

Actes des 15^{èmes} journées sur les

Approches Formelles dans l'Assistance au Développement de Logiciels

Édités par Aurélie Hurault et Nicolas Stouls
Les 7 et 8 Juin 2016 à l'Institut FEMTO-ST de Besançon



Préface

L'atelier francophone AFADL sur les Approches Formelles dans l'Assistance au Développement de Logiciels, se tiendra pour sa quinzième édition les 7 et 8 juin 2015 au Département d'Informatique des Systèmes Complexes (DISC) de l'Institut FEMTO-ST à Besançon. Cette année encore, cet atelier est organisé conjointement avec les journées du GDR-GPL, la 5^{ème} Conférence en Ingénierie du Logiciel (CIEL) et la 10^{ème} Conférence francophone sur les Architectures Logicielles (CAL). Comme l'année dernière, l'atelier AFADL est organisé en collaboration avec les groupes de travail Méthodes de Test pour la Vérification et la Validation (MTV2) et Méthodes Formelles pour le Développement Logiciel (MFDL), qui auront chacun une session dédiée.

L'atelier AFADL rassemble de nombreux acteurs académiques et industriels intéressés par la mise en œuvre des techniques formelles aux divers stades du développement des logiciels et/ou des systèmes. Il a pour objectif de mettre en valeur les travaux récents effectués autour de thèmes comme :

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

Nous aurons l'honneur d'accueillir, en association avec les journées du GDR, Pascal Cuoq (Trust In Soft) comme conférencier invité.

Cette année, l'appel à contributions a été orienté pour se concentrer sur des contributions en format court. Ces formats plus synthétiques ont l'avantage d'ouvrir plus largement la portes aux discussions. Les contributions attendues étaient des articles courts (6 pages), des démonstrations d'outils (4 pages), des présentations de projets (2 pages), des résumés longs de travaux déjà publiés (2 pages) et des travaux de doctorants (3 pages). 24 travaux ont été retenus pour être présentés. Parmi eux, il y a notamment 6 contributions de doctorants, 10 articles courts et 3 résumés étendus.

Nous remercions les membres du comité de programme et des groupes MTV2 et MFDL pour leur travail qui a contribué à produire un programme dense et 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 du GDR-GPL 2016 qui ont pris en charge tous les aspects logistiques.

Le 12 Mai 2016,

Aurélie Hurault et Nicolas Stouls
Présidents du comité de programme AFADL 2016.

Comité de programme

Présidents du comité de programme

- Aurélie HURAUULT IRIT/INPT-ENSEEIH, Toulouse
- Nicolas STOULS, CITI/INSA-Lyon, Lyon

Organisateur local

- Frédéric DADEAU FEMTO-ST, Université de Bourgogne Franche-Comté

Membres du comité de programme

- Yamine AIT AMEUR, IRIT/INPT-ENSEEIH, Toulouse
- Sandrine BLAZY, IRISA - Université Rennes 1
- Frédéric BONIOL, ONERA, Toulouse
- Lydie DU BOUSQUET, LIG/Université Grenoble-Alpes, Grenoble
- Catherine DUBOIS, ENSIIE – Samovar, Evry
- Christèle FAURE, SafeRiver, Paris
- Akram IDANI, LIG, Grenoble
- Jacques JULLIAND, FEMTO-ST, Université de Bourgogne Franche-Comté
- Florent KIRCHNER, CEA, Saclay
- Nikolai KOSMATOV, CEA, Saclay
- Régine LALEAU, LACL – Université Paris-Est, Créteil
- Jean-Louis LANET, INRIA-RBA, Rennes
- Arnaud LANOIX, LINA, Université de Nantes
- Yves LEDRU, LIG/Université Grenoble-Alpes, Grenoble
- Pascale LEGALL, École Centrale, Paris
- Nicole LEVY, CEDRIC/CNAM, Paris
- Delphine LONGUET, LRI, Orsay
- Ioannis PARISSIS, LCIS, Valence
- Pascal POIZAT, LIP6, Paris
- Marie-Laure POTET, Verimag, Grenoble
- Marc POUZET, DIENS, Paris
- Antoine ROLLET, LaBRI, Bordeaux
- Vlad RUSU, INRIA, Lille
- Safouan TAHA, Supélec, Gif-sur-Yvette
- Sylvie VIGNES, Télécom Paris-Tech, Paris
- Laurent VOISIN, Systerel, Aix-en-Provence
- Virginie WIELS, ONERA, Toulouse
- Fatiha ZAIDI, LRI, Orsay

Relecteur additionnel

- Mohamed Amine AOUADHI, LINA, Université de Nantes

Sommaire

Session 1 : Vérification

Slicing relaxé : une version de slicing adaptée à la vérification	1
<i>Résumé étendu–Jean-Christophe Léchenet, Nikolai Kosmatov and Pascale Le Gall</i>	
Outillage pour la modélisation, la vérification et la génération d'applications temporisées et embarquées	2
<i>Article court–Pierre-Emmanuel Hladik, Silvano Dal Zilio, Olivier Pasquier, Sébastien Pillement and Bernard Berthomieu</i>	
Identification de propriétés pour la validation d'un système cyber-physique médical	8
<i>Doctorant–Yoann Blein</i>	
Comparaison des Approches SMT et CSP Appliquées à la Vérification de Réseaux Workflows ...	11
<i>Résumé étendu–Hadrien Bride, Olga Kouchnarenko, Fabien Peureux and Guillaume Voiron</i>	

Session 2 : Sécurité

Validation formelle d'implémentation des patrons de sécurité : Application aux SCADA	13
<i>Article court–Fadi Obeid and Philippe Dhaussy</i>	
GenISIS : un outil de recherche d'attaques d'initié en Systèmes d'Information	19
<i>Article court–Amira Radhouani, Akram Idani, Yves Ledru and Ben Rajeb Narjes</i>	
Approche de spécification et validation formelles de politiques RBAC au niveau des processus métiers	27
<i>Doctorant–Salim Chehida</i>	
Inférence et analyse de propriétés dans les protocoles de contrôle-commande	32
<i>Doctorant–Emmanuel Perrier</i>	

Session 3 : Outils d'aide à la vérification

Formalisation d'une Approche Compositionnelle Des Patrons de Propriétés	35
<i>Article court–Djamila Baroudi, Philippe Dhaussy and Safia Nait Bahloul</i>	
Formalisation des interactions asynchrones	43
<i>Doctorant–Florent Chevrou</i>	
SIMPA - Simpa Infers Models Pretty Automatically	47
<i>Outils–Catherine Oriat, Roland Groz and Emmanuel Perrier</i>	
DynIBEX : une boîte à outils pour la vérification des systèmes cyber-physiques	51
<i>Outils–Julien Alexandre Dit Sandretto and Alexandre Chapoutot</i>	

Session 4 : Mathématiques pour le développement logiciel

Préservation de la cohérence des transformations topologiques et géométriques	55
<i>Article court–Valentin Gauthier, Thomas Bellet, Hakim Belhaouari and Agnès Arnould</i>	
Algèbre linéaire pour invariants polynomiaux	61
<i>Article court–Steven De Oliveira, Virgile Prevosto and Saddek Bensalem</i>	
Vérification formelle de programmes de génération de données structurées	67
<i>Doctorant–Richard Genestier</i>	
Spécification et vérification formelle d'opérations sur les permutations	72
<i>Article court–Richard Genestier and Alain Giorgetti</i>	

Session 5 : MTV2 et AFADL

Quelle confiance peut-on établir dans un système intelligent ?	79
<i>Projet–Lydie Du Bousquet and Masahide Nakamura</i>	
Génération systématique de scénarios d'attaques contre des systèmes industriels	81
<i>Article court–Maxime Puys, Marie-Laure Potet and Jean-Louis Roch</i>	
MBeeTle - un outil pour la génération de tests à-la-volée à l'aide de modèles	88
<i>Outils–Julien Lorrain, Elizabeta Fourneter, Frédéric Dadeau and Bruno Legeard</i>	
BINSEC : plate-forme d'analyse de code binaire	92
<i>Résumé étendu–Adel Djoudi, Robin David, Josselin Feist, Sebastien Bardin and Thanh Dinh Ta</i>	

