Actes des 14e journées sur les Approches Formelles dans l'Assistance au Développement de Logiciels

Frédéric Dadeau, Pascale Le Gall

AFADL 2015

Actes des 14èmes journées sur les

Approches Formelles dans l'Assistance au Développement de Logiciels

Edités par Frédéric Dadeau et Pascale Le Gall

9 et 10 juin 2015

ENSEIRB-MATMECA de Bordeaux

Table des matières

Préface

La 14ème édition d'AFADL, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels, se tiendra les 9 et 10 juin 2015 à l'Ecole Nationale Supérieure d'Electronique, Informatique et Radiocommunication de Bordeaux (ENSEIRB-MATMECA). Elle est organisée conjointement avec 2 autres manifestations : CIEL 2015, Conférence en IngénieriE du Logiciel et les journées nationales du GDR Génie de la Programmation et du Logiciel (GPL). Cet événement permettra ainsi à toute la communauté francophone des chercheurs en génie logiciel de se retrouver et échanger. L'atelier AFADL rassemble de nombreux acteurs académiques et industriels intéressés par la mise en œuvre de 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 des thèmes comme :

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

Nous aurons l'honneur d'accueillir, en association avec la conférence CIEL, Bertrand Meyer (ETH Zurich) comme conférencier invité.

Les actes d'AFADL 2015 comprennent 2 articles longs, 2 articles courts présentant des résultats nouveaux.

Comme dans l'édition précédente, 7 présentations concernent des résultats déjà présentés dans d'autres circonstances (conférences, ateliers, projets, ...internationaux). Ces présentations sont réalisées conjointement avec les journées du GDR GPL, notamment les groupes de travail Méthodes Formelles pour le Développement Logiciel (MFDL) et Méthodes de Test pour la Vérification et la Validation (MTV2), dont les thématiques recoupent celles d'AFADL.

Cette année, nous avons fait le choix d'ouvrir l'Atelier AFADL à un nouveau type de contributions visant à présenter des travaux en cours dans l'objectif de susciter des discussions au sein de la communauté. Le programme de cette session, en grande partie à destination des jeunes chercheurs, a été ouvert à toutes les bonnes volontés de notre communauté. Nous avons inclus les résumés étendus de ces présentations dans les actes. Nous espérons que les discussions suscitées par cette session enrichiront cette édition de l'atelier AFADL.

Comme les années précédentes, les contributions couvrent un large éventail de techniques, méthodes et applications.

Nous remercions les membres du comité de programme pour leur travail qui a contribué à produire un programme de qualité, ainsi que tous les auteurs qui ont soumis un article et sans qui il n'y aurait plus d'atelier AFADL.

Nous remercions les membres du comité d'organisation des journées AFADL-CIEL-GPL 2015 qui ont pris en charge tous les aspects logistiques.

Le 26 mai 2015

Frédéric Dadeau et Pascale Le Gall
Présidents du comité de programme AFADL 2015

Comité de programme

Présidents

Frédéric Dadeau, FEMTO-ST, Besançon
Pascale Le Gall, Ecole Centrale Paris

Membres

Yamine Ait Ameur, IRIT/INPT-ENSEEIHT, Toulouse
Sandrine Blazy, IRISA - Université Rennes 1
Frédéric Boniol, ONERA, Toulouse
Pierre Casteran, LaBRI, Bordeaux
Lydie Du Bousquet, LIG, Grenoble
Catherine Dubois, ENSIIE - CEDRIC, Evry
Christèle Faure, Saferiver, Paris
Pascal Fontaine, LORIA, Nancy
Akram Idani, LIG, Grenoble
Jacques Julliand, FEMTO-ST, Besançon
Florent Kirchner, CEA, Saclay
Nikolai Kosmatov, CEA, Saclay
Régine Laleau, LACL - Université Paris-Est Créteil
Jean-Louis Lanet, Université de Limoges
Arnaud Lanoix, Université de Nantes
Yves Le Traon, Université du Luxembourg
Yves Ledru, LIG, Grenoble
Nicole Levy, CNAM, Paris
Delphine Longuet, LRI, Orsay
Jean-Marc Mota, Thales
Ioannis Parissis, LCIS, Valence
Pascal Poizat, LIP6, Paris
Marie-Laure Potet, Verimag, Grenoble
Marc Pouzet, LIENS, Paris
Antoine Rollet, LaBRI, Bordeaux
Vlad Rusu, INRIA, Lille
Nicolas Stouls, INSA, Lyon
Safouan Taha, Supélec, Gif-sur-Yvette
Sylvie Vignes, Télécom Paris-Tech, Paris
Laurent Voisin, Systerel
Virginie Wiels, ONERA, Toulouse
Fatiha Zaidi, LRI, Orsay

Relecteur additionnel

Stephan Merz, LORIA, Nancy

Vérification parallélisée de propriétés temporelles sur des traces d'exécution, par analyse dynamique formelle

Antoine Ferlin* antoine.ferlin@ifsttar.fr
Philippe Bon* philippe.bon@ifsttar.fr
Virginie Wiels† virginie.wiels@onera.fr
Simon Collart-Dutilleul* simon.collart-dutilleul@ifsttar.fr

Résumé

Les méthodes de vérification peuvent être classées suivant deux critères : une méthode peut être statique ou dynamique, ainsi que formelle ou informelle. Ce papier poursuit des travaux de thèse sur la vérification de propriétés temporelles sur des traces d'exécution par analyse dynamique formelle. L'approche proposée consiste à transformer une propriété LTL en automate de Büchi et à exécuter ce dernier sur une trace pour l'analyser. Le problème de fin de trace lié à l'utilisation de LTL sur des traces finies peut être contourné par le calcul d'informations statistiques à condition que la propriété suive un patron prédéfini. Pour des traces de très grande taille, cette approche est bien adaptée, mais nécessite que la trace soit vérifiée séquentiellement. Cet article propose de remédier à ce problème, en découpant la trace en plusieurs sous-traces analysables séparément, suivant une stratégie définie, ce qui permet un gain de temps significatif.

1 Introduction

Le développement de logiciels critiques est contraint par des standards de certification dépendant des domaines d'applications. Par exemple, le standard pour les logiciels avioniques est le DO-178 ; dans le ferroviaire, on utilise la norme IEC 50128. Bien que les objectifs pour chaque étape du développement soient définis par les standards, il incombe aux entreprises de choisir les méthodes de vérification permettant d'atteindre ces objectifs. Les standards peuvent proposer à titre indicatif des méthodes connues.

Une façon de classer les techniques de vérification est de considérer deux critères : statique ou dynamique, formelle ou non formelle. Une méthode statique signifie que la vérification s'effectue sans l'exécution du programme. À l'inverse, une méthode dynamique nécessite l'exécution du programme. Une méthode formelle mettra en œuvre une analyse formelle en s'appuyant sur un langage possédant une sémantique formelle. À titre d'exemples, les revues de code sont statiques et non formelles, le test est classiquement dynamique et non formel. Les méthodes formelles sont classiquement statiques : B [1], model checking [8], interprétation abstraite [9]... Nous nous intéressons dans cet article aux méthodes formelles dynamiques [1].

Ces travaux font suite à une thèse Cifre faite à Airbus et à l'Onéra, à propos de la vérification de propriétés temporelles [13, 12]. Pendant ces travaux, plusieurs logiciels embarqués ont été analysés pour extraire les propriétés difficiles à vérifier par les techniques de vérification existantes. Un langage adapté au contexte industriel a été défini pour formaliser ces propriétés. Ce langage combine la logique temporelle linéaire (LTL) et des expressions régulières, particulièrement adaptées aux propriétés de séquence. La thèse a permis de définir une méthode de vérification

*IFSTTAR/COSYS-ESTAS, 20 Rue Élisée Reclus, BP 70317, 59666 Villeneuve d'Ascq Cedex
†ONERA/DTIM, 2 avenue Édouard Belin, BP74025,31055 Toulouse

1. Conférences *Runtime Verification*, 2001-2015, `www.runtime-verification.org`

outillée adaptée à ces propriétés. L'approche développée fait partie des méthodes de *Runtime Verification*. La méthode développée se compose de deux étapes : la transformation de la propriété formalisée en automate de Büchi non-déterministe à l'aide de *Ltl2ba* [14], puis l'exécution de cet automate sur une trace d'exécution du programme à analyser. Comme LTL a une sémantique sur des traces infinies, une trace d'exécution finie est rendue infinie par bouclage sur le dernier état de la trace. Les effets de bord sont contrés par le calcul d'informations statistiques sur la trace pour guider l'interprétation du résultat, pour les cas litigieux. Le lecteur intéressé par plus de détails sur l'approche, notamment sur les effets de bord et les cas litigieux, pourra lire [13].

Cette approche est envisagée dans le domaine ferroviaire. Nous projetons de l'utiliser pour analyser des traces issues de la plate forme de simulation [2]. Cependant, la plateforme étant en pleine évolution dans le cadre du projet PERFECT [3], nous effectuerons ici des expérimentations sur des traces générées aléatoirement. Pour réduire davantage le temps de vérification lié à cette méthode, ce papier propose une version parallélisée de cette approche.

La section 2 positionnera cette approche suivant l'état de l'art et le contexte industriel. La section 3 résumera l'approche définie en [13]. La section 4 définira quelques notations utiles à la compréhension de ce papier, la section 5 formalisera précisément l'approche classique d'exécution d'un automate de Büchi sur une trace. On s'appuiera sur cette dernière pour formaliser l'approche parallélisée en section 6. Ensuite, la section 7 présentera les expérimentations menées, en mettant en relief les deux approches. Finalement, la conclusion (section 8) de cet article portera sur l'efficacité de l'approche parallèle et sur ses limitations.

2 Contexte

2.1 État de l'art

On peut classer les travaux de *Runtime Verification* en deux catégories : les méthodes *online*, qui effectuent des vérifications pendant l'exécution du programme, et les méthodes *offline* qui effectuent des vérifications sur des traces d'exécution de programme, à savoir des séquences d'états de programme. Il est à noter qu'un état correspond à un instant associé à l'ensemble des variables du programme et de leur valeur.

De nombreux travaux concernent la vérification *online*, en particulier pour les programmes Java [19, 11, 18, 17] et les communautés de programmation orientée aspects [23]. Certains travaux sont basés sur des règles de réécriture des propriétés [17] ou des techniques spécifiques de transformation de propriétés LTL en machines à état [16, 10].

La réduction du temps de vérification est un enjeu essentiel pour pouvoir traiter des logiciels industriels. De nombreux efforts sont faits en verification *online* dans ce sens. [5] propose de réduire le nombre de moniteurs à l'aide de l'analyse statique. [4] définit une approche parallélisant la vérification de plusieurs propriétés LTL trivaluées en utilisant le GPU [4] pour encoder les moniteurs. [25, 24, 20] dissocient les moniteurs des programmes à analyser.

L'approche proposée dans ce papier est une approche *offline*, les traces d'exécution seront issues à terme d'un simulateur ferroviaire. Le choix d'une approche de vérification a posteriori a été fait pour plusieurs raisons. L'une des plus importantes est qu'il ne faut pas perturber la simulation en ajoutant des processus dédiés à la vérification. En effet, nous souhaitons limiter le ralentissement de l'exécution dans l'application temps réel pour éviter des conséquences sur la véracité d'une propriété. Dans ce contexte de vérification a posteriori, trois facteurs jouent sur le temps de vérification : la taille de la trace, le nombre de variables dans la trace, et le format de la trace (binaire, Xml, ...). Ce dernier facteur n'est cependant pas optimisable puisque le simulateur est propriétaire, ce qui interdit également l'application d'une approche GPU qui nécessite une modification de la plateforme.

2. Plateforme propriétaire Ersa, European Rail Software Applications, `http://www.ersa-france.com`
3. `http://perfect.ifsttar.fr/Site`
4. *Graphic Processing Unit*

Dans plusieurs travaux, les traces sont obtenues par écoute de l'ensemble des variables du programme à analyser, même si seules quelques variables sont utiles pour vérifier une propriété donnée [21, 3]. Cette approche aboutit à la génération de traces imposantes. Il faut rappeler que le temps d'analyse est linéaire avec la taille de la trace et le nombre de variables.

Pour limiter la taille de ces traces, l'analyse statique [12] offre la possibilité de détecter tous les points de programme où un opérande de la propriété à vérifier varie. Un point d'observation est défini par des instructions d'extraction de variable(s) à un point de programme donné. Cependant, dans le cas d'une simulation de plusieurs heures, certaines variables changent constamment, ce qui augmente considérablement la taille de la trace.

Par conséquent, il est également utile d'améliorer l'efficacité de la méthode de vérification. [15] propose une méthode pour vérifier des traces d'applications parallèles. Les propriétés sont vérifiées en utilisant des processus parallèles (un pour chaque trace), au lieu de fusionner toutes les traces en une seule. Cela limite le champ d'application de cette méthode aux programmes parallèles. Si fragmenter la génération de la trace n'est pas possible, on peut découper la trace après sa génération pour analyser chaque sous-trace indépendamment les unes des autres. C'est ce que nous proposons dans cet article.

2.2 Le contexte ferroviaire

De nos jours, chaque nouvelle ligne au sein de l'Union Européenne doit nécessairement satisfaire les règles nationales et les règles de spécification ERTMS/ETCS [5]. Ces règles sont une proposition de l'Union Européenne au problème d'interopérabilité entre les différents pays, pour les systèmes embarqués vis-à-vis des infrastructures au sol. Ce nouvel environnement légal et technologique doit être implémenté dans tous les pays membres. En effet, chaque règle d'une nouvelle ligne ferroviaire satisfait la norme ERTMS/ETCS et les règles nationales.

Les logiciels considérés dans ce contexte sont critiques et leur vérification est essentielle pour éviter les collisions ou presque collisions [2]. Pour ce faire, une plateforme propriétaire permet de simuler les comportements des trains. Cette plateforme virtualise le centre de contrôle, les lignes avec les gares, balises, des trains et les communications avec le centre de contrôle. Un train peut être conduit par un opérateur comme un train réel, ou défini par le scénario. Nous proposons d'adapter et de paralléliser l'approche proposée dans [13] afin d'analyser des propriétés temporelles sur les traces d'exécution issues de simulations menées sur cette plateforme.

3 Approche de base

L'approche définie dans [13] est résumée dans cette section. Ce travail consistait à vérifier des propriétés temporelles sur des logiciels avioniques embarqués.

3.1 Un langage adapté au contexte industriel

Plutôt que d'utiliser un langage complexe préexistant, nous avons préféré définir un langage adapté à notre contexte. La première étape du travail a donc été d'étudier plusieurs logiciels avioniques pour en extraire les propriétés à vérifier. Le langage défini permet d'exprimer un grand nombre de propriétés temporelles (y compris des propriétés de durée et de séquence). Ce langage s'est révélé adapté au contexte ferroviaire également.

Les propriétés sont donc formalisées en utilisant une combinaison de

- LTL,
- expressions régulières,
- opérateurs numériques (entiers et flottants) : comparaison entre numériques, addition, soustraction, multiplication, division,
- opérateurs spécifiques ci-dessous :
  - $x?T$ est l'instant où x a été modifié

5. *European Railway Transport Management System/European Train Control System*

— $x?C$ est le nombre de modifications subi par la variable
— $x?n$ est la valeur de la variable n modifications auparavant,
— $x?n?T$ est l'instant où variable x a été modifiée n fois auparavant,
— $x?n?C$ est la valeur du compteur de modification, n modifications auparavant.

3.2 Génération de la trace

Les traces à analyser sont issues de simulations. L'approche propose une technique pour générer les points d'observation nécessaires à la vérification de la propriété considérée. Ces points d'observation sont définis comme tous les points de programme pour lesquels une variable intervenant dans la propriété est modifiée.

L'analyse statique a servi à collecter l'ensemble de ces points d'observation. Un plugin de Frama-C [7] appelé Breakpointer a été implémenté pour cette tâche. Il repose sur le plugin Frama-C Value Analysis, qui effectue une analyse sémantique de programme, et donc tient compte des alias de variables.

Dans ce papier, nous adaptons cette technique pour le simulateur ERTMS qui permet de générer les traces. Chaque trace du simulateur est sauvegardée dans une base de données. Une trace correspond à une table contenant des états, i.e. l'instant, un type de message et une valeur associée. Le type de message décrit les variables modifiées.

3.3 Vérification

La phase de vérification se fait en deux temps :

1. Transformation de la propriété temporelle en automate de Büchi via *Ltl2ba*
2. Vérification de la propriété par exécution synchrone de l'automate de Büchi sur la trace.

Lorsque la propriété suit un patron prédéfini, des informations statistiques sont calculées par un automate statistique déterministe exécuté en même temps que l'automate de Büchi. Pour cet automate, chaque transition possède une étiquette contenant un prédicat, et une liste d'opérations sur des compteurs. Les opérations sont effectuées sur les compteurs lorsque le prédicat est vrai. Les compteurs conservent les informations statistiques et sont fournis à l'opérateur en fin de vérification.

4 Rappels

4.1 Notations utilisées

Dans les sections suivantes seront utilisées les notations classiques pour la formalisation. Le sens de ces notations est rappelé ci-dessous :

— $\mathcal{P}(Q)$ est l'ensemble des parties de Q ;
— par soucis de clarté, nous allons adopter la convention suivante. Si E est un produit cartésien d'ensembles tel que $E = E_1 \times E_2 \times ... \times E_n$, alors $E_{|i}$ est la projection de E sur E_i. En d'autres termes, $E_{|i} = E_i$. Si E_i est un produit cartésien lui même, tel que $E_i = E_{i,1} \times ... \times E_{i,m}$, alors $E_{|i,j}$ fait référence à $E_{i,j}$. Cette convention est également applicable aux fonctions ; $f : X_1 \times ... \times X_n \to Y_1 \times ... \times Y_m$ est une fonction. $f_{|i}$ est la projection de f sur Y_i ;
— Si E est un ensemble, E^ω est le produit cartésien infini de E ;
— $[\![a; b]\!]$, tel que $(a, b) \in \mathbb{N}^2$ et $a < b$, est l'ensemble des entiers $\{a, a + 1, \ldots, b\}$;
— Une trace σ est une séquence d'états. Un état de trace $\sigma_i \in \Sigma$, tel que Σ est l'alphabet, à l'index $i \in [\![0; |\sigma| - 1]\!]$ de la trace σ est une fonction totale qui retourne pour chaque variable sa valeur. $|\sigma|$ est la taille de la trace σ. Dans ce papier, **une trace est considérée comme un mot** basé sur l'alphabet Σ, dans le sens de la théorie des langages [22].

On définit aussi des nouvelles notations :

— si $\mathbb{K}$ est un ensemble de valeurs, alors $\mathbb{K}^{¿} = \mathbb{K} \cup \{¿\}$, où ¿ est la valeur *inconnue* ;

— par la suite, pour éviter les confusions, on parlera d'un élément pour un état de trace, et d'un état, pour un état d'automate.

4.2 Rappel des définitions d'un automate de Büchi [6]

Dans cette section est rappelée la définition formelle d'un automate de Büchi. La définition formelle d'un automate de Büchi statistique est également proposée. Ensuite, l'exécution d'un automate de Büchi sera schématisée.

4.2.1 Automate de Büchi

La définition de l'automate de Büchi s'appuie sur celle d'un automate classique :

Définition 1 *Un automate est classiquement défini comme un 5-uplet $A = (Q, \Sigma, \rightarrow, q_0, F)$ tel que :*
— *Q est un ensemble d'état ;*
— *Σ est un alphabet ;*
— *$\rightarrow \subseteq Q \times \Sigma \times Q$ est une relation de transition ;*
— *$q_0 \in Q$ est l'état initial ;*
— *$F \subseteq Q$ est un ensemble d'états finaux.*

La condition d'acceptation d'un mot fini par un automate est donnée par la définition suivante :

Définition 2 *Un mot fini $w \in \Sigma^*$ est reconnu par un automate $A = (Q, \Sigma, \rightarrow, q_0, F)$, si et seulement s'il existe une séquence $(q)_{i \in [\![0;|w|]\!]}$, qui commence à l'état initial q_0, tel que $i \in [\![0; n-1]\!], (q_i, w_i, q_{i+1}) \in \rightarrow$ et tel que $q_n \in F$.*
Un mot w reconnu par A est écrit $w \in \mathcal{L}(A)$, où $\mathcal{L}(A)$ est l'ensemble des mots reconnus par A.

Un automate de Büchi est un automate classique (définition 4) dont la condition d'acceptation permet de manipuler les mots/traces infinies. On parlera d'états *acceptants* plutôt que d'états finaux. La condition d'acceptation d'une trace est définie par :

Définition 3 *Un mot infini $w \in \Sigma^\omega$ est un mot de $\mathcal{L}(B)$, avec $B = (Q, \Sigma, \rightarrow, q_0, F)$ l'automate de Büchi, si et seulement si :*
— *il existe une séquence $(q)_{i \in \mathbb{N}}$ telle que pour tout $i \in \mathbb{N}, (q_i, w_i, q_{i+1}) \in \rightarrow$;*
— *pour tout $j \in \mathbb{N}$, il existe $k \in \mathbb{N}$ tel que $k > j$ et $q_k \in F$.*

Dans ces travaux, une propriété est vérifiée sur une trace d'exécution. L'automate de Büchi correspondant est donc exécuté sur cette trace. L'alphabet utilisé est ainsi bâti sur les éléments.

Pour calculer des informations statistiques sur un mot infini donné, un automate de Büchi statistique est défini comme une extension d'un automate de Büchi déterministe. Lorsqu'une formule d'une transition donnée est vraie, des opérations primaires sont effectuées sur des compteurs donnés. À la fin de l'exécution de l'automate statistique, les compteurs quantifient des propriétés orthogonales à la propriété temporelle.

Exemple 1 *La propriété $\Box (\Diamond e)$ signifie que e se produit infiniment souvent. Le nombre d'occurrence de e est une information statistique.*

Dans cet article, l'automate de Büchi statistique remplace l'automate de Büchi classique lorsqu'une propriété temporelle correspond au patron de propriété de l'automate de Büchi statistique, contrairement à ce qui a été fait dans [13].

La première étape consiste à définir une opération statistique :

Définition 4 *Soit $\mathcal{C}$ un ensemble de variables entières appelées compteurs. L'ensemble des opérations statistiques $\Lambda_{\mathcal{C}} : (\mathcal{C} \rightarrow \mathbb{Z}^\delta) \rightarrow (\mathcal{C} \rightarrow \mathbb{Z}^\delta)$ est une liste d'action sur $\mathcal{C}$, dépendant de la valeur courante de tous les compteurs. Les opérations peuvent être :*

— *ne rien faire,*
— *affecter à un compteur une constante, la valeur d'un compteur, ou une expression numérique :*
  — *addition, soustraction,multiplication,division de compteurs/constantes*
  — *minimum, maximum de compteurs/constantes*

En cas d'exécution parallèle d'un automate de Büchi statistique déterministe, la valeur inconnue est nécessaire pour le calcul symbolique des compteurs.

Par définition, l'opération statistique λ, pour une transition donnée, est une liste d'actions.

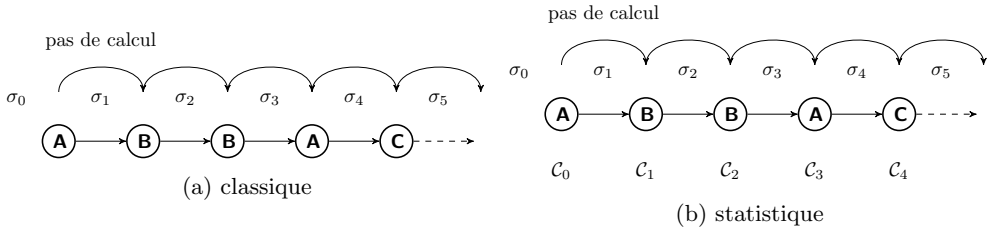
Enfin, l'automate de Büchi statistique est défini comme suit :

Définition 5 *Un automate de Büchi statistique est un 6-uplet $\mathcal{A} = (Q, \Sigma, \rightarrow, q_0, F, \mathcal{C}_0)$ tel que :*
— *Q est un ensemble d'états ;*
— *Σ est un alphabet ;*
— *$\rightarrow \subseteq Q \times \Sigma \times \Lambda_{\mathcal{C}} \times Q$ est une relation de transition ;*
— *$q_0 \in Q$ est un état initial ;*
— *$F \subseteq Q$ est un ensemble d'états acceptants ;*
— *$\mathcal{C}_0 : \mathcal{C} \rightarrow \mathbb{Z}^{\delta}$, est une fonction retournant la valeur initiale de chaque élément de $\mathcal{C}$.*

Par la suite, le terme *mot*$\in \Sigma^*$ est remplacé par trace d'exécution.

4.2.2 Exécution d'un automate de Büchi : trois cas étudiés

On utilisera par la suite trois cas d'automate de Büchi : déterministe, déterministe statistique et non déterministe. Avant de proposer une définition formelle d'une exécution d'automate pour chacun de ces cas, nous proposons une illustration dans la figure 1 de ces cas sur un exemple.



(a) classique (b) statistique

FIGURE 1 – Exécution d'un automate de Büchi déterministe

Un pas de calcul correspond à la consommation par l'automate de Büchi d'un élément pour activer les transitions accessibles. Lorsque l'automate de Büchi est déterministe (figure 1a), pour chaque pas de calcul, il n'y a qu'un état avant la consommation de l'élément courant et au plus un état après la consommation de cet élément. De ce fait, l'exécution d'un automate de Büchi peut être vue comme un chemin à plusieurs étapes (nœuds). Chaque nœud est un pas de calcul, et il n'y a qu'un seul lien entre deux nœuds.

L'exécution d'un automate de Büchi statistique déterministe correspond à l'exécution d'un automate de Büchi déterministe avec un ensemble de compteurs et leur valeur pour chaque état courant. Dans la figure 1b, les compteurs et leurs valeurs sont représentés par les $\mathcal{C}_i$.

Si l'automate de Büchi n'est pas déterministe, alors il y aura un ensemble d'états courants à la place d'un seul état. Le chemin est alors comparable à un treillis plutôt qu'à une séquence d'états. Dans la figure 2, chaque rectangle est l'ensemble des états courants après une étape de calcul. Le rectangle du niveau 1 est l'état initial, ensemble $\{A\}$. Celui du niveau 2 est $\{A, B\}$ après consommation du premier élément...

Schématiquement, un pas de calcul consiste à se déplacer d'un rectangle de niveau i à celui de niveau $i + 1$. Ainsi, la définition formelle d'une exécution de programme doit contenir deux ensembles d'états : un en début de pas de calcul, et un en fin de pas de calcul. Par la suite, cela sera pris en compte pour la formalisation d'une exécution.

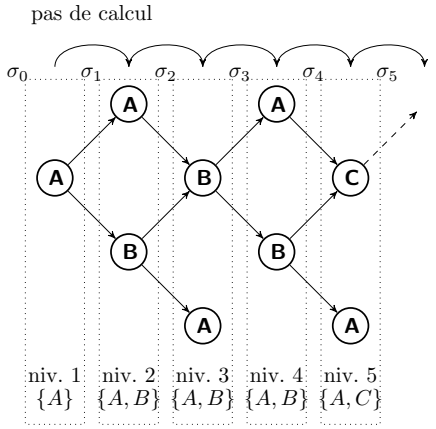


FIGURE 2 – Exécution d'un automate de Büchi non déterministe

L'exécution d'un automate de Büchi statistique n'est pas traitée ici. Par conséquent, dans la suite du papier, nous parlerons d'automate statistique pour un automate de Büchi statistique déterministe.

5 Exécution classique d'un automate de Büchi sur une trace

Cette section propose une formalisation de l'exécution séquentielle d'un automate de Büchi.

5.1 Exécution d'un automate non déterministe sur une trace

La formalisation de l'exécution d'un automate non déterministe servira de support aux formalisations proposées pour les autres cas. Dans l'approche de [13], les expressions régulières sont traduites en automates de Büchi déterministes avec un outil propriétaire AIRBUS, tandis que les formules LTL sont traduites en automates non déterministe par *Ltl2ba*.

Pour rappel, la définition 3 requiert une suite d'état, par conséquent, elle suffit pour traiter et formaliser complètement le premier cas. Cependant, dans le second cas, les automates de Büchi générés sont non déterministes. Pour le traitement de ce cas, la définition 3 est suffisante, néanmoins la formalisation complète nécessite la définition d'une suite non pas d'états mais d'ensembles d'états.

La formalisation d'une exécution d'un automate de Büchi non déterministe sur une trace repose sur la figure 2. Une exécution d'automate est une suite de pas de calcul. Chaque pas est une paire d'ensembles d'états : un ensemble avant la consommation d'un élément, et un après. Ainsi, nous définissons formellement l'exécution d'un automate de Büchi non déterministe de la manière suivante :

Définition 6 *Un mot infini $w \in \Sigma^\omega$ est un mot de $\mathcal{L}(B_{nd})$, où $B_{nd} = (Q, \Sigma, \rightarrow, q_0, F)$ est un automate de Büchi non déterministe, si et seulement si existe une suite $R_{i\in\mathbb{N}} \in \mathcal{P}(Q)^2$ tel que :*

- $R_0 = (\{q_0\}, \{q_0\}) \in \mathcal{P}(Q)^2$ (1)
- $\forall i \in \mathbb{N}^*$, $R_i \in \mathcal{P}(Q)^2$, *tel que :*
  - $R_{i|1} = R_{i-1|2}$ (2)
  - $\forall r_i \in R_{i|2}, \exists r_{i-1} \in R_{i|1}$, *tel que* $(r_{i-1}, w_i, r_i) \in \rightarrow$ (3)
- $\forall j \in \mathbb{N}$, $\exists k \in \mathbb{N}$ *tel que* $k > j$ *et* $\exists R_{k|1} \cap F \neq \emptyset$. (4)

La propriété 2 assure la consistance de l'exécution de l'automate : l'ensemble d'états courants en fin d'un pas de calcul est identique à l'ensemble d'états courants au début du pas de calcul

suivant. La propriété 3 est une partie de la définition d'un mot accepté par l'automate, traitant de l'existance d'un chemin pour un automate de Büchi donné. La propriété 4 est l'autre partie de la définition d'un mot accepté par l'automate, qui stipule l'existence d'un état acceptant infiniment souvent accessible.

Si un mot infini w n'est pas reconnu par un automate de Büchi non déterministe, alors soit il existe k tel que $R_{k|2} = \emptyset$ et pour tout $k' > k, R_{k'} = (\emptyset, \emptyset)$, soit la suite d'ensembles d'états courants contient un nombre fini d'états acceptants.

Pour la figure 2, on obtient la suite : $R_0 = (\{A\}, \{A\})$; $R_1 = (\{A\}, \{A, B\})$; $R_2 = (\{A, B\}, \{A, B\})$; $R_3 = (\{A, B\}, \{A, B\})$; $R_4 = (\{A, B\}, \{A, C\})$.

Dans notre contexte, les traces finies manipulées sont transformées en traces infinies par bouclage sur le dernier élément afin de permettre l'utilisation d'outils tel que *Ltl2ba* pour transformer des propriétés LTL en automate de Büchi, sans modification particulière de la sémantique de LTL. De plus, le problème de fin de trace n'est pas abordé ici, il a été traité dans [13].

La limite de cette méthode d'exécution est de devoir calculer séquentiellement la suite de pas de calcul $R_{i \in [\![0;|\sigma|-1]\!]}$.

5.2 Exécution d'un automate statistique sur une trace

La formalisation de l'exécution d'un automate statistique sur une trace est la suivante :

Définition 7 *Un mot infini $w \in \Sigma^\omega$ est un mot de $\mathcal{L}(B_S)$, où $B_S = (Q, \Sigma, \rightarrow, q_0, F, \mathcal{C}_0)$, si et seulement si il existe une suite $R^S_{i \in \mathbb{N}} \in (Q)^2 \times (\mathcal{C} \longrightarrow \mathbb{Z}^\lambda)^2$ telle que :*

- $R^S_0 = (q_0, q_0, \mathcal{C}_0, \mathcal{C}_0)$ (5)
- $\forall n \in \mathbb{N}^*$, $R^S_n \in (Q)^2 \times (\mathcal{C} \longrightarrow \mathbb{Z}^\lambda)^2$ *tel que :*
  - $R^S_{n|1} = R^S_{n-1|2}$ *et* $R^S_{n|3} = R^S_{n-1|4}$ (6)
  - $(R^S_{n|1}, w_i, \lambda_n, R^S_{n|2}) \in \rightarrow$ (7)
  - $R^S_{n|4} = \lambda_n(R^S_{n|3})$
- $\forall j \in \mathbb{N}$, $\exists k \in \mathbb{N}$ *tel que* $k > j$ *et* $R^S_{k|1} \in F$. (8)

Les propriétés 6 assurent la consistance de l'exécution de l'automate. La propriété 7 est une partie de la définition d'un mot accepté par l'automate. La propriété 8 est l'autre partie de la définition d'un mot accepté par l'automate, qui stipule qu'un état acceptant est infiniment souvent atteint. Dans la propriété 7, w_n est l'élément courant et λ_n l'opération courante appliquée aux compteurs.

Si on reprend la figure 1b, alors la suite sera : $R^S_0 = (A, A, \mathcal{C}_0, \mathcal{C}_0)$; $R^S_1 = (A, B, \mathcal{C}_0, \mathcal{C}_1)$; $R^S_2 = (B, B, \mathcal{C}_1, \mathcal{C}_2)$; $R^S_3 = (B, A, \mathcal{C}_2, \mathcal{C}_3)$; $R^S_4 = (A, C, \mathcal{C}_3, \mathcal{C}_4)$.

6 Exécution parallélisée d'un automate de Büchi sur une trace

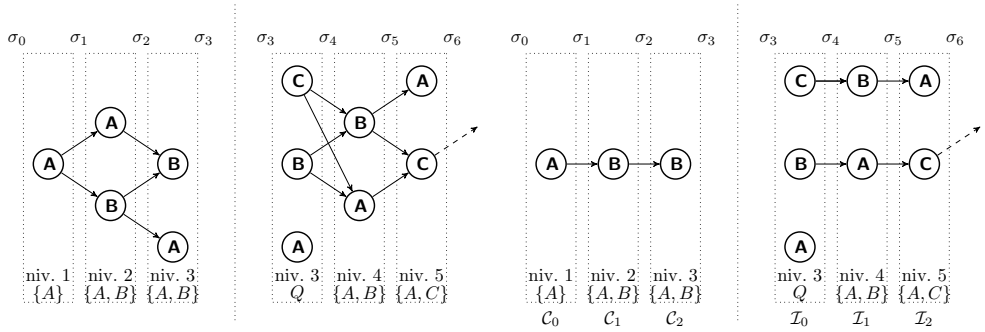
Nous arrivons maintenant au cœur de la problématique. Après en avoir expliqué les principes de l'approche, nous l'appliquons sur un automate de Büchi non déterministe, puis sur un automate statistique. Nous apportons ensuite la preuve de l'équivalence entre l'analyse séquentielle et parallèle d'une trace et évoquons quelques limites.

6.1 Principe

Jusqu'à présent, lors de l'analyse d'une trace, il était nécessaire de calculer la suite de pas de calcul $R_0, ..., R_{i-1}$, avant de pouvoir calculer le pas R_i. Nous proposons l'approche suivante :

1. découper la trace en plusieurs sous traces,
2. exécuter l'automate de Büchi B sur chaque sous trace indépendamment les unes des autres,
3. fusionner les résultats d'exécution intermédiaires pour déterminer si la trace est dans $\mathcal{L}(B)$.

La figure 3 schématise l'approche sur une trace divisée en deux, à l'état 3. L'automate de Büchi est exécuté sur les sous-traces $\sigma_0 \dots \sigma_2$ et $\sigma_3 \dots \sigma_5$. L'exécution de l'automate de Büchi sur la première trace suit la formalisation de la section précédente. L'exécution de l'automate de Büchi sur la seconde trace commence avec Q ($\{A, B, C\}$) comme ensemble d'états initiaux.



(a) non déterministe (b) statistique

FIGURE 3 – Principe de l'approche parallèle

La vérification de la propriété sur la trace entière est effectuée par l'utilisation d'une fonction agissant comme un raccourci entre le début et la fin de la seconde sous-trace.

Par exemple, si la trace se termine à l'état 5, alors on voit sur la figure 3a que :

— l'état A avant l'élément 3 ne conduit à rien après l'élément 5,

— les états B et C avant l'élément 3 conduisent tous les deux à A et C après l'élément 5.

Comme au niveau 3, l'ensemble des états courants est $\{A, B\}$, alors l'ensemble des états après l'élément 5 est $\{A, C\}$.

Après avoir illustré ce principe, nous allons le formaliser pour un automate de Büchi non déterministe et pour un automate statistique. Par souci de compréhension, la formalisation se fera sur une trace divisée en deux sous-traces, mais elle est généralisable à plusieurs divisions.

