-Marie PÉDROT et Nicolas TABAREAU. "An Effectful Way to Eliminate Addiction to Dependence". In : *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*. Sous la dir. de Joël OUAKNINE. 2017, p. 1-12.

[PT18] Pierre-Marie PÉDROT et Nicolas TABAREAU. "Failure is Not an Option - An Exceptional Type Theory". In : *Programming Languages and Systems - 27th European Symposium on Programming, ESOP 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. Sous la dir. d'Amal AHMED. 2018, p. 245-271.

Étendre OCaml par du filtrage par comotifs à l'aide d'une macro

Paul Laforgue et Yann Régis-Gianas

Ce texte est un résumé de notre article présenté à PPDP en 2017 [LR17].

Dans un langage paresseux comme Haskell, définir une séquence infinie est un jeu d'enfant :

```
fib = 1 : 1 : (zipWith ( + ) fib (tail fib))
```

Dans cet exemple, la suite infinie des nombres de Fibonacci est définie récursivement par une unique équation, très proche de sa définition mathématique classique.

Malheureusement, une retranscription directe de cette définition Haskell en OCaml est vouée à l'échec. En effet, la définition suivante :

```
let rec fib () = 1 :: 1 :: (map2 ( + ) (fib ()) (tl (fib ())))
```

pose problème car dès la première évaluation de `fib ()`, le programme se met à diverger. C'est la stratégie d'évaluation stricte d'OCaml qui est en cause : dès que la fonction `fib` est appliquée, elle a immédiatement besoin de la valeur de `fib ()`, ce qui provoque une boucle d'évaluation infinie.

La solution classique pour permettre la définition d'une séquence infinie s'appuie sur le module **`Lazy`** de la bibliothèque standard d'OCaml :

```
type 'a cell = InfiniteCons of 'a * 'a stream and 'a stream = 'a cell Lazy.t

let rec map2 f xs ys =
  lazy (match Lazy.force xs, Lazy.force ys with
        | InfiniteCons (x, xs), InfiniteCons (y, ys) ->
          InfiniteCons (f x y, map2 f xs ys))

let tail xs = let (InfiniteCons (_, xs)) = Lazy.force xs in xs

let head xs = let (InfiniteCons (x, _)) = Lazy.force xs in x

let rec fib = lazy (InfiniteCons (1, lazy (InfiniteCons (1, map2 ( + ) fib (tail fib)))))
```

Cette définition fonctionne mais est plus lourde syntaxiquement que le code Haskell initial et elle est aussi plus éloignée de la définition mathématique de la suite de Fibonacci. De plus, le raisonnement sur ce programme est pollué par les occurrences de **`lazy`** et de **`Lazy.force`**.

Le filtrage par comotifs et la définition de types coalgébriques permettent de réconcilier les langages fonctionnels stricts avec la définition d'objets infinis. Dans l'extension d'OCaml avec filtrage par comotifs que nous présentons ici, la suite de Fibonacci s'écrit ainsi :

```
type 'a stream = { Head : 'a; Tail : 'a stream }

let rec map2 : type a b c. (a -> b -> c) -> a stream -> b stream -> c stream
= fun f xs ys -> cofunction : c stream with
  | ..#Head -> f xs#Head ys#Head
  | ..#Tail -> map2 f xs#Tail ys#Tail

```

```
8  let corec fib : int stream with
9    | ..#Head -> 1
10   | ..#Tail : int stream with
11   | ...#Head -> 1
12   | ...#Tail -> map2 ( + ) fib fib#Tail
```

La déclaration de type de la ligne 1 introduit un type coalgébrique pour les séquences infinies de valeurs de type `'a`. Un type coalgébrique est défini par les *observations* que l'on peut faire de ses habitants : ici par exemple, on peut observer la tête d'une séquence avec l'observation **`Head`** et obtenir alors une valeur de type `'a` ; on peut aussi observer la suite d'une séquence avec l'observation **`Tail`** et obtenir alors une nouvelle séquence de type `'a stream`. Il n'est pas étonnant de reconnaître ici la syntaxe des déclarations des types d'enregistrements car il y a une similarité entre les types coalgébriques et ces derniers. Ils diffèrent cependant sur un aspect essentiel : les champs d'un enregistrement sont des valeurs tandis que les observations sont des calculs.

Ces calculs sont déclenchés par des applications d'observations à des habitants du type coalgébrique. Ainsi, `xs#Head` provoque le calcul effectif de la tête de la séquence `xs`. Ce calcul est défini au moment de la définition d'une valeur du type `'a stream` par le biais d'un filtrage par comotifs. La ligne 8 introduit ici la suite de Fibonacci en définissant les calculs déclenchés pour chacune de ses observations. Par exemple, en ligne 9, on déclare que "`fib#Head = 1`" tandis qu'en ligne 12, on déclare que "`fib#Tail#Tail = map2 ( + ) fib fib#Tail`". Ces définitions sont à la fois de haut-niveau et suffisamment paresseuses pour que la définition d'un objet infini (productif) puisse se faire sans provoquer de divergence incontrôlée.

Le filtrage par comotifs a été introduit par Abel, Pientka, Thibodeau et Setzer pour éviter l'écueil des coinductifs de Coq : la perte de la préservation du typage par l'évaluation. Notre contribution a été de montrer qu'il suffit qu'un langage de programmation fonctionnel soit muni d'un système de type incluant GADTs et polymorphisme d'ordre 2 pour que son extension avec un filtrage par comotifs se résume à l'écriture d'une macro, c'est-à-dire d'une transformation purement locale et syntaxique.

Non seulement la simplicité de cette transformation nous a permis d'étendre sans effort OCaml avec des types coalgébriques mais notre encodage offre aussi un gain en expressivité par rapport aux langages d'Abel *et al* : nos observations sont des valeurs de première classe et du premier ordre. Nous illustrons l'intérêt de cette propriété sur quelques exemples.

Références

[LR17] Paul Laforgue and Yann Régis-Gianas. Copattern matching and first class observations in OCaml, with a macro. *PPDP'17*. 2017. Pages 97–108. Editeurs : Wim Vanhoof, Brigitte Pientka.