Session 6 : MFDL et AFADL

Un processus de développement Event-B pour des applications distribuées	94
<i>Article court–Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix and Mamoun Filali</i>	
Projet ANR BINSEC : analyse formelle de code binaire pour la sécurité	101
<i>Projet–Sebastien Bardin</i>	
Combiner des diagrammes d'état étendus et la méthode B pour la validation de systèmes industriels	103
<i>Doctorant–Thomas Fayolle</i>	
Vers un développement formel non incrémental	106
<i>Article court–Thi-Kim-Zung Pham, Catherine Dubois and Nicole Levy</i>	

Slicing relaxé : une version de *slicing* adaptée à la vérification *

Jean-Christophe Léchenet^{1,2} Nikolai Kosmatov¹ Pascale Le Gall²¹ CEA, LIST, Laboratoire de Sûreté des Logiciels, PC 174, 91191 Gif-sur-Yvette
prenom.nom@cea.fr² Laboratoire de Mathématiques et Informatique pour la Complexité et les Systèmes
CentraleSupélec, Université Paris-Saclay, 92295 Châtenay-Malabry France
prenom.nom@centralesupelec.fr

Contexte et motivations. Le *slicing* est une technique consistant à simplifier un programme donné en un programme appelé *slice* qui a le même comportement que le programme initial vis-à-vis d'un critère donné, dit critère de *slicing*. Ce critère consiste généralement en une instruction du programme initial. La *slice* est construite en retirant du programme initial les instructions qui n'ont pas d'impact sur le critère. Appliquer des méthodes de vérification sur des *slices* plutôt que sur le programme initial peut être intéressant, mais cela requiert une solide justification théorique, notamment en présence d'erreurs et de non-terminaisons, pour garantir la correction des résultats sans alourdir les *slices*.

Nous nous intéressons à la forme classique du *slicing*, dit *slicing* statique arrière. Ce *slicing* est classiquement basé sur deux relations de dépendance, de contrôle et de donnée. Nous nous donnons un langage impératif simple dit langage WHILE contenant les instructions suivantes : `skip`, `x:=e`, `if`, `while`, avec des expressions usuelles et des tableaux de taille fixe. La propriété de correction classique du *slicing* sur ce langage qui ne contient pas d'erreurs, formalisée avec une sémantique à base de trajectoires, est la suivante : si un programme termine sur un état initial donné, alors sa *slice* termine aussi sur cet état et les deux exécutions s'accordent après chaque instruction préservée dans la *slice* sur la valeur des variables apparaissant dans cette instruction. Formellement, on définit la notion de projection, et ce théorème stipule que les projections des trajectoires du programme initial et de la *slice* sont égales. Ce résultat ne donne pas d'information en présence d'erreurs et de non-terminaisons. Tient-il dans ce cadre élargi ? On montre facilement sur deux contre-exemples que non. Dans le premier, une instruction provoque une erreur dans le programme initial et n'est pas préservée dans la *slice*. L'exécution de la *slice* ne peut évidemment pas échouer sur la même instruction que le programme initial, puisque cette instruction n'est pas dans la *slice*. Les deux exécutions peuvent donc diverger. Dans le deuxième, une instruction boucle infiniment dans le programme initial et n'est pas préservée dans la *slice*. Comme pour le premier contre-exemple, la *slice* n'exécute pas cette instruction, et son exécution diverge donc de celle du programme initial.

Approche et contributions. Pour généraliser la propriété de correction aux programmes pouvant provoquer des erreurs ou des non-terminaisons, nous choisissons de conserver des relations de dépendance similaires, et d'affaiblir la propriété de correction, ce que nous appelons le *slicing* relaxé. Nous élargissons notre langage en ajoutant l'instruction `assert` pour expliciter les erreurs à l'exécution. Nous faisons de plus l'hypothèse que les erreurs ne peuvent survenir que dans des assertions, les potentielles erreurs étant protégées par des assertions. Nous ajoutons une nouvelle relation de dépendance, dite dépendance d'assertion, de façon à ce que les instructions menaçantes ne puissent pas être conservées dans la *slice* sans les assertions qui les protègent. En dehors de cette nouvelle relation, les dépendances de contrôle et de donnée sont inchangées. Nous établissons une nouvelle propriété de correction, qui stipule que la projection de la trajectoire du programme initial est un préfixe de celle de la *slice* dans le cas général, et que les deux projections sont égales en l'absence d'erreurs et de non-terminaisons. Pour la vérification, nous déduisons deux théorèmes de cette propriété. Premièrement, si la *slice* ne provoque pas d'erreurs, les instructions du programme initial préservées dans la *slice* ne provoquent pas d'erreurs non plus. Deuxièmement, si la *slice* provoque une erreur, alors soit la même instruction produit une erreur dans le programme initial, soit une erreur ou une non-terminaison provoquée par une instruction non préservée la cache. Les deux derniers cas correspondent aux deux contre-exemples cités plus haut.

Tous les résultats de ce travail ont été formalisés et prouvés dans l'assistant de preuve Coq pour un langage similaire. Cette formalisation permet l'extraction en OCaml d'un *slicer* certifié pour ce langage. Le développement Coq est disponible sur <http://perso.ecp.fr/~lechenetjc/slicing/>.

*Cet article est un résumé étendu de l'article *Cut Branches Before Looking for Bugs: Sound Verification on Relaxed Slices* accepté à la conférence FASE 2016.

Outillage pour la modélisation, la vérification et la génération d'applications temporisées et embarquées

Pierre-Emmanuel Hladik¹, Silvano Dal Zilio¹, Olivier Pasquier², Sébastien Pillement² et Bernard Berthomieu¹

¹LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France

²Lunam Université, Université de Nantes, UMR CNRS 6164, Institut d'Électronique et de Télécommunications de Rennes (IETR), Polytech Nantes, France
{bernard,dalzilio,pehladik}@laas.fr, {olivier.pasquier, Sebastien.Pillement}@univ-nantes.fr

Résumé

Cet article présente un travail en cours pour mettre en place une chaîne d'outils dédiée à la conception, la vérification et l'exécution de systèmes embarqués temps réel. Ce travail se base sur la méthode MCSE et les modèles qu'elle préconise pour la description d'applications. Une traduction du modèle dans le langage formel Fiacre est appliquée pour ensuite vérifier le système à l'aide du model-checker Tina. Afin de faciliter cette analyse et la génération d'un exécutif, la notion de *Logical Execution Time* est utilisée pour décrire le comportement temporel. Nous présentons ces différentes méthodes et outils avant d'exposer l'état d'avancement des différents composants de la chaîne.

1 Introduction

Le projet présenté dans cet article est en cours de réalisation et vise la mise en place d'une chaîne d'outils pour la modélisation, la vérification et la génération d'applications embarquées temps réel. Une des motivations principales de ce travail est d'intégrer une approche formelle basée sur le model-checking dans une démarche de conception orientée modèles. Ce travail porte sur la branche descendante du cycle de développement classique, dit en V, c'est-à-dire de la spécification des besoins à l'implantation. Cette réalisation devrait déboucher sur un outil facile à prendre en main et pouvant être employé, dans un premier temps, pour des projets d'enseignement ou comme démonstrateur.

L'approche développée dans ce projet a pour cadre la méthode MCSE, proposée par Jean-Paul Calvez dans les années 1990 [4]. Elle offre l'avantage, par rapport à d'autres approches, d'être simple, de couvrir complètement le cycle de développement et d'avoir un langage de modélisation avec une syntaxe restreinte, mais expressive. Une partie de la chaîne logicielle repose sur la boîte à outils Tina [2], qui est utilisée comme outil de vérification formelle, mais aussi comme format intermédiaire pour l'exécution.

L'étude présentée dans cet article se focalise uniquement sur la génération d'un exécutable logiciel à partir d'un modèle décrit dans le langage MCSE. Ce travail est une première étape pour aborder des problématiques d'aide à la conception conjointe matérielle-logicielle de systèmes embarqués, c.-à-d. des méthodes pour aider un concepteur à définir l'architecture d'un système en répartissant les composants entre le matériel et le logiciel et tout en garantissant les exigences fonctionnelles et non-fonctionnelles.