6.2 Exécution d'un automate de Büchi non déterministe

La trace σ est coupée en deux à l'état d'indice c. Il est cependant nécessaire que l'élément d'indice c soit complet. En d'autres termes, il faut que la valeur de chaque variable soit définie dans cet état, même si celle-ci n'est pas modifiée. Il suffit alors de définir une fonction pour cet état statuant sur la modification d'une variable ou non à cet état σ_c. L'automate sera exécuté sur chacune des deux sous-traces. Pour ce faire, deux suites de pas de calculs sont définies : $\mathcal{R}_{n\in[\![0;c]\!]}$ et $\overline{\mathcal{R}}_{n\in[\![0;|\sigma|-c+1]\!]}$. $\mathcal{R}_{n\in[\![0;c]\!]}$ est la suite de pas de calcul débutant à l'élément σ_0 comme définie dans la section 5. Ainsi, pour tout $n \in [\![0; c]\!]$, $\mathcal{R}_n = R_n$.

La suite de pas de calcul sur la seconde trace est définie de la façon suivante :

Définition 8 *La suite* $\overline{\mathcal{R}}_{n\in[\![0;|\sigma|-c+1]\!]} \in (\mathcal{P}(Q))^2 \times (Q \longrightarrow \mathcal{P}(Q))^2$ *est telle que :*

— *La définition des deux premiers composants de* $\overline{\mathcal{R}}_n$ *est la suivante :*

— $\overline{\mathcal{R}}_{0|1} = Q$ (9)

— $\forall n \in [\![0; |\sigma| + 1 - c]\!], \overline{\mathcal{R}}_{n|1} = \overline{\mathcal{R}}_{n-1|2}$. (10)

— $\forall n \in [\![0; |\sigma| + 1 - c]\!]$, $\forall r'_n \in \overline{\mathcal{R}}_{n|2}$, $\exists r_n \in \overline{\mathcal{R}}_{n|1}$, *tel que* $(r_n, \sigma_{c+n+1}, r'_n) \in \rightarrow$ (11)

— *La définition des deux derniers composants de* $\overline{\mathcal{R}}_n$ *correspond à la fonction* α *définie ci-après :*

— $\forall n \in [\![0; |\sigma| + 1 - c]\!], \overline{\mathcal{R}}_{n|3} = \alpha_n$, (12)

— $\forall n \in [\![1; |\sigma| + 1 - c]\!], \overline{\mathcal{R}}_{n|4} = \overline{\mathcal{R}}_{n+1|3}$, (13)

— $\forall q \in Q$, $\alpha_0(q) = \{q\}$. (14)

— $\alpha_n(q) = \{q'' | \exists q' \in \alpha_{n-1}(q), (q'\sigma_{c+n+1}, q'') \in \rightarrow\}$. (15)

Concrètement, $\overline{\mathcal{R}}$ est la suite de pas de calcul de l'automate débutant à l'élément d'index $c+1$ et finissant à l'élément d'index $|\sigma|-1$. L'ensemble d'états initiaux est Q. En effet, l'exécution de l'automate sur chaque sous-trace étant indépendante, on ne connaît donc pas l'ensemble des états courants de l'automate en début de la seconde sous-trace. C'est l'opération de fusion qui permet de recalculer a posteriori cette information, grâce à la fonction α_n. En effet, celle-ci sauvegarde le lien entre l'ensemble des états à l'élément $c+1$ et l'ensemble des états à l'élément $|\sigma|-1$.

La propriété 9 signifie que la suite de pas de calcul est initialisée avec Q comme ensemble d'états initiaux. Les propriétés 10 et 11 assurent respectivement la consistance de l'exécution et l'acceptation d'une trace par l'automate. α est initialisée par la fonction identité (propriété 14). Pour chaque état, α est calculée récursivement en fonction des transitions activées (propriété 15).

Si on reprend l'exemple de la figure 3a, alors la suite sera : $\mathcal{R}_0 = (\{A\}, \{A\})$; $\mathcal{R}_1 = (\{A\}, \{A, B\})$; $\mathcal{R}_2 = (\{A, B\}, \{A, B\})$; $\overline{\mathcal{R}}_0 = (\{A, B, C\}, \{A, B\}, \alpha = id, \{\alpha(A) = \emptyset, \alpha(B) = \{A, B\}, \alpha(C) = \{A, B\}\})$, $\overline{\mathcal{R}}_1 = (\{A, B\}, \{A, C\}, \{\alpha(A) = \emptyset, \alpha(B) = \{A, B\}, \alpha(C) = \{A, B\}\}, \{\alpha(A) = \emptyset, \alpha(B) = \{A, C\}, \alpha(C) = \{A, C\}\})$.

L'exécution de l'automate de Büchi sur chacune des deux sous-traces données est indépendante. Les résultats de l'analyse de chaque sous-trace doivent être assemblés après, grâce à la fonction α.

Pour effectuer une exécution parallélisée d'un automate de Büchi en plus de deux sous-traces, la démarche à suivre est la même que pour deux sous-traces. En effet, le préfixe de la trace est analysé normalement et pour chaque autre sous-trace on utilise les définitions de $\overline{\mathcal{R}}$.

6.3 Exécution d'un automate statistique

Soit σ la trace, $B_S = (Q, \Sigma, \rightarrow, q_0, F, \mathcal{C}_0)$ est un automate de Büchi statistique déterministe.

La suite de pas de calcul ressemble à $\overline{\mathcal{R}}$ définie dans la section 6.2. Une fonction additionnelle est calculée comme la fonction α, pour construire les opérations appliquées aux compteurs entre le début et la fin de chaque sous-trace.

$\mathcal{R}^S_{n\in[\![0;c]\!]}$ est la suite de pas de calcul débutant à l'élément σ_0 comme défini dans la section 5.2. Ainsi, pour tout $n \in [\![0; c]\!]$, $\mathcal{R}^S_n = R^S_n$.

L'autre suite de pas de calcul est construite comme celle de la section précédente, en tenant compte du calcul des informations statistiques :

Définition 9 $\overline{\mathcal{R}}^S_{n\in[\![0;|\sigma|+1-c]\!]} \in (\mathcal{P}(Q))^2 \times (Q \rightarrow \mathcal{P}(Q))^2 \times (Q \rightarrow \Lambda_{\mathcal{C}})^2$*, où, pour tout* $n \in [\![0; |\sigma| + 1 - c]\!]$ *:*

- $\overline{\mathcal{R}}^S_{n|1} = \overline{\mathcal{R}}_{n|1}$, $\overline{\mathcal{R}}^S_{n|2} = \overline{\mathcal{R}}_{n|2}$
- $\overline{\mathcal{R}}^S_{n|3}$ *(fonction* α*) est proche de* $\overline{\mathcal{R}}_{n|3}$*. La dernière équation à propos de* α *définie à la section 6.2 est la seule qui change :*
  - $\forall q \in Q, \alpha_n(q) = \bigcup q''$ *tel que* $\exists q \in \alpha_{n-1}, (q', \sigma_{c+n+1}, \lambda_{n,q'}, q'') \in \rightarrow$
  - *la propriété* $\overline{\mathcal{R}}^S_{n|4} = \overline{\mathcal{R}}^S_{n+1|3}$ *est préservée*
- $\forall q \in Q, \overline{\mathcal{R}}^S_{0|5}(q) = Inconnu.$
- $\forall q \in Q, \overline{\mathcal{R}}^S_{n|6}(q) = \lambda_{(n,q')}(\overline{\mathcal{R}}^S_{n|5}(q))$.
- $\forall q \in Q, \overline{\mathcal{R}}^S_{n|6}(q) = \overline{\mathcal{R}}^S_{n+1|5}(q)$.

Si on reprend la figure 3b, en ajoutant les informations statistiques, alors la suite sera :
$\mathcal{R}_0 = (\{A\}, \{A\}, \mathcal{C}_0, \mathcal{C}_0)$; $\mathcal{R}_1 = (\{A\}, \{A, B\}, \mathcal{C}_0, \mathcal{C}_1)$; $\mathcal{R}_2 = (\{A, B\}, \{A, B\}, \mathcal{C}_1, \mathcal{C}_2)$;
$\overline{\mathcal{R}}_0 = (\{A, B, C\}, \{A, B\}, \alpha = id, \{\alpha(A) = \emptyset, \alpha(B) = \{A\}, \alpha(C) = \{B\}\}, \mathcal{I}_0, \mathcal{I}_1)$;
$\overline{\mathcal{R}}_1 = (\{A, B\}, \{A, C\}, \{\alpha(A) = \emptyset, \alpha(B) = \{A\}, \alpha(C) = \{B\}\}, \{\alpha(A) = \emptyset, \alpha(B) = \{C\}, \alpha(C) = \{A\}\}, \mathcal{I}_1, \mathcal{I}_2)$. Si la transition de X à Y provoque le calcul statistique $\lambda_{X,Y}$, alors :
$\mathcal{I}_2(A) = Inconnu$,
$\mathcal{I}_2(B) = \lambda_{A,C}(\mathcal{I}_1(B)) = \lambda_{A,C}(\lambda_{B,A}(Inconnu))$
$\mathcal{I}_2(C) = \lambda_{B,A}(\mathcal{I}_1(C)) = \lambda_{B,A}(\lambda_{C,B}(Inconnu))\}$

6.4 L'opération de fusion

L'opération de fusion consiste à rassembler les informations calculées sur les deux sous traces. L'ensemble d'états courants obtenu après l'exécution de l'automate sur la première sous trace se combine avec les informations de la fonction α calculée pendant l'exécution de l'automate sur la seconde sous-trace.

Soit $\mathcal{F}$, l'ensemble des états de l'automate à l'élément $|\sigma| - 1$. Alors $\mathcal{F} = \{q \in \overline{\mathcal{R}}_{|\sigma|-1|2}$, tel que $\exists q' \in \mathcal{R}_{c|2}$ et $q \in \alpha_n(q')\}$.

Théorème 1 *La vérification de la propriété par exécution séquentielle d'un automate de Büchi est équivalent à la vérification de cette propriété par exécution parallélisée de l'automate de Büchi.*

Si la trace est découpée en plus de deux sous-traces, il suffit d'assembler les résultats séquentiellement.

Preuve 1 *Par construction, la fonction α calcule un raccourci d'exécution : pour chaque état de $\overline{\mathcal{R}}_{0|2}$ à l'élément σ_c, on a l'ensemble des états qu'il produit $\overline{\mathcal{R}}_{|\sigma||1}$ en fin d'exécution. L'opération de fusion consiste à déterminer les éléments de $\overline{\mathcal{R}}_{|\sigma||1}$ à conserver. Les éléments à conserver sont ceux que génère l'ensemble $\mathcal{R}_c$ par application de la fonction α. Par définition de $\mathcal{F}$, on a donc $\mathcal{F} = R_{|\sigma||1}$*

De ce fait, une propriété peut être vérifiée en utilisant l'algorithme de fin de trace défini dans [13] et $\mathcal{F}$.

6.5 Limitations

Dans [13], des opérateurs spécifiques sur des variables ont été définis pour calculer à la volée des compteurs de modification de ces variables, ou obtenir une valeur antérieure à un nombre de modifications donné. Des exemples sont donnés dans la section 3.1.

Comme ces éléments sont calculés à la volée, il est impossible d'utiliser la méthode de vérification parallèle. Pour contourner ce problème, on pourrait ajouter des nouvelles variables dans la trace pour obtenir ces informations directement.

7 Résultats expérimentaux

Des résultats expérimentaux sont présentés dans cette section. Les essais sont réalisés sur des traces génériques pour les raisons évoquées dans la section 1. Le but de ces expériences est d'évaluer l'efficacité de la nouvelle approche et de la comparer, en terme de temps d'exécution, avec l'approche séquentielle.

7.1 Implémentation d'un prototype

Pour des raisons de confidentialité, il est impossible de fournir le code source du programme implémentant l'approche de ce papier. Cependant, certains éléments techniques sont fournis. Le prototype est écrit en C++ et comporte 32900 lignes de code, dont 13000 lignes de fichier *header*.

7.1.1 Une architecture distribuée

Les deux approches (séquentielles et parallèles) ont été conçues sur une architecture distribuée, en vue de partager efficacement les ressources matérielles disponibles. Ainsi, l'opération de lecture de la trace est séparée de l'exécution de l'automate de Büchi. La figure 4 illustre la relation entre les différents processus de l'approche séquentielle. Le processus d'exécution et le processus de lecture communiquent via une interface TCP/IP. Le processus d'exécution charge les informations à la volée d'un ou plusieurs processus de lecture. En effet, chaque processus peut lire une sous-trace différente simultanément avec les autres processus. La figure 5, quant à elle, schématise les relations de communication entre les différents processus de l'approche parallèle ; néanmoins, l'opération de

fusion n'apparaît pas. Chaque processus d'exécution peut interagir avec un ou plusieurs processus de lecture.

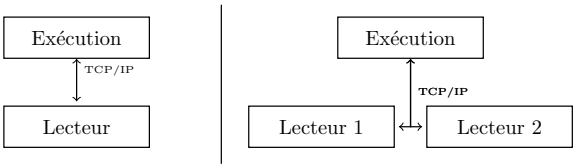


FIGURE 4 – Implémentation séquentielle

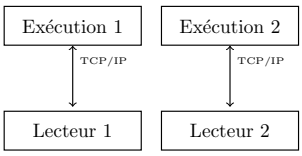


FIGURE 5 – Implémentation parallèle

7.1.2 Le format de trace

Un format de trace basé sur Xml a été implémenté. La trace est découpée en plusieurs blocs qui contiennent le même nombre d'éléments $|b|$. Ainsi, lorsque le processus d'exécution a besoin d'un élément σ_i, il appelle le processus de lecture qui retourne le bloc complet contenant σ_i. Ce bloc σ_i contient donc la sous-trace $\sigma_i ... \sigma_{i+|b|-1}$. Les blocs sont utilisés pour deux raisons. D'une part, cela limite les problèmes de latence dus au réseau. Des essais ont montré que le temps de vérification augmentait considérablement si les éléments étaient envoyés les uns après les autres. D'autre part, les sous-traces de la version parallélisée suivent la segmentation par bloc. En effet, les traces utilisées sont partielles [6], mais les éléments σ_i et $\sigma_{i+|b|-1}$ sont complets [7] pour chaque bloc b commençant à σ_i.

7.2 Expériences de charge

Deux expériences sont faites sans information statistique, et avec informations statistiques. Les essais sont menés sur la même trace générique pour les deux expériences. Il s'agit d'une trace à 10 millions d'éléments et ne contenant qu'une variable appelée x. La valeur de la variable change à chaque élément. Le domaine de variation de la variable est $[\![-10; 10]\!]$. Pour effectuer une vérification sur plusieurs tailles de trace, le processus peut être arrêté après un nombre donné d'états. Ainsi, on peut vérifier un préfixe de la trace pour simuler des traces plus petites.

La figure 7 rassemble les résultats pour les deux expériences. La trace a été découpée en 2, 4, 8 ou 10 sous-traces. La vérification a été effectuée sur chaque sous-trace et le temps maximum pour chaque classe de division a été placé dans ce graphique. Le processus de lecture a été lancé sur un ordinateur de 24Go-RAM, tandis que le processus d'exécution a été exécuté sur un ordinateur de 4Go-RAM avec un processeur Core-5i (2Ghz, 3Mo de cache, 64-bits).

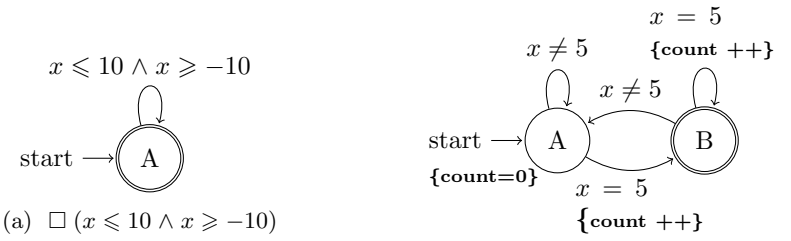
Sans information statistique La propriété à vérifier sur cette trace est $\Box\ (x \leqslant 10 \wedge x \geqslant -10)$. Cela signifie que la contrainte $x \leqslant 10 \wedge x \geqslant -10$ est vraie pour tout élément de la trace. Le but de cet essai est de comparer les temps de vérification entre l'approche séquentielle et l'approche parallèle lorsqu'il n'y a pas d'information statistique

L'automate de Büchi de cette propriété obtenue avec *Ltl2ba*, est représenté en figure 6a.

En comparaison avec le temps de vérification séquentiel, le temps de vérification est réduit de 40% si la trace est divisée en deux sous-traces. Si la trace est divisée en 10 sous-traces, le temps de vérification est réduit de 90%. Le temps de fusion reste négligeable, à moins de 0.01 seconde lorsque la trace est divisée en 10.

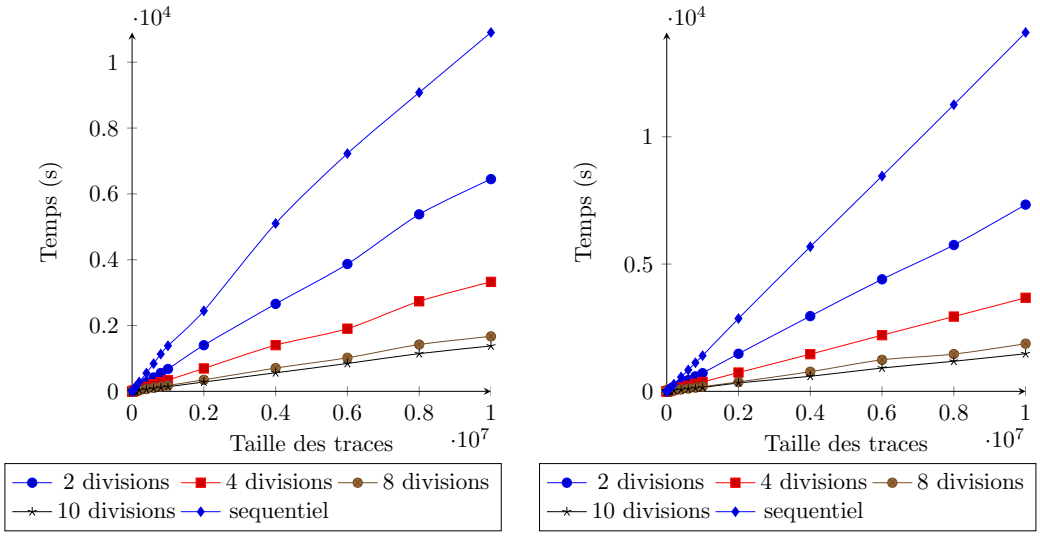
Avec information statistique La propriété étudiée est $\Box\ (\Diamond\ (x = 5))$. Elle signifie que la contrainte $x = 5$ se produit infiniment souvent. Pendant l'analyse de la trace, le nombre d'éléments où la propriété $x = 5$ survient est calculé à l'aide du compteur *count*, dont la valeur initiale est 0.

6. Un élément ne contient que les variables modifiées
7. Toutes les variables et leur valeurs courantes sont dans cet état.



(a) $\square\,(x \leqslant 10 \wedge x \geqslant -10)$ (b) $\square\,(\Diamond\,(x = 5))$

FIGURE 6 – Automates de Büchi des expériences



(a) Sans information statistique (b) Avec information statistique

FIGURE 7 – Temps maximum requis pour la vérification d'une trace

La figure 7b montre que la vérification de la propriété est plus efficace lorsque la trace est découpée, que lorsque la trace est analysée séquentiellement. Le temps de vérification est environ divisé par deux lorsque la trace est divisée par deux, divisé par 4 lorsque la trace est divisée par 4... Le temps de fusion reste négligeable, moins de 0.01 seconde lorsque la trace est divisée en 10.

Cette approche parallèle est donc plus efficace que l'approche séquentielle. L'opération de fusion des résultats intermédiaires est négligeable pour dix divisions. La prochaine étape de ces travaux doit consister à analyser une trace d'exécution provenant d'un cas réel industriel, avec des propriétés plus complexes et des traces contenant plus de variables. En effet, les automates de Büchi générés par des propriétés plus complexes, ont généralement plus d'états et de transitions que ceux des expériences menées ci-dessus.

8 Conclusion

Dans cet article, nous avons formalisé l'exécution sur une trace d'automates de Büchi de type non-déterministe, déterministe ou statistique. Ces formalisations reposent sur la définition formelle d'un automate de Büchi et les conditions d'acceptation d'un mot, qui ne décrivent pas précisément chaque pas de calcul. Cette formalisation est un socle pour la contribution de ce papier qui consiste à paralléliser l'exécution d'un automate de Büchi sur une trace.

La trace analysée est découpée en deux sous-traces, puis l'automate de Büchi est exécuté sur

chacune d'entre elles. L'exécution sur la première sous-trace est classique tandis que sur l'autre elle varie. En effet, l'automate de Büchi est initialisé avec l'ensemble des états de l'automate au lieu de l'état initial. Pendant l'exécution de l'automate sur la seconde sous-trace, un raccourci entre un état au premier élément de la sous-trace et l'ensemble des états qu'il génère au dernier élément de la sous-trace est calculé. Ce raccourci est ensuite utilisé pour fusionner les résultats d'analyse obtenus entre la première et la seconde sous-trace et permet de déduire le résultat global.

Cette nouvelle approche a été appliquée sur des traces génériques de différentes tailles. Le temps de vérification de l'approche parallèle a été comparé à celui de l'approche séquentielle. Le temps de vérification est divisé environ par le nombre de coupures effectuées sur la trace. Cette nouvelle approche limite donc le temps de vérification et la mémoire requise pour analyser une trace, puisque l'analyse indépendante des sous-traces est possible. La prochaine étape de ces travaux consistera à analyser des traces d'exécution issues de cas industriels réels, et d'effectuer les comparaisons entre l'approche séquentielle et l'approche parallèle.

Un léger inconvénient de cette nouvelle approche est que les opérateurs spécifiques développés en [13] ne peuvent pas être employés, puisqu'ils induisent un calcul à la volée non parallélisable. Malgré tout, nous avons montré que le temps de vérification reste meilleur avec l'approche parallèle. Cette approche inclut également le calcul d'informations statistiques sur une trace.

Une des perspectives de ces travaux pourrait être d'enrichir la trace avec les informations nécessaires à l'utilisation des opérateurs spécifiques. Une autre piste de recherche serait de déterminer le nombre de coupures de trace maximum permettant un gain de temps de calcul.

Remerciement

Ce travail est financé par l'ANR dans le contexte du projet Perfect. Ce projet est également supporté par le pôle de compétitivité national ITrans.

Références

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

[2] F. Aguirre, M. Sallak, W. Schon, and F. Belmonte. Application of evidential networks in quantitative analysis of railway accidents. Proceedings of the Institution of Mechanical Engineers, Part O, Journal of Risk and Reliability, 227(4) :368–384, Nov 2013.

[3] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Eagle does space efficient ltl monitoring. Technical report, Nasa, 2003.

[4] Shay Berkovich, Borzoo Bonakdarpour, and Sebastian Fischmeister. Gpu-based runtime verification. In Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, IPDPS '13, pages 1025–1036, Washington, DC, USA, 2013. IEEE Computer Society.

[5] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. In Software Engineering, 2010 ACM/IEEE 32nd International Conference on, volume 1, pages 5–14, May 2010.

[6] J.Richard Büchi. On a decision method in restricted second order arithmetic. In Saunders Mac Lane and Dirk Siefkes, editors, The Collected Works of J. Richard Büchi, pages 425–435. Springer New York, 1990.

[7] CEA-LIST and INRIA-Saclay. The frama-c platform for static analysis of c programs, 2008.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., 8(2) :244–263, April 1986.

[9] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, 1977.

[10] Marcelo d'Amorim and Grigore Rosu. Efficient monitoring of omega-languages. In CAV'05, pages 364–378, 2005.

[11] Doron Drusinsky. The temporal rover and the atg rover. In K. Havelund, J. Penix, and W. Visser, editors, SPIN Model Checking and Software Verification, volume 1885 of Lecture Notes in Computer Science. Springer, 2000.

[12] A. Ferlin and V. Wiels. Combination of static and dynamic analyses for the certification of avionics software. In Software Reliability Engineering Workshops (ISSREW), 2012 IEEE 23rd International Symposium on, pages 331–336, Nov.

[13] Antoine Ferlin. Vérification de propriétés temporelles sur des logiciels avioniques par analyse dynamique formelle. PhD thesis, Institut Supérieur de l'Aeronautique et de l'Espace (ISAE), Université de Toulouse, ED-MITT, September 2013.

[14] P. Gastin and D. Oddoux. Fast LTL to Büchi automata translation. In Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01), volume 2102 of LNCS. Springer, 2001.

[15] Markus Geimer, Felix Wolf, BrianJ.N. Wylie, and Bernd Mohr. Scalable parallel trace-based performance analysis. In Bernd Mohr, JesperLarsson Träff, Joachim Worringen, and Jack Dongarra, editors, Recent Advances in Parallel Virtual Machine and Message Passing Interface, volume 4192 of Lecture Notes in Computer Science, pages 303–312. Springer Berlin Heidelberg, 2006.

[16] D. Giannakopoulou and K. Havelund. Automata-based verification of temporal properties on running programs. In Automated Software Engineering, 2001.

[17] K. Havelund and G. Rosu. Monitoring programs using rewriting. In Automated Software Engineering, pages 135 – 143, 2001.

[18] Klaus Havelund and Kestrel Technology. A rewriting-based approach to trace analysis. Automated Software Engineering, 12 :2005, 2002.

[19] Patrick O'Neil Meredith, Dongyun Jin, Dennis Griffith, Feng Chen, and Grigore Roşu. An overview of the mop runtime verification framework. International Journal on Software Tools for Technology Transfer, 14 :249–289, 2012.

[20] R. Pellizzoni, P. Meredith, M. Caccamo, and G. Rosu. Hardware runtime monitoring for dependable cots-based real-time embedded systems. In Real-Time Systems Symposium, 2008, pages 481–491, Nov 2008.

[21] A. Pnueli and A. Zaks. Psl model checking and run-time verification via testers. In FM 2006 : Formal Methods, volume 4085 of LNCS. Springer, 2006.

[22] Grzegorz Rozenberg and Arto Salomaa, editors. Handbook of Formal Languages, Vol. 1 : Word, Language, Grammar. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[23] Volker Stolz and Eric Bodden. Temporal assertions using aspectj. Electronic Notes in Theoretical Computer Science, 144(4) :109 – 124, 2006. Proceedings of the Fifth Workshop on Runtime Verification (RV 2005).

[24] Haitao Zhu, M.B. Dwyer, and S. Goddard. Predictable runtime monitoring. In Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on, pages 173–183, July 2009.

[25] C.B. Zilles and G.S. Sohi. A programmable co-processor for profiling. In High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on, pages 241–252, 2001.

Une argumentation pour des exigences temps réel

Thomas Polacsek, Virginie Wiels, Frédéric Boniol
ONERA, Département Traitement de l'Information et Modélisation
2, avenue Edouard Belin BP74025, 31055 TOULOUSE Cedex 4
{prénom.nom}@onera.fr

Abstract

Cet article propose des patrons d'argumentation permettant de construire une argumentation de la tenue des exigences temps réel d'une architecture modulaire embarquée. Ces patrons s'appuient sur une description systématique des exigences temps réel applicables à ces architectures dans le domaine aéronautique. Ces travaux s'inscrivent dans un projet de définition d'un référentiel de certification des architectures modulaires embarquées intégrant une description des composants de telles architectures, des exigences attendues et des moyens de conformité acceptables pour démontrer la tenue de ces exigences.

1 Introduction

Dans le domaine aéronautique, la certification est une contrainte structurante à tous les niveaux de développement de l'avion. Des standards de certification existent pour chacun des domaines de conception de l'avion ; ces standards fixent des objectifs à satisfaire afin que l'avion soit autorisé à voler. Par exemple, le DO-178 (standard de certification du logiciel) définit des objectifs pour le développement et la vérification des logiciels embarqués en fonction de leur niveau de criticité.

Un dossier de certification regroupe tous les documents, toutes les justifications, qui valident qu'un système satisfait les objectifs spécifiques. Pour le UK Ministry of Defence Standard [9], ce dossier consiste en *"une argumentation raisonnée et auditable, créée pour soutenir l'affirmation selon laquelle un système satisfait des exigences"*. Concernant la forme de ce dossier si les premières versions étaient un ensemble de documents sans le moindre graphique, il semble aujourd'hui que la tendance soit d'en donner une représentation graphique. Bien qu'il n'y ait pas de véritable consensus sur une unique représentation, elles sont toutes de la forme d'un graphe. Par exemple, Eurocontrol dans [10] a choisi la représentation GSN (Goal Structured Notation) [6].

L'IMA (pour "Integrated Modular Avionics") est un nouveau concept d'architecture embarquée qui est utilisé pour l'implémentation des systèmes embarqués depuis l'A380 pour Airbus et le B777 pour Boeing et dans une version militaire depuis le Rafale pour Dassault Aviation. Un standard de certification spécifique a été défini (DO-297) mais n'est pour l'instant pas utilisé par tous les industriels. La certification de cette première génération d'IMA n'a pas été simple (voir [12]) notamment à cause de problèmes liés au partage de ressources mis en œuvre dans ces architectures et aux effets de bords résultants.

L'ONERA mène des travaux en collaboration avec le DGA/TA (en charge pour la France de la certification des avioniques civiles et militaires) pour définir un référentiel appelé AMEM (pour Avioniques Modulaires Embarquées Militaires). L'objectif de ce référentiel est de formaliser les exigences attendues pour les architectures IMA et les moyens de conformité acceptables pour la tenue de ces exigences. Ce référentiel en construction pourrait servir de base à l'évaluation de solutions d'architectures en vue de leur certification.

Cet article présente une partie du référentiel AMEM : nous nous focalisons sur l'argumentation pour les exigences temps réel des architectures modulaires embarquées. La section 2 rappelle les principes de l'IMA et définit les concepts utilisés par la suite. La section 3 donne une vue d'ensemble du référentiel AMEM, cadre de ces travaux. La section 4 propose une classification et une décomposition des exigences temps réel applicables aux fonctions, systèmes,

logiciels et matériels embarquées. La section 5 présente les patrons d'argumentation que nous avons définis afin de construire l'arbre d'argumentation permettant de justifier la tenue des exigences temps réel de la section 2. La section 6 conclut et décrit les travaux futurs.

2 Architectures IMA (Integrated Modular Avionics)

Les architectures IMA (appelées aussi dans la suite "architectures modulaires embarquées") sont apparues au début des années 2000 pour répondre au besoin de réduire le nombre de ressources embarquées. L'idée, classique dans les autres domaines de l'informatique, était d'introduire des ressources de calcul (calculateurs) et de communication (réseaux) suffisamment génériques pour les partager entre l'ensemble des applications exécutées dans l'avion (pilote automatique, freinage, fuel, etc.)[1]. Les systèmes avioniques sont donc désormais des systèmes concurrents, au sens où les applications concourent pour l'accès aux ressources. Pour gérer cette concurrence, un modèle d'exécution et de communication a été défini par les standards ARINC 653 [1] et ARINC 664 [2] adoptés par Airbus depuis l'A380 et par Boeing depuis le B777. L'objectif de ces standards est d'assurer un certain niveau de "prédictibilité temporelle", malgré la concurrence entre fonctions, en introduisant des principes de "partitionnement". L'ARINC 653 définit les règles de partage des calculateurs. Chaque application est implantée par un ensemble de tâches logicielles ordonnancées dans une ou plusieurs partitions. Chaque partition est allouée sur un calculateur dans une "tranche de temps" séquencée statiquement selon une trame cyclique (appelée MAF pour MAjor time Frame). L'ordonnancement des tâches à l'intérieur d'une partition est en revanche laissé libre au concepteur de l'application. Dans la pratique, cet ordonnancement est souvent un ordonnancement temps réel à priorités fixes (du type Rate Monotonic). L'ARINC 664 définit les règles de partage du réseau de communication inter-calculateurs. Les tâches communiquent d'une partition à une autre via des liens virtuels (notés VLs pour "virtual links"), eux-mêmes implantés par chemins statiques à travers le réseau (par exemple des arbres de diffusion sur un réseau commuté). Un VL est essentiellement caractérisé par un BAG (Bandwidth Allocation Gap) définissant l'intervalle de temps minimal séparant l'émission de deux trames consécutives à partir d'une même partition. Le respect de cette contrainte par chaque partition assure la fluidité du trafic dans le réseau, et une certaine prédictibilité des temps de communication.

Les applications implantées sur des architectures IMA sont souvent critiques. Il est donc impératif de pouvoir montrer que leur implantation est correcte vis-à-vis d'un ensemble d'exigences. Cet article s'intéresse à cette question et étudie la notion d'argumentation de bon fonctionnement pour des applications concurrentes s'exécutant sur une architecture modulaire embarquée.

3 Vue d'ensemble du référentiel AMEM

Le référentiel nommé AMEM (Architectures Modulaires Embarquées Militaires) est constitué

- d'un modèle de données central définissant les concepts nécessaires à la description d'une architecture modulaire embarquée et leurs relations ;
- d'un modèle d'exigences ciblant deux vues particulières : la sûreté de fonctionnement, et le temps réel ;
- et d'un modèle d'argumentation associant à chaque exigence des justifications (basées sur des normes, des raisonnements, des moyens de démonstration, etc).

Ce référentiel n'est pas défini dans le but de concevoir des architectures modulaires embarquées, mais pour d'une part formaliser précisément les besoins (les propriétés essentielles) de ce type d'architectures (volet exigences du référentiel), et d'autre part définir face à ces besoins des moyens de conformité acceptables et la façon de les composer (volet argumentation du référentiel). Ce référentiel pourra ensuite être utilisé par la DGA pour poser les bonnes questions (par rapport aux exigences de certification) sur des solutions proposées par les industriels. Il permettra également de mieux évaluer l'adéquation des moyens de conformité mis

[1] Voir [3] pour un bref historique sur les architectures avioniques et l'IMA.

en regard des exigences (moyens proposés par les industriels pour démontrer la conformité du produit par rapport aux objectifs définis par la certification).

3.1 Modèle de données

Le modèle de données est composé de quatre niveaux : Fonctionnel, Logiciel, Topologie, Matériel. Ces quatre niveaux permettent de décrire une architecture modulaire embarquée à différents niveaux d'abstraction. Trois modèles supplémentaires permettent de décrire les mappings entre les niveaux (fonctionnel sur logiciel, logiciel sur topologie, logiciel et topologie sur matériel). Nous ne décrirons pas ici le détail de ces sept niveaux, mais ils correspondent aux différents niveaux de description utilisés lors de la conception d'architectures embarquées dans le domaine aéronautique.

Ce modèle multi-niveaux permet de décrire une architecture modulaire embarquée à différents niveaux d'abstraction. Les 3 modèles de mappings permettent de plus de s'assurer de certaines propriétés structurelles de la description.

Le modèle de données inclut également des vues spécifiques dont l'objectif est de définir des concepts spécifiques à des points de vue donnés. Les deux vues considérées sont la sûreté de fonctionnement et le temps réel.

3.2 Modèle d'exigences

Le modèle d'exigences a pour objectif de permettre l'expression d'exigences attachées aux architectures modulaires embarquées. Il définit un concept général d'exigences et un concept de justification. A chaque exigence est associée une justification qui permet de faire le lien avec le modèle d'argumentation. Nous nous sommes intéressés particulièrement aux exigences de sûreté de fonctionnement et aux exigences temps réel. Nous avons proposé une classification des exigences dans ces deux domaines et formalisé cette classification dans deux modèles dédiés.

3.3 Modèle d'argumentation

Parallèlement au modèle d'exigence, nous avons développé un modèle d'argumentation. Le but de l'argumentation est de pouvoir présenter sous une forme générique comment des éléments de preuve, tels qu'une pratique acceptée, des tests, l'utilisation de méthodes formelles, etc. peuvent être autant de justifications acceptables en vue d'établir une propriété et de montrer comment sont structurés ces éléments dans un format auditable. Pour cela, dans [11], nous avons proposé un schéma d'argumentation en nous inspirant de divers travaux issus de l'argumentation et plus précisément des travaux de Stephen Toulmin [13].

Ainsi, nous avons considéré que notre modèle permet de représenter un pas d'argumentation, c'est-à-dire l'étape qui permet d'établir une *Conclusion* en partant d'un ensemble d'éléments de preuve que nous appelons *Supports*. A cela, nous ajoutons la *Stratégie*, qui explicite clairement comment, à partir des Supports, il est possible d'inférer la conclusion et les *Fondements*, qui justifient pourquoi une stratégie est acceptable. La Figure 1 présente un exemple simple d'application de notre modèle.