Cet article est organisé de la manière suivante : la section 2 introduit la méthodologie MCSE ; la section 3 présente les éléments qui composent la chaîne d'outils ; la section 4 s'attarde sur les hypothèses sous-jacentes à la machine d'exécution ; la section 5 fait le point sur les avancées et l'état du développement des éléments de la chaîne ; et pour finir la section 6 conclut le travail et ouvre sur les suites envisagées.

2 La méthodologie MCSE

Le processus de développement d'un systèmes embarqués peut être décrit de manière classique par un cycle en V. La conception y est considérée comme un raffinement des composants du système dont le résultat sert de référence pour le programmeur (ou l'outil) en charge d'écrire le code spécifique à la plate-forme d'exécution. La branche montante du cycle est consacrée à l'intégration des composants du systèmes. À cela s'ajoute des étapes de vérification et de validation par des tests lors des phases en aval et par des méthodes ad-hoc pendant la conception. Ces dernières activités sont particulièrement importantes afin de corriger au plus tôt les éventuelles erreurs de conception.

Pour mener à bien ces étapes de développement, il est nécessaire de disposer d'un cadre méthodologique. De nombreuses propositions existent dans le domaine des systèmes embarqués depuis les années 1970 telles que SADT [10] ou plus récemment ARCADIA [9]. Au tournant des années 1990, Jean-Paul Calvez a proposé dans [4] une méthode baptisée Méthodologie de Conception des Systèmes Embarqués (MCSE). C'est une méthode de conception descendante appliquée aux systèmes embarqués qui couvre les phases de spécification, de conception, d'implantation et de validation. Cette méthode est supportée par CoFluent Studio¹ racheté en 2011 par Intel, pour la modélisation et la simulation de systèmes électroniques embarqués.

Conjointement à la méthode MCSE, un langage spécifique de modélisation à base de composants est défini et permet de couvrir les dimensions parallèles et séquentielles d'une application. La dimension parallèle est liée à la structure fonctionnelle basée sur quatre types d'éléments : la fonction et les trois relations possibles que sont l'événement, la variable partagée et la file de messages. La dimension séquentielle est liée au comportement de chaque fonction qui est basé sur des éléments tels que l'opération, la boucle, le test, l'entrée et la sortie. À ces éléments peut être associée une notion de temps.

La sémantique du langage MCSE n'est pas formellement définie et la méthode ne préconise pas des techniques spécifiques de vérification ou de validation. Par contre, la méthode est particulièrement intéressante de part son caractère efficace et dédié à la conception des systèmes embarqués. Ainsi, sa syntaxe réduite permet de raisonnablement proposer un outils couvrant la totalité du modèle et d'en fixer la sémantique formellement.

3 Chaîne d'outils

Le travail présenté ici n'aborde que la partie vérification et génération de code pour des applications logicielles embarquées à partir d'un modèle fonctionnel décrit en MCSE. La vérification ne porte que sur des propriétés temporelles et comportementales du systèmes. Les aspects numériques tels que le domaine des valeurs ou la simulation des algorithmes sont traités par d'autres outils tels que ceux disponibles dans CoFluent.

1. <http://www.intel.com/content/www/us/en/cofluent/intel-cofluent-studio.html> [consulté 02/2016].

Afin de disposer d'un outillage adapté, nous nous appuyons sur la boîte à outils Tina [2]. Celle-ci est dédiée à la vérification formelle de systèmes décrits en réseaux de Petri temporels et offre des méthodes pour construire les graphes de comportements ainsi que des model-checkers pour les logiques temporelles LTL et CTL. Tina peut être couplé avec Fiacre [1], langage formel de plus haut niveau que les réseaux de Petri qui permet de modéliser des systèmes d'un point de vue comportemental et temporel. Plusieurs compilateurs sont proposés pour traduire un modèle Fiacre vers des outils d'analyse. Dans le cas de Tina, un modèle Fiacre est compilé par l'outil `frac` sous la forme d'un *Time Transition Systems* (TTS).

Le travail présenté dans cet article consiste à organiser, définir et produire les outils nécessaires pour réaliser la chaîne décrite par la figure 1. Cette chaîne s'inscrit dans la phase de conception d'un cycle de développement de MCSE et comprend les étapes suivantes : (i) le résultat de la conception du système est modélisé selon le modèle fonctionnel MCSE, (ii) le modèle fonctionnel est traduit en Fiacre, (iii) le modèle est ensuite compilé à l'aide de l'outil `frac` sous forme d'un TTS qui servira d'entrée à Tina ; (iv) le modèle est utilisé par Tina afin de vérifier les propriétés temporelles et comportementales du système ; (v) un générateur, baptisé `hippo`, produit un exécutable depuis le TTS garantissant la même sémantique que le modèle vérifié.

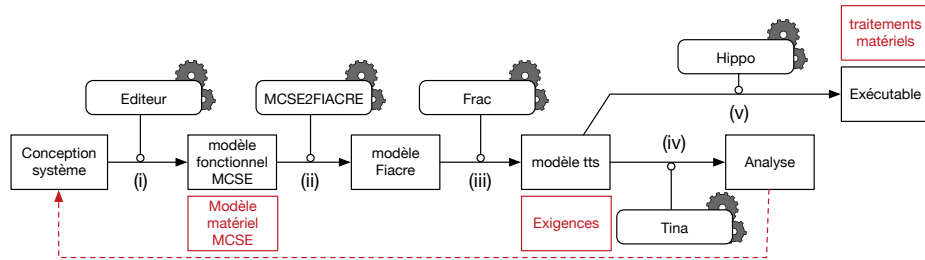


FIGURE 1 – Chaîne d'outils pour la conception, la vérification et l'exécution d'un système embarqué avec des contraintes temporelles. En rouge, les éléments non traités dans cet article.

Cette chaîne est construite suivant des principes classiques pour vérifier le comportement de systèmes temporels. L'intérêt de passer par Fiacre est double [1] : d'une part simplifier la traduction du modèle fonctionnel MCSE en utilisant les notions de processus, de synchronisation et de composants offertes par Fiacre (que l'on ne retrouve pas dans les formalismes bas-niveau comme les réseaux de Petri ou les automates), et d'autre part de permettre l'utilisation d'outils de vérification autres que Tina sans avoir besoin de redéfinir la traduction.

Une fois cette chaîne réalisée, ce travail sera complété par d'autres outils pour par exemple (voir éléments en rouge sur la figure 1) : remonter les résultats de l'analyse au niveau du modèle MCSE, exprimer et traduire automatiquement les exigences en propriétés vérifiables, prendre en considération les composants matérielles du système dans l'analyse et la génération.

Actuellement, les outils de vérification sont disponibles et matures, c'est-à-dire que la partie (iii)-(iv) de la chaîne existe et est outillée². Les étapes (i) et (ii) suivent une démarche basée sur l'ingénierie des modèles avec la définition d'un méta-modèle du langage MCSE afin de produire un éditeur sous la forme d'un plugin Eclipse à l'aide du projet Sirius². Ce méta-modèle est aussi employé lors de l'étape (ii) afin d'écrire une transformation entre les modèles MCSE et Fiacre. La génération de code de l'étape (v) est détaillée dans la section suivante.

2. Les outils liés à Tina sont disponibles sur <http://projects.laas.fr/tina/> et ceux permettant de compiler du Fiacre sur <http://projects.laas.fr/fiacre/>.

2. Sirius est un projet soutenu par la fondation Eclipse et dédié à la conception d'interfaces graphiques. <https://projects.eclipse.org/projects/modeling.sirius> [consulté en février 2016]

4 Hypothèses d'exécution

La génération d'un exécutable depuis un modèle est un problème largement traité dans la littérature du génie logiciel. Cependant, au delà des méthodes issues du développement logiciel, il existe des problématiques propres aux systèmes embarqués, en particulier pour rester fidèles aux contraintes temporelles exprimées dans le modèle.

Une de ces problématiques est la différence sémantique pouvant exister entre le modèle produit par le concepteur, le modèle vérifié et l'exécutable. Les travaux de Mathias Brun [3] et de Cédric Lelionnais [7] exposent clairement les différentes approches possibles. Dans notre cas, nous générons un code à partir du modèle le plus proche de celui vérifié, à savoir le TTS. Le support d'exécution n'est alors qu'une machine virtuelle pouvant exécuter le format TTS et garantissant ainsi le respect de la sémantique. Cette approche est similaire à ce qu'offre, par exemple, BIP [11] pour des systèmes non temporisés.

Un second point spécifique à l'exécution d'un modèle temporisé est la prise en charge du temps, aussi bien au niveau des modèles vérifiés que de l'exécutable. Classiquement, deux approches sont distinguées en fonction de la manière dont l'écoulement du temps est considéré et du moment de la prise en considération des événements, les démarches synchrone et asynchrone [12].

Il existe une troisième approche, dite *asynchrone conduite par le temps* (*Time-Triggered*) et qui sépare les aspects logiques (fonctionnalité et chronométrage) des aspects physiques (placement et ordonnancement). La production des événements et leur prise en compte par le système se fait à des instants connus sous l'hypothèse synchrone (temps logiquement nul), mais à la différence des approches synchrones, les traitements ne sont pas supposés se réaliser entre chaque instant. Christoph Kirsch et Raja Sengupta dans [6] décrivent ce principe à l'aide du *Logical Execution Time* (LET), qui est la durée séparant l'instant où un traitement peut commencer à s'exécuter et la date où il doit produire son résultat. Ainsi la date de production des données est parfaitement connue et ne dépend pas de la durée effective du traitement. Il est par contre nécessaire lors de la génération du système exécutable de s'assurer que l'exécution d'un traitement respecte bien son LET. Le *framework* pour exécutifs temps réel OASIS [8] est un exemple mettant en œuvre cette démarche tout comme Giotto [5] ou Ptide [13].

Cette approche est intéressante pour deux raisons : (i) l'abstraction du temps physique facilite la vérification du comportement d'un système, en particulier en ignorant la préemption ; (ii) la notion de LET offre plus de souplesse que l'approche purement synchrone. Par contre, l'ordonnancement du système doit d'être vérifié lors de la génération afin de garantir que les traitements respectent bien leur LET.

La sémantique d'exécution d'un modèle fonctionnel MCSE est définie comme étant séquentielle au sein de chaque fonction et concurrentes entre les fonctions. Au niveau du temps, les synchronisations (attente, message) l'écriture de données partagées et les éléments de contrôle (boucle, condition, etc.) sont supposés à temps nuls (équivalent de l'hypothèse synchrone) et les *opérations* (équivalent à un traitement) sont associées à une durée.

L'hypothèse LET est alors particulièrement adaptée pour gérer et vérifier le modèle MCSE, mais nécessite une légère adaptation de son comportement. Pour cela, il est nécessaire d'ajouter une échéance relative à chaque opération (c.-à-d. la date à laquelle l'opération doit être terminée en fonction de sa date de démarrage) en plus de sa durée d'exécution. La sémantique d'exécution de MCSE est ainsi alors modifiée en introduisant une lecture et une écriture (c'est ce dernier point qui est nouveau) des *opérations* à des instants parfaitement définis et donc vérifiables.

5 Outils et travaux en cours

Plusieurs outils de la chaîne sont encore au stade de développement ou de preuve de concept. Nous donnons dans cette section un aperçu rapide des solutions envisagées et de l'avancée des travaux.

Méta-modèle MCSE. Le premier élément de la chaîne a pour objectif de saisir le modèle de l'application en se basant sur une approche multi-fonctions (parallélisme) et comportemental (séquentiel) avec notion de temps. Il est en cours de développement à l'aide de l'environnement Sirius. Il s'agit de redéfinir un méta-modèle du modèle fonctionnel préconisé par la méthodologie MCSE. L'objectif de ce travail est d'avoir un méta-modèle ouvert mais en aucun cas de concurrencer les possibilités de CoFluent Studio qui en plus permet de faire de la simulation fonctionnelle et de l'extraction de performances en générant un code basé sur SystemC.

Transformation du modèle MCSE vers Fiacre. Cette transformation couvre tous les aspects comportementaux du modèle fonctionnel de MCSE ce qui permet ainsi d'en fixer formellement la sémantique. Un méta-modèle de Fiacre, en plus de celui de MCSE, a été produit et les règles de transformations ont été écrites à l'aide du langage ATL. Cependant, le prototype existant doit être revu pour prendre en considération les hypothèses sur l'exécution ainsi que les dernières modifications du méta-modèle MCSE. Par la suite, il serait aussi intéressant d'assurer, lors de cette transformation, une traçabilité des propriétés temporelles et comportementales afin de pouvoir faire remonter les éventuels échecs au niveau du modèle manipulé par le concepteur (cette traçabilité existe déjà entre le TTS et Fiacre).

Moteur d'exécution. Pour une question de place, nous n'entrerons pas dans les détails techniques de la machine assurant l'exécution du TTS. La version prototype est basée sur un Linux patché temps réel (Xenomai) pour gérer l'ordonnancement des traitements et pour avoir accès à des mécanismes qui garantissent la cohérence des états aux instants logiques. La machine d'exécution est alors une simple couche intermédiaire entre le système d'exploitation et les traitements applicatifs (idem aux couches OASIS et Giotto). Son rôle est de contrôler l'exécution des threads qui encapsulent les traitements et de faire avancer l'état du TTS en garantissant sa cohérence. Une version statique au niveau mémoire est implantée et sera portée sur un microcontrôleur sous OSEK afin d'en évaluer les surcoûts temporels et mémoriels.

La prise en considération d'événements asynchrones extérieurs au système a aussi été implantée, mais nécessite une étude plus approfondie pour valider l'architecture retenue. Cela rejoint les difficultés pour modéliser et vérifier un système dans son environnement.

Analyse du Logical Execution Time. La notion de LET attachée aux traitements impose de vérifier le comportement temporel du système du point de vue de l'ordonnabilité. Actuellement cet aspect n'a pas été traité, mais plusieurs pistes sont envisagées pour attaquer ce point. Le comportement temporel des traitements étant bien spécifié dans le modèle, il est possible d'utiliser les approches dites analytiques pour évaluer l'ordonnabilité. Un autre moyen serait de conduire l'analyse par model-checking en ajoutant des motifs propres au comportement de l'ordonnanceur dans le modèle Fiacre ou dans le TTS ; mais cela est au risque d'une explosion du nombre d'états du système.

6 Conclusion

Cet article présente un travail en cours sur la production d'une chaîne outillée pour la modélisation, la vérification et la génération d'applications embarquées temps réel. La méthodologie MCSE sert de cadre à cette étude et guide la démarche mise en place. Nous avons montré que certains outils sont matures, comme ceux pour la vérification, alors que d'autres sont encore au stade de prototype (éditeur, transformation de modèle et machine d'exécution) voire de projet (validation de l'ordonnancement).

Ce travail permettra à terme de disposer d'une chaîne complète pour générer un exécutable temps réel. Cet outillage aura l'avantage d'être inscrit dans un processus de conception et d'assurer la cohérence entre les outils. À plus long terme, cette chaîne servira de base pour aborder d'autres problèmes telle que la gestion des propriétés (traçabilité et sémantique) ou la modélisation de l'environnement.