Ce modèle d'argumentation nous permet de représenter et d'organiser les différents moyens utilisés pour montrer la conformité d'une architecture modulaire embarquée par rapport à ses exigences.

A partir de ces concepts, des patrons d'argumentation sont définis qui représentent des stratégies fréquemment utilisées lors de l'argumentation de la tenue des exigences d'une architecture modulaire embarquée. On peut définir des patrons d'argumentation correspondant à des processus standards (par exemple processus de sûreté de fonctionnement issu de l'ARP-4754) ou à des analyses classiquement effectuées pour la vérification d'exigences issues de la certification.

4 Exigences temps réel

En ce qui concerne l'avionique embarquée, les exigences temps réel ne font pas l'objet d'un processus spécifique. Elles sont distribuées dans divers documents sans forcément être identifiées

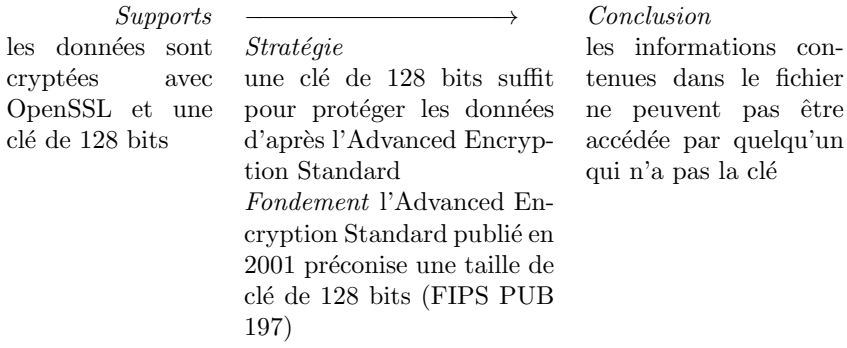


Figure 1: Exemple du schéma d'argumentation

comme exigences temps réel. On peut ainsi trouver des objectifs concernant des propriétés temps réel dans

- le standard de certification des IMA (DO-297) : partitionnement temporel par exemple,
- le standard de certification du logiciel (DO-178) : compatibilité avec le hardware (Worst Case Execution Time par exemple),
- les exigences fonctionnelles des systèmes considérés ("telle fonction doit répondre en moins de x ms").

Nous proposons ici une classification et une décomposition des exigences temps réel qui concernent des fonctions implantées sur une architecture IMA. La décomposition proposée prend en compte les objectifs des standards de certification mentionnés ci-dessus mais elle tente également de partir d'une vue d'ensemble de la "correction temps réel" d'une fonction. L'analyse de la classification proposée par rapport notamment à des travaux de même nature dans d'autres domaines applicatifs, fait partie des travaux futurs. Nous pensons cependant qu'elle apporte une contribution originale dans le domaine aéronautique.

Plus précisément, nous nous intéressons à l'exigence de très haut niveau suivante : *"une fonction implantée sur une architecture modulaire embarquée est correcte du point de vue temps réel"*. Dans le cadre de fonctions dites échantillonnées (c'est-à-dire dont les entrées sont échantillonnées) telles qu'une fonction de pilotage ou de guidage, cette exigence se décompose en deux types d'exigences :

1. Latence de bout-en-bout sur les entrées-sorties de la fonction : le temps entre la réception d'une entrée par la fonction, par exemple une rafale de vent (vue par une hausse soudaine de la pression externe), et la production de la sortie répondant à cette entrée, par exemple un braquage de gouverne de vol pour maintenir l'avion sur sa trajectoire, doit être borné supérieurement (on parle de latence maximale) et inférieurement (on parle de latence minimale). Le terme "bout-en-bout" exprime le fait que le chemin suivi entre l'entrée et la sortie peut passer par plusieurs calculateurs, plusieurs tâches logicielles au sein de ces calculateurs, et plusieurs moyens de transmission entre les calculateurs (par exemple des réseaux). L'exigence de latence porte sur l'ensemble du chemin.
2. Cohérence temporelle sur les entrées : l'écart entre les dates de production d'entrées servant au calcul de la même sortie à un temps donné est inférieur à un seuil. Par exemple, pour des raisons de tolérance aux pannes, le braquage de la gouverne est calculé en fonction de trois valeurs de pression mesurées indépendamment. Il est alors important de s'assurer que ces trois valeurs de pression sont mesurées "à peu près" en même temps pour que leur comparaison ait un sens. En d'autres termes, elles doivent être cohérentes lors de leur arrivée à l'entrée de la fonction.

Ces exigences portent sur le niveau fonctionnel du modèle de données (Fonction et Flow). Quand on considère les niveaux suivants, les objectifs temps réel se rangent en deux catégories : les objectifs concernant les temps de calcul et les objectifs concernant les temps de communication ou temps de traversée. Dans la suite on s'intéressera uniquement à la décomposition de l'exigence de latence en fonction des temps de calcul des tâches intervenant dans les fonctions

et des temps de communication entre ces différentes tâches. L'argumentation de l'exigence de cohérence ne sera pas abordée.

4.1 Temps de calcul

Pour les temps de calcul, on a besoin de savoir calculer le

a) WCRT (Worst Case Response Time) de chaque tâche intervenant dans la fonction : il s'agit du temps de réponse d'une tâche en prenant en compte les autres applications qui s'exécutent sur la même partition et donc la politique d'ordonnancement de cette partition. Cet objectif concerne donc les niveaux logiciel et topologique du modèle de donnée AMEM (Application et Partition).

Le calcul du WCRT va ensuite lui-même s'appuyer sur le calcul du

b) WCET (Worst Case Execution Time) d'une tâche sur un CPM (Core Processing Module) : il s'agit du temps d'exécution de la tâche sur un calculateur donné en faisant abstraction des autres tâches s'exécutant sur le même calculateur (pas d'interruption notamment). Cet objectif concerne les niveaux logiciel et matériel du modèle de donnée AMEM (Application et CPM).

4.2 Temps de traversée

Pour les temps de communication, on a besoin de savoir calculer le

c) WCTT (Worst Case Traversal Time) de chaque chemin entre les différentes tâches : il s'agit du temps de traversée d'une donnée sur un chemin en prenant en compte les autres données empruntant le même chemin. Cet objectif concerne les niveaux logiciel, topologique et matériel du modèle de donnée AMEM (DataFlow, TopoPath, Bus).

4.3 Arbre d'exigences

La figure 2 synthétise les liens entre les différents types d'exigence.

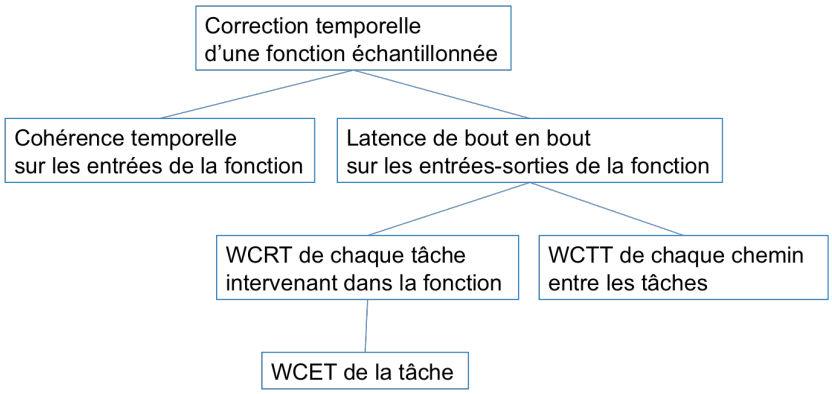


Figure 2: Une partie de l'arbre des exigences temps réel

Dans la suite de l'article nous nous concentrons uniquement sur l'exigence de latence et les trois sous-exigences qui participent à son argumentation.

5 Argumentation temps réel

Dans cette section, nous allons, premièrement, présenter trois patrons d'argumentation, chacun correspond à une des exigences présentées ci-dessus. Pour chacun de ces trois patrons génériques nous reprenons le schéma d'argumentation présenté précédemment. Ensuite, nous verrons comment nous pouvons étendre notre patron dans le cadre de l'utilisation d'un outil

logiciel. En effet, nos patrons étant générique, ne portant pas sur une technique particulière, ils explicitent seulement la forme que doit avoir l'argumentation qui répond à une exigence, mais ne donnent pas les restrictions qui pourraient être liées à l'emploi d'une technique particulière. Puis nous présenterons brièvement le concept de questions critiques et nous terminerons en donnant les instanciations de nos patrons génériques pour des moyens de vérification particuliers.

5.1 Patrons d'argumentation spécifiques

Nous avons défini trois patrons qui, en se composant, permettent de construire l'arbre d'argumentation :

- le patron WCL (worst case latency),
- le patron WCTT (worst case transit time),
- le patron WCRT (worst case response time).

Dans ces trois patrons, nous différencions un type de support particulier (indiqué en rouge sur fond grisé) : les supports qui font référence aux éléments d'architecture définis dans le modèle de données AMEM.

Notation : dans la suite, nous avons choisi de nous rapprocher de la notation GSN. Sur nos représentations visuelles, les boites arrondies représentent les *Conclusions* et les *Supports*, les boites trapézoïdales représentent les *Stratégies*, et les boites ovales représentent le *Domaine d'emploi* (placé à gauche de la stratégie) et les *Fondements* de cette stratégie (placés à droite de la stratégies) (voir paragraphe 3.3). De plus, les schémas se lisent de bas en haut, les supports étant placés en bas et la conclusion du raisonnement étant placée au sommet.

5.1.1 Argumentation WCL

Ce patron, Figure 3, permet de capturer l'argumentation d'un calcul de temps de latence (WCL). Le WCL se calcule pour une chaîne fonctionnelle donnée. Ce calcul nécessite de connaître (a) pour chaque tâche le WCRT, la période, l'échéance (deadline), ainsi que le décalage entre la date prévue du début d'exécution et le début de la période (décalage appelé souvent "offset"), (b) le WCTT de tous les chemins impliqués dans la chaîne fonctionnelle considérée.

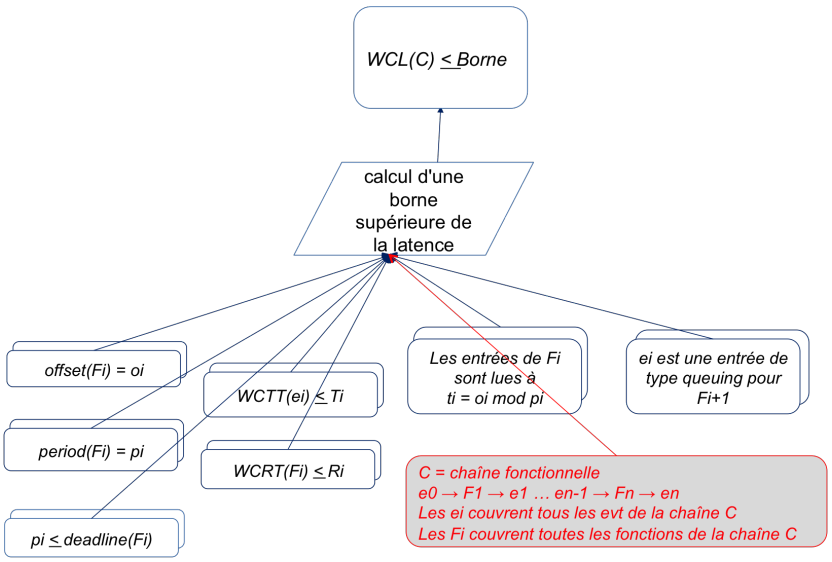


Figure 3: Argumentation WCL

La feuille rouge sur fond grisé décrit la chaîne fonctionnelle considérée, avec toutes les tâches F_i impliquées et toutes les données e_i échangées entre ces tâches. Les deux feuilles juste au-dessus détaillent le type d'échanges utilisées entre les F_i. Les feuilles de gauche du

schéma représentant les valeurs d'offset, de période, de deadline et de WCRT pour toutes les F_i, ainsi que le WCTT de tous les e_i. Le patron exprime le fait que tous ces supports sont nécessaires au calcul de la borne supérieure de la latence d'une chaîne fonctionnelle. Nous ne détaillerons pas ici tous ces paramètres. Notre objectif dans cet article est de montrer comment définir des patrons et les utiliser.

5.1.2 Argumentation WCTT

Le calcul du pire temps de traversée réseau (WCTT) d'une trame le long d'un VL (Virtual Link) à travers un réseau avioniques IMA peut être effectué par différentes méthodes telles que le Network Calculus [7], l'Approche de Trajectoires [8], etc. Si elle se différencient par le technique sous-jacente, ces techniques nécessitent toutes la connaissance d'un même ensemble d'information. Le patron, Figure 4, exprime ces dépendances sous la forme d'un arbre d'argumentation.

- le WCTT se calcule pour un VL C et une trame tr donnés;
- ce calcul nécessite de connaître pour chaque VL de la plateforme la taille maximale des trames et le BAG (Bandwidth Allocation Gap);
- il nécessite également la connaissance la connaissance du routage R de tous les VL à travers le réseau, ce routage devant être statique;
- et enfin, il est nécessaire dans le calcul de considérer tous les VL de la plateforme.

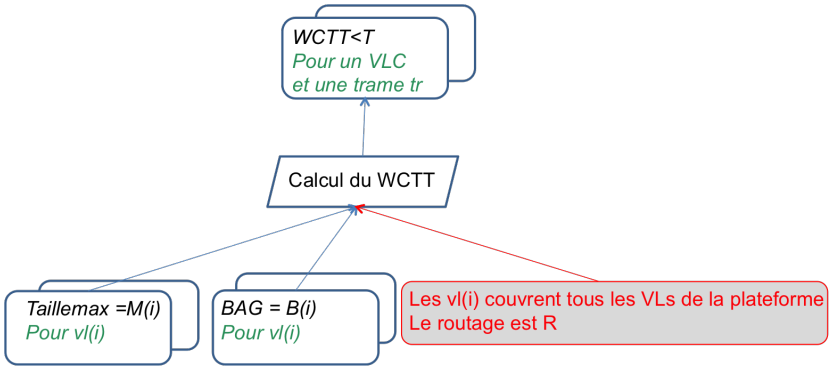


Figure 4: Argumentation WCTT

On retrouve ici la même organisation que pour le patron précédent : la description des données considérées en rouge-grisé, puis tous les éléments nécessaires au calcul du WCTT.

5.1.3 Argumentation WCRT

Ce patron, Figure 5, permet de capturer l'argumentation d'un calcul de temps de réponse (WCRT). Le WCRT se calcule pour une tâche i dans une partition donnée. Ce calcul nécessite de connaître pour chaque tâche de la partition le WCET et la période.

Sur ce patron, on peut noter la conjonction intéressante de deux supports : le WCET de chaque tâche est nécessaire pour le calcul du WCRT, ce calcul est fait en prenant l'hypothèse qu'il n'y a pas de préemption sur la tâche et que cette tâche s'exécute sur un processeur sans système d'exploitation (mention "Pas d'OS" pour "Pas d'Operating System") ; il faut donc ajouter un second support pour s'assurer qu'il n'y a effectivement pas de préemption et pas de système d'exploitation sur le processeur hébergeant la partition, ou que leur coût est négligeable. Ce dernier jugement ne peut être donné que par un expert.

5.2 Patrons d'argumentation portant sur les moyens de conformité

Dans la section précédente, nous avons proposé des patrons d'argumentation portant sur la nature de la conclusion considérée mais ne considérant pas les moyens mis en oeuvre pour

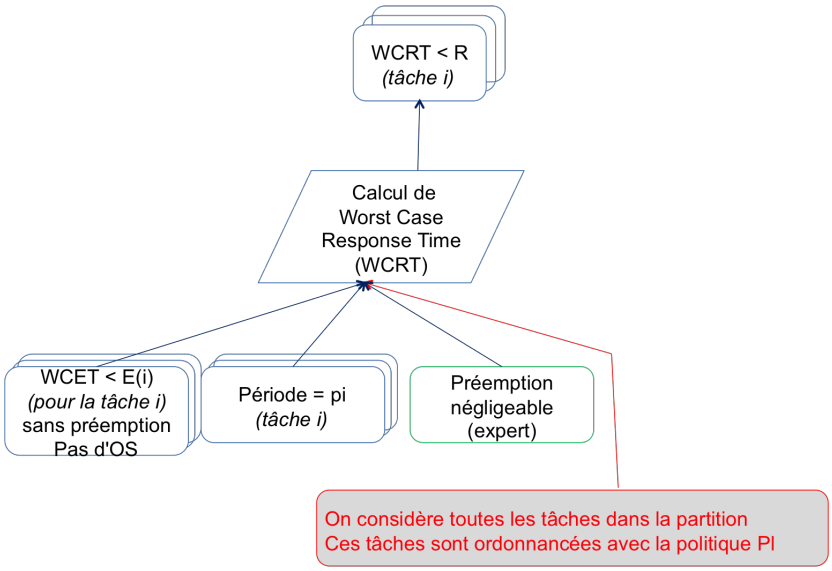


Figure 5: Argumentation WCRT

obtenir un résultat. Dans cette section, nous présentons un patron d'utilisation d'un outil logiciel. Nous combinerons ensuite ces deux types de patrons.

Dans le cadre de l'utilisation d'un logiciel, la Conclusion est le résultat donné par le logiciel et les éléments de preuve sont les informations dont a besoin le logiciel pour produire son résultat. Mais cela n'est pas suffisant, en effet l'usage d'un logiciel est soumis à un ensemble de conditions, de prérequis. Par exemple, on peut imaginer un logiciel qui garantit des propriétés mais seulement sur des systèmes déterministes. Par conséquent l'ensemble des éléments de preuve pour l'utilisation d'un logiciel est composé de ce qui relève de ses conditions d'utilisation et des données nécessaires à la production de la Conclusion.

De plus, un logiciel n'est pas utilisable hors de tout contexte, il dispose d'un domaine d'emploi. Par conséquent, dans le cadre du patron d'argumentation d'utilisation d'un outil logiciel, il est indispensable d'expliciter clairement le *Domaine d'emploi* afin de vérifier que celui-ci couvre correctement les conditions d'utilisation données dans l'argumentation. Notons que la vérification de l'adéquation entre le domaine d'emploi et les conditions d'utilisations est a priori à la charge de l'autorité de vérification.

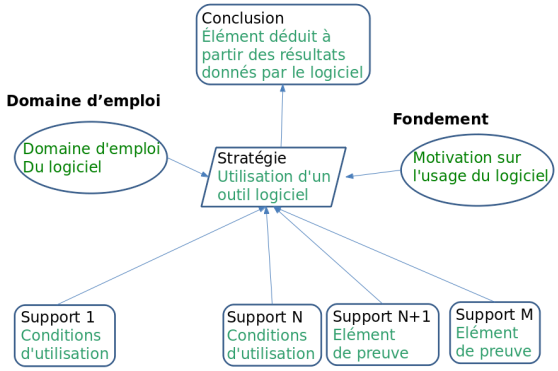


Figure 6: Argumentation : Stratégie d'utilisation d'un logiciel

Finalement, nous proposons le patron d'argumentation générique dédié à l'*utilisation d'un outil logiciel* comme stratégie en vue d'établir un résultat Figure 6. Sur ce nouveau patron, nous voyons apparaître trois familles de Supports. Le premier ensemble, de gauche à droite, les Supports de 1 à N concernent les éléments de preuve relatifs aux conditions d'utilisation d'un outil logiciel. C'est cet ensemble qui permet de vérifier que le logiciel est bien employé dans son *Domaine d'emploi*. Le deuxième ensemble, les supports de N+1 à M, sont les éléments de preuve qui servent à la démonstration.

5.3 Questions critiques

Pour pouvoir aider l'auditeur dans sa tâche de compréhension et d'analyse d'une argumentation, nous avons défini ce que nous appelons les questions critiques. Elles sont le lien entre l'argumentation et la vérification. Notons que ces questions, en plus d'aider l'auditeur, aident aussi celui qui réalise l'argumentation puisqu'elles le forcent à s'interroger sur ce qu'il fait et à vérifier qu'il ne tire pas de conclusions hâtives.

Nos questions critiques sont en fait une adaptation des travaux de Douglas Walton sur la parole d'un expert ([4, 5]). Dans notre cas, nous ne nous intéressons pas à la parole d'un expert, mais à la validité des résultats donnés par un logiciel. Par conséquent nous proposons d'ajouter à notre schéma les trois questions critiques suivantes :

1. (Fondement) Est-ce que ce logiciel est crédible ?
2. (Domaine d'emploi) Est-ce que ce logiciel est utilisable dans ce cadre d'emploi ?
3. (Vérification) Est-ce que le logiciel permet de conclure très clairement ce qui est dit en conclusion ?

La question 1 porte sur la crédibilité du logiciel. En d'autres termes, la question 1 renvoie au problème de l'utilisation de l'outil logiciel dans sa globalité : est-ce un outil connu ? Est-il crédible (par exemple grâce à l'utilisation de méthodes formelles pour le vérifier) ? Son utilisation est-elle reconnue pour les opérations de V&V ? Est-il qualifié ? A t-il déjà été utilisé avec succès pour cet usage et si oui combien de fois ? Etc. Dans notre schéma d'argumentation, cette question oblige à fournir ce que nous avons appelé un fondement.

La question 2 relève du domaine d'emploi. Un logiciel n'est pas utilisable hors de tout contexte, il dispose d'un domaine d'emploi. Par conséquent, il est indispensable d'expliciter clairement le domaine d'emploi afin de vérifier que celui-ci couvre correctement la façon dont le logiciel a été employé. Dans la pratique, la question 2 renvoie à un document de réponse qui établit l'adéquation entre le domaine d'emploi du logiciel et l'usage qui en est fait. Cette adéquation peut être triviale, par correspondance directe entre le domaine d'emploi et les conditions d'utilisation, ou au contraire nécessiter elle-même une argumentation.

Enfin, parce que le résultat issu de l'utilisation d'un logiciel peut lui-même relever d'une expertise, le but de la question 3 est de renvoyer à un document explicitant sans ambiguïté le lien entre le résultat donné par le logiciel et la conclusion de l'argumentation. A titre d'exemple et pour rester dans notre domaine, un outil qui calcule le WCRT d'un ensemble de tâches ne tient pas compte en général du temps pris par l'OS et l'ordonnancement lui-même, ou plutôt considère que ce temps est nul. Du résultat de l'outil (le WCRT des tâches) on ne pourra conclure que ces tâches sont ordonnançables que si leurs WCRT sont strictement inférieurs à leurs échéances et si le temps (inévitablement non nul) pris par l'OS et l'ordonnanceur est négligeable. Sans ce dernier jugement d'expert, il serait imprudent de déduire la conclusion (les tâches sont ordonnançables) à partir du résultat de l'outils (les WCRT des tâches).

5.4 Combinaison entre les patrons temps réel et le patron *utilisation d'un outil logiciel*

Pour illustrer cette combinaison, reprenons l'argumentation du WCET et du WCRT d'une tâche selon les patrons correspondants Figures 5 et 5, et supposons, à titre d'exemple, que les moyens utilisés sont les outils logiciels aiT et Cheddar.

5.4.1 Instanciation pour le WCET

Sur la Figure 7 nous donnons un exemple d'argumentation en vue d'établir que le WCET d'une tâche existe (c'est-à-dire n'est pas infini) et qu'il est compatible (c'est-à-dire inférieur)

à la taille temporelle de la partition dans laquelle la tâche est exécutée notée ici E. S'il existe plusieurs outils permettant de calculer le WCET nous avons choisi d'utiliser dans notre exemple aiT[2].

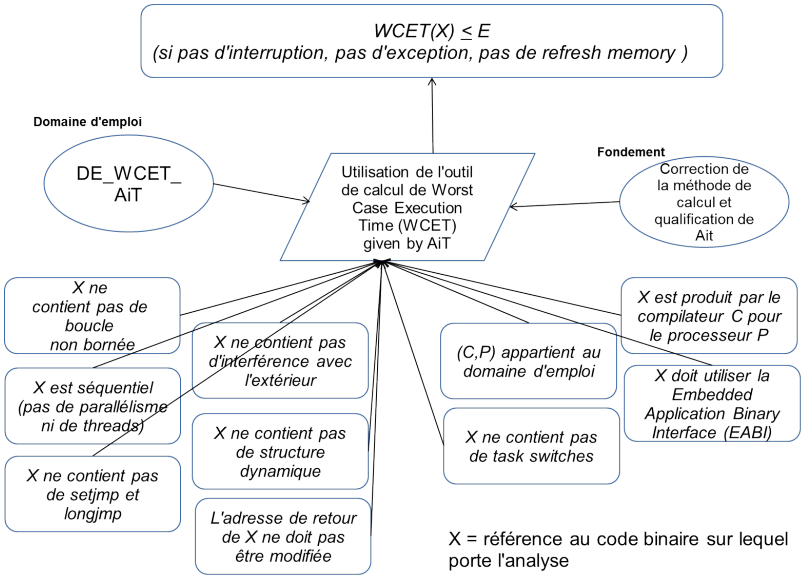


Figure 7: Argumentation WCET

Le Fondement attaché à la stratégie d'emploi d'aiT explicite simplement pourquoi l'utilisation d'aiT dans notre contexte est acceptable. Nous avons choisi, dans le cadre de cet exemple, de mettre une référence concernant la correction des calculs mais aussi de considérer que l'outil était qualifié. Les explications plus détaillées concernant la confiance dans les résultats donnés par aiT correspondent en fait à la réponse à la question critique 1.

Le domaine d'emploi pour l'utilisation d'aiT correspond aux spécifications données par l'éditeur comme la liste des processeurs supportés ou le fait que le code ne doit pas contenir de *task switches*. Dès lors, répondre à la question critique 2, peut se raffiner en répondre aux questions telles que : est-ce que le modèle de processeur appartient à la liste des processeurs couverts par l'outil logiciel ? Est-ce que le compilateur utilisé pour produire le code objet exécutable est compatible avec celui qui doit être utilisé pour ce processeur là avec aiT ? Est-ce qu'il y a des boucles dans la tâche, et si oui sont-elles bornées par une constante, et si oui ces constantes sont-elles renseignées et sont-elles correctes ? Est-ce que la tâche n'est pas soumise à des interférences venant de l'extérieur (une préemption par exemple) ? Etc. (voir liste complète des questions Figure 7)

Concernant la question critique 3, remarquons qu'ici nous ne pouvons pas directement conclure que le $WCET \leq E$, mais que $WCET \leq E$ sous réserve qu'il n'y ait pas d'interruption, d'exception et de *refresh memory*.

5.4.2 Instanciation pour le WCRT

Dans notre exemple, nous considérons l'utilisation de l'outil Cheddar[3]. Cheddar est un logiciel libre, c'est un simulateur d'ordonnanceurs temps réel qui permet de calculer une estimation précise du WCRT d'applications non-préemptives.

Figure 8, nous pouvons voir la combinaison du patron WCRT et de l'instanciation du patron d'argumentation *utilisation d'un outil logiciel*.

La question du Fondement est la même que dans l'exemple du WCET. Le domaine d'emploi est l'ensemble des politiques d'ordonnacement prises en compte par l'outil, la réponse à la question critique 2 est donc de vérifier que le support correspondant est correct.

[2] `http://www.absint.com/ait`
[3] `http://beru.univ-brest.fr/~singhoff/cheddar/`

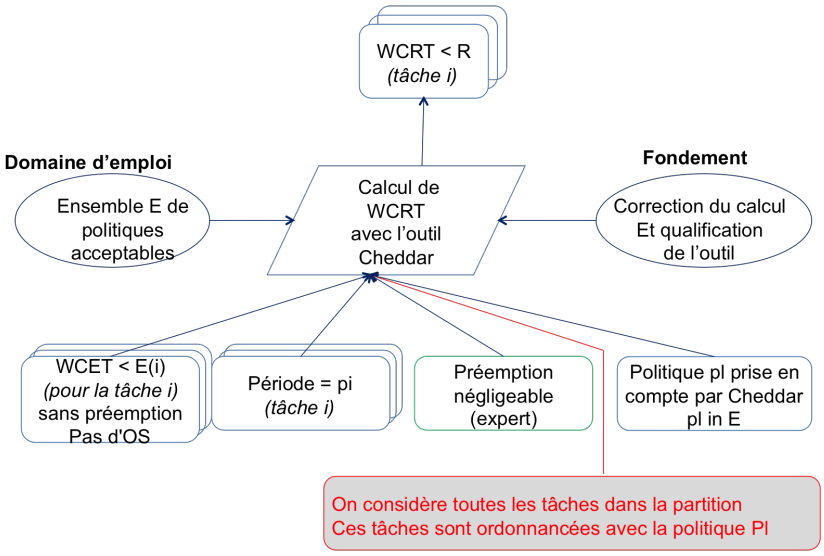


Figure 8: Argumentation WCRT

6 Conclusion

Dans cet article, nous avons présenté des travaux en cours sur la définition de patrons d'argumentation permettant de construire une argumentation de la tenue des exigences temps réel d'une architecture modulaire embarquée. Ces patrons s'appuient sur une description systématique des exigences temps réel applicables à ces architectures dans le domaine aéronautique. Ces travaux s'inscrivent dans un projet de définition d'un référentiel de certification des architectures modulaires embarquées intégrant une description des composants de telles architectures, des exigences attendues et des moyens de conformité acceptables pour démontrer la tenue de ces exigences.

Des travaux similaires vont être menés pour l'argumentation des exigences de sûreté de fonctionnement de ces architectures. Nous travaillons également sur la question de l'argumentation incrémentale, afin de fournir un support à la certification incrémentale des architectures modulaires embarquées.

References

[1] ARINC 653, Aeronautical Radio Inc. *Avionics Application Software Standard Interface*, 1997.

[2] ARINC 664, Aeronautical Radio Inc. *Aircraft Data Network, Part 1 : Systems Concepts and Overview*, 2002.

[3] P. Bieber, F. Boniol, M. Boyer, E. Noulard, and C. Pagetti. New Challenges for Future Avionic Architectures". *Journal AerospaceLab*, 4, 2012.

[4] David J Godden and Douglas Walton. Advances in the theory of argumentation schemes and critical questions. *Informal Logic*, 27(3):267–292, 2007.

[5] David M Godden and Douglas Walton. Argument from Expert Opinion as Legal Evidence: Critical Questions and Admissibility Criteria of Expert Testimony in the American Legal System. *Ratio Juris*, 19(3):261–286, 2006.

[6] Tim Kelly and Rob Weaver. The Goal Structuring Notation – A Safety Argument Notation. In *Proc. of Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[7] Jean-Yves Le Boudec and Patrick Thiran. *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag, Berlin, Heidelberg, 2001.

[8] Steven Martin and Pascale Minet. Schedulability analysis of flows scheduled with fifo: application to the expedited forwarding class. *Parallel and Distributed Processing Symposium, International*, 0:167, 2006.

[9] Ministry of Defence. *Defence Standard 00-42 Issue 2, Reliability and Maintainability (R&M) Assurance Guidance Part 3 R&M Case*, 2003.

[10] European organisation for the safety of air navigation. *Safety Case Development Manual*, 2006.

[11] Thomas Polacsek. Réflexions sur les liens possibles entre argumentation et v&v pour le logiciel. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2014.

[12] Jean-François Sicard, Ghilaine Martinez, and Florian Many. Specific Certification Issues Concerning IMA. In *Embedded Real-Time Software and Systems (ERTS2 2012)*, February 2012.

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

Vers un outil de vérification formelle légère pour OCaml

Thomas Genet[1], Barbara Kordy[2], and Amaury Vansyngel[1]

[1]IRISA, Université de Rennes 1, France
[2]IRISA, INSA de Rennes, France

Résumé

Si l'on décrit, par une grammaire, l'ensemble des entrées possibles d'un programme fonctionnel, peut-on connaître la grammaire des sorties de celui-ci ? Il existe des outils en réécriture à même de répondre à cette question, pour certaines fonctions. On peut utiliser ce genre de calcul pour détecter des bugs ou, à l'inverse, pour prouver des propriétés sur ces fonctions. Dans cet article, nous présentons un travail en cours visant à concevoir un outil de vérification formelle légère pour OCaml. Si l'essentiel des résultats théoriques et outils de réécriture existent déjà, leur application à la vérification de programmes OCaml réalistes nécessite de résoudre un certains nombre de problèmes. Nous donnerons l'architecture d'un interprète abstrait pour OCaml, basés sur ces principes et outils, et nous verrons quelles sont les briques manquantes pour finaliser son développement.

1 Une vérification aussi automatique que l'inférence de type

Certains langages de programmation fortement typés (comme Haskell, OCaml, Scala et F#) disposent d'un mécanisme d'inférence de type. Ce mécanisme permet, entre autres, de détecter automatiquement certains types d'erreurs dans les programmes. Pour vérifier plus que le bon typage d'un programme, il est possible de définir des propriétés et, ensuite, de les prouver à l'aide d'un assistant de preuve ou d'un prouveur automatique. Cependant, la formulation de ces propriétés en logique et la construction de la preuve requièrent généralement une certaine expertise.

On s'intéresse ici à une famille restreinte de propriétés : des propriétés "régulières" portant sur les structures manipulées par ces programmes. On décrit, par une grammaire (d'arbres, régulière), l'ensemble de structures données en entrée à une fonction et on cherche à construire la grammaire des structures pouvant être obtenues en sortie (ou une approximation). La grammaire de sortie pourra révéler un bug potentiel dans le programme ou, à l'inverse, montrer que celui-ci vérifie une propriété régulière.

La famille de propriétés démontrables de cette façon est restreinte, même si elle dépasse ce que l'on peut exprimer par typage simple. On peut par exemple distinguer la liste vide de la liste non vide. Une telle distinction est également faisable avec des GADTs (Generalized Algebraic DataTypes) mais, à notre connaissance, pas le type de raisonnement sur des structures récursives que l'on propose ici. Il existe d'autres approches où le système de type est enrichi par des formules logiques et de l'arithmétique [17, 3]. Cependant ces techniques nécessitent généralement d'annoter le programme pour que la vérification de type réussisse. Les propriétés que nous considérons ici sont volontairement plus simples pour éviter au maximum le travail d'annotation. Pour ces propriétés, l'objectif est de proposer une forme de vérification formelle "légère". La vérification est formelle car elle *démontre* que les résultats d'une fonction ont une certaine forme. La vérification est légère car, d'une part, la preuve est automatique, et d'autre part, il n'est pas nécessaire de définir la propriété attendue par une formule logique mais simplement d'observer le résultat d'un calcul abstrait.

Concernant le procédé de calcul lui-même (produire la grammaire de sortie à partir de la grammaire d'entrée), un certain nombre de travaux abordent ce sujet dans la communauté de programmation fonctionnelle [13, 15] comme dans la communauté de réécriture [10, 7]. Dans la communauté de programmation fonctionnelle, l'analyse du flot de contrôle des programmes fonctionnels d'ordre supérieur est une thématique de recherche extrêmement active [2, 14] mais l'objectif est différent puisqu'il ne vise pas à déterminer l'ensemble des résultats. Dans [10, 7], à partir d'un système de réécriture de termes codant une fonction et d'un automate (ou d'une grammaire) reconnaissant les entrées de la fonction, il est possible de produire automatiquement un automate reconnaissant une *sur-approximation* des sorties de cette fonction. On dispose donc d'un outil pouvant interpréter ou évaluer, de façon *abstraite*, une fonction (exprimée par un système de réécriture) non pas sur une entrée, comme le fait un interprète classique, mais sur un ensemble d'entrées. Voyons maintenant ce qui manque pour réaliser, à partir de ces outils de réécriture, un outil de vérification pour OCaml.

2 A quoi pourrait ressembler un interprète abstrait OCaml ?

Imaginons que l'on dispose d'une notation, inspirée des expressions régulières, pour définir des langages réguliers de listes. Notons `[a*]` le langage des listes avec 0 occurrence ou plus du symbole `a`. Notons `[a*,b*]` le langage des listes avec 0 occurrence ou plus de `a`, suivies de 0 occurrence ou plus de `b`. On note `[(a|b)*]` toute liste avec 0 occurrence ou plus de `a` ou `b` (dans n'importe quel ordre). On souhaite définir, par exemple, une fonction de suppression supprimant toutes les occurrences d'un élément dans une liste en OCaml. Voici une première définition (erronée) de cette fonction :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then t else h::(delete x t);;
```

Il est bien sûr possible de réaliser des tests sur cette fonction en utilisant l'interprète OCaml :

```
# delete 2 [1;2;3];;
-:int list= [1; 3]
```

Si l'on disposait d'un interprète *abstrait* OCaml travaillant sur des grammaires, nous pourrions lui poser la question suivante : quel est le langage des listes obtenu en appliquant `delete` à `a` et n'importe quelle liste de `a` et de `b` ?

```
# delete a [(a|b)*];;
-:abst list= [(a|b)*];;
```

La réponse obtenue n'est pas celle attendue. Dans la mesure où toutes les occurrences de `a` auraient dû être supprimées, nous espérions le résultat `[b*]`. Comme l'interprète abstrait donne une sur-approximation de la grammaire de sortie, cela ne démontre pas l'existence d'un bug. Cela met quand même la puce à l'oreille : toutes les occurrences de `a` ne seraient-elles pas supprimées ? Effectivement, il devrait y avoir un appel récursif de `delete` dans la branche `then`. Prenons maintenant cette deuxième version de la fonction :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if x=x then (delete x t) else h::(delete x t);;
```

Si on pose à l'interprète la même question, on obtient :

```
# delete a [(a|b)*];;
-:abst list= [];;
```

Cette fois-ci on a la garantie de la présence d'un bug, car la sur-approximation ne contient même pas le langage attendu : `[b*]`. Si on corrige cette dernière erreur (`if x=x...`), on obtient la fonction suivante :

```
let rec delete x l= match l with
  | [] -> []
  | h::t -> if h=x then (delete x t) else h::(delete x t);;
```

Sur cette fonction l'interprète abstrait nous donnera finalement :

```
# delete a [(a|b)*];;
-:abst list= [b*];;
```

Ce résultat prouve deux propriétés : (1) `delete` supprime toutes les occurrences d'un élément dans la liste et (2) `delete` ne supprime *pas* toutes les occurrences des autres éléments dans la liste. Ce ne sont que *quelques* propriétés de `delete` mais celles-ci ont pu être démontrées sans avoir dû, au préalable, les formaliser en logique. De plus, la preuve est automatique. Comme autre exemple, prenons les fonctions suivantes définissant un tri par insertion :

```
let rec insertion e l= match l with
    [] -> [e]
  | h::t -> if e=h then e::h::t else h::(insertion e t)

let rec sort l= match l with
    [] -> []
  | h::t -> insertion h (sort t)
```

Comme pour la fonction `delete`, un test un peu rapide de la fonction avec l'interprète OCaml peut nous donner l'illusion que cette fonction est correcte :

```
# sort [3;2;1];;
-:int list= [1;2;3];;
```

Un interprète OCaml abstrait nous permettrait, lui, de détecter rapidement un problème : la liste en sortie n'est pas nécessairement triée (on suppose que `a` est inférieur ou égal à `b`).

```
# sort [(a|b)*];;
-:abst list= [(a|b)*];;
```

Si l'on corrige l'erreur dans la condition du `if` de la fonction `insertion`, on obtient la fonction :

```
let rec insertion e l= match l with
    [] -> [e]
  | h::t -> if e<=h then e::h::t else h::(insertion e t)

let rec sort l= match l with
    [] -> []
  | h::t -> insertion h (sort t)
```

Avec cette fonction, un interprète abstrait nous donnera un résultat plus proche de nos attentes, comme `[a*,b*]`. Sur ces exemples, la sur-approximation est suffisamment fine pour que l'interprète abstrait révèle une erreur ou, au contraire, prouve la propriété. Cependant, si l'approximation est trop grossière, le langage obtenu ne fournit que peu d'information. Par exemple, si nous analysons de la même façon la fonction de tri par fusion nous obtenons le langage `[(a|b)*]` qui ne garantit pas la propriété attendue. Cependant, nous verrons dans la Section 4 qu'il existe des pistes pour raffiner automatiquement une approximation lorsqu'elle est trop grossière.

3 Construire un interprète abstrait pour OCaml

Comme nous avons vu dans la première section, il existe des travaux qui visent le même objectif que l'interprète abstrait mais pour des systèmes de réécriture. En utilisant Timbuk [8], il est possible d'effectuer les preuves présentées ci-dessus, en traduisant les fonctions OCaml en systèmes de réécriture. Par exemple, la fonction `delete` (corrigée), peut être traduite manuellement dans le système de réécriture $\mathcal{R}$ suivant :

$$
\begin{array}{l}
delete(X, nil) \to nil \\
delete(X, cons(Y, Z)) \to ite(eq(X, Y), delete(X, Z), cons(Y, delete(X, Z))) \\
ite(true, X, Y) \to X \quad ite(false, X, Y) \to Y \\
eq(a, a) \to true \quad eq(a, b) \to false \quad eq(b, a) \to false \quad eq(b, b) \to true
\end{array}
$$

où les constantes a et b, le prédicat d'égalité eq et ite (pour if then else) sont générés en complément de la définition de $delete$ [1]. D'autre part, l'expression `delete a [(a|b)*]` peut être traduite manuellement en un automate d'arbre $\mathcal{A}$ (et non une grammaire) reconnaissant le langage régulier des termes correspondant à cette requête. Enfin, avec Timbuk il est possible de produire automatiquement un autre automate d'arbre, $\mathcal{A}^*$, reconnaissant une sur-approximation de l'ensemble des termes obtenus par réécriture des termes reconnus par $\mathcal{A}$. Si l'on s'intéresse au sous-langage de $\mathcal{A}^*$ représentant des valeurs [2], celui-ci correspond effectivement à `[b*]`. Actuellement, nous complétons un interprète OCaml avec ces fonctionnalités abstraites. Le principe est résumé par la Figure 1.

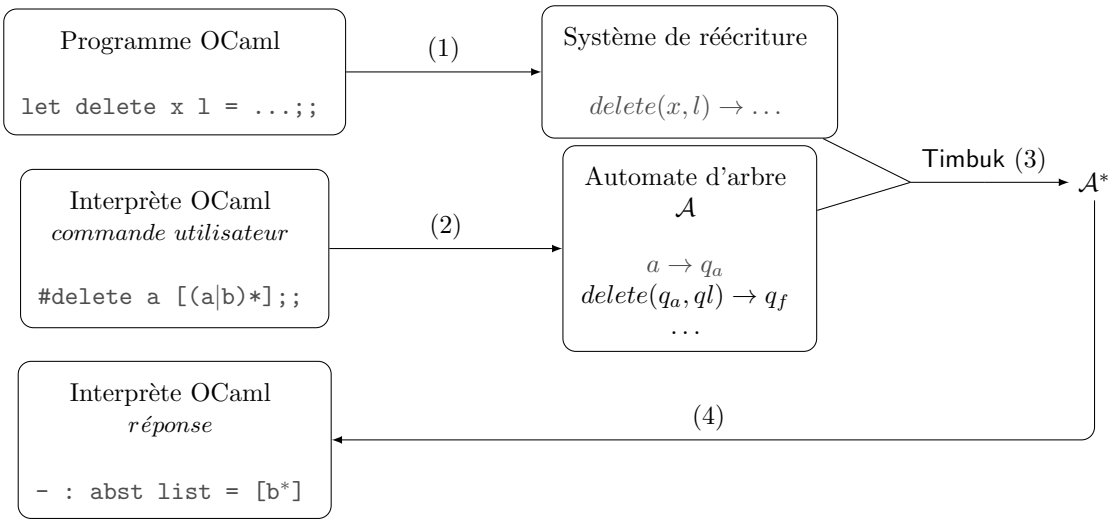


FIGURE 1 – Utilisation de Timbuk pour la réalisation de l'interprète abstrait

(1) La définition d'une fonction OCaml dans l'interprète est complétée par la génération d'un système de réécriture ;

(2) Les requêtes abstraites faites dans l'interprète sont interceptées pour être évaluées séparément. On génère automatiquement un automate d'arbre reconnaissant le langage décrit par l'expression régulière ;

(3) Dans le cas d'une requête classique, on laisse l'interprète OCaml standard faire son travail. Dans le cas d'une requête abstraite, celle-ci est redirigée vers Timbuk pour calculer $\mathcal{A}^*$;

(4) L'automate résultat $\mathcal{A}^*$ est réinterprété sous la forme d'une expression intelligible par l'utilisateur.

Le sous-ensemble du langage OCaml auquel nous nous intéressons actuellement est le suivant : programmes fonctionnels purs (pas d'effets de bord, pas de références, pas de structures mutables, pas d'objets), avec définition de fonctions mutuellement récursives, d'ordre supérieur, avec pattern-matching, sans exceptions, sans types prédéfinis (`int`, `string`, etc.) mais avec des types algébriques. Pour disposer d'un interprète abstrait, sur ce sous-ensemble d'OCaml, il reste quelques obstacles à surmonter. Le premier est de définir formellement une traduction de OCaml vers la réécriture utilisée par (1). Cette traduction doit préserver la sémantique du programme OCaml de départ. D'autres travaux étudient ce type de transformation [16, 4] pour des langages plus riches que ce que nous visons pour l'instant. Par exemple, ces traductions couvrent les types

1. A la place du codage avec ite, on aurait pu utiliser des règles de réécriture conditionnelle mais celles-ci ne sont que partiellement supportées par Timbuk.

2. L'automate $\mathcal{A}^*$ reconnaît tous les calculs intermédiaires jusqu'aux résultats. On peut ne garder que les résultats en calculant un produit entre $\mathcal{A}^*$ et un automate reconnaissant uniquement les valeurs résultats.

prédéfinis, les effets de bords et même la concurrence. En revanche, la traduction attendue en (1) devra produire un système de réécriture suffisamment complet à partir d'un programme OCaml dont le pattern-matching est exhaustif. En effet, la complétude suffisante est nécessaire pour garantir la terminaison de l'interprète abstrait [7]. D'autre part, cette traduction devra prendre en compte les fonctions d'ordre supérieur. Les expérimentations menées avec un codage des fonctions d'ordre supérieur en réécriture, proposé par Jones et Andersen dans [13], sont encourageantes. Si les résultats obtenus par Jones et Andersen ont été dépassés depuis (par [15]), ce n'est pas la faute du codage mais plutôt de la précision de leur algorithme de calcul de grammaires. Notre calcul étant plus précis, nous avons constaté qu'il était possible de traiter les fonctions d'ordre supérieur considérées dans [15] avec ce codage [11]. La question en suspens est de savoir si la terminaison du calcul de $\mathcal{A}^*$ (3), qui est assurée au premier ordre [7], est toujours garantie à l'ordre supérieur avec ce codage. Enfin pour les étapes (2) et (4), comme dans le cas des mots, il existe des formats d'expressions régulières et des algorithmes de passage des expressions vers les automates pour les arbres [5]. Cependant, ces formats d'expressions sont parfois plus difficiles à écrire ou à lire que dans le cas des mots. Par exemple, l'expression régulière (telle qu'elle est définie dans [5]) correspondant à notre notation `[a*]`, est l'expression $nil + cons(a, \Box)^{*,\Box}.{}_{\Box} nil$, où *cons* et *nil* sont les constructeurs de listes. Nous cherchons une solution plus naturelle et facilement utilisable par des programmeurs OCaml pour définir des langages d'éléments de types algébriques usuels.

4 Conclusion et perspectives

Dans cet article, nous donnons quelques pistes pour exploiter des outils de réécriture pour la vérification formelle légère de programmes OCaml. La vérification est *formelle* car elle permet de *prouver* certaines propriétés sur ces programmes. D'autre part, elle est *légère* car la preuve est automatique et la famille de propriétés considérées est restreinte aux propriétés régulières. Il reste un certain nombre de problèmes théoriques à dépasser, comme la prise en compte des fonctions d'ordre supérieur. Il reste également des problèmes plus conceptuels, comme la bonne façon d'interagir avec l'interprète. Car si, dans les exemples présentés dans cet article, les expressions régulières sont lisibles, cela ne sera pas toujours le cas : les expressions de langages pourront être trop complexes. Cependant, dans un interprète abstrait il est également possible de poser des requêtes de la forme : `delete(a,[(a|b)*])=[b*]` où la propriété attendue est donnée comme une post-condition du calcul. A noter que, sous certaines conditions, on pourra bien parler d'une égalité. En effet, la précision des approximations est quantifiable [10] et dans ce cas particulier il est possible de garantir à la fois que `delete(a,[(a|b)*])`$\subseteq$`[b*]` et que `delete(a,[(a|b)*])`$\supseteq$`[b*]`. En donnant ainsi l'ensemble des résultats attendus, si le résultat est `true`, il sera possible d'éviter la phase de lecture et d'analyse de l'expression résultat. Cependant, on a vu que la précision de l'approximation n'était pas toujours garantie : on a mentionné que c'était le cas pour la fonction de tri par fusion. L'autre intérêt de cette forme est de donner le langage attendu pour le résultat et de permettre ainsi un calcul par raffinement d'abstraction automatique (CounterExample Guided Abstraction Refinement : CEGAR), dans le cas où l'approximation se révèle trop grossière. Ceci est utilisé dans l'analyse de programmes fonctionnels [15] et a été expérimenté avec succès sur Timbuk [1]. Parfois, la représentation des valeurs n'est pas seulement trop complexe, elle est inaccessible : c'est le cas des types abstraits qui masquent les détails de leur implantation. Ceci empêche d'exprimer le langage de sortie d'une fonction par une expression régulière ou par un automate. En revanche, ceci n'interdit pas la définition et la vérification de propriétés à l'aide de prédicats, comme dans le cas des preuves réalisées par les assistants de preuve. Dans le cas du type abstrait `Set` d'OCaml, on pourra ainsi exprimer et vérifier des propriétés de la forme :

`not(mem a (remove a add*((a|b),empty)))`, où l'expression régulière sera utilisée pour exprimer de façon abstraite l'ensemble construit dans la requête et le résultat sera booléen et non sous la forme d'une expression régulière.

Sur la route menant à la conception d'un interprète abstrait, il existe d'autres embûches pour lesquelles nous avons déjà quelques éléments de réponse. C'est le cas des stratégies d'évaluation (appel par valeur) et des types prédéfinis (`int`, `string`, etc.). Quand l'exécution du programme repose

sur une stratégie d'évaluation, le calcul de l'automate $\mathcal{A}^*$ doit prendre en compte cette stratégie au risque de produire des résultats trop imprécis. Récemment, nous avons montré que cela était possible pour la stratégie d'appel par valeur [12] utilisée par OCaml. D'autre part, les programmes OCaml ne manipulent pas uniquement des termes mais également des valeurs *prédéfinies* comme des `int`, `string`, etc. Nous savons maintenant que l'algorithme de calcul de l'automate A^* peut être étendu de façon naturelle pour prendre en compte ces valeurs particulières [9]. Les automates obtenus marient des transitions classiques (pour la partie terme/structure) et des éléments d'un domaine abstrait de l'interprétation abstraite [6] (pour les ensembles de valeurs prédéfinies). Les expérimentations menées dans [9] permettent de manipuler des langages réguliers de termes où certaines feuilles peuvent être des intervalles abstrayant des ensembles de valeurs entières prédéfinies. Pour l'instant, ces automates ne permettent pas de définir de relation entre les valeurs associées aux feuilles. Par exemple, on ne peut pas décrire le langage des listes triées. En revanche, ils devraient permettre de définir et de raisonner sur des expressions régulières hybrides plus riches dans l'interprète abstrait. Par exemple, une expression de la forme `[(1;+oo)*]`, pourra représenter les listes de longueur quelconque dont tous les éléments sont des entiers appartenant à l'intervalle $[1; +\infty[$.

Remerciements Les auteurs remercient les relecteurs pour leurs remarques et commentaires.

Références

[1] Y. Boichut, B. Boyer, T. Genet, and A. Legay. Equational Abstraction Refinement for Certified Tree Regular Model Checking. In *ICFEM'12*, volume 7635 of *LNCS*. Springer, 2012.

[2] C. H. Broadbent, A. Carayol, M. Hague, and O. Serre. C-shore : a collapsible approach to higher-order verification. In *ICFP'13*. ACM, 2013.

[3] G. Castagna, K. Nguyen, Z. Xu, H. Im, S. Lenglet, and L. Padovani. Polymorphic functions with set-theoretic types : part 1 : syntax, semantics, and evaluation. In *POPL'14*. ACM, 2014.

[4] F. Chalub and C. Braga. A Modular Rewriting Semantics for CML. *J.UCS*, 10(7) :789–807, 2004.

[5] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, C. Löding, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http://tata.gforge.inria.fr`, 2008.

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

[7] T. Genet. Towards Static Analysis of Functional Programs using Tree Automata Completion. In *WRLA'14*, volume 8663 of *LNCS*. Springer, 2014.

[8] T. Genet, Y. Boichut, B. Boyer, V. Murat, and Y. Salmon. Reachability Analysis and Tree Automata Calculations. IRISA / Université de Rennes 1. `http://www.irisa.fr/celtique/genet/timbuk/`.

[9] T. Genet, T. Le Gall, A. Legay, and V. Murat. A Completion Algorithm for Lattice Tree Automata. In *CIAA'13*, volume 7982 of *LNCS*, pages 134–145, 2013.

[10] T. Genet and R. Rusu. Equational tree automata completion. *Journal of Symbolic Computation*, 45 :574–597, 2010.

[11] T. Genet and Y. Salmon. Tree Automata Completion for Static Analysis of Functional Programs. Technical report, INRIA, 2013. `http://hal.archives-ouvertes.fr/hal-00780124/PDF/main.pdf`.

[12] T. Genet and Y. Salmon. Reachability Analysis of Innermost Rewriting. In *RTA'15*, LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. To be published.

[13] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1-3) :120–136, 2007.

[14] N. Kobayashi. Model Checking Higher-Order Programs. *Journal of the ACM*, 60.3(20), 2013.

[15] L. Ong and S. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *POPL'11*. ACM, 2011.

[16] S. Owens. A Sound Semantics for OCamllight. In *ESOP'08*, volume 4960 of *LNCS*, pages 1–15. Springer, 2008.

[17] N. Vazou, P. Rondon, and R. Jhala. Abstract Refinement Types. In *ESOP'13*, volume 7792 of *LNCS*. Springer, 2013.

La composition de services dans le monde asynchrone
Formalisation et vérification en TLA+

Florent Chevrou Aurélie Hurault Philippe Mauran Meriem Ouederni
Philippe Quéinnec Xavier Thirioux

IRIT – Université de Toulouse

Résumé

Les architectures orientées services (SOA) permettent de répondre à deux défis importants du génie logiciel : la réutilisabilité et la décomposition. Néanmoins elles amènent de nouveaux problèmes, notamment liés à la répartition des services et la non-centralisation du contrôle. Les services étant indépendants et autonomes, il faut s'assurer que mis ensemble ils sont capables de communiquer et que leurs interactions n'introduisent pas de mauvais fonctionnement global. Dans le monde asynchrone, plus proche de la réalité, cette vérification devient non triviale, et cela d'autant plus qu'il existe de multiples modèles asynchrones, plus ou moins libéraux dans ce qu'ils autorisent. Nous exposons dans ce papier nos travaux en cours autour des modèles asynchrones et de la vérification des compositions de services paramétrées par ces modèles.

1 La composition de service...

Les architectures orientées services sont intéressantes de deux points de vue. Elles ont l'avantage de découper un problème en sous-problèmes plus simples et elles permettent de réutiliser les services. Ces architectures étant distribuées et si possible non contrôlées de façon centralisée, une des problématiques est la réalisation de l'assemblage [MP09] pour s'assurer que les services communiquent correctement ensemble et ont le comportement souhaité.

1.1 Obtention des compositions de services

Les systèmes distribués constitués d'un ensemble de processus sont généralement construits de manière ad-hoc. Mais la composition peut aussi découler d'une spécification. Dans ce cas, deux approches prédominent : l'orchestration de services et la chorégraphie de services [BGG+06]. L'orchestration conduit à la génération d'un contrôleur centralisé en charge de contrôler les échanges de messages entre les services. La distribution de ce contrôleur reste un problème majeur. Une chorégraphie spécifie, d'un point de vue global, les interactions (par messages) entre les participants d'une collaboration. Une chorégraphie ne peut pas être exécutée, elle est jouée quand les participants exécutent leur rôle. On dit qu'une chorégraphie est réalisable si, quand chaque participant agit indépendamment en fonction de son comportement, le comportement global joue la chorégraphie.

1.2 Vérification des compositions

De nombreux travaux s'intéressent à la vérification de compositions de services. Différents formalismes sont utilisés pour représenter les services : machine à états finis [DOS12, CLB08,

BCT04], algèbre de processus [BCPV04], réseaux de Petri [LFS+11, Mar03]. Différents critères sont utilisés pour représenter la compatibilité : absence d'interblocage [DOS12], réception non spécifiée [BZ83, DOS12], état terminal accessible [DOS12, BCT04, LFS+11], état terminal nécessairement atteint [BCT04, BCPV04], absence de famine [FUMK04], divergence [BCPV04]. Les modèles de communications utilisés sont principalement synchrones [DOS12, BCT04, FUMK04, BCPV04], plus rarement asynchrones [PS12].

Dans le cas asynchrone, chaque travail possède son propre modèle et propose une approche specifique pour ce modèle, en ignorant la diversité des modèles asynchrones. Par exemple, [PS12] propose une approche pour vérifier la réalisabilité d'une chorégraphie BPMN 2.0 dans le monde synchrone ou asynchrone et [BBO12] donne une condition nécessaire et suffisante pour prouver la réalisabilité d'une chorégraphie en asynchrone. Mais tous deux sont restreints à une forme particulière de communication asynchrone (FIFO n-1 dans la suite).

1.3 Nos choix

Nous nous intéressons à la diversité des modèles asynchrones. Il est important de noter que dans le monde asynchrone, c'est le médium qui décide du message délivré : les applications spécifient les canaux qu'elles écoutent, mais elles ne peuvent pas imposer l'ordre de délivrance des messages. En ce sens, c'est le modèle de communication qui *pousse* les messages vers les services, en respectant certaines propriétés d'ordre. Par ailleurs, nous utilisons différents critères de compatibilité : la terminaison (tous les processus terminent dans un état d'acceptation), l'absence de service définitivement bloqué en attente de message, l'absence de message inattendu (le modèle de communication délivre un message à un service qui ne sait pas le traiter à ce point). Enfin, nous avons choisi de décrire les services par des systèmes de transitions qui découlent d'une description CCS (sans la règle de communication synchrone, voir section 3.1), mais les résultats sur la comparaison des modèles de communication ne sont pas restreints à ces descriptions.

2 . . . dans le monde asynchrone

Le comportement et la correction d'ensemble d'une composition de services reposent sur l'exécution et l'ordonnancement des interactions entre composants ou services. La mise en œuvre des interactions a donc un impact direct sur la viabilité de l'application globale. Or, dans un contexte réparti, il existe des écarts considérables entre les différents protocoles d'interaction, en termes d'efficacité et de faisabilité. Il est donc essentiel de situer les différents modèles d'interactions les uns par rapport aux autres au regard de leur facilité et économie de mise en œuvre. De plus il faut que le contrat définissant le service d'interaction lui-même soit complètement explicité, afin de permettre d'évaluer et vérifier la composition de services qu'il emploie.

2.1 Les modèles

Une application répartie est vue comme un ensemble de services (ou sites) séparés, communiquant par échange de messages au travers d'un réseau asynchrone. Chaque site dispose d'une mémoire locale, privée. La communication est *asynchrone*, c'est-à-dire qu'il n'existe pas de borne maximale sur le temps d'acheminement d'un message par le réseau. Cet asynchronisme, caractéristique des environnements répartis, induit de nombreux résultats d'impossibilité [FLP85, BZ83]. Par exemple, il est impossible de détecter l'arrêt d'un site distant, puisqu'il est impossible de savoir si un site muet est effectivement en panne, ou a émis des

messages non encore parvenus. Enfin, le système est ouvert et dynamique : à tout moment de l'exécution, des sites peuvent rejoindre ou quitter l'ensemble des sites connectés.

Les différents modèles asynchrones pratiquement utilisés et que nous avons étudiés sont :

FIFO n-n Les messages sont ordonnés globalement et sont consommés dans leur ordre d'émission. Ce modèle repose sur un objet centralisé/partagé (par exemple une unique FIFO). Il reste éloigné d'une implantation répartie efficace et ne constitue qu'une première étape pour découpler les événements d'émission et de réception et ainsi s'éloigner du modèle synchrone.

FIFO n-1 Pour un service donné, les messages sont consommés dans leur ordre global d'émission. Ce modèle est souvent confondu avec FIFO 1-1 alors qu'il induit un ordre plus fort : alors même que les messages émis peuvent être totalement indépendants, l'ordre de leur délivrance est leur ordre d'émission *dans le temps absolu*. Ainsi, un événement d'émission est ordonné implicitement par rapport à l'ensemble des émissions vers le même destinataire.

Causal Les messages sont consommés dans un ordre respectant la causalité de leurs émissions [Lam78] : si un message m_1 est émis causalement avant un message m_2 (c-à-d. qu'il existe un chemin causal d'une émission à l'autre), alors un même site ne peut délivrer m_2 avant m_1. La réalisation de ce modèle nécessite la capture de la relation de causalité, par des histoires causales ou des vecteurs/matrices d'horloges.

FIFO 1-1 Les messages d'un même émetteur pour un même récepteur sont consommés dans leur ordre d'émission. Il peut être implanté grâce à une file associée à chaque couple de sites.

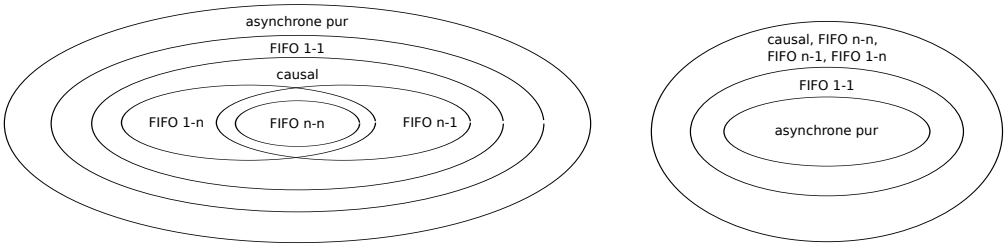
Asynchrone (pur) Il n'y a aucune contrainte sur l'ordre de consommation. Le réseau peut être vu comme un multi-ensemble où les sites déposent et d'où le réseau extrait arbitrairement des messages pour les délivrer.

FIFO 1-n les messages d'un même émetteur sont consommés dans leur ordre d'émission. Ce modèle est le pendant du FIFO n-1, avec, par exemple, sur chaque site, une file des messages émis.

2.2 Liens entre les modèles

Les modèles imposent plus ou moins de contraintes sur l'ordre de délivrance des messages. Si nous nous intéressons aux exécutions (suites d'émissions et consommations de messages), nous disons que pour un modèle de communication M et une exécution σ, $M \models \sigma$, si σ respecte l'ordre de consommation du modèle M. Pour deux modèles de communications M_1 et M_2, $M_1 \subseteq M_2 \triangleq \forall \sigma : M_1 \models \sigma \Rightarrow M_2 \models \sigma$. La figure 1(a) résume les liens entre les modèles. Certains résultats sont connus depuis longtemps (lien asynchrone, FIFO 1-1, causal, FIFO n-n), les preuves pour les modèles FIFO 1-n et FIFO n-1 ont été réalisées.

Nous nous intéressons aussi au comportement du système dans son ensemble. Pour deux modèles de communication M_1 et M_2, nous notons $M_1 \sqsubseteq M_2$ si tout système qui ne conduit pas à un état d'erreur (délivrance d'un message que le système ne sait pas traiter) dans M_1 ne conduit pas à un état d'erreur dans M_2. La figure 1(b) montre la hiérarchie des modèles dans le cas de système où tout message finit par être consommé. On remarque que les modèles causal, FIFO n-n, FIFO n-1 et FIFO n-1 se confondent. En effet si une exécution causale mène à un état d'erreur, il est également possible d'en trouver une similaire (au réordonnancement des émissions près), conforme au modèle FIFO n-n, qui mène également à l'état d'erreur. Les preuves sont en cours de réalisation.



(a) $\subseteq$ (b) $\sqsubseteq$

FIGURE 1 – Inclusion des modèles de communication

3 Formalisation et implantation

Nous avons défini formellement les modèles de communication et développé un outil de vérification de compatibilité. L'outil permet de vérifier automatiquement des compositions pour différents modèles de communication. Cet outil peut également servir à la vérification de chorégraphie selon la démarche de [PS12], qui lui ne gère qu'un seul modèle de communication.

3.1 Formalisation

Soit $\mathcal{C}$ un ensemble énumérable de **canaux**. Un canal n'est pas restreint à un unique émetteur ni à un unique récepteur. Les canaux peuvent être partitionnés en groupes auxquels sont associés différents modèles de communication.

Un **service** est défini par un système de transition étiqueté dont les étiquettes sont :
— $c!$: envoi d'un message sur le canal c ($c \in \mathcal{C}$) ;
— $c?$: réception de message sur le canal c ($c \in \mathcal{C}$) ;
— τ : action interne.

Un **modèle de communication** est élémentaire, comme fifo 1-1 pour un ensemble des canaux, ou composite, associant des modèles élémentaires différents à des groupes de canaux. Il est défini par un système de transition étiqueté dont les étiquettes sont :
— un sous-ensemble de $\mathbb{N} \times \bigcup_{c \in \mathcal{C}} \{c!\}$: envoi par le service i d'un message sur le canal c ;
— un sous-ensemble de $\mathbb{N} \times \bigcup_{c \in \mathcal{C}} \{c?\} \times \mathcal{P}(\mathcal{C})$: réception par le service i d'un message sur le canal c alors que le service est en écoute sur un ensemble de canaux ;
— τ : action interne.

Un **système** est composé d'un ensemble de services et d'un modèle de communication. Il s'agit d'un produit synchronisé entre les services et le modèle de communication, où une transition est la synchronisation entre une émission (resp. réception) d'un service, et une action d'émission (resp. de réception) du modèle de communication. Enfin plusieurs **critères de compatibilité** ont été définis : la terminaison, l'absence d'interblocage, l'absence de message jamais consommé, la délivrance d'un message inattendu... Ces notions de système, services, critères de compatibilité et modèles de communication ont été formalisées en TLA+ [Lam03], dans le but d'en donner une description non ambiguë et de bénéficier des outils associés. Le choix s'est porté sur TLA+ car un modèle de spécification de haut niveau et expressif était nécessaire et que nous bénéficions des outils associés : le vérificateur de modèle TLC et l'assistant de preuve TLAPS.

3.2 Implantation

Un outil a été développé pour vérifier la composition de service, paramétrée par les modèles de communication et les critères de compatibilité considérés. Partant d'une description CCS

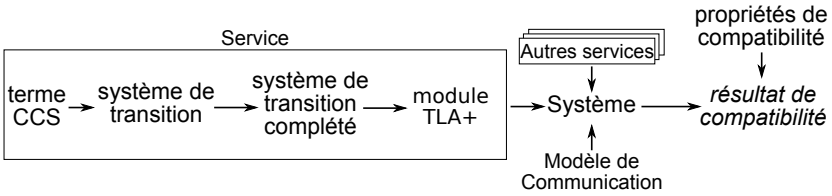


FIGURE 2 – Étapes principales de l'implantation

des services, l'outil génère le système de transitions correspondant à chaque service (sans la règle de communication synchrone de CCS), le traduit en une spécification TLA+, et construit le produit avec les modèles de communication étudiés (cf figure 2). L'outil appelle ensuite TLC, le vérificateur de modèle de TLA+, pour vérifier si les services sont compatibles pour un critère de compatibilité donné. Une étape importante de la conversion en TLA+ est la complétion du service : pour détecter la situation où un message inattendu est délivré au service, chaque état est complété pour assurer qu'il a une transition de réception de messages pour tous les canaux écoutés. Ces transitions ajoutées conduisent à un état d'erreur.

3.3 Exemple

Considérons un système faisant intervenir quatre processus (un étudiant, un responsable de formation, un secrétaire et un enseignant) et représentant les interactions lorsqu'un étudiant doit passer une seconde session d'un examen. Voici les termes CCS décrivant ces processus :

$$\begin{aligned}
\textbf{Responsable} &\triangleq nom! \cdot nom! \cdot session2! \cdot (ok? \cdot 0 + ko? \cdot annule! \cdot note! \cdot 0)\\
\textbf{Secretaire} &\triangleq nom? \cdot note? \cdot 0\\
\textbf{Etudiant} &\triangleq session2? \cdot (\tau \cdot ko! \cdot 0 + \tau \cdot EtudiantOK)\\
EtudiantOK &\triangleq ok! \cdot examreq! \cdot documents? \cdot exam? \cdot reponses! \cdot 0\\
\textbf{Enseignant} &\triangleq nom? \cdot (annule? \cdot 0 + examreq? \cdot EnseignantExam)\\
EnseignantExam &\triangleq documents! \cdot exam! \cdot reponses? \cdot note! \cdot 0
\end{aligned}$$

En analysant cette composition avec notre outil, nous obtenons les résultats suivants :

	fifo n-n	fifo n-1	fifo 1-n	causal	fifo 1-1	async.
Terminaison avec un réseau vide	✔	✔	✔	✔	×	×
Terminaison partielle (secrétaire)	✔	✔	✔	✔	×	×
Pas de message inattendu	✔	✔	✔	✔	×	×
Pas de blocage en attente de msg	✔	✔	✔	✔	×	×

La délivrance causale est nécessaire car il y a un lien de causalité entre *nom* (*Responsable* → *Enseignant*) et *examreq* (*Etudiant* → *Enseignant*) via *session2* (*Responsable* → *Etudiant*). Par ailleurs, la terminaison de la secrétaire n'est pas triviale, la note pouvant lui parvenir par deux chemins (via le responsable si annulation de l'étudiant ou via l'enseignant si l'étudiant décide de repasser l'examen).

4 Conclusion et perspectives

Ces travaux visent à éclairer la diversité du monde asynchrone et à fournir des outils pratiques adaptés à cette multiplicité. Ainsi nous pouvons vérifier automatiquement la correction de la composition de services, paramétrée par une combinaison de modèles de communication. L'utilisation du vérificateur de modèle TLC limite les vérifications aux systèmes ayant un nombre fini d'états. Des travaux sont en cours pour réaliser des preuves avec TLAPS (TLA Proof System) de façon à traiter des systèmes plus complexes, notamment des systèmes

paramétrés par le nombre de sites et qui présentent une certaine symétrie. Par ailleurs, les travaux en cours finalisent les relations entre les modèles et l'impact des opérateurs de communication sur ces relations, afin de clarifier la diversité du monde asynchrone et les spécificités de chaque modèle.

Références

[BBO12] Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*, pages 191–202, January 2012.

[BCPV04] Antonio Brogi, Carlos Canal, Ernesto Pimentel, and Antonio Vallecillo. Formalizing web service choreographies. *Electron. Notes Theor. Comput. Sci.*, 105 :73–94, December 2004.

[BCT04] Boualem Benatallah, Fabio Casati, and Farouk Toumani. Analysis and management of web service protocols. In *Conceptual Modeling – ER 2004*, volume 3288 of *LNCS*, pages 524–541. Springer, 2004.

[BGG+06] Nadia Busi, Roberto Gorrieri, Claudio Guidi, Roberto Lucchi, and Gianluigi Zavattaro. Choreography and orchestration conformance for system design. In *Coordination Models and Languages*, volume 4038 of *LNCS*, pages 63–81. 2006.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2) :323–342, April 1983.

[CLB08] Heung Seok Chae, Joon-Sang Lee, and Jung Ho Bae. An approach to checking behavioral compatibility between web services. *International Journal of Software Engineering and Knowledge Engineering*, 18(2) :223–241, 2008.

[DOS12] Francisco Durán, Meriem Ouederni, and Gwen Salaün. A generic framework for n-protocol compatibility checking. *Science of Computer Programming*, 77(7-8) :870–886, July 2012.

[FLP85] Michael Fischer, Nancy Lynch, and Michael Paterson. Impossibility of distributed consensus with one faulty process. *J. of the ACM*, 32(2) :374–382, April 1985.

[FUMK04] Howard Foster, Sebastian Uchitel, Jeff Magee, and Jeff Kramer. Compatibility verification for web service choreography. In *IEEE International Conference on Web Services*, pages 738–, 2004.

[Lam78] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7) :558–565, July 1978.

[Lam03] Leslie Lamport. *Specifying Systems*. Addison Wesley, 2003.

[LFS+11] Xitong Li, Yushun Fan, Q. Z. Sheng, Z. Maamar, and Hongwei Zhu. A Petri net approach to analyzing behavioral compatibility and similarity of web services. *Trans. Sys. Man Cyber. Part A*, 41(3) :510–521, May 2011.

[Mar03] Axel Martens. On compatibility of web services. *Petri Net Newsletter*, pages 12–20, 2003.

[MP09] Annapaola Marconi and Marco Pistore. Synthesis and composition of web services. In *Formal Methods for Web Services*, volume 5569 of *LNCS*, pages 89–157. 2009.

[PS12] Pascal Poizat and Gwen Salaün. Checking the Realizability of BPMN 2.0 Choreographies. In *27th Symposium On Applied Computing (SAC 2012)*, pages 1927–1934, March 2012.