Références

- [1] B. BERTHOMIEU, J.-P. BODEVEIX, P. FARAIL, M. FILALI, H. GARAVEL, P. GAUFILLET, F. LANG et F. VERNADAT : Fiacre : an intermediate language for model verification in the topcased environment. *In ERTS 2008*, 2008.
- [2] B. BERTHOMIEU, P.-O. RIBET et F. VERNADAT : The tool TINA – construction of abstract state spaces for Petri nets and time petri nets. *Int. Journal of Production Research*, 2004.
- [3] M. BRUN : *Contribution à la considération explicite des plates-formes d'exécution logicielles lors d'un processus de déploiement d'application*. Thèse de doctorat, Univ. de Nantes, 2010.
- [4] J.-P. CALVEZ : *Spécification et Conception des Systèmes : une Méthodologie*. Masson, 1990.
- [5] T. HENZINGER, B. HOROWITZ et C. KIRSCH : Giotto : a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1), 2003.
- [6] C. KIRSCH et R. SENGUPTA : *Handbook of Real-Time and Embedded Systems*, chapitre The Evolution of Real-Time Programming. Chapman and Hall/CRC, 2007.
- [7] C. LELIONNAIS, J. DELATOUR, M. BRUN, O. Henri ROUX et C. SEIDNER : Formal synthesis of real-time system models in a MDE approach. *International Journal on Advances in Systems and Measurements*, 7, 2014.
- [8] S. LOUISE, M. LEMERRE, C. AUSSAGUES et V. DAVID : The oasis kernel : A framework for high dependability real-time systems. *In IEEE HASE*, 2011.
- [9] P. ROQUES : Mbse with the arcadia method and the capella tool. *In 8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.
- [10] D. T. ROSS : *Applications and extensions of SADT : structured analysis and design technique*. SoftTech, 1984.
- [11] J. SIFAKIS : A framework for component-based construction. *In Third IEEE International Conference on Software Engineering and Formal Methods*, 2005.
- [12] F. SIMONOT-LION et Y. TRINQUET : *Encyclopédie de l'informatique et des systèmes d'information*, chapitre Exemples de systèmes temps réel et choix d'implémentation. Vuibert, 2005.
- [13] J. ZOU : *From Ptides to PtidyOS, Designing Distributed Real-Time Embedded Systems*. Thèse de doctorat, UC Berkeley, 2011.

Identification de propriétés pour la validation d'un système cyber-physique médical (Session doctorant)

Yoann Blein

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

CNRS, LIG, F-38000 Grenoble, France

yoann.blein@imag.fr

Résumé

On s'intéresse ici à la validation de systèmes cyber-physiques médicaux (SCPM). Plus précisément, on cherche à établir la conformité de traces d'exécution par rapport des propriétés données. Ces propriétés seront exprimées par des ingénieurs dans un langage dédié (DSL) de haut niveau et adapté aux besoins. Il est critique de trouver le juste niveau d'abstraction pour ce langage afin qu'il soit adoptable par nos partenaires industriels.

Cet article rapporte sur les premières étapes de la définition de ce langage dédié. À partir d'une étude réelle de SCPM (pose assistée de prothèse totale du genou), nous identifions les besoins des industriels du domaine, ainsi que les propriétés qu'ils souhaiteraient vérifier sur ce système. Un premier résultat de cette étude est l'identification de trois familles de propriétés de natures radicalement différentes : elles portent sur le comportement du système, sur l'interface homme-machine et les informations affichées par le système, et enfin sur des propriétés de géométrie 3D.

1 Introduction

L'utilisation de systèmes cyber-physiques médicaux (SCPM) est de plus en plus fréquente dans les hôpitaux et les cliniques. Ces appareils ont pour but de faciliter, voire automatiser, le traitement d'un patient. Par exemple, la pompe à infusion administre une solution dans le système sanguin d'un patient, et les assistants de chirurgie permettent d'augmenter la fiabilité des gestes opératoires d'un chirurgien. Même s'ils peuvent avoir des modes de fonctionnement très différents, les SCPM ont pour nature même d'intervenir sur l'organisme d'un patient. De ce fait, leur sûreté de fonctionnement est critique [4].

Dans le cadre du projet ANR MODMED, nous avons l'opportunité d'étudier un SCPM réel permettant d'effectuer la pose assistée d'une prothèse totale du genou. On dispose notamment des traces d'exécution produites par ce système sur un nombre important de chirurgies. Le projet MODMED souhaite concevoir un outil permettant de s'assurer que les exécutions d'un système, reflétées par ses traces, vérifient certaines propriétés.

2 Étude de cas : l'arthroplastie totale du genou assistée par ordinateur

Dans ce travail, nous étudions un système de guidage assisté par ordinateur pour l'arthroplastie totale du genou : ExatechGPS, conçu conjointement par les sociétés Exatech et BlueOrtho. L'arthroplastie totale du genou est une intervention chirurgicale qui consiste à remplacer certaines parties de l'articulation du genou par une prothèse. Le but de cette opération est de soulager la douleur causée par un genou arthritique, tout en maintenant ou améliorant sa fonctionnalité. Afin d'installer la prothèse, il est nécessaire de couper une partie du tibia et du fémur. ExatechGPS aide le chirurgien à réaliser ces

coupes précisément grâce à l'installation pas à pas de guides de coupe dans la *bonne* position. Cette position est déterminée en combinant l'objectif cible donné par le chirurgien et la reconstitution spatiale de la scène établie par le système. Il est actuellement utilisé à l'échelle mondiale.

ExatechGPS est une solution clé en main qui fournit à la fois la prothèse du genou et le système de guidage. Ce dernier comprend plusieurs composants : une machine capable de communiquer avec le chirurgien via un écran tactile, une caméra trois dimensions, un ensemble de traqueurs visibles par cette caméra et un ensemble d'instruments mécaniques permettant de fixer les traqueurs et les guides de coupe sur le tibia et le fémur.

L'installation des guides de coupes est réalisée au travers d'une succession d'étapes que doit effectuer le chirurgien. La nature de ces étapes ainsi que l'ordre dans lequel elles sont réalisées sont, dans une certaine mesure, configurables. Cette configuration, appelée *profil*, est déterminée avec le chirurgien selon ses préférences opératoires. Dans tous les cas, la séquence des étapes prend la forme macroscopique suivante : calibration des capteurs, acquisition de points anatomiques, vérification des acquisitions, ajustement des paramètres cibles, et enfin, réglage des guides de coupe.

ExatechGPS est équipé d'un système d'enregistrement de trace d'exécution. Pour toute chirurgie effectuée, la trace d'exécution correspondante est envoyée à BlueOrtho. L'entreprise dispose ainsi d'un corpus d'environ 4500 traces de chirurgies ayant eu lieu dans des conditions réelles. Chaque trace se compose d'un journal d'événements, d'une description hiérarchique des étapes de la chirurgie contenant les valeurs acquises et calculées, et de l'ensemble des captures d'écran réalisées pour chaque étape. Cet ensemble d'informations permet de comprendre a posteriori le déroulement d'une chirurgie et éventuellement d'identifier des défaillances.

BlueOrtho souhaiterait exploiter le corpus des traces accumulées sur plusieurs années pour attester la robustesse du logiciel et valider son utilisation dans des environnements non contrôlés. En effet, pendant le développement du produit, un certain nombre d'hypothèses ont été faites à la fois sur son environnement et sur son utilisation. Ces hypothèses sont aujourd'hui traduites dans des recommandations d'utilisation. Afin d'entraver le moins possible le travail du chirurgien en le laissant poursuivre une opération dans des conditions anormales, les développeurs ont choisi que le système soit tolérant au non-respect des recommandations. Concrètement, ces hypothèses peuvent être traduites en propriétés que les traces devraient satisfaire.

Afin de vérifier automatiquement des propriétés sur un ensemble de traces, il est nécessaire de formaliser ces propriétés. Pour cela, on veut déterminer un langage dédié de haut niveau qui est

- adapté au domaine des SCPM ;
- facile à apprendre et manipuler afin d'augmenter ses chances d'adoption dans l'industrie ;
- et qui permet d'exprimer des propriétés vérifiables sur des traces finies.

Un enjeu important est de trouver le juste niveau d'abstraction pour ce langage dédié : il doit être pratique à utiliser et peu sujet aux erreurs, tout en ayant une expressivité suffisante pour décrire des propriétés intéressantes.