Reverse Engineering for Smart Grid Modeling in the Scope of the SESAM Grids Project†

Gabriel Pedroza and Pascale Le Gall
CentraleSupélec, MAS Laboratory
Grande Voie des Vignes,
92295, Châtenay-Malabry, France
{gabriel.pedroza, pascale.legall}@centralesupelec.fr

Christophe Gaston
CEA, LIST, LISE Laboratory
Point Courrier 174,
91191, Gif-sur-Yvette, France
christophe.gaston@cea.fr

Fabrice Bersey
SCLE-SFE
25 Chemin de Paleficat,
31200, Toulouse, France
fabrice.bersey@scle.fr

***Abstract*—Smart Grids are the evolution of the progressive integration of smarter SW-based components into the electricity grid. As long as Smart Grids gain in connectivity and automation, new concerns on their safety, security, and interoperability arise. In particular, significant risks and extended impact due to misbehaviors or failures are recognized. A promising strategy to effectively address these issues is to join research and industry expertises. Among the initiatives to improve the safety and security of Smart Grids is the SESAM Grids project†. Following a model driven paradigm, the project undertakes the challenge of analyzing these complex widely-distributed systems. This paper summarizes our experience in the development of a reference model to support safety and security analyses. Yet an experimental Smart Grid infrastructure is available, a lack of specification, confidential technology, and constrained technical resources are among the faced drawbacks. Our proposal consists in a reverse engineering method to overcome these limitations. This work shows how the method was successfully applied to obtain a well-founded model. Some perspectives are finally given to strengthen model conformity and to proceed with analyses.**

I. Introduction

As stated by the European Network of Transmission System Operators for Electricity (ENTSOE), around 60% of the electricity in Europe comes from non-renewable sources (fossil fuels and nuclear) [1]. This fact, along with the pervasive use of electricity, have motivated the need for an efficient use of resources. The adopted strategy consists in deploying systems for better driving electricity demand and generation response. It comprises to boost de development of renewable source technology (wind, solar, hydraulic) and to keep customers informed about their consumption, sell prices, and environmental effects. The so named Smart Grids are the technology committed to support just referred strategy. They emerged as a result of the progressive integration of computerized technology into the electricity grid. The assortment of electric, electronic, and programmable components is quite complex and distributed along wide areas. The dependency of modern industrial society on the grid has led to consider certain risks. As explained in [2] and [3], the impact in case of safety or security incidents is non-negligible. Referred incidents range from accidental misbehaviors, due to failures or flawed applications, up to premeditated network misuse, due to intrusions or attacks by third parties. The assessment and evaluation of potential risks have increasingly attracted the attention from several sectors. In particular, SESAM Grids is a joint academy-industry project (2013-2016) settled to address safety and security of Smart Grids (the acronym stands for "**S**afety, s**E**curity, and **S**tand**A**rdisation for s**M**art **Grids**"). It is a long term investment program focused on the development of generic modules for a safe-secure embedded architecture. The consortium is composed by CentraleSupélec, TRIALOG, COFELY-INEO, and CEA-LIST.

This paper is dedicated to describe the SESAM Grids project and some initial results. In particular, how to conduct analyses relying upon an experimental Smart Grid without models or specifications, and with constrained technical resources. Since a model driven approach is adopted, we summarize our experiences in the development of a well-founded model relying upon reverse engineering techniques. To do so, we first introduce Smart Grids in section II. The project motivations and scope are presented in section III. The target Smart Grid subsystem is also precised there. The section IV explains the main drawbacks faced during modeling tasks, and the method proposed to tackle them. The initial results and technical achievements from applying the method come in section V. Finally, some tasks for improving our approach and better supporting further analyses are listed in section VI.

II. Smart Grids

A. Overview

The Smart Grids are seen as the coupling of the electricity grid and computerized technology. The grid supports the electricity flow, from generation sites up to customer areas, relying upon a variety of transmission, distribution, and storage assets. The digital technology has progressively improved the ability to monitor and control the overall infrastructure. More precisely, the so named Supervisory Control and Data Acquisition System (SCADA) has enlarged network connectivity and automation [4]. The SCADA is a network of programmable electronic devices interconnected via a variety of links and buses. The goals imposed to this widely distributed architecture are to be safe, economically viable, and environmentally adequate. To accomplish its goals, the SCADA is dedicated to monitor the grid so as to assess and control its status. To

† National research project supported by the Economy Industry and Digital Technology Ministry of France

do so, several parameters are periodically sampled by sensors at field level in order to obtain electricity consumption, power generation, events occurrence, etc. The sampling architecture is also referred as Advanced Metering Infrastructure (AMI). As shown in figure 1, the data sampled by the AMI are later gathered by a global network center to be processed by estimation and optimization algorithms.

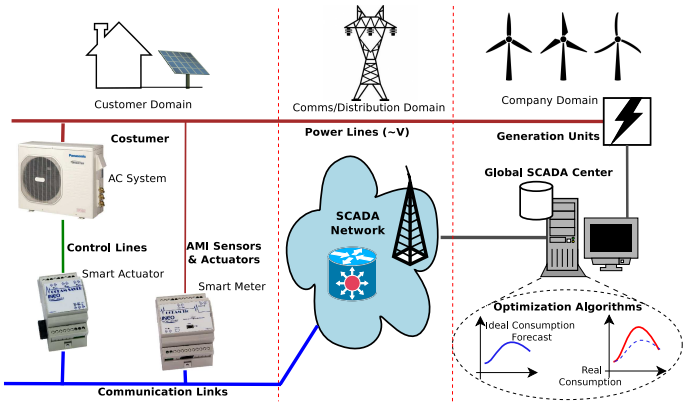


Fig. 1. Scheme showing a global overview of a Smart Grid

The outcomes finally help to assess and control the overall grid operation via AMI actuators. In addition, most of the data are not time critical and thus network bottlenecks are not an issue. Nevertheless, certain safety-critical messages should respect stringent time thresholds, like the commands to protect substations from overload peaks (3ms) [2]. The smart electronic devices are composed by proprietary and open source technology. Indeed, along with proprietary middleware, the distributed applications can rely upon a variety of protocols like DNP3, MODBUS, IEC61850, and TCP/IP [2]. This imposes certain difficulties with regard to networks interoperability. The complexity of this widely-distributed highly-networked system imposes a challenge to ensure safety goals. First, many of the SCADA and AMI sub-systems deployed at field level were originally designed to operate locally. Consequently, they may be poorly protected and become vulnerable to attacks [5]. Secondly, grid misbehaviors due to flawed applications or intrusions can be propagated and possibly impact human safety. Finally, since the collected data can be linked to sensitive information - e.g., identities - the privacy of customers may also be at stake [5].

B. Particular Objectives

The particular objectives pursued by the Smart Grids are:

1) *Reinforce the safety of the electricity transport grid so as to limit the consequences in case of failure*
2) *Settle adequate trade-offs between electricity demand and bulk generation considering production/sell prices, grid capacity, and pollution effects*
3) *Deploy the technology to integrate intermittent renewable energy sources - like wind turbines or solar panels - to the electricity grid*
4) *Deploy the technology to decentralize electricity production* - e.g., *by connecting solar panels owned by customers - and to measure its contribution*
5) *Optimize the use of the overall electricity infrastructure*

III. The SESAM-Grids Project

A. Motivations and Scope

The following issues are identified as crucial to accomplish Smart Grids objectives:

- **Complex technology:** Most of the Smart Grid goals are driven the development of new electronic components and applications. The developments are progressively merged into the target electricity assets from their very design leading to embedded systems with increased capacities and also complexity.
- **Heterogeneous domain network:** Customers are not supposed to access Smart Grid components, even if they are installed at customer areas. On the other hand, electricity companies may be unable to efficiently supervise components at customer side. The Smart Grid technology should stay compliant with the policies to rule the complex heterogeneous domain network.
- **Interoperability mismatches:** Neighbor Smart Grids may be prompted to operate together, *e.g.*, to cover electricity demand during peak hours. The wide variety of technology and criteria upon which Smart Grids are deployed imposes a challenge to ensure interoperability.

The complexity of the evolving technologies may impact overall Smart Grids safety. The analysis of security is also paramount in order to prevent Smart Grid misuse by third parties or attackers. More specifically, the expected operation of the grid can only be ensured if the grid is adequately protected against its threats. Consequently, Smart Grid safety also depends upon an adequate threats, risk, and countermeasures assessment. Last but not least, the means to cope with interoperability issues should also be proposed. The SESAM Grids project emerged as a joint effort of academy and industry to undertake the referred issues. The project outcomes should finally lead to a first draft of guidelines aligned with the IEC-61508 standard [6]. IEC-61508 is a reference in industry for the development of safety-critical systems based upon electric, electronic, and programmable devices. Finally, the validation guidelines are intended to summarize project experiences.

B. Approach Rationale

To undertake the issues referred in previous subsection III-A, a reference Smart Grid is first delimited. The objective is to target a Smart Grid subsystem identified as safety and security critical. The targeted subsystem should make evident Smart Grids complexity and their typical interoperability conflicts. According to that, the project was finally oriented to the so named Micro-Grids. As agreed by several European standardization committees like CEN-CENELEC [7], the Micro-Grid is a heterogeneous architecture deployed over multiple domains from field to enterprise - see figure 2. More precisely, the Micro-Grid consists of assets like the AMI, including part

of the SCADA, Distributed Energy Resources (DER) - *e.g.*, load management units -, and auxiliary automation systems.

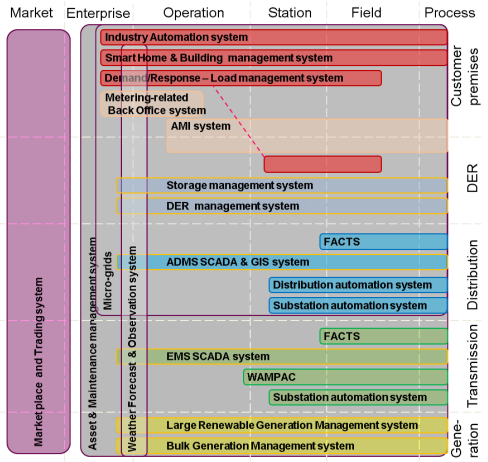


Fig. 2. The Micro-Grid as defined by CEN-CENELEC-ETSI

The Micro-Grid border circumvents a reference architecture. The reference architecture consists of the electronic components - HW and SW -, the communication channels, as well as the applications upon which safety and security analyses will be conducted. This Micro-Grid is already deployed in a demonstration site at Toulouse named Smart ZAE [8]. According to that, the project strategy can be summarized in three stages. In the first stage, the Micro-Grid subsystem is modeled and a methodology is proposed to analyze and improve its safety and security. Secondly, and based upon the IEC-61508 standard, a first draft of ground rules for Micro-Grid validation is addressed. Finally, and as a proof of concept, a Micro-Grid demonstrator is settled in order to provide evidence of feasibility.

C. SESAM Grids Consortium

The accomplish its goals, the SESAM Grids project encompasses the expertise of several academy and industry partners. The consortium is briefly presented in the next paragraphs.

- ***COFELY-INEO [9]:*** It is a branch of the GDF-SUEZ group, the latter a company specialized in the development of systems for the energy industry. COFELY-INEO provides engineering solutions for the design, integration, and deployment of systems for multiple industries: electric, public lighting, transport, telecommunications, bulk generation and transmission.
- ***CEA-LIST [10]:*** It is a research institute from the Commissioner for Atomic Energy bureau (CEA) of the french government. The CEA-LIST is focused on the design and improvement of digital systems. It is specialized in the research for the development of interactive, embedded, sensor, and signal processing systems for a variety of sectors: energy, health, security, transport and manufacturing.
- ***CentraleSupélec-MAS [11]:*** It is a research team of the CentraleSupélec, the latter a complex of engineering schools and research centers. Among others, the MAS laboratory brings experience and expertise in component-based systems modeling and validation. Its expertise includes research on methods for the analysis of critical systems based upon conformity testing.
- ***TRIALOG [12]:*** It is a consulting company that provides support for the development of new embedded and industrial technologies. Its main fields of activity are on intelligent transport, energy, and machine-to-machine systems. Among others, TRIALOG brings expertise in the development of HW/SW-based architectures according to the Model Driven Engineering approach.

IV. Reverse Engineering Method

In previous section, a Micro-Grid subsystem was delimited. This section explains the motivations for considering reverse engineering techniques and how they were applied in the context of the project.

A. Motivations

The SESAM Grids project is committed to improve the safety and security of Smart Grids. As shown in [13], the reinforcement of security by modeling is possible. Thus, we rely upon modeling to drive the respective analyses. Since the Micro-Grid subsystem is already deployed at the Smart ZAE site, we plan to develop a reference model by observation of the Micro-Grid behavior. The observation can simply consists of a sequence of actions executed by the system. From the objectives listed in sections III-A and III-B, it results evident that most of the project goals depend upon safety and security analyses. Thus, we are prompted to propose a framework adequate to conduct both modeling and analysis. Among the languages that have proven to be effective for the design and analysis of systems is the Unified Modeling Language (UML) [14]. Complementary, the consortium can propose and develop other required technology and frameworks. After all, design a model of the Micro-Grid subsystem becomes a project milestone. Nevertheless, the following aspects interfered with the steady development of the model:

1) *Technical specification:* The Micro-Grid subsystem is composed by an assortment of components running different distributed applications and no specification exists for the overall behavior. The lack of a specification is common during the development of systems used for experimentation or research purposes.
2) *Code accessibility:* Many of the source code running on the Micro-Grid components is proprietary and intellectual property policies apply. The technology is often confidential and should not be disclosed to non-intended parties, even in the context of a collaborative project.
3) *Micro-Grid ambit:* The Smart ZAE site is a decisive infrastructure for experimentation, tests, and validation. However, its access is limited and the development of any testing workbench requires the deployment of important resources. The feasibility of experiments involving the Smart ZAE is subjected to previous constraints.

B. Reverse Engineering Principle

According to the drawbacks listed in previous subsection IV-A, it is concluded that a behavior learning, non-interfering, and cost-effective method needs to be proposed. The method should ultimately disclose information about the system in a non-invasive way. Moreover, it should rely upon the Smart ZAE "as is" and minimize the deployment of further resources. Since reverse engineering techniques are almost compliant with these restraints, they are proposed to overcome the drawbacks. As depicted in figure 3, reverse engineering roughly consists in inferring a behavior by analyzing the target system from an outer perspective. Certain accessibility to the system should be assumed but it is initially perceived as a reactive black box. By providing certain stimuli to the system, the respective response provides information about its behavior. As long as system accessibility is ensured, numerous aspects can be inferred, since even a quiescence can contribute to disclose system details. As demonstrated in [15], the use of reverse engineering techniques has proven to be effective even for unveiling security flaws.

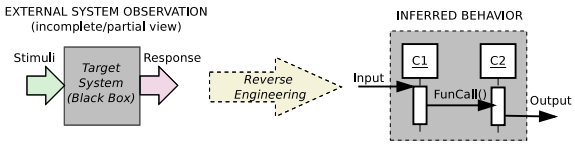


Fig. 3. Scheme showing the reverse engineering principle

C. Proposed Reverse Engineering Method

A method is proposed in order to adapt and apply reverse engineering techniques in the context of the project. First, a way to observe the system should be found. Typically, systems can be inspected via a packet analyzer, *e.g.*, Wireshark [16]. However, a simpler option is to rely on the data logger mechanism implemented in a vast majority of systems. Thus, our method takes as input a file from the data logger mechanism implemented in the Micro-Grid subsystem. The so called log file contains information about system actions, events, and time occurrence as well as references to the involved Micro-Grid components and modules. The method is described in the following items relying on figure 4.

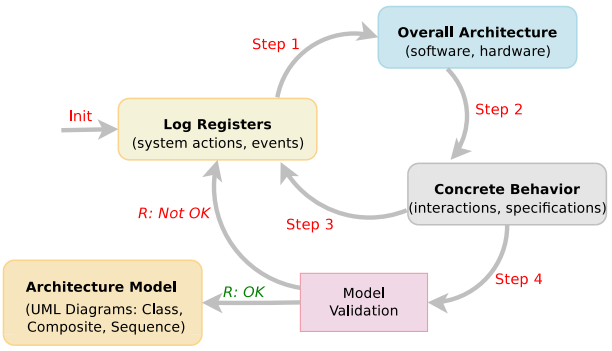


Fig. 4. Phases of the reverse engineering method for the Micro-Grid

- *Step 1:* The source log file is parsed so as identify the involved Smart-Grid components or SW modules and their exchanges. Once identified, they are associated to the architecture assets composing the Smart-Grid subsystem. The Smart ZAE site is taken as reference to identify the architecture assets.
- *Step 2:* Since the log registers are time stamped, the exchanges between components/modules can be ordered. Thus, by ordering registers, a sequence of interactions can be proposed and an initial concrete behavior inferred. A first draft of a technical specification is also elaborated.
- *Step 3:* The inferred behavior is modeled and the technical specification enriched. The model and the technical specification are cross-checked with respect to the source log file. Further log files can be used to complement cross-checking and enlarge its validity.
- *Step 4:* After iterating on steps 1 to 3, a first model of the Micro-Grid subsystem and technical specification are realized. Both, the model and technical specification, are finally validated by engineers and operators of the Smart ZAE site. In case of model deviation, further iterations on steps 1 to 4 should be performed.

The validation process yields a reference model useful to conduct safety and security analyses. Any subsequent deviation of the model w.r.t. the behavior of the Micro-Grid can also be treated by iterating on the method phases.

V. Initial Results

This section is dedicated to show the application of the method proposed in previous section IV and to summarize the outcomes.

A. Micro-Grid subsystem

A general description of our target Micro-Grid subsystem was already provided in subsection III-B. Now, its main components, including HW and SW modules, are precised. The Micro-Grid subsystem is mostly placed at field level within the customers domain. It is a subset of the AMI architecture mainly composed by two kind of components - see figure 5. Placed at customers households, the so named Smart Meters are committed to continuously measure electricity consumption. The so called Smart Controller supervises the operation of Smart Meters and is also committed to collect their measures.

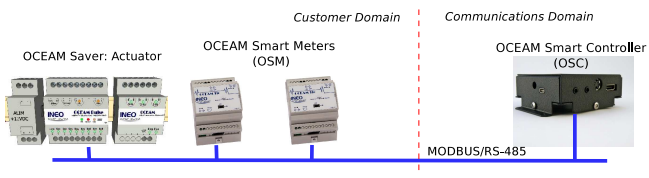


Fig. 5. Closer view of the target subsystem in the Smart ZAE site

Since the Controller operates as a gateway between customers and company domains, it is able to enforce a consumption policy by regulating the operation of the grid. For instance,

an electric device at customer side can be switched off via an Actuator. The components of the Micro-Grid subsystem are networked via a MODBUS channel [17]. The MODBUS is a serial master/slave protocol typically used in industry by its simplicity and reliability. It is recalled that, according to the initial safety and security risks analyses [18], the Micro-Grid subsystem is critical with regard to the safety and security of the overall Smart Grid. The system is deployed along customer and communications domains and it is virtually out of the company surveillance scope. Also, the Micro-Grid architecture integrates open and proprietary technology. Consequently, it is a good candidate for studying interoperability concerns.

B. Reverse Engineering Results

As a result of parsing and analyzing the registers of the source log file, several SW modules were identified as well as a variety of internal function calls and external exchanges. According to the information and by considering the reference Micro-Grid architecture, it is concluded that the file was truly generated by a Smart Controller. As can be seen in figure 6, the log registers are time stamped and contain information about actions/events and involved components/modules.

```
20140916 11:12:08.016 DEBUG  oceam-gateway.log (15024)- Téléchargement des dernières donnée
20140916 11:12:08.016 DEBUG  oceam-gateway.log (15024)- Définition du numéro d'esclave 4
20140916 11:12:08.016 DEBUG  oceam-gateway.log (15024)- Envoi numéro 1 de trame lv puis ls à l'adresse 4
20140916 11:12:08.026 DEBUG  liboceam.log (15024)- Envoi de la trame lv ...
20140916 11:12:08.026 DEBUG  libmodbus.log (15024)- Frame number is 1, data length is 1, data length in network order is 0x100
20140916 11:12:08.026 DEBUG  libmodbus.log (15024)- Data length in message is 0x00 0x01
20140916 11:12:08.526 DEBUG  libmodbus.log (15024)- 0 bytes flushed
20140916 11:12:08.526 DEBUG  libmodbus.log (15024)- [04][33] [03][F1] [6C][76] [01] [00][01] [01][D0][F2]
```

Fig. 6. Excerpt of a source log file from a Smart ZAE component

By ordering the registers and assuming a synchronous communication policy, an association between concerned modules and a concrete behavior was made - see figure 7.

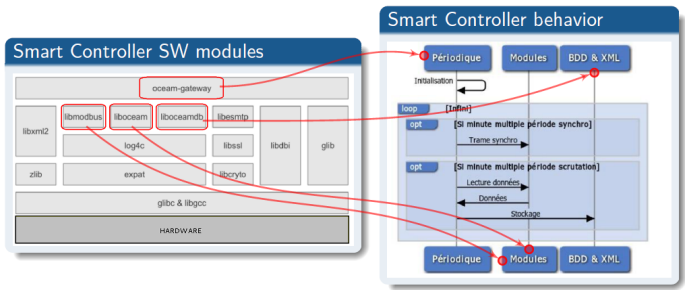


Fig. 7. Reverse engineering: inferring a behavior for a Micro-Grid component

As a result of this work a first model of the Micro-Grid subsystem was elaborated. The model was developed in Papyrus [19], a UML-Eclipse-based profile adequate to harmonize Class, Composite, and Sequence Diagrams. These diagrams were used in order to consider architectural aspects that are relevant for the safety and security analyses. An excerpt of the UML-based interactions model is presented in figure 8. In order to efficiently cross-check the model with the log file registers, we rely on a framework and tool named Diversity [20]. Diversity is developed by the CEA-LIST and

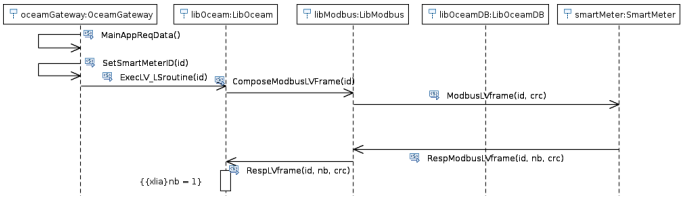


Fig. 8. Excerpt of the UML interaction model for the Micro-Grid subsystem

supports the translation of UML-based models into a syntax adequate to conduct formal analyses. More precisely, the Diversity algorithm is able to resolve on the behavior of Timed Input-Output Symbolic Transition Systems (TIOSTS) [21]. Among its features, Diversity allows to determine whether a sequence of Inputs-Outputs stays compliant with the TIOSTS of an UML model. The cross-checking was semi-automatically performed in two phases. In the first phase, the log file was translated into the Input-Output format accepted by Diversity. Afterwards in the second phase, the conformity between our UML model and the transformed log file is tested - this procedure is also known as offline testing. The overall outcomes were summarized in a technical specification which was eventually approved by the engineers and operators of the Smart ZAE site. A summary of our findings after applying the reverse engineering method is presented in table I

TABLE I
SUMMARY OF OUTCOMES FROM THE ANALYSIS OF THE LOG FILE

Trace Parameter	**Outcome**
Nb. of log registers	11,828
Identified words	110,476
Registers Date(s)	20140916
Trace duration	13 min 44.177 s
Nb. of interleavings	0
SW modules	oceam-gateway, liboceam, libmodbus, liboceamdb
Smart-ZAE component	OCEAM Smart Controller
Nb. of external Smart Meters	15
Nb. of MODBUS messages	230
Identified MODBUS frames	lv & ls
Minimum response delay	0.018 s
Maximum response delay	0.234 s
Nb. of main thread cycles	103
Associated behavior	For each Smart Meter: 1.- Request nb. of bytes to transfer 2.- Transfer and acknowledge data 3.- Convert and store data

C. Technical Achievements

In order to apply the reverse engineering method, three technical tasks were mainly accomplished - see figure 9.

- **T1:** The first task was to extend the translation from UML models in Papyrus into the TIOSTS syntax. This modification was crucial, since several modeling aspects were introduced in order to better capture the system.
- **T2:** Originally, the Diversity algorithm supports sequences where every Input or Output contains a single symbolic data. Yet, the exchanges within the Micro-Grid

subsystem usually convey multiple parameters. Thus, the second tasks consisted in adapting Diversity in order to align the algorithm with a manifold Input-Output format.

- **T3:** The log registers are stored in a high-level human readable format. This format is not really adequate to conduct conformity tests as implemented by Diversity. In order to efficiently conduct the cross-checking, a log file parser was developed to automatically translate log files into the Diversity Input-Output format.

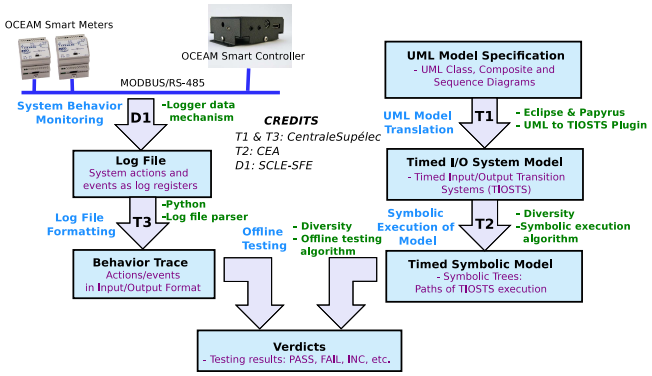


Fig. 9. Technical achievements for the initial phase of the project

VI. Future Tasks

Some tasks that can be conducted to improve our approach are described in the following paragraphs.

With regard to the improvement of the reference model, we plan to strengthen its conformity with respect to the Micro-Grid subsystem by considering more nominal and outside-of-specification log files.

With regard to the detection of nominal and flawed behaviors, we plan to validate our model by testing log files containing errors or misbehaviors. Error sequences can be generated by altering the nominal log files or by stressing the operation of the Micro-Grid subsystem. For instance, the system can be stressed by injecting corrupted or dummy packets into the MODBUS channel.

With regard to the detection of secure/insecure behaviors, we still have to consider the presence of intruders or attackers useful to analyze intrusions. The introduction of an attacker model in the Micro-Grid is not a minor issue and should be thoroughly considered

Finally, our approach is able to resolve on the conformity between a model and a specific behavior (cross-checking). Nevertheless, we ought to elaborate a method to evaluate its effectiveness. Such a method shall estimate or measure the coverage during the cross-checking activities. As summarized in table II, wrong verdicts can occur during the conformity tests. A wrong verdict occurs when a nominal sequence is wrongly rejected or when an flawed one passes the conformity test. Evaluations may be necessary in order to assess the impact of wrong verdicts on the overall safety and security.

TABLE II
Cases for effectiveness evaluation during conformity tests

	Conformity verdict	
	PASS	FAIL
Nominal Input-Output sequence	OK	X
Flawed Input-Output sequence	X	OK

References

[1] European Network of Transmission System Operators for Electricity, "Electricity in Europe: Synthetic overview of ENTSO-E electric system consumption, generation and exchanges during 2013," In https://www.entsoe.eu/.

[2] Wenye Wang and Zhuo Lu, "Cyber security in the Smart Grid: Survey and challenges." *Computer Networks*, vol. 57, no. 5, pp. 1344–1371. Elsevier SicenceDirect, 2013.

[3] L. Zhou and S. Chen, "A survey of research on smart grid security," in *Network Computing and Information Security*, ser. Communications in Computer and Information Science. Springer Berlin Heidelberg, 2012, vol. 345, pp. 395–405.

[4] Al Hamadi, Hussam M.N. and Yeun, ChanYeob and Zemerly, Mohamed-Jamal, "A Novel Security Scheme for the Smart Grid and SCADA Networks," *Wireless Personal Communications*, pp. 1–13. Springer US, 2013.

[5] Gao, J. and Xiao, Y. and Liu, J. and Liang, W. and Chen, C.L., "A survey of communication/networking in Smart Grids," *Future Generation Computer Systems*. Elsevier SicenceDirect, 2012.

[6] International Electrotechnical Commission, "Functional safety of electrical/electronic/programmable electronic safety-related systems," In http://www.iec.ch/functionalsafety/.

[7] The European Committee for Standardization and the European Committee for Electrotechnical Standardization, "The CEN-CENELEC Website," In http://www.cencenelec.eu/.

[8] SCLE-SFE, "The experimentation Micro-Grid site Smart ZAE," In http://www.cofelyineo-gdfsuez.com/smart-zae.

[9] COFELY-INEO from GDF-SUEZ group, "COFELY-INEO Website," In http://www.cofelyineo-gdfsuez.com/en/.

[10] CEA-LIST Laboratory, "CEA-LIST Website," In http://www-list.cea.fr/index.php/en/.

[11] The CentraleSupélec MAS Laboratory, "MAS Website," In http://www.mas.ecp.fr/.

[12] TRIALOG, "TRIALOG Website," In http://www.trialog.com/home/.

[13] Bannour, B. and Escobedo, J. and Gaston, C. and Le Gall, P. and Pedroza, G., "Designing Sequence Diagram Models for Robustness to Attacks," in *Proc. of Workshop SECTEST*. IEEE, 2014.

[14] Object Management Group, "The UML standard specification," In http://www.omg.org/spec/UML/2.4.1/.

[15] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *Security and Privacy (SP), 2010 IEEE Symposium on*. IEEE, may 2010, pp. 447–462.

[16] The Wireshark packet analyzer, "The Wireshark Foundation," In https://www.wireshark.org/.

[17] The Modbus Organization, Inc., "The Modbus Protocol Specification V1.1b3," In http://www.modbus.org/docs/.

[18] Le Gall, Pascale and Gaston, Christophe and Escobedo, Jose and Bannour, Boutheina and Pedroza, Gabriel, "Methodology for Modeling Smart Girds Systems: Attack Impact Analysis on Interaction Scenarios," The Sesam-Grids Consortium, Tech. Rep., December 2013.

[19] The Eclipse Foundation, "The Papyrus framework," In http://www.eclipse.org/papyrus/.

[20] Rapin, N. and Gaston, C. and Lapitre, A. and Gallois, J.-P., "Behavioural unfolding of formal specifications based on communicating automata," in *Proc. of Workshop ATVA*, 2003.

[21] P. Krcal and W. Yi, "Communicating timed automata: The more synchronous, the more difficult to verify," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Ball and R. Jones, Eds. Springer Berlin Heidelberg, 2006, vol. 4144, pp. 249–262.

MBT_Sec – Model-Based Testing for Security Components

Julien Botella[1], Frédéric Dadeau[2], Elizabeta Fourneret[2], Bruno Legeard[1,3], Julien Lorrain[2] and Romain Sibre[2]

[1]Smartesting Solutions & Services, Besançon, France
[2]FEMTO-ST/DISC - Université de Franche-Comté, Besançon, France

Résumé

Nous présentons dans ce papier le projet ANR ASTRID Maturation MBT_Sec (Model-Based Testing for Security Components). Ce projet vise à l'intégration et à l'industrialisation de travaux de recherche ayant pour objectif le test fonctionnel de sécurité, appliqué aux composants cryptographiques.

1 Contexte du projet

Les composants de sécurité présentent un caractère stratégique pour la sécurité des systèmes informatiques, tant dans le domaine Défense (pour le chiffrement et l'authentification par exemple) que dans le domaine Civil (au travers des cartes à puce et des systèmes de contrôle d'accès par exemple). Le test de sécurité de ces composants vise à garantir que les propriétés de sécurité définies, telles que la confidentialité ou l'intégrité, sont correctement implémentées au sein du composant. Il s'agit aussi de tester des vulnérabilités éventuelles du composant lorsque celui-ci est soumis à des attaques ou à des comportements malveillants.

Le projet MBT_Sec [1] vise le développement d'un environnement de génération automatique de tests de sécurité dédiés aux composants de sécurité. Il s'agit d'accroître la maturité technologique des résultats obtenus au sein du projet ASTRID OSeP [2] pour la faire passer d'un TRL 4 [3] actuellement à un TRL 5-6 et au delà.

Le projet MBT_Sec est fondé sur le partenariat entre Smartesting Solutions & Services, éditeur de logiciel, une PME innovante issue de la recherche, spécialisée sur les techniques de génération automatique de tests et l'Institut FEMTO-ST et notamment le Département Informatique des Systèmes Complexes. Dans le cadre des projets ASTRID (Accompagnement Spécifique des Travaux de Recherches et d'Innovation Défense), MBT_Sec s'effectue également en collaboration avec la DGA Maîtrise de l'Information / Sécurité des Systèmes d'Information, Département "Expertise de Logiciels Sécurisés" assure le suivi technique du projet, en étant partie prenante pour l'expérimentation et la validation des résultats.

Nous présentons dans ce résumé les résultats obtenus dans les projets précédents et faisant l'objet de ce présent besoin de maturation. Nous décrivons ensuite les premiers résultats obtenus suite à l'industrialisation de cette approche, au sein de l'outil de génération de tests Smartesting CertifyIt.

2 Combiner les approaches pour le test de sécurité

L'approche proposée dans le cadre du projet MBT_Sec s'articule autour du principe de test à partir de modèle [1] (MBT). Celui-ci consiste à utiliser un modèle fonctionnel de l'application

1. http ://projects.femto-st.fr/mbt_sec/
2. http ://osep.univ-fcomte.fr/
3. TRL = Technology Readiness Level

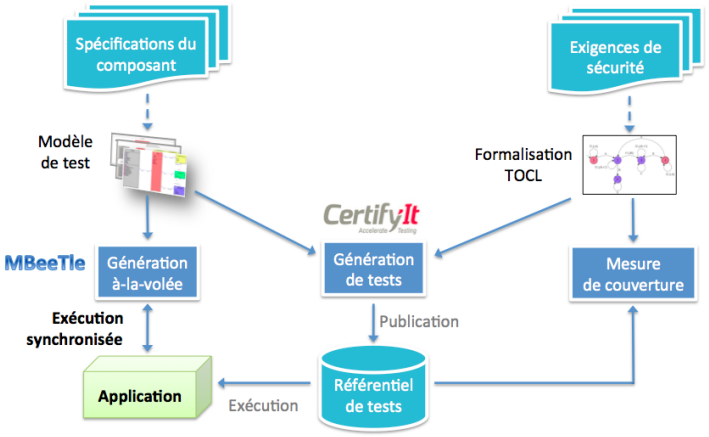


FIGURE 1 – Le processus de test de sécurité fonctionnelle proposé par MBT_Sec

pour décrire son comportement, et ainsi générer des tests, dits abstraits, visant à couvrir les comportements présents dans le modèle.

La figure 1 présente le processus MBT que nous souhaitons industrialiser. Le processus MBT classique démarre sur le côté gauche : un modèle de test est conçu par un ingénieur validation pour décrire le comportement du système qu'il veut valider [3]. Ici, nous utilisons des modèles décrits en UML/OCL. Un diagramme de classe permet de décrire le modèle du composant. Il est complété par un diagramme d'objets qui décrit l'état initial du système. La partie du dynamique du modèle, à savoir les comportements des opérations spécifiées, est décrite à l'aide de pré- et postconditions OCL, définissant les branches d'exécution du code. Suite à cette étape de modélisation, des critères de sélection de tests sont appliqués pour produire des cibles de tests, visant à exhiber chacun des comportements présents dans le code OCL. Un moteur de génération de tests calcule ensuite automatiquement une séquence d'opérations, qui, depuis l'état initial, permettra d'activer le comportement ciblé. Ces séquences constituent alors le référentiel de tests abstraits. Celui-ci pourra être concrétisés dans différents formats, dépendant du banc de test adressé, nécessitant le développement d'un adaptateur liant les données abstraites et les données concrètes. Ce processus MBT classique, dénomé également *off-line*, est implanté au sein de l'outil Smartesting CertifyIt [2].

Dans le cadre du projet MBT_Sec, les améliorations de ce processus sont de deux axes. D'une part, nous souhaitons améliorer la couverture de besoins de tests spécifiques, en utilisant des propriétés permettant de décrire les aspects de sécurité fonctionnelle du système. Ces propriétés peuvent être utilisées pour qualifier une base de tests, en évaluant si les tests existants couvrent la propriété, et/ou pour alimenter cette base de tests, en produisant des cibles pertinentes par rapport à la propriété considérée. D'autre part, nous nous sommes intéressés à la définition et à l'utilisation d'une technologie de génération de tests à la volée et en ligne, permettant de réaliser de larges explorations du modèle, couplant la génération et l'exécution des tests. Nous décrivons ces deux approches respectivement dans Section 2.1 et Section 2.2.

2.1 Test à partir de propriétés

Les propriétés utilisées dans notre approche sont basées sur les patrons de propriété de Dwyer [5]. Ceux-ci spécifiquent qu'une propriété est décrite par une combinaison de deux éléments : un motif (pattern) et une portée (scope) qui constitue une fenêtre d'observation de l'exécution du système, durant laquelle le motif, exprimant des occurrences ou des absences d'événements, doit être valide. Le langage TOCL [6], initialement proposé dans le cadre du projet ANR TASCCC, permet ainsi de décrire des propriétés temporelles à l'aide de ces constructions.

Par exemple, considerons la specification PKCS#11 qui définit l'API Cryptoki, une interface pour la gestion de la securité et l'interoperabilité des composants cryptographiques. La spécification définit differentes propriétés que nous sommes en mesure d'exprimer en TOCL. Ainsi, une des propriétés est définie comme suit : "Un utilisateur ne peut signer un message (opération C_SignInit) sans une authentification prélable (opération C_Login)". Le langage TOCL nous permet d'exprimer 2 propriétés qui formalisent cette exigence de sécurité fonctionnelle :

eventually isCalled(C_Login, @CKR :OK) **before** isCalled(C_SignInit, @CKR :OK)

qui définit qu'une opération de signature qui réussit (comportement CKR :OK) doit être précédée d'un opération de login ayant également été exécutée avec succès. On note ici la présence d'une portée (before - avant la première occurence de l'événement), et d'un motif (eventually - l'événement qui suit doit être présent). Les événements sont ici des appels aux opérations du système sous test, dans lesquelles on précise les comportements (ici, des cas de succès identifiés par @CKR :OK dans le code OCL de ces opérations).

Une seconde propriété vient compléter la précédente, pour exprimer qu'après une perte d'authentification, il y a nécessairement une étape de login pour pouvoir réaliser une signature.

eventually isCalled(C_Login,@CKR :OK)
between isCalled(C_Logout, @CKR :OK) **and** isCalled(C_SignInit, @CKR :OK)

Une fois la propriété décrite, celle-ci est automatiquement convertie en automate qui permet de donner une représentation visuelle des événements ainsi spécifiés, telle que réprésentée sur la Figure 2.

Cet automate sert de base à deux mécanismes complémentaires.

Mesure de couverture des tests. Il est tout d'abord possible de mesurer la couverture de l'automate (et, de ce fait, de la propriété associée) en détectant, pour chaque test, les transitions de l'automate qui sont couvertes. Ce processus, illustré par la Figure 2, est implanté au sein de l'outil CertifyIt. On distingue sur l'automate un état correspondant à un état d'erreur représentant une violation de la propriété. Si un test permet d'atteindre cet état, on se retrouve en présence d'une erreur, qui peut être dûe soit à la propriété (si celle-ci est trop stricte), soit au modèle (si celui-ci est trop laxiste).

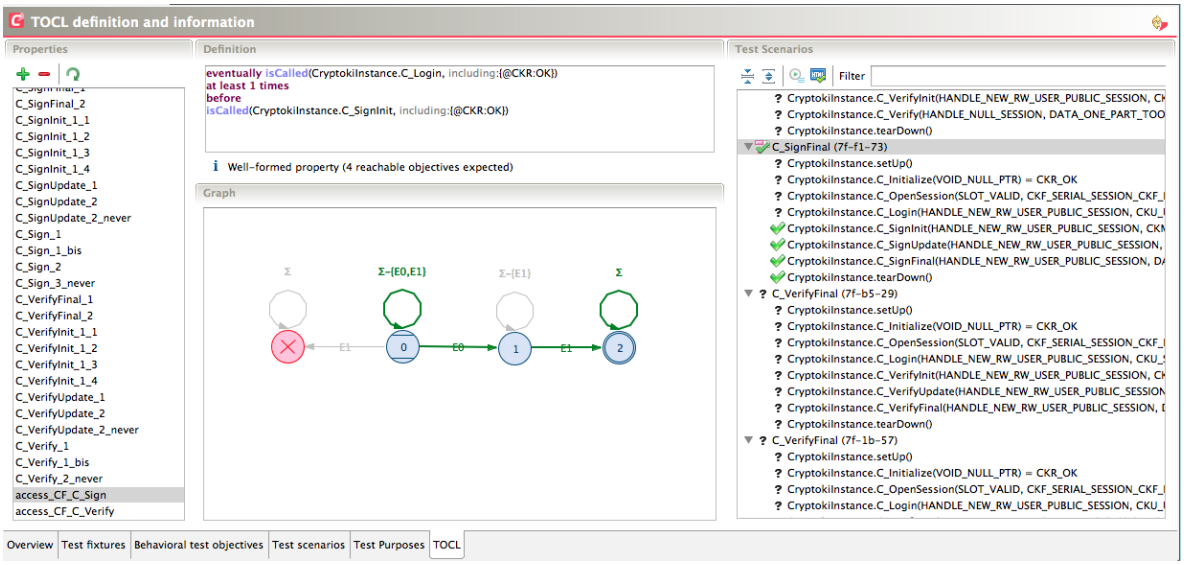


FIGURE 2 – L'interface TOCL de l'outil CertifyIt

Génération de tests complémentaires. Une fois les tests évalués, si la totalité des transitions de l'automate n'a pas été couverte, il est possible de générer des cas de tests supplémentaires dans CertifyIt. Ceci permet de venir compléter la base de tests pour exercer la propriété dans des configurations du système jusqu'alors ignorées.

2.2 Test on-line

MBeeTle est un outil de génération de tests *on-line*. La dénomination "on-line" signifie que l'outil génère et exécute les tests sur le système sous test en lui étant directement connecté. Cet outil se base sur la technologie d'animation de modèle extraite de l'outil CertifyIt. Il réutilise par ailleurs les éléments de la démarche off-line avec l'outil CertifyIt : il réutilise le modèle, la couche d'adaptation et les tests déjà généré par CertifyIt.

Le calcul des tests est similaire de l'idée du fuzzing, le moteur de MBeeTle va chercher à activer aléatoirement les opérations offertes par le système, ayant pour but de stimuler un comportement inhabituel et ainsi découvrir des vulnerabilités dans le système. Chaque pas calculé est ainsi envoyé au banc de test qui renvoie le résultat de l'exécution. La génération d'un pas de test consiste à sélectionner aléatoirement une opération, puis de valider le pas par un sélecteur. Le sélecteur valide le pas selon plusieurs stratégies : *all-steps* (chaque pas est valide), *tag-based* (en fonction de la couverture des comportements), *flower* (se base sur la reconnaissance des cas (non)-passants) ou en combinant ces stratégies unitaires dans une stratégie dite complexe.

3 Application aux composants de sécurité

Dans le contexte du partenariat avec la DGA, nous nous intéressons à des composants cryptographiques. A des fins de démonstration, nous avons expérimenté notre approche sur le standard PKCS#11 (Public Key Cryptographic Standards) [4]. Celui-ci est un standard visant la sécurité et interopérabilité des composants. PKCS#11 défini l'API Cryptoki, une interface de composants de sécurité.

Les travaux et les premières experimentations nous ont permis de tirer les conclusions suivantes.

Concernant l'utilisation des propriétés TOCL : (*i*) les propriétés TOCL permettent d'exprimer des exigences de sécurité présentes dans le document de spécification intial sans rajouter d'information superflue dans le modèle de test, (*ii*) les tests fonctionnels initialement calculés par CertifyIt ne couvrent que très partiellement ces propriétés, Les propriétés TOCL ne sont couvertes que partiellement par les tests fonctionnels, dont environ 35% de propriétés TOCL non couverts par ces tests (i.e., la situation décrite dans la propriété n'est pas exercée par les tests), (*iii*) en se basant sur les automates, CertifyIt permet de calculer en temps restreint les cibles de test, et ainsi de couvrir les propriétés TOCL à moindre effort.

Concernant l'utilisation de l'outil MBeeTle de génération de tests on-line : (*i*) la mise en oeuvre de cette approche présente un coût minimal, du fait de la réutilisation de composants de concrétisation de tests déjà existants, (*ii*) cet outil permet de couvrir de larges pans du système et d'activer des conditions limites liées à l'infrastructure qui n'étaient pas prises en compte par l'approche fonctionnelle, (*iii*) les traces produites par MBeeTle tendent à améliorer la couverture initiale des propriétés TOCL définies sur le modèle.

Finallement, nous avons constatés que les approches on-line et off-line, test fonctionnel classique, sont complémentaires. L'approache on-line peut être employée en tâche de fond pour éprouver le système de manière intensive, une fois que les premières campagnes de qualification du système sont effectuées.

4. http ://france.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm

4 Conclusion et travaux actuels

Le projet MBT_Sec propose d'industrialiser une technologie mature, initialement définie et expérimentée dans le cadre de projets ANR antérieurs : le projet ANR TASCCC (2009–2012) qui a permis la définition du langage de propriétés temporelles TOCL, et le projet OSeP (2012–2013) qui a montré l'intérêt de ces techniques pour le test de sécurité fonctionnelle des composants cryptographiques. Cette technique s'avère intéressante car elle répond à des besoins spécifiques de nos partenaires de la DGA dans le cadre de la validation de composants cryptographiques. Par ailleurs, l'approche proposée se veut plus générale et ne se limite pas à ce type d'applications. En effet, les expérimentations menées dans le cadre du projet TASCCC ont également montré l'intérêt de cette approche pour la validation de systèmes embarqués, tels que les cartes à puces. De plus, la couverture des propriétés permet de donner des garanties concernant les tests produits qui correspondent aux attentes des évaluateurs Critères Communs.

Nos efforts se focalisent actuellement sur l'implantation de critères de test de robustesse visant à exercer le système vis-à-vis en allant provoquer les événements indésirables [4]. Par ailleurs, même si les premiers résultats sont concluants en ce sens, nous souhaitons évaluer le pouvoir de détection d'erreur des deux approches proposées dans ce projet.

Références

[1] Boris Beizer. *Black-box testing - techniques for functional testing of software and systems.* Wiley, 1995.

[2] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux. A test generation solution to automate software testing. In *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, pages 45–48, New York, NY, USA, 2008. ACM.

[3] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting. A subset of precise UML for model-based testing. In *Proceedings of the 3rd International Workshop on Advances in Model Based Testing*, pages 95–104, 2007.

[4] F. Dadeau, K. Cabrera Castillos, and J. Julliand. Coverage criteria for model-based testing using property patterns. In *9th Workshop on Model-Based Testing, Satellite workshop of ETAPS 2014*, volume 141 of *EPTCS*, pages 29–43, Grenoble, France, April 2014. Open Publishing Association.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 411–420, New York, NY, USA, 1999. ACM.

[6] B. Kanso and S. Taha. Temporal constraint support for ocl. In Krzysztof Czarnecki and Gorel Hedin, editors, *Software Language Engineering*, volume 7745 of *LNCS*, pages 83–103. Springer Berlin Heidelberg, 2013.

Des réels aux flottants : préservation automatique de preuves de stabilité de Lyapunov

Olivier Hermant et Vivien Maisonneuve
MINES ParisTech, PSL Research University, France
prenom.nom@mines-paristech.fr

Abstract

Les programmes contrôle-commande doivent assurer la stabilité d'un système dynamique autour d'un point d'équilibre. De tels programmes sont généralement issus d'une modélisation du système, en partie intégrée au programme, qui calcule alors à chaque cycle la commande adéquate à envoyer au système.

Les concepteurs de tels systèmes s'appuient sur la théorie de Lyapunov pour déterminer les paramètres internes nécessaires à la stabilité. Il s'agit donc d'un cas rare où l'on dispose *en amont* d'une preuve que le système respecte la propriété voulue.

Nous nous intéressons au problème du transfert de ces preuves de la modélisation vers l'implémentation, et plus particulièrement au problème du passage des nombres réels, utilisés pour la modélisation, aux nombres flottants, qui sont utilisés dans l'implantation du programme sur microcontrôleur.

Pour cela, nous introduisons l'outil LyaFloat qui permet de vérifier à quelles conditions de précision une preuve de stabilité, donnée en tant qu'annotation dans une logique de Hoare, est préservée ou non lors du passage aux flottants.

1 Introduction

La stabilité est un attribut essentiel des systèmes de contrôle, en particulier lorsqu'ils commandent des systèmes physiques dits critiques, dont un dysfonctionnement peut engendrer des pertes humaines ou matérielles importantes. De nombreuses techniques complémentaires doivent donc être mises en œuvre pour s'en assurer.

Pour les programmes de type contrôle-commande, qui sont l'objet d'étude de l'automatique, la théorie de Lyapunov joue un rôle central. Elle permet, après modélisation de l'environnement physique dans lequel évolue le système contrôlé, de déterminer les valeurs des constantes qui permettront aux variables d'état du système de rester contenues dans une *enveloppe bornée*, autrement dit, d'assurer la *stabilité*, et ce en boucle fermée ou en boucle ouverte, c'est à dire en considérant le modèle de l'environnement ou non.

Ainsi, les concepteurs de tels contrôleurs doivent fournir tout le travail de modélisation et de preuve nécessaire. Mais ce travail s'arrête le plus souvent à un modèle du programme, généralement du code de haut niveau, de type MATLAB ou Simulink. Tout ce travail est perdu ensuite lors des étapes successives menant au code exécuté sur microcontrôleur, et doit parfois être retrouvé [2, 6] a posteriori.

Pour pallier ce problème, Eric Féron [3] propose d'utiliser une logique de Hoare [5], qui permet de propager les invariants quadratiques de Lyapunov vers du code plus bas niveau (en l'occurrence, du code `C`). Cependant, bien que les étapes (par exemple multiplication par une matrice) soient de ce fait plus atomiques, le raisonnement continue de se faire sur des nombres réels, alors que les variables et les constantes du code exécuté sont à précision finie et que les opérations arithmétiques introduisent des erreurs d'arrondi.

C'est ce problème que nous proposons d'étudier. Étant donné un programme et une preuve de sa stabilité par la théorie de Lyapunov, le programme LyaFloat que nous introduisons permet

de vérifier automatiquement, si cela est possible, que cette preuve est toujours valide lorsque l'on considère non plus les nombres réels mais les flottants. De plus, il est possible de faire varier dans LyaFloat la précision des nombres considérés, permettant ainsi de déterminer à quelles conditions de précision sur les nombres la preuve reste valide.

À la suite de Féron [3], nous considérerons donc donné un programme *numérique*, comportant uniquement des constantes, des opérations arithmétiques ou de saturation ($\max, \min$), ainsi que des assignations de variables. Ce programme est accompagné, pour chaque instruction, de deux invariants *quadratiques*, la pré- et la postcondition, donnés en logique de Hoare, qui sont supposés valides pour des nombres réels et des opérations arithmétiques exactes.

Ces invariants quadratiques proviennent de la structure particulière du code des programmes contrôle-commande et sont donnés par la théorie de Lyapunov. L'outil LyaFloat propage l'invariant d'entrée en prenant en compte les constantes altérées et les erreurs d'arrondi des nombres flottants. En fin de programme, la stabilité du système en nombres flottants est vérifiée. Elle se traduit par l'inclusion de l'ellipsoïde (domaine résultant d'un invariant quadratique borné) final dans l'ellipsoïde initial.

D'autres travaux se sont récemment intéressés au problème de la stabilité des programmes *concrets*. Citons en particulier l'approche d'Astrée [2] ainsi que celle de Pierre Roux [6]. Toutes deux s'intéressent au seul problème en boucle ouverte, et s'efforcent de reconstruire des invariants corrects en suivant une approche par interprétation abstraite (sur des domaines ellipsoïdaux). Dans le cas de [6], un ensemble de conditions générales est même donné. Notre travail adopte une démarche légèrement différente, comme il a déjà été souligné : nous proposons de réutiliser les connaissances spécifiques et les preuves de haut niveau développées par les concepteurs, ce qui nous permet entre autre de traiter les deux cas boucle ouverte et boucle fermée.

Après un survol du domaine des programmes contrôle-commande provenant de l'automatique, nous présentons la structure de la traduction que nous proposons, validée en Coq. L'outil concret LyaFloat est ensuite introduit, ainsi que son application au cas d'étude présenté par Féron [3]. Nous discutons ensuite du travail encore à réaliser. Ces travaux ont fait l'objet d'une partie de la thèse de doctorat de Vivien Maisonneuve [4].

2 Structure d'un programme contrôle-commande

Un programme contrôle-commande reçoit des entrées, provenant du système physique et d'un opérateur, et, en fonction de celles-ci et de son état interne, produit des sorties, destinées à des actuateurs, qui eux-mêmes influent sur le système, comme décrit par la figure 1.

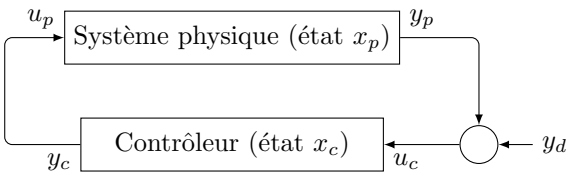


Figure 1: Schéma d'un système contrôle-commande

Le système physique est d'abord modélisé de façon continue, avec des équations différentielles. Ce modèle est ensuite linéarisé autour du point où l'on recherche la stabilité, afin de permettre la conception du contrôleur. Le contrôleur final est, quant à lui, discret ; le code lira donc ses entrées et produira ses sorties en boucle à une fréquence donnée. Il est d'autre part linéaire (avec éventuelle saturation des entrées/sorties) ; les opérations mathématiques en jeu sont donc des opérations matricielles, comme dans le code MATLAB de la figure 2, issu de [3].

```
1  Ac = [0.4990, -0.0500; 0.0100, 1.0000];
2  Bc = [1; 0];
3  Cc = [564.48, 0];
4  Dc = -1280;
5  xc = zeros(2, 1);
6  receive(y); receive(yd);
7  while (1)
8      yc = max(min(y - yd, 1), -1);
9      skip;
10     u = Cc*xc + Dc*yc;
11     xc = Ac*xc + Bc*yc;
12     send(u, 1);
13     receive(y, 2);
14     receive(yd, 3);
15     skip;
16 end
```

Figure 2: Code MATLAB d'un contrôleur masse-ressort [3]

Les étapes de modélisation du problème, de linéarisation, de conception du contrôleur ainsi que de passage du continu au discret sont l'objet d'étude de l'automatique, qui fournit ensuite des garanties sur le système obtenu, sous la forme de preuves de stabilité, sur le système en boucle fermée (représenté par l'intégralité du schéma de la figure 1) ainsi que sur le système en boucle ouverte (lorsque l'on considère le contrôleur de la figure 1 seul).

Ceci se traduit sur le code de la figure 2 en des invariants elliptiques qui sont déterminés par la théorie de Lyapunov et propagés par le code, et que l'on peut présenter en une logique de Hoare.

3 Structure et validité de la traduction

Ainsi, comme indiqué par Féron [3], chaque instruction se retrouve décorée d'une paire pré-/postcondition, la postcondition de l'instruction i étant à la fois:

- le résultat de la transformation de la précondition par i, et éventuellement de l'application de théorèmes,
- et la précondition de l'instruction suivante.

La précondition initiale est que le vecteur des variables d'état du contrôleur se trouve dans une certaine enveloppe: $x_c \in \mathcal{E}_P \Leftrightarrow x_c^{\mathrm{T}} \cdot P \cdot x_c \leq 1$, avec P une matrice définie positive. La postcondition de fin de corps de boucle est $x_c \in \mathcal{E}_R \Leftrightarrow x_c^{\mathrm{T}} \cdot R \cdot x_c \leq 1$, avec R définie positive. La stabilité des itérations, et donc du contrôleur, est alors une conséquence de $\mathcal{E}_R \subseteq \mathcal{E}_P$.

C'est ce que nous cherchons à vérifier avec une traduction en deux étapes, comme illustré dans la figure 3, où sont schématiquement représentés l'instruction i et les invariant d'entrée d et de sortie d'. Ce dernier est le résultat de l'application du théorème p à l'instruction i en supposant les variables dans le domaine d .

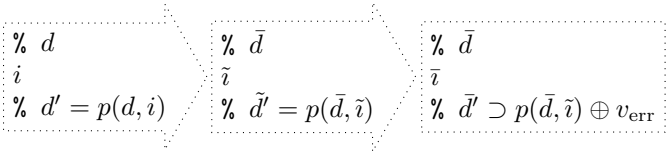


Figure 3: Schéma de la traduction

Cette traduction de l'instruction i est effectuée en deux étapes, qui ont un impact sur les pré- et postconditions : tout d'abord, nous générons $\tilde{\imath}$, où les constantes réelles sont remplacées par leur représentation flottante, mais où les opérations arithmétiques sont toujours exactes.

La précondition d a potentiellement changé, en particulier dans le cas où l'instruction i n'est pas la première instruction de la boucle. C'est pourquoi l'on condière un invariant d'entrée modifié, $\bar{d}$, qui est propagé avec le même argument p, appliqué maintenant à l'instruction $\tilde{\imath}$.

La seconde étape de la traduction remplace les opérateurs arithmétiques exacts de $\tilde{\imath}$ par leurs représentations en flottant pour obtenir $\bar{\imath}$. Malheureusement, les résultats de la théorie de Lyapunov ne s'appliquent qu'aux opérations exactes (par exemple, l'addition flottante n'est pas associative). Il est ainsi impossible d'appliquer directement p à l'instruction $\bar{\imath}$ résultante. Aussi, nous conservons l'invariant de l'étape précédente $p(\bar{d}, \tilde{\imath})$, que nous élargissons en rajoutant l'erreur maximale obtenue par des opérations arithmétiques v_{err}, qui dépend de $\tilde{\imath}$ et de $\bar{d}$. Cette erreur est fonction des bornes sur les opérandes (connues grâce à $\bar{d}$).

Comme nous chercherons, en pratique, à conserver une structure d'ellipsoïde pour les invariants, de manière à rester dans le cadre de la théorie de Lyapunov, nous considérons en réalité une surapproximation $\bar{d}'$ de cet invariant.

Cette approche théorique a été formalisée en Coq, dans le cas des opérateurs binaires. L'objet primitif de cette formalisation sont les domaines, des sous-ensembles de réels (cas abstrait) ou de flottants (cas concret), avec des conditions de cohérence issues de l'interprétation abstraite entre les deux. De cette manière, nous restons le plus générique possible. Les théorèmes sont des transformateurs de domaine qui sont supposés valides sur n'importe quel domaine, y compris non ellipsoïdal. En effet il suffit, dans ce dernier cas, de renvoyer l'ensemble $\mathbb{R}$ entier, ce qui équivaut à une perte totale d'information. Pour plus de détails, voir [4], chapitre 5.

4 LyaFloat

LyaFloat est une implantation concrète de l'architecture présentée figure 3. Il s'agit d'un programme Python, qui s'appuie sur les deux librairies SymPy, pour manipuler les calculs symboliques, et Mpmath, pour l'arithmétique flottante en précision arbitraire. Le calcul de l'erreur maximale v_{err}, en fonction des bornes sur les valeurs des variables, est défini par la norme IEEE 754.

Comme mentionné dans la section. 3, nous choisissons de surapproximer la postcondition obtenue, afin de conserver la forme ellipsoïdale de l'invariant. Comme il n'y a pas de critère unique de minimalité pour une ellipsoïde devant contenir un certain domaine, un choix heuristique est fait : voir [4] pour plus de détails.

En boucle ouverte, la ligne 11 de l'exemple de la figure 2 se traduit par la figure 4. La précondition de cette ligne est l'invariant suivant, où `Zc` est le vecteur à trois dimensions composé de `xc` et de `yc`, P la matrice de ℓ_P et μ un paramètre réel déterminé par la théorie de Lyapunov (cf. [3, 4]):

$$\texttt{Zc} \in \mathcal{E}_{Q_\mu}, Q_\mu = \left(\begin{smallmatrix} \mu P & 0_{2\times 1} \\ 0_{1\times 2} & 1-\mu \end{smallmatrix}\right), u^2 \leq (C_c\ D_c) \cdot Q_\mu^{-1} \cdot (C_c\ D_c)^{-1}$$

On vérifie finalement l'inclusion de l'ellipsoïde élargie dans $\mathcal{E}_P$, ce qui est possible dans le cas de flottants sur 64 bits. Il a aussi été possible de vérifier la stabilité d'exemples tirés de la littérature, notamment de [6].

Comme il est d'autre part possible de régler la précision des nombres flottants (grâce à la primitive `setfloatify`), on s'aperçoit que le système de la figure 2 reste stable jusqu'à une précision de 17 bits pour les nombres flottants, ce qui permettrait par exemple d'opter pour un microcontrôleur manipulant des flottants moins précis.

Dans le cas où nous choisissons une précision inférieure, nous ne sommes plus en mesure de conclure : soit le système est réellement instable, soit trop d'imprécision a été introduit, notamment dans le calcul de l'ellipsoïde englobante $\bar{d}' \subseteq p(\bar{d}, \tilde{\imath})$ ou dans le calcul de l'erreur arithmétique maximale v_{err}.

Il est intéressant de noter que nous avons aussi été en mesure de traiter également le cas de la boucle fermée, ce qui va au delà des approches par interprétation abstraite [3, 6]. Il s'avère

```
from lyafloat import *
# Parameters
setfloatify(constants=True, operators=True, precision=53)

# Definition of $\EllP$
P = Rational("1e-3") * Matrix(rationals(["0.6742 0.0428", "0.0428 2.4651"]))
EP = Ellipsoid(P)

# Definition of $\EllQmu$
mu = Rational("0.9991")
Qmu = mu * P
Qmu = Qmu.col_insert(2, zeros(2, 1)).row_insert(2, zeros(1, 3))
Qmu[2,2] = 1 - mu
EQmu = Ellipsoid(Qmu)

# Symbols
xc1, xc2, yc = symbols("xc1 xc2 yc")
Xc = Matrix([[xc1], [xc2]])
Yc = Matrix([[yc]])
Zc = Matrix([[xc1], [xc2], [yc]])

# Constant matrices
Ac = Matrix(constants(["0.4990 -0.0500", "0.0100 1.0000"]))
Bc = Matrix(constants(["1", "0"]))
Cc = Matrix(constants(["564.48 0"]))
Dc = Matrix(constants(["-1280"]))

# Definition and verification of $\EllR$
AcBc = Ac.col_insert(Ac.cols, Bc)
R = (AcBc * Qmu.inv() * AcBc.T).inv()
ER = Ellipsoid(R)
print("ER included in EP :", ER <= EP)

# Computation and verification of $\widefp\EllR$
i = Instruction({Xc: Ac * Xc + Bc * Yc},
       pre=[Zc in EQmu], post=[Xc in ER])
ERbar = i.post()[Xc]
print("ERbar =", ERbar)
print("ERbar included in EP :", ERbar <= EP)
```

Figure 4: Vérification de stablité en boucle ouverte avec LyaFloat

impossible de conclure à la stabilité dans le cas de flottants sur 64 bits ; il nous faut pour ce faire passer à une précision d'au moins 117 bits, par exemple des flottants en quadruple précision. Notons que le modèle du système physique continue à avoir des constantes et opérations réelles, car il ne correspond à aucun code concret. Ainsi les étapes de traduction ne se font que sur le contrôleur.

5 Conclusion

De nombreuses améliorations peuvent encore être apportées à l'outil LyaFloat d'analyse semi-automatique de contrôleurs de systèmes présenté dans cet article. Tout d'abord, pour atteindre un plus grand degré de confiance dans la preuve de stabilité, nous pourrions soit développer et prouver correct notre vérificateur en Coq, soit demander à LyaFloat de générer un script de preuve Coq démontrant la stabilité en flottants, par exemple en utilisant la librairie Flocq[1]. C'est cette seconde approche que nous privilégierons en premier lieu.

Des cas d'études à plus grande échelle sont aussi envisagés, en collaboration avec le Centre automatique et systèmes à MINES ParisTech. Afin d'en faire un outil facilement utilisable par les automaticiens, il faudra aussi développer une interfaçe homme-machine moins bas niveau.

References

[1] Sylvie Boldo and Guillaume Melquiond. Flocq, 2010.

[2] Patrick Cousot, Radhia Cousot, Jérôme Feret, Antoine Miné, and Xavier Rival. The astrée static analyzer, 2013.

[3] Eric Feron. From control systems to control software. *IEEE Control Systems Magazine*, 30(6):50–71, December 2010.

[4] Vivien Maisonneuve. *Static Analysis of Control-Command Systems – Floating-Point and Integer Invariants*. PhD thesis, MINES ParisTech, 2015.

[5] Doron Peled. *Software reliability methods*. Springer, 2001.

[6] Pierre Roux. *Analyse statique de système contrôle commande: synthèse d'invariants non linéaires*. PhD thesis, Université de Toulouse, 2013.

Proving soundness of a procedure for verifying RL Formulas in Coq

Andrei Arusoaie

INRIA Lille-Nord Europe

Abstract. In this paper we give a brief description of some ongoing work on proving the soundness of a procedure for verifying Reachability Logic properties in Coq. The procedure has been introduced in our previous work and it has been proved sound. Reachability Logic is a formalism that can be used to state properties about program executions. The procedure can be seen as an implementation of a strategy for applying the Reachability Logic's proof system. Our objective is to implement this procedure in a way that it can generate certificates for proofs. To this aim, we encode the procedure in Coq and we prove its soundness. We report here the progress that we have made and the main difficulties that arise when dealing with such a proof.

1 Introduction

Reachability Logic [5–7, 3] (RL) is a formalism which can be used for stating properties about program execution and for specifying operational semantics of programming languages. It mainly consists of a language-independent proof system which has been shown sound and relatively complete. This proof system includes a special deduction rule which allows the use of a RL formula in its own proof. RL formulas are pairs of the form $\varphi \Rightarrow \varphi'$, where φ, φ' are called *patterns* and they represent a set of states. A RL formula denotes a *reachability relation* between two such sets of states. A set of RL formulas generates a transition system over states, where two states are in the transition relation if the sets of states they belong to are in a reachability relation. Depending on the interpretation of a formula, the reachability relation can express programming-language semantic steps, or properties over programs in the language in question.

In previous work [1], we proposed a procedure (Figure 1) for verification of RL properties and we proved it sound. The procedure `prove` can be regarded as an implementation of a certain strategy for applying the rules of the proof system of RL . Its benefit is that it allows us to mechanise the proving process. Our objective is to implement `prove` in a way that it can be used, not only for verification of RL formulas, but also for generating certificates. To this aim, we have chosen the Coq system to both implement our procedure and generate certificates. This system fits our requirements because the implementation of `prove` in Coq can be extracted as certified OCaml code (which can be further used for verification) and also because it is able to generate certificates from proofs. For the latter, we also need to encode in Coq the soundness of our procedure.

procedure `prove`$(\mathcal{S}, G_0, G)$

1: **if** $G = \emptyset$ **then return** `success`
2: **else choose** $\varphi \Rightarrow \varphi' \in G$
3: **if** $M \models \varphi \rightarrow \varphi'$ **then return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\})$
4: **else if** there is $\varphi_c \Rightarrow \varphi'_c \in G_0$ s. t. $M \models \varphi \implies (\exists \mathit{FreeVars}(\varphi_c))\varphi_c$ **then**
5: **return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \Delta_{\varphi_c \Rightarrow \varphi'_c}(\varphi \Rightarrow \varphi'))$
6: **else if** φ is $\mathcal{S}$-derivable **then**
7: **return** `prove`$(\mathcal{S}, G_0, G \setminus \{\varphi \Rightarrow \varphi'\} \cup \Delta_{\mathcal{S}}(\varphi \Rightarrow \varphi'))$
8: **else return** `failure`.

Fig. 1. RL verification procedure.

2 Proving soundness in Coq

The soundness result for our procedure is stated and proved in [1]. It essentially says that if `prove` returns `success` then the transition system generated by the input $\mathcal{S}$ (a set of RL formulas, usually representing an operational semantics) satisfies all the goals in G_0 (a set of RL formulas representing the goals to be proved). A charateristic feature of RL is that the initial goals G_0 can be used in the proof of other goals or in their own proofs. The only restriction is to do at least one deduction step using an RL formula from $\mathcal{S}$ before applying goals from G_0. Thus, when calling `prove` we initially set the third argument, namely G, to be the set of goals derived from G_0 using $\mathcal{S}$. When running, the procedure chooses a goal $\varphi \Rightarrow \varphi'$ from the current set of goals G and either eliminates it completely from G, or replaces it with a set of new goals under some conditions. If none of these conditions holds then `failure` is returned.

In order to prove the soundness in Coq we have to face a series of challenges raised by the procedure itself.

First, the procedure might not terminate. For instance, this might happen when G_0 contains a single goal which cannot be used in its own proof and the procedure keeps applying formulas from $\mathcal{S}$. Since Coq does not accept functions that do not terminate, we have to introduce a new parameter which is intended to limit the number of recursive calls.

Second, the proof in [1] is based on an additional construction which stores all the intermediate information generated by the recursive calls of `prove`. However, in practice, storing all that information might be very inefficient, especially if we take into account that fact that we need this information only for soundness. Thus, we have to find a way to handle that intermediate information in Coq such that it is used only when proving soundness.

Another issue here is that the intermediate information is said to be generated by each recursive call of `prove`. However, there is no explicit (formally defined) link between each recursive call and the corresponding generated information. Thus, we have to add the missing parts to the proof before encoding it into Coq.

Finally, a more general issue is that the formal background needed to encode `prove` in Coq is quite large. Also, some of the lemmas needed by the proof are very complex and have to be broken into several pieces to be amenable for the Coq proof.

Future work Currently, we introduced all the needed definitions in Coq and we adapted and stated our lemmas in Coq. Moreover, we adapted the proofs of the lemmas such that they can be easily encoded and we are in the process of proving them in Coq. After proving the soundness, we want to integrate our procedure into a tool, such that we can use the existing operational semantics of real-life programming languages (e.g. [4],[2], ...) for verifying programs written in those languages. This will also require some translation of the semantics into Coq. In the end, we aim that this tool will accept specifications of program properties (in the form of RL formulas) as input, and it will output the result of the verification, accompanied by a certificate (in case it ended successfully).

References

1. Andrei Arusoaie, Dorel Lucanu, David Nowak, and Vlad Rusu. Verifying Reachability-Logic Properties on Rewriting-Logic Specifications (Extended Version). Technical Report TR 15-01, "Al.I.Cuza" University of Iaşi, Faculty of Computer Science, 2015. http://www.infoiasi.ro/ tr/tr.pl.cgi.
2. Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015.
3. Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.
4. Chucky Ellison and Grigore Roşu. An executable formal semantics of C with applications. In *Proceedings of the 39th Symposium on Principles of Programming Languages (POPL'12)*, pages 533–544. ACM, 2012.
5. Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *Proceedings of the 28th Symposium on Logic in Computer Science (LICS'13)*, pages 358–367. IEEE, June 2013.
6. Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the 27th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'12)*, pages 555–574. ACM, 2012.
7. Grigore Rosu and Andrei Stefanescu. From hoare logic to matching logic reachability. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)*, volume 7436 of *Lecture Notes in Computer Science*, pages 387–402. Springer, 2012.

Configuration en langue naturelle du fonctionnement d'une maison intelligente

D. Sadoun[1] C. Dubois[2] Y. Ghamri-Doudane[3] B. Grau[4]

1 : LIMSI, Driss.Sadoun@limsi.fr.

2 : ENSIIE, CEDRIC, catherine.dubois@ensiie.fr.

3 : Univ. de La Rochelle, L3i, yacine.ghamri@univ-lr.fr.

4 : LIMSI, ENSIIE, bg@limsi.fr.

Résumé

Cet article décrit les résultats obtenus dans le projet Envie Verte. L'objectif est de permettre à un utilisateur de configurer son environnement intelligent en décrivant en langage naturel les règles de comportement de l'environnement. Notre approche consiste à traduire ces spécifications en spécifications formelles en passant par une représentation ontologique. Des vérifications sont faites sur les deux modèles.

1 Introduction

« C'est si pratique, tout communique », disait Mme Arpel [1], alors qu'elle montrait sa villa aux multiples gadgets technologiques - à l'utilité discutable - à sa voisine. Depuis l'illustration donnée par Jacques Tati en 1958, la domotique, qui regroupe l'ensemble des technologies de l'électronique, de la physique du bâtiment, de l'informatique et des communications au service de la maison, a fait bien des progrès. Une fois le système domotique choisi et installé dans une maison, reste à déterminer la configuration adaptée aux besoins et aux envies des utilisateurs, soient les personnes qui vivent dans la maison. Dans le cadre du projet Envie Verte [2], terminé à ce jour, notre objectif était de permettre à un utilisateur de configurer son propre environnement intelligent simplement en décrivant ses besoins, i.e. les règles de comportement de l'environnement, en langage naturel (LN). Cela suppose de pouvoir produire un retour à l'utilisateur sur la qualité des exigences qu'il aura rédigées : complétude, cohérence, conformité aux contraintes du domaine. Dans cet article, nous présentons les contributions présentées dans la thèse du premier auteur [6], principaux résultats du projet Envie Verte. Ces travaux se placent dans la lignée de travaux plus généraux concernant le passage automatique (ou semi-automatique) de spécifications rédigées en langue naturelle vers des spécifications formelles [2, 4, 5] (pour d'autres références consulter [6]).