3 Travaux connexes sur la formalisation de propriétés

On distingue deux types de formalismes dédiés à la spécification formelle de propriétés. D'une part, il existe les formalismes de bas niveau qui ont une sémantique non ambiguë mais qui sont difficiles à manipuler. On peut citer la logique temporelle linéaire proposée par Pnueli [6] ou des extensions plus récentes de celle-ci [2, 5]. On pense aussi aux travaux modernes de Barringer *et al.* [1] où les propriétés sont spécifiées via des automates expressifs.

D'autre part, nous avons les langages dédiés de haut niveau dont le but est de simplifier la lecture et l'écriture de propriétés. La sémantique de ces langages est souvent définie par rapport à un formalisme de la catégorie précédente. On pense notamment aux travaux initiaux de Dwyer *et al.* sur les patrons de propriétés [3] ou un raffinement de ceux-ci comme proposé par Smith *et al.* [7].

4 Conclusion et perspectives

Le travail que je mène actuellement vise à déterminer un langage dédié pour la formalisation des propriétés attendues. Il consiste à étudier les documents de spécification et le plan de test détaillé pour identifier les propriétés pertinentes et les exprimer dans différents langages (LTL, QEA, Dwyer, ...). À ce jour, trois classes de propriétés distinctes ont été identifiées :

1. Des propriétés portant sur le comportement du système, c'est-à-dire sur la séquence d'événements internes du logiciel. Par exemple « la phase d'acquisition a toujours lieu après la phase de calibration des capteurs » ;
2. Des propriétés portant sur l'interface homme-machine et plus particulièrement sur les informations affichées et les actions disponibles. Par exemple « l'utilisateur doit toujours pouvoir revenir à l'étape précédente » (sur toutes les captures d'écran présentes dans une trace, le bouton « précédent » est présent) ;
3. Des propriétés portant sur la géométrie de la scène 3D construite par le système à partir des acquisitions. Ces propriétés permettent notamment de valider l'utilisation du système. Par exemple, « les traqueurs ne sont jamais exploités en dehors du champ recommandé par rapport à la caméra » (la précision de la caméra n'est pas garantie en dehors de cette zone).

Finalement, nous avons également observé des propriétés formées par combinaisons de ces trois classes, telle que « l'utilisateur doit refaire l'acquisition d'un nuage de points si sa qualité ne paraît pas assez bonne ». La première partie correspond à un événement observé par le système (classe 1) et la seconde à un critère géométrique (classe 3). La suite de ce travail comportera une analyse qualitative de la lisibilité et l'utilisabilité de ces langages par les ingénieurs de développement.

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

Références

- [1] H. Barringer, Y. Falcone, K. Havelund, G. Regeer, and D. E. Rydeheard. Quantified event automata : Towards expressive and efficient runtime monitors. In *FM 2012 : Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, pages 68–84, 2012.
- [2] D. A. Basin, F. Klaedtke, S. Müller, and E. Zalinescu. Monitoring metric first-order temporal properties. *J. ACM*, 62(2) :15, 2015.
- [3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the 1999 International Conference on Software Engineering, ICSE'99, Los Angeles, CA, USA, May 16-22, 1999.*, pages 411–420, 1999.
- [4] E. A. Lee. Cyber physical systems : Design challenges. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 363–369, 2008.
- [5] R. Medhat, Y. Joshi, B. Bonakdarpour, and S. Fischmeister. Accelerated runtime verification of LTL specifications with counting semantics. *CoRR*, abs/1411.2239, 2014.
- [6] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57, 1977.
- [7] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil. PROPEL : an approach supporting property elucidation. In *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 11–21, 2002.

Comparaison des Approches SMT et CSP Appliquées à la Vérification de Réseaux Workflows

Hadrien Bride, Olga Kouchnarenko, Fabien Peureux et Guillaume Voiron

Institut Femto-st – UMR CNRS 6174, Université de Franche-Comté
16, route de Gray, 25030 Besançon, France
{hbride,okouchna,fpeureux,gvoiron}@femto-st.fr

Résumé

Cet article propose une évaluation et une comparaison de deux méthodes de résolution de systèmes de contraintes, SMT (Satisfiability Modulo Theory) et CSP (Constraint Satisfaction Problem), appliquées à la vérification de spécifications modales. Ce processus de vérification vise à garantir la conformité de processus métier, représentés sous la forme de réseaux de Petri de type Workflow, vis-à-vis de comportements nécessaires ou admissibles, modélisés par des formules modales.

1 Méthode de vérification

La comparaison des approches de résolution SMT et CSP a été menée sur la base de formules logiques du premier ordre produites par la méthode de vérification de spécifications modales présentée dans [1]. Pour remédier au problème d'atteignabilité EX-SPACE [4], cette méthode de vérification, qui s'applique à des réseaux de Petri Workflows, procède par étapes successives de raffinement de l'ensemble des exécutions valides du réseau à vérifier. Les constructions de ces raffinements s'appuient sur le calcul de sur-approximations et de sous-approximations qui sont évaluées à l'aide de systèmes de contraintes dédiés. Ce sont précisément ces systèmes qui ont été utilisés comme support à la comparaison des deux approches de résolution (SMT et CSP). La satisfiabilité de ces systèmes de contraintes permet de déterminer la validité ou l'invalidité de la spécification modale à vérifier.

2 Protocole expérimental

Afin de comparer les performances des approches SMT et CSP vis-à-vis de la méthode de vérification, un protocole expérimental a été défini. Un premier outil génère des réseaux Workflows d'une certaine classe et d'une certaine taille, ainsi qu'une spécification modale associée (de type *may* ou *must*, et dont la validité est connue). Un second outil dérive les systèmes de contraintes associés aux Workflows et aux spécifications obtenus, et calcule leur satisfiabilité à l'aide du solveur SMT *Z3* [3] et CSP *SICStus Prolog* [2].

3 Synthèse des résultats expérimentaux

La figure 1 présente un extrait des résultats obtenus vis-à-vis des temps de résolution des solveurs selon le nombre de nœuds des Workflows traités.

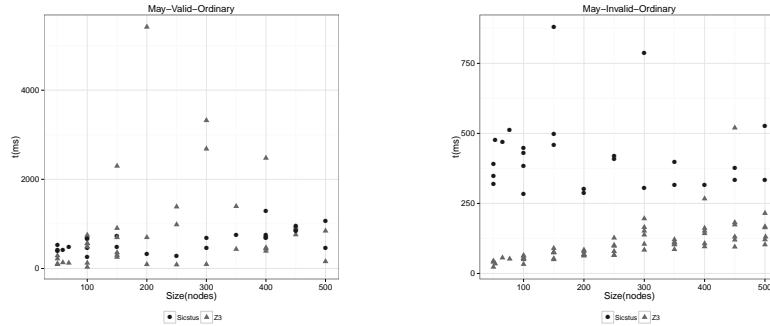


FIGURE 1 – Temps de résolution de spécifications *may* sur réseaux ordinaires

De manière globale, en terme de temps d'exécution, les expérimentations ont mis en évidence une meilleure efficacité de *Z3* sur les réseaux de type ordinaire, State-Machine, et Free-Choice. En revanche, *SICStus* a été globalement plus performant dans le cas particulier des Marked Graphs. Une étude avancée de ces résultats tend à conclure que ce phénomène est directement lié au nombre de points de choix induits par les systèmes de contraintes dérivés : plus le nombre de points de choix, à explorer potentiellement lors de la résolution, est élevé, plus *Z3* prend l'avantage sur *SICStus*.

Ces résultats ont également montré la capacité de passage à l'échelle de l'approche proposée dans [1], qui est capable de traiter des réseaux de grande taille (au moins jusqu'à 500 nœuds) de façon quasi instantanée.

4 Conclusion

Ce court article présente la synthèse d'une étude portant sur la comparaison des temps de résolution des solveurs *Z3* (SMT) et *SICStus* (CSP) pour vérifier des spécifications modales sur des réseaux Workflows. Les résultats obtenus tendent à montrer une supériorité du solveur *Z3* dès que le nombre potentiel de points de choix, à explorer pour résoudre les formules, augmente.