2 Approche générale

Pour l'utilisateur des services domotiques qui décrira des règles de fonctionnement, définir ses besoins en LN reste le moyen le plus efficace. Un exemple [3] d'une telle spécification est le suivant : *When I enter a room the door opens automatically. If a movement is detected in the bedroom, turn on the light. Each time someone is detected in a room shut its doors.* Pour pouvoir les vérifier, nous proposons de traduire automatiquement ces spécifications écrites en LN en spécifications formelles. Outre les ambiguïtés propre au langage naturel, se pose le problème de l'implicite qui requiert des connaissances extérieures aux textes pour comprendre les spécifications et les traduire

1. Réplique extraite du film *Mon Oncle* de Jacques Tati (1958)
2. financé par DIGITEO, projet DIM LSC 2010
3. Les textes considérés sont en anglais, cependant l'approche est indépendante de la langue.

dans un langage formel. L'ampleur du fossé entre spécifications LN et spécifications formelles peut se combler en passant par une représentation intermédiaire. Différents formalismes ont été proposés pour jouer le rôle de pivot dans la transformation. Les plus couramment utilisés sont les langages naturels contrôlés comme ACE ou SBVR, ou encore le langage semi-formel UML. Nous avons choisi d'utiliser une ontologie OWL-DL comme modèle pivot. Une telle ontologie permet de représenter des connaissances mais aussi de faire des inférences. L'utilisation d'une ontologie comme représentation intermédiaire dans un processus de formalisation automatique de spécifications LN n'avait pas encore été explorée à notre connaissance.

Notre approche consiste à définir au préalable une ontologie (voir Fig. 1) qui décrit le comportement général d'une maison intelligente, ou plus précisément de son système de capteurs, d'actionneurs etc. Cette ontologie est élaborée par un expert du domaine ; elle est utilisée et instanciée pour comprendre les spécifications de l'utilisateur. Chaque texte est donc analysé en utilisant l'ontologie initiale afin de la peupler, i.e. instancier des concepts et des propriétés de l'ontologie. Ensuite, des vérifications sont faites sur cette ontologie instanciée grâce au mécanisme d'inférence de OWL. On peut alors faire un premier diagnostic sur le fonctionnement spécifié et se retourner vers l'utilisateur si besoin. Enfin les informations présentes dans l'ontologie instanciée sont traduites en une spécification Maude [3] qui est à son tour animée et analysée pour détecter d'autres problèmes. Dans la suite, nous présentons rapidement les différentes étapes (voir [6] et [7, 8] pour plus de détails).

3 Du texte à l'ontologie

Ontologie de domaine

Définition d'une ontologie. Une ontologie OWL [1] est composée de concepts (ou classes), de propriétés et d'instances. Un concept désigne un groupe d'individus qui partagent les mêmes caractéristiques et sont organisés hiérarchiquement selon une taxonomie. Il existe en OWL deux types de propriétés : (i) propriété de type relation entre deux individus d'une classe ou de plusieurs classes, et (ii) une propriété de type attribut entre une valeur ou donnée et un individu d'une classe, Enfin une instance est un individu d'un concept qui peut prendre les caractéristiques définies par les propriétés. OWL s'appuie sur une logique de description (fragment de la logique des prédicats) et permet de faire des raisonnements sur une ontologie. Ainsi on peut vérifier la cohérence des données d'une ontologie, déduire des connaissances nouvelles et extraire des informations.

Contenu de l'ontologie. On y distingue deux types de concepts (cf. figure 1), *Composant* et *Type* qui héritent du concept racine Thing. Les individus de *Composant* sont les entités du système décrit : capteurs, actionneurs etc. Ils partagent des propriétés qui les distinguent des autres composants et possèdent des propriétés qui les distinguent les uns des autres, Les individus du concept *Type* sont analogues à des types (mouvement, température etc). Ils étendent les types simples prédéfinis OWL pour caractériser les composants du système.
Les propriétés de type attribut de l'ontologie décrivent les caractéristiques des composants. Ces attributs sont définitionnels (par exemple Fixed-in qui permet d'indiquer où un composant est installé) ou évolutifs (comme Status qui permet d'indiquer si un dispositif est activée ou non).

Conceptualisation des règles de comportement. Le comportement du système est décrit à l'aide de règles de la forme $P_1(i_j, i_x) \wedge ... \wedge P_m(i_y, i_n) \Rightarrow P_k(i_j, i_y) \wedge ... \wedge P_z(i_x, i_n)$ avec P_k un prédicat binaire correspondant à une propriété et i_x un individu, une valeur ou une variable. Dans l'ontologie de domaine, à ce stade, sont représentées les règles qui décrivent le comportement générique du système. Celles-ci, issues de la connaissance du domaine, sont indépendantes des spécifications de l'utilisateur. Ce sont plus précisément des patrons de règles.

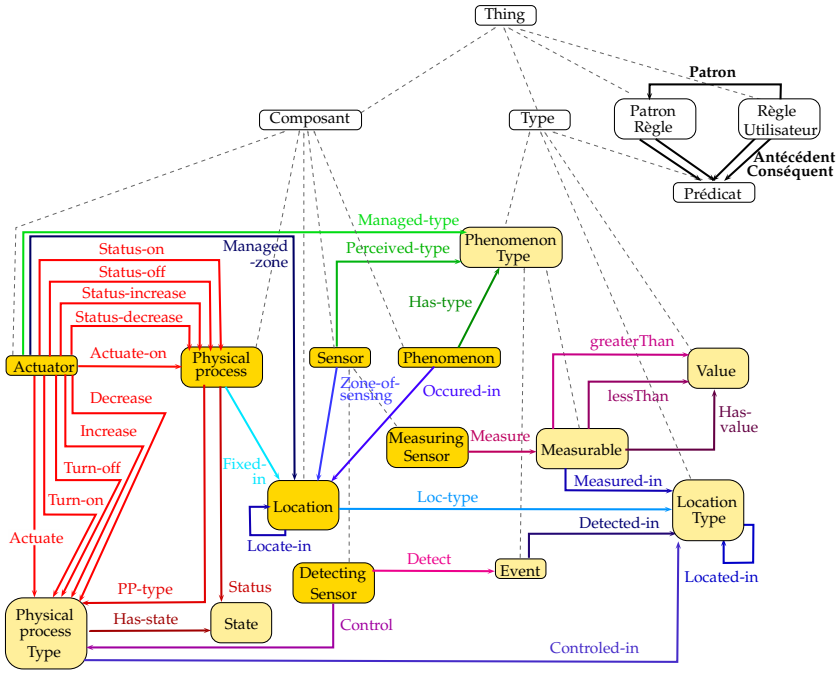


FIGURE 1 – Ontologie du comportement d'une maison intelligente

Patron-Règle
Detected-in(t,l)
Controlled-in(p,l)
Has-type(?ph,t)
Occurred-in(?ph, ?loc)
Perceived-type(?s,t)
Zone-of-sensing(?s, ?loc)
Managed-type(?a,t)
Managed-zone(?a, ?loc)
Loc-type(?loc,l)
→ Actuate(?a,p)

Spécialisation

Règle-Utilisateur
Detected-in(movement-in,room)
Controlled-in(door,room)
Has-type(?ph,movement-in)
Occurred-in(?ph, ?loc)
Perceived-type(?s,movement-in)
Zone-of-sensing(?s, ?loc)
Managed-type(?a,movement-in)
Managed-zone(?a, ?loc)
Loc-type(?loc,room)
→ Turn-on(?a,door)

FIGURE 2 – Règles de comportement

Peuplement de l'ontologie

Les deux étapes décrites dans cette partie et dans la suivante sont spécifiques à une spécification utilisateur. Ainsi, à la suite de l'analyse syntaxique de chaque phrase du texte, des instances des concepts figurant dans l'ontologie sont extraites et ajoutées dans l'ontologie si ces instances vérifient les propriétés énoncées dans l'ontologie. Des patrons de règles guident la formation de règles spécifiques, qui viennent également peupler l'ontologie. La figure 2 montre un patron de règle et son instanciation produite par la phrase « When I enter a room the door opens automatically ». Les éléments à droite en couleur sont les instances identifiées dans le texte. Les éléments précédés d'un point d'interrogation sont des variables. Dans cette étape, les propriétés jouent un rôle crucial. En effet, l'analyse du texte en LN consiste d'une part à repérer les propriétés par des enchainements syntaxiques spécifiques à ces propriétés et d'autre part à utiliser les propriétés en question pour identifier des termes et les concepts qu'ils dénotent. Il y a ici un apprentissage progressif. Nous ne décrivons pas plus longuement le processus de peuplement qui relève de techniques de TAL.

Vérifications

Une première série de vérifications peut se faire sur l'ontologie peuplée. Elles concernent l'applicabilité et la cohérence des règles utilisateur. Vérifier l'applicabilité d'une règle revient à vérifier

que chaque prédicat impliqué dans la règle peut être satisfait, i.e. qu'il existe au sein de l'ontologie au moins une instance de la classe décrite par le prédicat. Une fois l'applicabilité de la règle de comportement vérifiée, un raisonnement sur l'ontologie est effectué afin de vérifier que les instances participant à cette règle ne génèrent pas d'incohérences. Enfin la cohérence de la règle par rapport aux autres règles de l'ontologie doit être vérifiée. Les règles ayant une même précondition mais des conclusions contradictoires peuvent être détectées par OWL. En revanche, deux règles ayant des préconditions différentes mais qui peuvent s'appliquer simultanément ne sont pas vérifiables sous OWL. En effet, il est nécessaire ici d'explorer l'ensemble des états du système, ce que ne peut faire OWL. Le passage à Maude dans la suite du processus permettra de prendre en considération ce type de vérification.

4 De l'ontologie à la spécification Maude

Il s'agit ici de dériver du modèle OWL les spécifications Maude suffisantes à la vérification du comportement dynamique du système modélisé. Ce comportement est défini majoritairement par les règles utilisateur identifiées à partir de l'analyse des spécifications d'exigences.

Traduction automatique vers Maude

En Maude, l'espace d'états d'un système est représenté par une spécification équationnelle (Σ, $\mathcal{E}$), où Σ est une signature et $\mathcal{E}$ est l'ensemble des axiomes équationnels. La dynamique du système est décrite à l'aide de règles de réécriture. Ces règles décrivent les transitions simultanées possibles dans le système. Nous avons choisi de suivre la présentation objet de Maude qui permet de décrire aisément des systèmes d'objets concurrents. Cette syntaxe nous permet d'obtenir une traduction des éléments ontologiques vers des spécifications formelles, en identifiant de manière assez naturelle certains concepts syntaxiques, comme par exemple la classe OWL et la classe Maude. Le tableau 3 illustre les différentes correspondances relatives à notre contexte d'utilisation. L'état d'un système, appelé configuration, est un multi-ensemble d'objets et de messages. Les algorithmes de traduction sont détaillés dans [6] et [8].

Vérifications avec Maude

Le mécanisme de réécriture permet d'animer une spécification et de vérifier certaines propriétés comme par exemple, l'atteignabilité ou la non-atteignabilité de certains états. Il s'agit ici principalement de vérifier quelques invariants en utilisant la commande *search* de Maude. L'utilisation de cette commande demande la donnée d'une configuration sur laquelle seront recherchées certaines propriétés. A ce stade, les configurations à analyser sont choisies manuellement. Par exemple, on cherchera à vérifier s'il existe une configuration dans laquelle certains messages sont créés : un *actionneur* pourra-t-il à un moment envoyer un message *allumer* à un *appareil* ? La vérification proposée par la commande *search* a l'avantage d'explorer le modèle de manière complètement automatique. Lorsqu'une erreur est trouvée, un contre-exemple est produit. Un autre usage intéressant de la commande *search* porte sur la validation de règles de comportement. Cette validation se fait par l'animation du modèle.

5 Conclusion

L'approche que nous avons proposée s'appuie sur l'utilisation d'une ontologie pivot entre les spécifications en langue naturelle et les spécifications formelles. Ce modèle intermédiaire permet déjà de faire des vérifications formelles. A chaque étape du processus sont gardées des informations qui font le lien entre le texte et les informations stockées dans les modèles ontologique et formel. Ainsi toute erreur découverte sous OWL ou sous Maude peut être retournée à l'utilisateur en lui indiquant la portion de texte qui a permis de mettre à jour l'erreur.

Ontologie OWL	Module orienté-objet Maude
Individu du concept *Composant*	Objet
Individu du concept *Type*	Valeur d'attribut
Concept *Composant*	Classe
Concept *Type*	Sorte *Oid*
Propriété (relation)	Message
Propriété (attribut)	Attribut
Instance de *Règle-Utilisateur*	Règle de réécriture

FIGURE 3 – Correspondance entre l'ontologie et Maude

Pour expérimenter notre approche, nous avons mis en place une plate-forme d'acquisition de textes. Cette interface décrit de manière graphique et textuelle la configuration physique de l'environnement à configurer ainsi que les différents dispositifs qui y sont intégrés. L'utilisateur peut alors spécifier ses besoins en langage naturel et les enregistrer lorsque ceux-ci lui conviennent. Nous avons ainsi récupéré plus de 100 phrases.

L'approche pivot mise en place a été conçue de manière à résoudre des formulations ambiguës ou implicites, s'adapter à différents langages dotés d'analyseurs syntaxiques et s'adapter à différents domaines d'application. Une des perspectives est de transporter cette approche sur un nouveau domaine d'application, l'effort à fournir étant en particulier la conception de l'ontologie de domaine.

Références

[1] G. Antoniou, F. V. Harmelen, and F. V. Harmelen. Web ontology language : Owl. In *Handbook on Ontologies in Information Systems*, pages 67–92, 2003.

[2] B. R. Bryant, D. Johnson, and B. Edupuganty. Formal specification of natural language syntax using two-level grammar. COLING '86, pages 527–532, Stroudsburg, PA, USA, 1986. Association for Computational Linguistics.

[3] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude : specification and programming in rewriting logic. *Theor. Comput. Sci.*, 285(2) :187–243, 2002.

[4] A.-J. Fougères and P. Trigano. Rédaction de spécifications formelles : élaboration à partir des spécifications écrites en langage naturel. *In Cognito - Cahiers Romans de Sciences Cognitives*, 1(8) :29–36, 1997.

[5] A. Guissé, F. Lévy, and A. Nazarenko. From regulatory texts to BRMS : how to guide the acquisition of business rules ? In *RuleML 2012, Montpellier, France*, volume 7438 of *LNCS*, pages 77–91. Springer, 2012.

[6] D. Sadoun. *Des spécifications en langage naturel aux spécifications formelles via une ontologie comme modèle pivot.* PhD thesis, Univ. Paris 11, juin 2014.

[7] D. Sadoun, C. Dubois, Y. Ghamri-Doudane, and B. Grau. From natural language requirements to formal specification using an ontology. In *IEEE 25th International Conference on Tools with Artificial Intelligence, Herndon, VA, USA*, pages 755–760. IEEE Computer Society, 2013.

[8] D. Sadoun, C. Dubois, Y. Ghamri-Doudane, and B. Grau. Formal rule representation and verification from natural language requirements using an ontology. In *RuleML 2014, Prague, Czech Republic*, pages 226–235, 2014.

Évaluation de l'impact de fautes matérielles sur le logiciel par Model Checking

Didier Bassole[1], Jean-Louis Lanet[2], and Axel Legay[2]

[1] Université de Ouagadougou
Burkina Faso
dbassole@univ-ouaga.bf
[2] INRIA-RBA, LHS-PEC,
Campus de Beaulieu, 263 Avenue Général Leclerc,
35042 Rennes
jean-louis.lanet@inria.fr

Abstract. La mutation de programme dû à un défaut d'environnement est susceptible de générer des problèmes de sécurité comme la non exécution de procédure de vérification, le saut de bloc d'instruction `etc.`. Pour les éradiquer il est nécessaire de comprendre les effets de ces perturbations sur le logiciel. Souvent les travaux explorent les effets au niveau d'un langage de haut niveau sans prendre en compte les effets des architectures matérielles sous-jacentes. Nous proposons de travailler au niveau du langage binaire en prenant en compte l'architecture matérielle. Nous modélisons à la fois l'unité centrale et le programme et nous définissons une transformation de l'état par mutation en fonction du modèle de faute défini.

Keywords: Faute matérielle, vérification, carte à puce

1 Introduction

L'injection de faute est une technique utilisée de longue date par la communauté de la sûreté de fonctionnement en particulier pour étudier les mécanismes de tolérance aux fautes et de recouvrement d'erreur. Les attaques par injection de fautes exploitent les perturbations qui peuvent être provoquées durant l'exécution d'un programme. Ces perturbations peuvent être provoquées sur un appareil en utilisant des moyens physiques, tels que la variation de la tension d'alimentation , la modification de la fréquence de l'horloge, des différents fronts des signaux, ou encore un éclairage intensif du circuit. Les résultats erronés peuvent, dans certains cas, être exploités afin d'obtenir des informations sur le fonctionnement du système ou même des données manipulées. Les attaques par injection sont devenues en quelques années une véritable menace pour les systèmes embarqués. Lors de la conception des composant des contre mesures sont systématiquement intégrées afin de détecter cette menace. De plus, lors des évaluations par les CESTI ces attaques sont évaluées sur les produits afin d'évaluer leur résistance.

Un point important est l'étude de l'impact d'une faute dans le système. En particulier il est important de s'assurer que l'évaluation de la faute correspond bien à l'effet physique. Il s'agit de la précision du modèle de la faute quelle soit transitoire ou permanente. Il est largement admis qu'une faute physique du matériel peut être émulée par une solution logicielle d'injection de faute.

L'impact de la faute sur le logiciel lui même est plus difficile à cerner et peut avoir des conséquences très diverses allant de l'absence d'effet jusqu'à des problèmes de sécurité qu'un attaquant pourrait souhaiter exploiter. Si l'effet d'une faute physique sur la matériel a été longuement étudié, les conséquences sur le logiciel sont souvent plus difficile à évaluer. Nous proposons dans cette thèse de transformer un programme au niveau binaire afin de pouvoir prendre en compte les aspects architecturaux vers une représentation permettant une vérification par modèle et un module de transformation de l'état du système représentatif du modèle de faute choisi.

Le reste de ce papier est organisé de la façon suivante dans une première section nous présentons la problématique des attaques par perturbation sur le matériel. dans une seconde section nous justifions le choix de travailler au niveau du langage assembleur. Dans une dernière section nous présentons les principes de résolution que nous souhaitons mettre en avant dans cette thèse.

2 Fautes physique sur le matériel

Les premiers travaux dans l'étude de l'injection de faute ont été publiés en 1996 par [2] qui ont étudié l'utilisation des perturbations aux schémas de signature à clé publique et aux schémas d'authentification. Une extension de leurs travaux pour les crypto-systèmes à clé privée a été publiée en 1997 par [1]. Plusieurs techniques permettent en perturbant l'environnement du système les plus efficaces sont actuellement les attaques par impulsion, par laser et par impulsion électromagnétiques.

2.1 Attaque en faute

Les attaques par impulsion sont destinées à modifier les trois signaux pouvant être sous le contrôle de l'attaquant: l'alimentation, l'horloge et le reset. Des lecteurs spécifique comme le MP300 de Microprossent spécifiquement dédiés à modifier tous les paramètres de ces signaux. Une variation brève de l'alimentation peut provoquer un basculement de l'état de certaines portes dans le circuit électronique. Une réduction momentanée la période d'un cycle de la fréquence d'horloge peut avoir comme effet une mauvaise lecture des données ou bien le passage à l'instruction suivante.

Les attaques par illumination laser cherchent à injecter de l'énergie dans le silicium qui se transforme alors en courant électrique apte à faire basculer là aussi des portes dans le circuit. Skorobogatov et Anderson [**?**] mentionnent pour la première fois les attaques par illumination. Généralement afin d'éviter les couches de protection métallique ces attaque se font par l'arrière du circuit et

demandent une ouverture chimique ou physique du circuit afin de se rapprocher du silicium.

Dans le premier article étudiant les attaques électromagnétiques [6], Quisquater et Samyde montrent influence d'une sonde électromagnétique sur le fonctionnement d'un circuit. Ce type d'attaque Ce type d'attaque ne nécessite pas de modifier le circuit, il est moins onéreux à mettre en place et dispose de capacités de modification du système assez proche des capacités d'un laser.

2.2 Effet de la faute

Les effets des fautes sont assez bien compris au niveau du silicium, son effet sur le logiciel est assez varié. Les registres, les bascules, les mémoires, les bus de données ou d'adresses, les caches sont susceptibles d'être modifiés. Les mémoires RAM et EEPROM peuvent être perturbées et leur contenu modifié. La perturbation des *latches* de bus peuvent aussi entrainer une modification des données de la mémoire ROM (le programme) lors de leur transfert. Il est important de noter que les perturbations affectent à la fois les données et les programmes. Le pipeline peut aussi être affecté avec des effets multiples sur les instructions en cours de traitement.

2.3 Modèle de faute

Les effets de ces attaques peuvent être catégorisés suivant différents modèles d'effet de fautes connus, suivant la capacité de l'attaquant à générer une ou plusieurs attaques synchronisée, la précision de la perturbation. [4] répertorie et explique ces modèles de fautes :

- Erreurs d'un octet précis au moment où l'attaquant choisit la faute elle n'affecte qu'un seul octet (donnée ou code),
- Erreurs d'un octet non précis, la faute n'affecte qu'un seul octet mais l'attaquant ne contrôle pas précisément lequel,
- Erreurs aléatoires la faute modifie un nombre quelconque d'octet avec une valeur indéterminée.

On considère généralement que le modèle de l'octet précis est à la portée d'un attaquant disposant du matériel adéquat. Pour pouvoir raisonner sur les logiciels les auteurs adaptent le modèle de faute du matériel à des modèles de faute pour des langages de haut niveau [3], [5] en restreignant les effets à deux catégories la modification d'une variable et la modification du flot de contrôle du programme.

3 Limites des approches existantes

L'exemple suivant permet de bien comprendre les limites de ces approches réalisées dans un langage de haut niveau. La fonction décrite dans le listing 1.2 montre une comparaison en temps constant d'un PIN code candidat `arrayCandidat`

Listing 1.1: Comparaison de deux tableaux

```
bool byteArrayCompare (byte* arrayCandidat, byte* arrayRef)
{
int i = 0;
int size = 4;
int diff = false;
/* Sensitive data */
int status = false;

for (i=0; i<size; i++) {
   if (arrayCandidat[i] != arrayRef[i]) { diff = true; }
  }
if ((i==size) && (diff == false)) {
   status = true;
   }
/* Sensitive data */
return status;
}
```

avec le tableau de référence `arrayRef`. On souhaite que la variable `status` ne soit à vrai que si les deux tableaux sont identiques.

Le problème est donc de détecter dans ce programme C, comment la variable `status` peut être à la valeur `true` alors que les tableaux d'entrée sont différent. Le programme étant correctement écrit, seule une perturbation peut amener à ce résultat. Au niveau source, il y a deux possibilités. La première dans le dernier tour de boucle, l'assignation prend une valeur `false` dans la comparaison suivante la variable `status` prend la valeur `true`. La seconde attaque est une perturbation dans la second partie du `if` après la boucle afin de donner une valeur vraie à la comparaison, le résultat sera donc la mise à `true` de la variable `status`. Si on raisonne au niveau assembleur et pour une cible choisie, `arm SC100` par exemple on obtient d'autres effets qui ne peuvent être obtenus au niveau du C.

Listing 1.2: Fragment d'assembleur

```
...
843c ! 14300be5   str         r3, [fp, #-20]
8440 ! 0030a0e3   mov         r3, #0
...
84d0 ! 0300a0e1   mov         r0, r3
...
```

Les deux premières lignes correspondent à l'initialisation de la variable `status` avec le modèle de faute d'un octet on transforme le code en `0000a0e1 mov r0, r0` et donc la variable `status` devient initialisée à vrai car le registre `r0` contient l'adresse du tableau candidat. De même, la fin de la fonction transfert le contenu de la variable `status` dans le registre `r0` pour le retour de la fonction. En modifiant la ligne 0x80d0 du programme avec `ff30a0e3 mov r3, #256`suivant notre modèle de faute le programme retourne toujours `true`.

Nous proposons donc de raisonner au niveau de l'assembleur en prenant en compte l'ensemble de l'état du système, des mutations possibles en fonction du modèle de faute, une cible choisie et d'étudier la possibilité d'avoir une trace amenant à un résultat non souhaité.

4 Utilisation d'un Model Checker pour trouver les contre-exemples

La vérification de modèle au niveau assembleur possède un certain nombre d'avantages par rapport à un langage source. En étant en fin de chaine de production du logiciel nous sommes capable de prendre en compte les problèmes liés à des optimisations, des compilateurs spécialisés (les produits carte à puce possèdent des instruction spécifiques non documentées et donc des compilateurs adaptés) et les mécanismes liés au matériel. Nous n'avons pas besoin du langage source en travaillant au niveau du binaire et les constructions complexes au niveau du C (arithmétique sur les pointeurs, pointeurs de fonctions, *etc*) disparaissent au niveau assembleur. L'inconvénient réside dans l'état du système à prendre en compte et la liaison avec l'architecture et le matériel cible.

L'outil doit prendre en entrée un fichier binaire `ELF` l'invariant devant être vérifié. L'idée de la thèse est de diviser le problème en différents modules :

- Un traducteur de programme assembleur dans le langage de spécification du vérificateur de modèle (en fait il faut concevoir un traducteur par architecture),
- Un outil de vérification de modèle, qui pour un état initial donné vérifie si un programme valide une propriété,
- Un module de mutation qui modifie l'état initial avant de le donner au vérificateur de modèle.

L'état du système comprend tous les éléments pouvant être muté soit le processeur d'un cote et les mémoires de l'autre. Le processeur sera représenté dans le cas du arm7 par les registres spécialisés pc, sp, lr le registre d'état cpsr et les registres généraux r0...r12. La mémoire comprendra la mémoire programme, la pile et le tas. Une première analyse devra être réalisée afin de réduire la taille de l'état à sauvegarder (registre non utilisé, mémoire non utilisée `etc`).

Le module de mutation doit réaliser plusieurs analyses sur le code afin de limiter les mutations. En effet, par exemple le registre CPSR comprend les bit de contrôle et les bits de mode. Sa mutation n'a d'intérêt que lors de l'évaluation

des conditions, il faut donc faire une analyse de flot de donnée pour trouver les instruction en lecture de ce registre et en déduire les mutations potentielles. De même pour le registre `pc` si à un moment donné ce registre est incrémenté d'une manière indue il peut amener à retourner `true`. Dans notre exemple le registre `r3` est utilisé pour la variable `i` qui est incrémentée, si le `pc` est modifié afin d'atteindre la fin de la fonction alors c'est cette valeur qui sera retournée et donc forcément `true`. Il faut donc faire une recherche si l'assignation peut être atteinte par une altération du `pc` suivant le modèle de faute et si le registre en cause est vivant.

5 Conclusions

Nous proposons d'aborder le problème de l'analyse de programme sous hypothèse de faute matérielle. Ceci correspond aux attaques les plus difficiles à contrer dans le monde de la carte à puce. Nous cherchons à obtenir l'ensemble des états initiaux amenant une propriété à ne pas être vérifiée à la fin d'une exécution. Pour ce faire nous intégrons à l'état le programme exécutable et un module de mutation qui transforme l'état initial du système. Avec une telle architecture nous pensons être capable de traiter un programme sans distinction des aspects donnée et code qui seront traité de manière uniforme à travers les mutations.

References

1. Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems, 1997.
2. Dan Boneh, Richard A. Demillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14:101–119, 2001.
3. Maria Christofi. *Preuves de sécurité outillées d'implémentations cryptographiques.* Thèse, Université de Versailles-St Quentin, Février 2013.
4. K. O. Gadellaa. Fault attacks on java card, phd thesis, technische universiteit, 2005.
5. Xavier Kauffmann-Tourkestansky. *Security analysis of smart card C code using simulated physical attacks.* Theses, Université d'Orléans, November 2012.
6. Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *E-smart*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001.

Testium : outil de génération automatique de données de test pour les systèmes synchrones

Mouna Tka Mnad, Christophe Deleuze , Ioannis Parissis

Univ. Grenoble Alpes, LCIS, F-26000, Valence , France

Mots clés. Test logiciel, approche synchrone, génération automatique de données de test, programmation logique par contraintes.
Key words. Software testing, synchronous approach, automatic test data generation, Constraint Logic Programming.

1 Introduction

Les systèmes réactifs synchrones sont largement utilisés dans l'industrie pour l'automatisation de divers processus. Le test de ces systèmes est une activité coteuse, c'est pour cela que des outils tels que Lurette [4] , Gatel [1] et Lutess [3] ont été conus pour l'automatiser. Ces outils génèrent des séquences de test partir de description du système et/ou les objectifs du test. Notre travail est axé sur une catégorie spécifique de contrleurs synchrones produite par une entreprise internationale "Crouzet Automatisme" [1]. Les utilisateurs peuvent développer des applications complexes grâce à un environnement de programmation graphique et convivial.

Actuellement, le test de ces applications reste manuel. L'objectif de ce travail est de concevoir un outil de test efficace et facile à utiliser. Les utilisateurs pourront exprimer divers scénarios de test à l'aide du langage SPTL (Synchronous Programs Testing Language). Celui-ci a été décrit dans [2, 5] et permet la description de l'environnement où le système va fonctionner ainsi que les scénarios souhaités.

Un outil, baptisé "Testium", utilise la description SPTL pour générer des séquences de test. Nous décrivons cet outil dans la suite.

2 Présentation de l'outil

"Testium" suit l'approche de test en "bote noire". Le code du programme sous test est supposé inconnu. Les programmes synchrones ayant un comportement cyclique, le générateur de test est également est cyclique et l'outil de test

1. `http://www.em4-remote-plc.com/`

coordonne leurs exécutions. L'outil est basé sur deux principaux éléments : La spécification du programme SPTL, et le deuxième élément est la "compilation" de ce dernier vers un programme exécutable.

2.1 Le langage SPTL

Un programme SPTL est composé de groupes de contraintes classés dans des catégories . Chaque groupe est constitué de contraintes basées sur la connaissance de l'environnement où le système fonctionne et la faon dont ses composants peuvent influencer les valeurs des entrées du système. Le test sera guidé par les profils d'utilisation, qui peuvent tre prédéfinis par les concepteurs du système. Un profil est obtenu par la composition de différents groupes. Un profil peut tre alors enrichi par des scénarios de Test.
Pour écrire un programme SPTL, nous devons :

- spécifier les entrées et les sorties du système à tester et éventuellement leur valeurs initiales,
- définir les catégories et les groupes des contraintes fondés sur l'étude de la spécification fonctionnelle du système et sur les données que nous avons sur l'environnement où il évolue,
- éventuellement, rédiger des scénarios.

Ci-dessous un extrait d'un programme SPTL pour un système d'éclairage de résidence. Nous voyons les 3 parties décrites ci-dessus, à savoir, la déclaration des variables, les catégories contenant les groupes de contraintes et enfin un scénario appelé ici "Lightening".

```
var
input bool Button1
input int Pot1=5
input bool Button2
input int Pot2
input bool CaptCrep
input bool CaptPres
output int LInt
output int LExt
output bool Tempo
output bool Permanent
```

```
categ TypeHabitat
{group Boarding{
    Button1= prob(true,0.1) ;
    Button2 = prob(true,0.2)}

 group RetirementHome{
    Button1= prob(true,0.03);
    Button2 = prob(true,0.03)
    }}
categ Place
{ group Town
{CaptPres = prob(true,0.3) }
 group CountrySide
{CaptPres = prob(true,0.0001)
}}
```

```
scenario Lightening
var
time tvariable1
time tvariable2
begin
{ CaptPres = true;Button1 = true;
 tvariable1.start }|
[ Button1 = true;
( tvariable1 > 2 ) ]|
{ Button1 = false;Button2 = true;
Pot2= 5;tvariable2.start}|
[ CaptPres=false ;Button1 = false;
Button2 = true;
Pot2 = 5 (tvariable2 > 1) ]
end
```

2.2 Traduction de SPTL vers PRolog

La génération des entrées de test à partir d'un programme SPTL nécessite la résolution, à chaque cycle d'exécution, d'un ensemble de contraintes. Pour ce faire, le programme SPTL est d'abord traduit en un programme Prolog comme le montre la figure 1 (notre prototype utilise Swi-Prolog [2]). Une analyse syntaxique extrait un arbre abstrait à partir du programme SPTL. En parcourant

2. http://www.swi-prolog.org/

cet arbre, l'outil génère un ensemble de prédicats en Prolog.

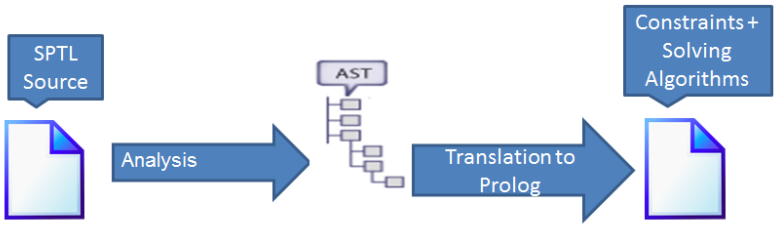


FIGURE 1 – Traduction de SPTL vers Prolog

L'algorithme principal débute par une étape d'initialisation. Il effectue ensuite une boucle. A chaque étape de la boucle, les contraintes invariantes et les contraintes de l'étape courante du scénario sont considérés. Pour la génération d'un vecteur d'entrée de test, il faut trouver une solution au système de contraintes. A chaque étape, une solution entre toutes les solutions possibles est sélectionnée. Une fois qu'une solution a été trouvée, le vecteur d'entrée est envoyé au système sous test. Ce dernier produit une sortie qui est utilisée pour calculer l'état suivant, et ainsi de suite, comme l'illustre la figure 2.

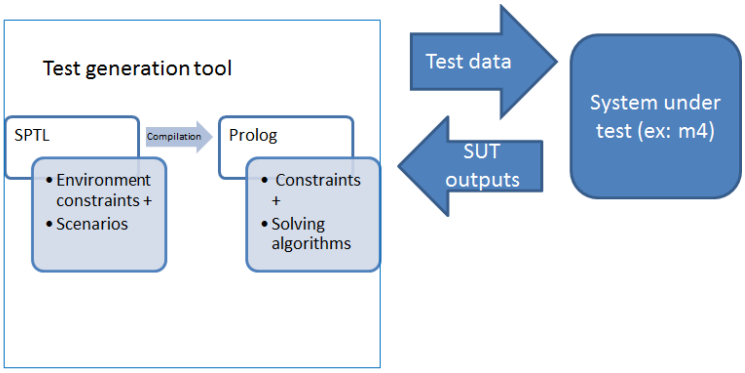


FIGURE 2 – Processus de génération des données de test

3 Conclusion

Nous avons proposé une méthode pour générer automatiquement des séquences de test pour systèmes synchrones basée sur l'utilisation du langage SPTL pour décrire les propriétés invariantes de l'environnement ainsi que son évolution dynamique. L'un des principaux objectifs de ce travail est de permettre à des non

experts en tests de faire des tests réalistes. Les notions de catégories d'utilisation, de groupes et de profils guideront la rédaction des tests et réduiront le temps et l'effort pour les préparer. Certains profils standard peuvent tre prédéfinis par les constructeurs ou par un tiers, pour aider davantage les testeurs. La possibilité d'exprimer des scénarios de test permet de prendre en compte des objectifs de test. Nous avons mis en place et expérimenté un prototype de générateur de données de test basé sur la traduction des programmes SPTL en Prolog. Les travaux futurs comprennent l'inclusion des oracles de test dans SPTL pour comparer les sorties du système avec celles attendues.

Références

[1] Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electr. Notes Theor. Comput. Sci.*, 111 :93–111, 2005.

[2] Mouna Tka Mnad, Christophe Deleuze, and Ioannis Parissis. Synchronous programs testing language (SPTL). In *Computational Science and Its Applications - ICCSA 2014 - 14th International Conference, Guimarães, Portugal, June 30 - July 3, 2014, Proceedings, Part I*, pages 683–695, 2014.

[3] Virginia Papailiopoulou, Besnik Seljimi, and Ioannis Parissis. Revisiting the Steam-Boiler Case Study with LUTESS : Modeling for Automatic Test Generation. In *Proceedings of the 12th European Workshop on Dependable Computing, EWDC 2009*, page 8 pages, Toulouse, France, May 2009. Helène WAESELYNCK.

[4] P. Raymond, D. Weber, X. Nicollin, and N. Halbwachs. Automatic testing of reactive systems. In *19th IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[5] Mouna TKA Mnad, Christophe Deleuze, and Ioannis Parissis. Synchronous Programs Testing Language (SPTL). MSR 2013 - Modélisation des Systèmes Réactifs, 2013.

Validation du standard RBAC ANSI 2012 avec B

Nghi Huynh[1-2], Marc Frappier[1], Amel Mammar[3], Régine Laleau[2] and Jules Desharnais[4]

[1]Université de Sherbrooke, Québec, Canada
[2]Université Paris Est-Créteil, Val de Marne, France
[3]Télécom Sud-Paris,Essonne, France
[4]Université Laval, Québec, Canada

RBAC est l'un des modèles de contrôle d'accès les plus connus et cités dans la littérature scientifique, qui a également su se répandre dans l'industrie. C'est aujourd'hui un standard ANSI qui a vu le jour en 2000 [6], et qui a connu des révisions majeures dont la dernière date de 2012 [1]. Nos projets de recherches nous ont conduit à évaluer RBAC pour du contrôle d'accès aux dossiers médicaux informatisés. Malgré l'utilisation d'une notation mathématique dans le standard, l'analyse du standard faite en B montre que plusieurs erreurs subsistent après la relecture par plus de 144 personnes. RBAC permet l'assignation de permissions d'effectuer des opérations sur des objets à des usagers par le biais de rôles. Les différents rôles sont donc des agrégats de permissions qui peuvent être endossés par les utilisateurs qui peuvent en endosser plusieurs à la fois durant une session. Le standard est rédigé en utilisant une notation mathématique proche du Z mais les définitions mathématiques n'ont pas été vérifiées syntaxiquement ni sur type. De par ce fait, plusieurs ambiguïtés et erreurs sont présentes dans le document. Nous avons fait le choix d'utiliser la méthode B pour son outillage riche et aussi par le fait qu'elle requiert la preuve de la préservation des invariants, alors que l'invariant est inclus dans la définitions des opérations en Z comme utilisé dans le standard.

Nous avons donc spécifié la globalité du modèle RBAC en B tout en gardant la séparation modulaire des composants RBAC. La phase de modélisation a permis de relever des ambiguïtés, des typos et des erreurs. La phase de validation et de preuve a permis de révéler des erreurs plus graves par des violations d'invariants. Parmi les erreurs retrouvés, on a : des termes utilisés avec plusieurs sens (ambiguïtés), des erreurs logiques, des symboles non déclarés, des symboles non utilisés ou encore des mauvaises définitions. Notre modèle fait en B est donc exempt de ces erreurs grâce à la validation via les preuves avec l'outil AtelierB [2] mais également via animation avec des outils comme ProB [5, 3]. Pour prendre en compte les différentes combinaisons possibles entre les divers composants RBAC, l'inclusion de machines B a été utilisée avec la prise en compte de divers problèmes comme la visibilité de certaines opérations et de variables, ainsi que le changement des préconditions des opérations. D'autre part, la modélisation en B permet également de vérifier des propriétés non présentes dans le standard comme par exemple l'absence de cycle dans la hiérarchie de rôles. La relecture humaine ne suffit donc pas à faire des standards internationaux exempts d'erreurs. L'article présenté a été accepté à la conférence ABZ 2014[4].

Références

[1] ANSI. Role Based Access Control, INCITS 359-2012, 2012.

[2] Clearsy. Atelier B. `http://www.atelierb.eu/`.

[3] M. Leuschel *et al.* ProB. `http://www.stups.uni-duesseldorf.de/ProB`.

[4] Nghi Huynh, Marc Frappier, Amel Mammar, Régine Laleau, and Jules Desharnais. Validating the RBAC ANSI 2012 standard using B. In Yamine Aït Ameur and Klaus-Dieter Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*, pages 255–270. Springer, 2014.

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

[6] R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control : Towards a unified standard. In 5^{th} *ACM Workshop on Role-based access control*, RBAC '00, page 47–63. ACM, 2000.

Construction de programmes parallèles en Coq avec des homomorphismes de listes

Frédéric Loulergue[1,2] Wadoud Bousdira[2] Julien Tesson[3]
1: Inria πr^2, PPS, Univ. Paris Diderot, Paris, France
2: Univ Orléans, INSA Centre Val-de-Loire, LIFO EA 4022, Orléans, France
3: Université Paris Est Créteil, LACL, Créteil, France

Résumé d'un article soumis au *Symposium on High-Level Parallel Programming and Applications (HLPP)*

De nos jours les architectures parallèles sont partout : des ordiphones aux super-calculateurs et fermes d'ordinateurs. Toutefois la plupart des programmeurs ne maîtrisent pas la programmation parallèle. Il y a donc un besoin de nouvelles abstractions de programmation pour rendre la programmation parallèle plus aisée, et de bibliothèques implantées en parallèle pour masquer les détails du parallélisme aux programmeurs. Le parallélisme étant présent dans tous les domaines d'applications, il est également important de s'intéresser à la correction des programmes parallèles.

Les techniques de transformation de programmes permettent d'élaborer des programmes efficaces de manière formelle. Un programme efficace (c'est-à-dire de complexité faible) est dérivé pas à pas à travers une séquence de transformations qui en préserve la sémantique et conséquemment la correction. Avec des structures de données appropriées, le calcul de programme peut être utilisé pour développer des programmes parallèles [2]. Une fois qu'une formulation correcte et efficace du programme est obtenue par transformations, le programme est souvent implanté en utilisant un langage de programmation répandu, la plupart du temps impératif, et une bibliothèque de squelettes algorithmiques. Les squelettes algorithmiques peuvent être vus comme des fonctions d'ordre supérieur implantées en parallèle. Cependant, il n'y a pas de correspondance formelle entre le programme obtenu par transformation et le programme conçu avec les squelettes.

Le système SyDPaCC est un ensemble de bibliothèques pour l'assistant de preuve Coq permettant d'écrire des programmes fonctionnels naïfs, et de les transformer en des versions plus efficaces qui peuvent être automatiquement parallélisées avant d'être extraites de Coq produisant ainsi du code OCaml enrichi par des appels à la bibliothèque de programmation parallèle fonctionnelle *Bulk Synchronous Parallel ML* ou BSML [3]. Le travail présenté est une extension de SyDPaCC permettrant de construire des programmes parallèles corrects par construction en se basant sur la théorie des homomorphismes de listes.

1 Une introduction à la théorie des listes

Une *join-list* (ou plus simplement une liste) est une séquence finie de valeurs du même type (dénombrable). Elle peut être : la liste vide $[]$, un singleton $[a]$ (pour un élément a), la concaténation $x +\!\!+ y$ de deux listes x et y. La concaténation est associative. Nous écrivons $[a_0; \ldots; a_n]$ plutôt que $[a_0] +\!\!+ \ldots +\!\!+ [a_n]$.

Pour définir une fonction sur les listes, il faut spécifier le résultat de l'application sur chaque cas de construction d'une liste. Par exemple, définissons deux combinateurs très classiques sur les listes : *map* et *reduce*.

$$\begin{cases} map\ f\ [] & = & [] \\ map\ f\ [x] & = & [f\ x] \\ map\ f\ (xs +\!\!+ ys) & = & (map\ f\ xs) +\!\!+ (map\ f\ ys) \end{cases}$$

Pour la réduction, l'argument $\oplus$ est un opérateur associatif avec pour élément neutre $i_\oplus$:

$$\begin{cases} reduce\ (\oplus)\ [] & = & i_\oplus \\ reduce\ (\oplus)\ [x] & = & [x] \\ reduce\ (\oplus)\ (xs +\!\!+ ys) & = & (reduce\ (\oplus)\ xs) \oplus (reduce\ (\oplus)\ ys) \end{cases}$$

$$
\begin{array}{lccccccccccc}
 & \mathsf{P_0} & & \dots & & \mathsf{P_i} & & \dots & & \mathsf{P_{p-1}} & & \\
 & h\ (& [a_0;\dots;a_{n_0-1}] & +\!\!+ & \dots & +\!\!+ & [a_{n_{i-1}};\dots;a_{n_i-1}] & +\!\!+ & \dots & +\!\!+ & [a_{n_{p-2}};\dots;a_{n_{p-1}-1}] &) \\
= & \{\text{ map phase }\} & & & & & & & & & & \\
 & \mathit{reduce}\ \oplus\ (& [f\,a_0;\dots;f\,a_{n_0-1}] & +\!\!+ & \dots & +\!\!+ & [f\,a_{n_{i-1}};\dots;f\,a_{n_i-1}] & +\!\!+ & \dots & +\!\!+ & [f\,a_{n_{p-2}};\dots;f\,a_{n_{p-1}-1}] &) \\
= & \{\text{ reduce phase }\} & & & & & & & & & & \\
 & & \bigoplus_{k=0}^{n_0-1} f\ a_k & \oplus & \dots & & \bigoplus_{k=n_{i-1}}^{n_i-1} f\ a_k & \oplus & \dots & \oplus & \bigoplus_{k=n_{p-2}}^{n_{p-1}-1} f\ a_k & \\
= & \bigoplus_{k=0}^{n_{p-1}-1} f\ a_k & & & & & & & & & &
\end{array}
$$

FIGURE 1 – Homomorphisme $h = (\!|\oplus, f|\!)$ comme composition de *map* et *reduce*

Nous nous intéressons plus particulièrement aux fonctions homomorphiques. Une fonction h est dite $\oplus$-*homomorphique* si pour toutes listes x et y, $h\ (x +\!\!+ y) = (h\ x) \oplus (h\ y)$ pour une opération binaire $\oplus$.

Pour fonction homomorphique h, $(\mathrm{img}(h),\ \oplus,\ h\ [\,])$ est un monoïde. On définit alors classiquement un homomorphisme de liste comme suit :

Définition 1.1. *Une fonction h est un* homomorphisme de liste *(ou simplement un homomorphisme), si elle est définie récursivement par :*

$$h\ [\,] = i_\oplus \qquad h\ [a] = f\ a \quad h\ (x +\!\!+ y) = (h\ x) \oplus (h\ y)$$

Comme h est déterminée de façon unique par f, $i_\oplus$ et $\oplus$, nous notons, classiquement, $h = (\!|\oplus, f|\!)$.

On peut remarquer que *map* $f = (\!|+\!\!+\,, f|\!)$ et *reduce* $\oplus = (\!|\oplus, id|\!)$.

Pour le parallélisme, les homomorphismes sont importants car ils ont une propriété intéressante connue sous le nom de premier théorème d'homomorphisme : tout homomorphisme $h = (\!|\oplus, f|\!)$ peut être écrit comme la composition de *map* et *reduce* : $h = (\mathit{reduce}\ \oplus) \circ (\mathit{map}\ f)$.

En se basant sur cette propriété, les homomorphismes peuvent être parallélisés comme illustré à la figure 1 où les P_i sont les p processeurs de la machine parallèle. En supposant que la liste est répartie sur les processeurs, chaque processeur applique la fonction *map* sur le morceau de la liste qu'il possède puis réduit la liste obtenue à l'aide de *reduce*. Pour calculer le résultat final, les réductions partielles doivent être échangées, de telle sorte qu'elles puissent être réduites ensemble pour obtenir le résultat final. Il y a plusieurs schémas de communication et de calcul qui peuvent être utilisés pour cette dernière étape, la plus simple étant de rassembler tous les résultats des réductions partielles sur un processeur qui se charge de faire la réduction finale.

Deux autres théorèmes d'homomorphisme sont classiques [1], nous présentons ici le troisième.

Définition 1.2 (Fonction $\oplus$-vers-la-gauche, fonction $\oplus$-vers-la-droite)**.** *Une fonction h est dite $\oplus$-vers-la-gauche pour un opérateur binaire $\oplus$, si pour toute liste x et tout élément a, $h\ ([a] +\!\!+ x) = a \oplus h\ x$.*

Une fonction h est dite $\oplus$-vers-la-droite pour un opérateur binaire $\oplus$, si pour toute liste x et tout élément a, $h\ (x +\!\!+ [a]) = (h\ x) \oplus a$.

Théorème 1.3 (Troisième théorème d'homomorphisme)**.** *Soient h une fonction, $\oplus$ et $\otimes$ des opérateurs binaires. Si h est $\oplus$-vers-la-gauche et $\otimes$-vers-la-droite, alors h est un homomorphisme.*

Informellement, puisque le premier théorème d'homomorphisme permet d'exprimer un homomorphisme comme une composition de *map* et *reduce*, et que cette dernière peut être parallélisée, le troisième théorème d'homomorphisme permet à partir de deux programmes séquentiels, l'un parcourant la liste de gauche à droite, l'autre de droite à gauche, d'obtenir un programme parallèle. Malheureusement, ce théorème n'est pas *constructif*. Nous utilisons une variante plus faible reposant sur la notion d'inverse droit faible : h' est un *inverse droit faible* de h si et seulement si pour toute liste x, $h\ x = h(h'(h\ x))$.

Théorème 1.4 (Troisième théorème d'homomorphisme faible)**.** *Soient h une fonction, h' un inverse droit faible de h, $\oplus$ et $\otimes$ des opérateurs binaires. Si h est $\oplus$-vers-la-gauche et $\otimes$-vers-la-droite, alors h est un homomorphisme $(\!|\odot, f|\!)$ où $f\ a = h\ [a]$ et $a \odot b = h((h'\ a) +\!\!+ (h'\ b))$.*

Nous présentons brièvement la dérivation d'un programme efficace pour le problème du calcul du maximum des sommes des préfixes d'une liste. Par exemple $mps\ [\underline{1;2;-1;2;-1;-1;3};-4] = 5$ (le préfixe dont la somme est maximale est souligné).

Une première version de la fonction *mps* est $mps = maximum \circ (map\ sum) \circ prefix$. La fonction *maximum* est l'homomorphisme $(\!|\uparrow, id|\!)$, où $a \uparrow b = b$ si $a < b$ et $a \uparrow b = a$ si $b \leq a$. *maximum* n'est pas définie sur la liste vide. La fonction *sum* est l'homomorphisme $(\!|+, id|\!)$. La fonction *prefix* génère la liste de tous les préfixes de son argument. Il est facile de vérifier que *mps* est $\oplus$-vers-la-gauche mais il n'y a pas de $\otimes$ tel que *mps* soit $\otimes$-vers-la-droite. Toutefois on peut apparier deux fonctions f et g : pour tout x, $(f \triangle g)\ x = (f\ x, g\ x)$. Considérons $ms = mps \triangle sum$. La fonction *ms* est $\oplus$-vers-la-gauche et $\otimes$-vers-la-droite avec $a \oplus (b_m, b_s) = (0 \uparrow (a + b_m), a + b_s)$ et $(a_m, a_s) \otimes b = (a_m \uparrow (a_s + b), a_s + b)$.

La fonction $ms'(m, s) = [m; s - m]$ est un inverse droit faible de *ms*. On peut alors appliquer le troisième théorème d'homomorphisme faible, et on obtient que *ms* est l'homomorphisme $(\!|\odot, f|\!)$ avec

$$\begin{cases} f\ a & = & (mps\ [a], sum\ [a]) = (0 \uparrow (a + mps\ []), a) = (0 \uparrow a, a) \\ (a_m, a_s) \odot (b_m, b_s) & = & ms(ms'\ (a_m, a_s) +\!\!+ ms'\ (b_m, b_s)) = \ldots = (0 \uparrow a_m \uparrow (a_s + b_m), a_s + b_s) \end{cases}$$

Si on note *fst* la projection de la première composante d'un couple, par le premier théorème d'homomorphisme, on a : $mps = fst \circ (map\ f) \circ (reduce\ \odot)$ qui peut être parallélisé comme indiqué figure 1.

2 Paralléliser avec des homomorphismes dans SyDPaCC

Dans la première section nous avons vu que la propriété centrale pour les homomorphismes est le fait pour une fonction h: list A →B d'être ⊕-homomorphique où ⊕ est une opération binaire et A et B sont deux ensembles. Nous modélisons cette propriété en Coq sous la forme d'une classe :

```
Class Homomorphic `(h:list A→B) `(op:B→B→B) := { homomorphic : ∀x y, h (x++y) = op (h x) (h y) }.
```

On peut alors montrer que ⊕ est nécessairement une opération associative, et que si h est définie pour la liste vide, elle forme un monoïde sur *l'image* de h. Une variante de la définition 1.1, est de dire que h est un homomorphisme si pour un monoïde $(\mathsf{B}, \oplus, i_\oplus)$, on a les trois équations de la définition 1.1. Cette définition capture moins de fonctions puisque l'on impose que l'opération binaire soit associative et ait un neutre sur tout B, pas seulement sur l'image de h. Nous formalisons cette variante de la définition 1.1, sous forme d'une classe :

```
Class Homomorphism_f `(h : list A →B) `(f: A →B) := { homomorphism_f : ∀(a:A), h [a] = f a }.
Class Homomorphism `(h:list A→B) `(op: B→B→B) `(f:A →B) `{LMonoid B op e} `{Homomorphic A B h op}
  `{Homomorphism_f A B h f} := { homomorphism_nil : h [] = e }.
```

Les deux dernières équations de la définition 1.1, sont toutes les deux modélisées par des classes et sont en argument de la classe Homomorphism. Ceci permet de créer des instances de ces classes et de bénéficier de la résolution d'instance lors de la création d'instances de la classe Homomorphism.

Nous pouvons prouver que si une fonction h:list A→B est ⊕-homomorphique alors (img h,⊕,h[]), où img h dénote l'image de h, est un monoïde et donc que h restreinte à son image est un homomorphisme au sens de la classe Homomorphism. Nous omettons ces détails ici. En utilisant cette définition d'homomorphisme et les définitions de map et reduce, on peut alors énoncer le premier théorème d'homomorphismes :

```
Theorem first_homomorphism_theorem: ∀`{Homomorphism A B h op f e}, ∀l, h l = ((reduce op) ∘(map f)) l.
```

La preuve se fait facilement par induction sur la liste l. Cet énoncé toutefois ne construit pas une version équivalente à h. Il est ainsi préférable de décomposer en :

```
Definition first_hom_thm_fun `(Homomorphism A B h op f e) := (reduce op) ∘(List.map f).
Lemma first_hom_thm_fun_prop: ∀`{hom:Homomorphism A B h op f e}, ∀l, h l = first_hom_thm_fun hom l.
```

Le troisième théorème d'homomorphisme nécessite que soient formalisées les notions de ⊕-vers-la-gauche et ⊗-vers-la-droite. Nous le faisons avec deux nouvelles classes et les fonctions fold usuelles :

```
Class Rightwards `(h:list A→B)`(op:B→A→B)`(e:B) := { rightwards: ∀l, h l = List.fold_left op l e }.
Class Leftwards (h:list A→B)`(op:A→B→B)`(e:B) := { leftwards: ∀l, h l = List.fold_right op e l }.
```

En définissant la classe RightInverse par :

```
Class Right_inverse '(h:list A →B)(h':B→list A) := { right_inverse: ∀l, h l = h(h'(h l)) }.
```

on peut alors énoncer une variante du troisième théorème d'homomorphisme faible :

```
Instance third_homomorphism_theorem '{h:list A→B}'{inv:Right_inverse A B h h'} '{Hl:Leftwards A B h opl e}
  '{Hr:Rightwards A B h opr e} : Homomorphic h (fun l r ⇒h( (h' l)++(h' r))).
```

La conclusion de ce théorème est que la fonction h est homomorphique. On a directement la version h est un homomorphisme par :

```
Program Instance third_hom_thm '(h:list A→B)'{inv:Right_inverse A B h h'} '{Hl:Leftwards A B h opl e}
  '{Hr:Rightwards A B h opr e} : Homomorphism (rstrct_h h) (rstrct(fun l r⇒h(h' l++h' r))) (fun a⇒(rstrct_h h)[a]).
```

où les fonctions rstrct limitent le codomaine de leurs arguments à l'image de h.

On peut définir la version naïve de mps puis ms comme suit, où les variantes de la composition prennent en compte le fait que maximum n'opère que sur des listes vides, et que prefix ne produite que des listes non vides :

```
Definition mps_spec : list t →t := maximum ∘' (map sum) ∘'' prefix.
Definition tupling '(f:A→B)'(g:A→C) := fun x ⇒(f x, g x).
Definition ms_spec := tupling mps_spec sum.
```

On montre alors que ms' défini comme suit est un inverse droit de ms_spec :

```
Definition ms' (p:t∗t) := let (m,s) := p in [ m; s + −m].
Program Instance ms_right_inverse : Right_inverse ms_spec ms'. Next Obligation. (∗ omis ∗) Qed.
```

Le troisième théorème d'homomorphisme, peut alors être appliqué :

```
Instance ms_homomorphic : Homomorphic ms_spec (fun l r⇒ms_spec(ms' l ++ ms' r)) := third_hom_thm.
```

On peut construire de façon très similaire à ce qui est fait dans la première section, une version optimisée de l'opérateur binaire, et des théorèmes garantissent que l'homomorphisme obtenu est extensionnellement égal à la fonction de départ.

Par le premier théorème d'homomorphisme, on peut alors obtenir ms_spec sous forme d'une composition efficace de map et reduce. Pour paralléliser cette composition, nous utilisons une axiomatisation de la bibliothèque de programmation parallèle *Bulk Synchronous Parallel ML* (BSML) pour programmer des versions parallèles de map et de reduce. La sémantique purement fonctionnelle des primitives de BSML qui manipulent une structure de données parallèle, est donnée par le type de module PRIMITIVES. Nous parallélisons automatiquement l'homomorphisme dérivé (en se basant sur le mécanisme des types de classes de Coq) puis appliquons une projection pour obtenir la version parallèle de mps_spec :

```
Module MPS_Parallel(Bsml : PRIMITIVES). (∗ Modules contenant les squelettes paralleles omis ∗)
  Definition par_ms := Eval simpl in Parallel.left_parallel (f:=first_hom_thm_fun optimised_ms).
  Definition par_mps := fst ∘par_ms.
End MPS_Parallel.
```

L'impression du terme par_ms donne :

```
par_ms = fun plst : Bsml.par (list t) ⇒Map_reduce.mapReducePar f odot plst : Bsml.par (list t) →img ms_spec
```

Cette fonction ne prend plus en entrée des listes, mais des valeurs de type Bsml.par(list t), c'est-à-dire des vecteurs parallèles de listes. Bsml.par est un type de données polymorphe fourni par (la formalisation en Coq de) la bibliothèque BSML. On peut voir une valeur de type Bsml.par(list t) comme une liste répartie (figure 1). On utilise l'extraction de Coq et la bibliothèque BSML en OCaml pour obtenir un programme parallèle exécutable.

Références

[1] J. Gibbons. The third homomorphism theorem. *Journal of Functional Programming*, 6(4) :657–665, 1996.

[2] Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in calculational forms. In *POPL*, pages 316–328. ACM, 1998.

[3] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML : Modular Implementation and Performance Prediction. In *International Conference on Computational Science (ICCS)*, volume 3515 of *LNCS*, pages 1046–1054. Springer, 2005.

Vérification formelle du module d'adressage virtuel de l'hyperviseur Anaxagoros avec Frama-C : une étude de cas *

Allan Blanchard[1,3] Nikolai Kosmatov[1] Matthieu Lemerre[1] Frédéric Loulergue[2,3]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] Inria πr^2, PPS, Univ Paris Diderot, CNRS, Paris, France
[3] Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
firstname.lastname@univ-orleans.fr

Les applications en lignes et mobiles sont de plus en plus populaires. Beaucoup d'applications migrent vers des systèmes d'informatique en nuage (*cloud*) et deviennent des *"Software as a Service"* (SaaS). Nous utilisons également de plus en plus les services de stockage dans le *cloud*. Avoir des environnements *cloud* fiables, sûrs et sécurisés devient donc une nécessité.

Ce travail concerne la vérification formelle du module d'adressage virtuel d'Anaxagoros, un micronoyau d'hyperviseur de *cloud* développé au CEA LIST, et conçu pour l'isolation et la protection des tâches invitées. Ce module gère les pages de mémoire sous la forme d'une hiérarchie, avec au plus bas les pages de données, et un ou plusieurs niveaux supérieurs de tables de pages. Une table de page *référence* une page de niveau inférieur en inscrivant son numéro dans la liste des pages qui lui sont accessibles. Chaque page pouvant être partagée entre plusieurs tables, on lui associe un compteur indiquant le nombre de tables de pages qui y font référence. De cette manière, le système ne permet pas de nettoyer et libérer une page qui serait toujours référencée, ce qui induirait un risque de fuite ou de corruption des données.

Nous nous attachons plus particulièrement à la vérification de la fonction en charge de modifier les référencements au sein d'une table de pages (et donc de maintenir les valeurs des compteurs à jour). Cette fonction peut être appelée de manière concurrente par différents processus.

Nous avons d'abord prouvé cette fonction dans le cadre de son exécution séquentielle. La preuve est effectuée de manière automatique avec Frama-C [1], la plateforme d'analyse de code C développée au CEA LIST, et son greffon de preuve déductive, Wp.

Nous avons pris en compte la dimension concurrente de la fonction par la réalisation d'une simulation qui consiste à modéliser les contextes des exécutions concurrentes par un ensemble de tableaux associant à chaque *thread* (fil d'exécution) la valeur de chacune de ses variables locales et sa position dans l'exécution de la fonction. Chacune des opérations atomiques est placée dans une fonction prenant en paramètre le numéro du *thread* effectuant l'action en question. Enfin, une boucle infinie choisit aléatoirement à chaque tour un *thread* et le fait avancer d'une opération.

Nous prouvons également cette simulation de manière automatique avec Frama-C et Wp. Quelques lemmes supplémentaires relatifs aux nombres d'occurrences de valeurs au sein des pages ont été prouvés à l'aide de l'assistant de preuve Coq par extraction depuis Wp.

Nous présentons les résultats de cette étude de cas incluant la simulation des exécutions concurrentes et la formalisation de l'invariant. Nous fournissons des retours d'expérience, notamment l'effort nécessaire pour réaliser la spécification et la preuve, et les difficultés liées au comptage de références. Nous discutons les bénéfices et inconvénients de la méthode utilisée et ses perspectives d'application, ainsi que sa validité en présence de modèles mémoire faibles.

*Cet article est un résumé étendu de l'article "A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C" accepté au workshop "20th Int. Workshop on Formal Methods for Industrial and Critical Systems (FMICS 2015)". Ce travail a été partiellement financé par le projet CEA CyberSCADA et le programme EU FP7 (projet STANCE, grant 317753).

1. Disponible sur http://frama-c.com

Verifying Reachability-Logic Properties on Rewriting-Logic Specifications

Andrei Arusoaie[1], Dorel Lucanu[2], David Nowak[3], and Vlad Rusu[1]

[1]INRIA Lille-Nord Europe, France

[2]Faculty of Computer Science UAIC, Romania

[3]CRIStAL, University of Lille, France

Reachability Logic (RL) is a formalism which can be used for stating properties about program execution and for specifying operational semantics of programming languages. RL formulas are pairs of the form $\varphi \Rightarrow \varphi'$, where φ, φ' are called *patterns* and they represent sets of states. A RL formula denotes a *reachability relation* between two such sets of states, and a set of RL formulas induces a transition system over states. Depending on the interpretation of a formula, the reachability relation can express either programming-language semantic steps, or properties over programs in the language in question.

In [1] , we show that RL can be adapted for stating properties of systems described in Rewriting Logic (RWL). We propose an automatic procedure for proving RL properties of RWL specifications, we prove that it is sound, and we illustrate its use by verifying RL properties of a communication protocol written in Maude. Our contribution with respect to RL is a proved-sound mechanical verification procedure. Previous works include sound and relatively complete proof systems for RL . However, these proof systems lack strategies for rule applications, making them unpractical for automatic verification. Our procedure can be seen as such a strategy. With respect to RWL , our contribution is the adaptation of this procedure for verifying RWL theories against *reachability* properties $\varphi \Rightarrow \varphi'$, which say that all terminating executions starting with a state in φ eventually reach a state in φ'. Both φ and φ' denote possibly infinite sets of states. We note that RL properties for RWL theories are different from the reachability properties that can be checked in Maude using the `search` command. The difference resides in the possibility of using first-order logic for constraining the initial and the final state terms, and in the interpretation of RL formulas.

References

[1] Andrei Arusoaie, Dorel Lucanu, David Nowak, and Vlad Rusu. Verifying Reachability-Logic Properties on Rewriting-Logic Specifications, submitted to Festschrift Symposium in Honor of Jose Meseguer: Logic, Rewriting, and Concurrency. `http://meseguer-fest.ifi.uio.no/`.

Instrumentation de programmes C annotés pour la génération de tests*

Guillaume Petiot[1,2] Bernard Botella[1] Jacques Julliand[2]
Nikolai Kosmatov[1] Julien Signoles[1]

[1] CEA, LIST, Software Reliability Laboratory, PC 174, F-91191 Gif-sur-Yvette
firstname.lastname@cea.fr
[2] FEMTO-ST/DISC, Université of Franche-Comté, F-25030 Besançon Cedex
firstname.lastname@femto-st.fr

La vérification déductive des logiciels critiques repose sur des spécifications formelles de propriétés. Pour des logiciels développés dans un langage de programmation impératif et/ou objet, les spécifications sont décrites dans des langages d'annotations exprimant des pré et des post conditions, des invariants, des variants et diverses autres assertions. L'écriture de la spécification du programme d'une part et la preuve de la conformité d'autre part sont souvent des tâches rendues difficiles par l'imprécision du diagnostic en cas d'échec de preuve. En cas de non conformité, le prouveur n'indique notamment pas quelle est l'annotation à l'origine de l'échec de preuve. L'une des pistes pour aider l'ingénieur validation est de combiner des analyses dynamiques à la preuve afin de détecter de manière plus précise l'origine des non conformités. Plus précisemment, notre objectif est de combiner de la génération de tests structurels à de la preuve pour exhiber des contre-exemples qui déterminent l'origine des échecs de preuve. Pour cela, il faut ajouter au programme des chemins d'exécution qui sont susceptibles de violer les spécifications. Pour les engendrer, nous proposons dans cet article de traduire chaque annotation en un fragment de programme introduisant ces chemins.

Ce papier apporte des solutions à ce problème dans le contexte de l'environnement de vérification de programmes C appelé Frama-C [1]. Il prend en compte des spécifications décrites dans le langage d'annotations appelé acsl. Nous décrivons des règles de traduction automatique de ces annotations en fragments de programmes C qui sont destinés à être utilisés par un générateur de tests structurels pour engendrer des cas de test violant chaque annotation révèlant ainsi des non conformités entre le programme et ses spécifications. Ces traductions ont été mises en œuvre au sein d'un greffon de Frama-C appelé StaDy (pour combinaison d'analyses Statique et Dynamique) qui utilise l'outil de preuve Wp de Frama-C et le générateur de tests PathCrawler développés au CEA LIST.

Nos contributions sont :

— une description des règles de traduction en C des annotations du noyau exécutable du langage acsl appelé e-acsl pour combiner preuve et génération de tests,
— une implémentation de ces règles dans StaDy,
— une comparaison des traductions pour la génération de tests et pour la vérification des annotations à l'exécution,
— un compte-rendu d'expérimentations montrant l'efficacité de la traduction pour trouver des contre-exemples de non conformité entre des programmes C et leurs annotations e-acsl.

*Cet article est un résumé étendu de l'article "Instrumentation of Annotated C Programs for Test Generation" accepté à la conférence "14th Int. Working Conference on Source Code Analysis and Manipulation (SCAM 2014)". Ce travail a été partiellement financé par le programme EU FP7 (projet STANCE, grant 317753).

1. Disponible sur http://frama-c.com

Détection sûre et quasi-complète d'objectifs de test infaisables *

Sébastien Bardin[1] Mickaël Delahaye[1] Robin David[1]
Nikolai Kosmatov[1] Michail Papadakis[2] Yves Le Traon[2]
Jean-Yves Marion[3]

[1] CEA, LIST, Laboratoire pour la Sûreté des Logiciels, e-mail: prenom.nom@cea.fr
[2] Université du Luxembourg, SnT, e-mail: prenom.nom@uni.lu
[3] Université de Lorraine, CNRS et Inria, LORIA, e-mail: prenom.nom@loria.fr

Dans le domaine du test logiciel, la qualité des cas de tests est mesurée via des *critères de couverture*. Un critère de couverture spécifie des objectifs que les cas de test doivent couvrir. Nous nous concentrons ici sur les critères de couverture *structurels,* c'est-à-dire définis au niveau du code source du programme. Dans la pratique, la couverture est limitée du fait du problème d'infaisabilité qui peut se poser pour chacun des objectifs de test, ceux-ci étant définis a priori par rapport à la structure du programme, et non par rapport à sa sémantique. Cela pose des problèmes considérables en termes de génération de tests manuelle ou automatisée (perte de ressources à essayer de couvrir des objectifs impossibles), de calcul de score de couverture (apparaissant plus bas en présence d'objectifs infaisables), et en conséquence d'estimation de la qualité d'un jeu de tests et de l'opportunité de terminer la phase de test.

Pour répondre à ce problème, nous utilisons une combinaison de deux techniques de vérification, l'analyse de valeurs par interprétation abstraite et le calcul de plus faible précondition. Nous proposons une méthode « boîte grise » pour combiner ces deux techniques de façon complémentaire. Grâce à cela, nous ciblons la détection des objectifs de test infaisables de manière automatique et correcte, c'est-à-dire, nous ne déclarons infaisables que des objectifs l'étant réellement. De plus, notre méthode exploite avantageusement une représentation unifiée des critères de couverture introduite dans nos travaux précédents.

Intégrée au sein de plateforme Frama-C et en particulier de la boîte à outil de test LTest, cette approche a pu être évaluée sur plusieurs critères de couverture (conditions, conditions multiples et mutations faibles). Nos résultats montrent que la méthode proposée est capable de détecter quasiment tous les objectifs de test infaisables, 95% en moyenne, et cela en un temps raisonnable, rendant cette approche viable pour le test unitaire.

*Ce résumé est issue de l'article « *Sound and Quasi-Complete Detection of Infeasible Test Requirements* » présenté à ICST 2015, *International Conference on Software Testing and Verificaiton.* Ce travail a été partiellement financé par le programme EU-FP7 (projet STANCE, bourse 317753) et l'ANR (projet BINSEC, bourse ANR-12-INSE-0002).

Montre moi d'autres contre-exemples : une approche basée sur les chemins

Kalou Cabrera Castillos, Hélène Waeselynck
LAAS-CNRS and Univ. Toulouse
7 av Colonel Roche
31400 Toulouse cedex, France
kalou.cabrera.castillos@laas.fr, helene.waeselynck@laas.fr

Virginie Wiels
ONERA/DTIM
2 av Edouard Belin, BP74025
31055 Toulouse cedex, France
virginie.wiels@onera.fr

Résumé

Nous nous situons dans le contexte du développement basé modèles de systèmes réactifs ; nous considérons plus spécifiquement la vérification formelle de propriétés sur des modèles Simulink. Le model checking est utilisé comme moyen de debug, l'analyse se concentre sur un petit nombre de fonctions critiques, les contre-exemples produits par la vérification sont utilisés pour corriger le modèle considéré jusqu'à ce qu'il soit correct vis-à-vis de la propriété considérée.

Plusieurs problèmes se posent pour mettre en œuvre de façon efficace ce processus de debug. Tout d'abord, lorsque le model checker produit un contre-exemple long et impliquant de nombreuses variables, il est difficile de comprendre les causes de la violation de la propriété. Pour assister le concepteur dans cette tâche, nous avons développé un outil STANCE (Structural ANanalysis of Counter-Examples) interfacé avec Simulink. Cet outil permet d'extraire le sous-ensemble de valeurs d'entrée datées suffisantes pour reproduire la violation et montre graphiquement les parties du modèle qui jouent un rôle dans la violation. Il filtre donc toutes les données qui n'interviennent pas dans la violation pemettant ainsi au concepteur de se focaliser sur les données importantes.

Le deuxième problème qui se pose est que le model checking ne fournit classiquement qu'un seul contre-exemple à la fois, et aucune autre information ne peut être obtenue tant que le modèle (ou la propriété) n'a pas été corrigé. Or il peut exister d'autres erreurs dans le modèle ou d'autres scénarios permettant de déclencher l'erreur illustrée par le contre-exemple. Pour obtenir un processus de debug plus efficace, il serait intéressant de donner au concepteur le plus d'informations possibles sur le modèle courant afin de pouvoir le corriger au mieux. Pour cela, il faut être capable de produire plusieurs contre-exemples illustrant différents types de violations de la propriété considérée. C'est l'objet de l'approche proposée dans cet article.

La recherche de nouveaux contre-exemples s'appuie sur l'analyse structurelle mise en œuvre par STANCE. Nous utilisons les données collectées lors du rejeu du contre-exemple pour synthétiser des requêtes afin de guider le model checker vers des chemins de violation différents. L'approche a été implémentée, elle est appliquée à un exemple académique ainsi qu'à un exemple industriel du domaine automobile.

Cet article a été accepté à ICST 2015 (International Conference on Software Testing Verification and Validation).

Remerciement

Ce travail a été financé en partie par le RTRA STAE (projet BriefCase).