Références

- [1] H. Bride et al. Verifying modal workflow specifications using constraint solving. In *IFM'14*, LNCS, pages 171–186. Springer, 2014.
- [2] M. Carlsson et al. *SICStus Prolog user's manual (Release 4.2.3)*. Swedish Institute of Computer Science, Kista, Sweden, October 2012.
- [3] Leonardo De Moura and Nikolaj Bjørner. *Z3 : An efficient smt solver*. In *TACAS'08*, pages 337–340. Springer, 2008.
- [4] R.J. Lipton et al. *The reachability problem requires exponential space*. Research report. Department of Computer Science, Yale University, 1976.

Validation formelle d'implantation de patrons de sécurité: Application aux SCADA

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

Abstract

Les systèmes de contrôle et d'acquisition de données (SCADA) sont des systèmes qui contrôlent la majeure partie de nos infrastructures industrielles critiques comme, par exemple, des centrales nucléaires ou chimiques. De nombreux travaux se focalisent sur la conception de mécanismes architecturaux pour améliorer leur sécurité et leur résistance aux attaques. Ces architectures renforcées peuvent alors faire l'objet d'une modélisation logicielle composant les architectures et des modèles de patrons de sécurité. La combinaison entre une architecture SCADA et les patrons de sécurité doit alors être validée au regard des exigences de sécurité à implanter. Cet article rend compte de travaux en cours sur l'exploitation de techniques de vérification formelle des propriétés, par model-checking, peuvent aider à garantir la correction des modèles.

Mots Clés: Sécurité, Patron de Sécurité, Vérification formelle, Model Checking, SCADA

1 Introduction

Les systèmes de contrôle et d'acquisition de données (SCADA) posent aujourd'hui des défis pour les experts de sécurité. Compte tenu de leurs exigences, il est très difficile de parvenir à un bon niveau de sécurité dans les systèmes SCADA. De nombreux travaux [1] ont eu pour objectifs d'élaborer des solutions théoriques, des guides méthodologiques et recommandations, pour renforcer la sécurité en SCADA et la protection des services offerts.

Certains problèmes de sécurité étant très fréquents, des modèles de patrons de sécurité ont été proposés [2, 3] pour faciliter la conception des architectures. L'utilisation des patrons de sécurité nécessite de vérifier que les

Étude partiellement financée par la Direction Générale d'Armement - Maîtrise de l'Information (DGA-MI).

contre-mesures implantées dans les architectures sont fiables. Une contre-mesure est considérée fiable si elle résout un problème de sécurité sans affecter les autres exigences du système. L'objectif de notre travail est d'étudier les conditions et la méthodologie avec laquelle peuvent se composer, d'une manière optimale, des patrons et s'intégrer dans un modèle d'architecture pour répondre à une politique de sécurité donnée, en référence aux critères communs (ISO15408). Les techniques de la validation formelle telles que le *model – checking* (MC) [4, 5] sont une bonne opportunité pour mener ce processus et ils ont prouvé leur efficacité pour vérifier si un modèle de système est conforme à ses exigences.

Dans ce papier, nous présentons un travail de validation formelle d'architecture à base de *model – checking* et une illustration sur un exemple simple d'un système SCADA intégrant des modèles de patrons de sécurité. Nous utilisons le langage FIACRE¹ pour définir les comportements de notre système ainsi que ceux des attaquants. Nous prouvons que le modèle d'architecture, doté de patrons de sécurité, respecte les exigences avec l'outillage OBP². Nous formalisons nos exigences de sécurité avec le langage CDL [6] associé à cet outil.

2 Patrons de sécurité pour les modèles SCADA

De nombreux systèmes de type SCADA sont intégrés aujourd'hui dans les systèmes industriels et sont exploités pour contrôler et surveiller les différentes composantes de leurs infrastructures. Les dispositifs SCADA sont composés d'éléments répartis géographiquement ce qui les rend vulnérables face à des potentielles attaques. Certains liens de communications entre ces dispositifs peuvent être malicieusement corrompus. Le renouvellement et les mises à jour des composants impliquent une revalidation coûteuse des architectures. Pour des raisons économiques, les systèmes SCADA intègrent des matériels et logiciels de type *COTS* qui, pour la plupart, ne répondent pas aux exigences de sécurité. La modélisation des architectures, dotées de mécanismes de sécurité, peut permettre, d'une part, de faciliter la conception et la réutilisation des schémas de conception et, d'autre part, de vérifier formellement des propriétés de sécurité. Un patron de sécurité est une solution réutilisable à un problème de sécurité récurrent. Il fournit des lignes directrices détaillées sur la façon de mettre en œuvre une solution architecturale pour un problème de sécurité spécifique. Dans la littérature, des patrons de sécurité ont été proposés et sont décrits dans divers formats [7, 8]. Dans cet article, nous nous intéressons uniquement à deux patrons (*Autorization* et *CheckPoint*) [3, 7] qui proposent de résoudre le problème des droits d'accès entre un sujet (ex. utilisateur) et un objet (ex. infor-

¹Défini lors du projet TopCased (<http://www.topcased.org>).

²OBP est accessible librement sur <http://www.obpcdl.org>.

mation confidentielle). Le processus de validation formelle de l'architecture consiste à réaliser un modèle qui représente l'architecture du système ou une abstraction de celui-ci, et à formaliser les propriétés encodant les exigences de ce système. Le modèle, soumis à des comportements d'attaquants, est ensuite exploré, par simulation exhaustive, et les propriétés sont vérifiées lors de l'exploration de tous ses comportements possibles.

3 Cas d'étude illustratif

Notre cas d'étude concerne une centrale électrique qui répond aux besoins d'énergie d'une ville. Cette centrale est composée, de manière simplifiée, d'un serveur et de 3 secteurs dont chacune est dirigée par un "Programmable Logic Controller" (PLC). Le serveur agit aux changements des besoins d'électricité en changeant le niveau de travail de la centrale. Ces changements sont traduits par des messages envoyés par le serveur aux 3 PLC pour que chacun modifie le niveau de travail du secteur associé. La ville spécifie ses besoins en électricité, indépendamment de la production, sous la forme d'une valeur de niveau. Dans notre modèle, le niveau, correspondant au besoin exprimé, s'incrémente ou se décrémente d'une façon non déterministe et d'une valeur δ . La production d'électricité, quant à elle, s'incrémente ou se décrémente par pas de $3 \times \delta$ selon les besoins exprimés par la ville. Une contrainte est que le niveau de production doit être toujours supérieur ou égal au niveau de la consommation (*Exigence 1*). Pour éviter toute surproduction, le niveau de production ne doit pas être supérieur de $3 \times \delta$ au niveau de consommation (*Exigence 2*). La ville communique son besoin courant selon un protocole simple. Elle envoie le niveau de consommation à l'aide d'un message à la centrale. La centrale vérifie alors la nécessité de changement de production. Si un changement est nécessaire, un message est envoyé à tous les PLC pour modifier leur niveau de fonctionnement.

Le protocole d'échange entre la ville et la centrale peut subir des attaques. Pour le sécuriser, l'architecture est alors renforcée par les éléments architecturaux basés sur les patrons de sécurité de type *Autorization* et de type *CheckPoint*. Le patron d'autorisation nécessite l'utilisation du patron *CheckPoint* pour contrôler l'accès à la fonction d'autorisation. Le principe général des contrôles implantés par ces mécanismes de sécurité est le suivant: Avant l'envoi d'un message provenant d'une entité A (cf figure 1), le système contrôle que le récepteur B a le droit de lire la donnée émise. Pour cela, A émet un message *AreqChgeB* à l'entité *CheckPoint*₁ qui émet la demande d'autorisation $R(sB, sA, read)$ (signifiant : B a-t-il le droit de lire des informations de A ?) vers l'entité *Authorization*₁. Celle-ci renvoie *ok* à *CheckPoint*₁ si l'autorisation est accordée qui relaie l'ordre *AreqChgeB* en envoyant *AchgeB* à B . Après la réception d'un message, le système contrôle que l'envoyeur A possède le droit de modifier les données

détenues par le récepteur B . Pour cela, B émet un message $AreqChgeB$ à l'entité $CheckPoint_2$ qui émet la demande d'autorisation $R(sA, sB, write)$ (signifiant : A a-t-il le droit de modifier des informations de B ?) vers l'entité $Authorization_2$. Celle-ci renvoie ok à $CheckPoint_2$ si l'autorisation est accordée qui relaie l'ordre $AreqChgeB$ en envoyant $AchgeB$ à B . Au delà de l'intégration des patrons de sécurité déployés sur une architecture pour sécuriser des protocoles de communication, le chiffrement des messages peut être complémentaire. Mais dans notre travail, nous nous focalisons sur l'apport des patrons sur les exigences de sécurité et leur impact, d'une part, sur la préservation de la sûreté de fonctionnement, et, d'autre part, leur impact sur la complexité. Le modèle que nous expérimentons à une complexité d'une vingtaine d'automates avec 2 à 3 états chacun.

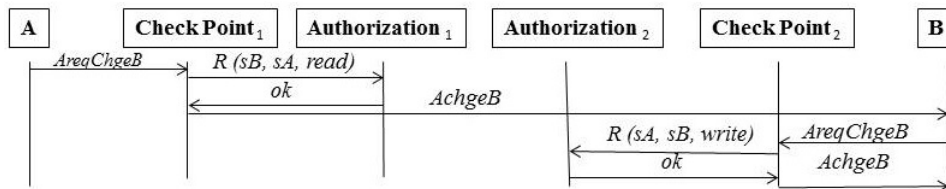


Figure 1: Diagramme de séquence d'un protocole sécurisé

4 Tests et Résultats

Une fois le modèle d'architecture du système conçu, intégrant les patrons de sécurité, nous le validons en vérifiant formellement les propriétés. Dans notre cas simplifié, les exigences *Exigence 1* et *Exigence 2* sont spécifiées en langage CDL suivant les principes suivants³: Une propriété CDL *pty*, de type observateur, fait référence à des occurrences d'événements *ev*. La propriété **property** *pty is : start -- // ev / - > reject* spécifie que l'observateur est mis en défaut (*faux*) si l'occurrence de l'événement *ev* est détectée lors de l'exploration du modèle. L'événement *ev* peut être défini par un changement d'état d'un prédicat *pred* comme par exemple: **event** *ev is : pred becomes true*. Enfin, le prédicat *pred* est défini de la façon suivante: **predicate** *pred is : x = 1* qui spécifie que *pred* est évalué à *vrai* si la variable *x* est égale à la valeur 1. Selon cette syntaxe, les 2 exigences de notre modèle sont spécifiées comme suit:

```

predicate pred1 is: consumption > production
event ev is: pred1 becomes true
property pty1 is: start -- // ev1 / -> reject    (Exigence 1)
predicate pred2 is: production - consumption >= 3 x DELTA.
event ev2 is: pred2 becomes true
property pty2 is: start -- // ev2 / -> reject    (Exigence 2)
    
```

³La documentation du langage CDL est accessible sur <http://www.obpcdl.org>.

Le seul cas où on accepte que ces exigences ne seront pas assurées, c'est quand la consommation vient de changer et un message vient d'être envoyé de la ville au serveur. Alors les prédicats sont modifiés de la manière suivante:

```
predicate pred1 is: consumption > production and
    (cityBwidth.length = 0 or cityBwidth[0].SenderID != IDcity)
predicate pred2 is: production - consumption < 3 x DELTA and
    (cityBwidth.length = 0 or cityBwidth[0].SenderID != IDcity)
```

La validation du modèle est exécutée selon 4 modes différents qui correspondent à 4 modèles différents : *NM* (*Normal Mode*): le modèle, sans mécanisme de sécurité, ne comporte pas d'attaquant. *NMUA* (*Normal Mode Under Attack*): un attaquant⁴ est ajouté au modèle pour mettre en évidence une faille de sécurité. *SM* (*Secured Mode*): les patrons de sécurité sont intégrés dans le modèle sans attaquant. *SMUA* (*Secured Mode Under Attack*): ajout d'un attaquant sur le modèle précédent pour vérifier l'efficacité des patrons de sécurité. Nous présentons dans la table 1 les résultats des explorations avec l'outil *OBP* pour les différents modèles en nombre de configurations et transitions explorées.

	NM	NMUA	SM	SMUA
nb. des configurations	319	3316	319	1515
nb. des transitions	460	7809	460	3952
Propriétés violées	-	2	-	-

Table 1: Résultats d'explorations pour les différentes configurations

Les résultats montrent des complexités faibles sur cet exemple qui est très simpliste mais illustratif dans le cadre de cet article. Aussi, le nombre de configurations pour les modes *NM* et *SM* sont identiques car les patrons de sécurité n'ont pas d'impact sur l'exploration en cas d'absence d'attaquant. Enfin, la complexité du mode *SMUA* est moindre que celle du mode *NMUA*, car, dans le mode *SMUA*, les patrons jouent un rôle restrictif en terme de comportement face aux attaques.

5 Conclusion

Cet article illustre, sur un exemple simplifié de modèle SCADA, l'exploitation de la technique de *model – checking* mis en œuvre avec l'outil *OBP*. L'idée poursuivie par ce travail préliminaire est d'étudier, sur des modèles de complexité plus réaliste, l'impact de l'intégration de patrons de sécurité dans des modèles d'architecture. L'ensemble des propriétés à vérifier répondent à des politiques de sécurité devant être implantées dans les systèmes. L'exploration

⁴Par simplification, le comportement de l'attaquant est spécifié ici dans un processus décrit en langage FIACRE.

des modèles permet de vérifier ces propriétés et de mesurer l'efficacité des patrons de sécurité pour répondre aux politiques choisies. Aussi, il est intéressant d'étudier la composition des différents patrons de manière à identifier une méthodologie d'assemblage optimale des patrons et la façon de les intégrer dans les modèles d'architecture. Le comportement des attaquants peut aussi être modélisé par des scénarios CDL, ce qui permet de profiter des travaux sur la réduction de la complexité lors des explorations des modèles [9], ce qui est un aspect problématique bien connu du *model – checking*.

References

- [1] Bonnie Zhu, Anthony Joseph, and Shankar Sastry. A taxonomy of cyber attacks on scada systems. In *Internet of things (iThings/CPSCoM), 2011 international conference on and 4th international conference on cyber, physical and social computing*, pages 380–388. IEEE, 2011.
- [2] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. *Urbana*, 51:61801, 1998.
- [3] Eduardo B Fernandez and Rouyi Pan. A pattern language for security models. In *proceedings of PLOP*, volume 1, 2001.
- [4] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.
- [5] Ronald Wassermann and Betty H.C. Cheng. Security patterns. Technical Report CSE 870, Software Engineering and Network Systems Lab., Michigan State University, 2003.
- [6] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, 2012.
- [7] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, 2013.
- [8] Munawar Hafiz, Paul Adamczyk, and Ralph E Johnson. Growing a pattern language (for security). In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, pages 139–158. ACM, 2012.
- [9] Luka Le Roux Ciprian Teodorov, Philippe Dhaussy. Environment-driven reachability for timed systems : Safety verification of an aircraft landing gear system. *Int. Software Tools for Technology Transfer (STTT)*, to be published, 2016.

GenISIS: un outil de recherche d'attaques d'initié en Systèmes d'Information

Amira Radhouani^{1,2,3,5}, Akram Idani^{1,2}, Yves Ledru^{1,2}, Narjès Ben Rajeb^{3,4}

¹ Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France

² CNRS, LIG, F-38000 Grenoble, France

³ LIP2-LR99ES18, 2092, Tunis, Tunisia

⁴ INSAT - Carthage University, Tunisia

⁵ FST - Tunis-El Manar University, Tunisia

Résumé. La sécurité des Systèmes d'Information est un domaine de recherche très actif abordant une grande variété de défis principalement dédiés à l'identification d'attaques externes : intrusion, déni de service, usurpation d'identité, etc. Les attaques internes, connues sous le nom " attaque d'initié ou insider attack ", sont moins étudiées malgré leur dangerosité et bien qu'elles puissent causer des dommages considérables. Ceci est dû au fait que l'attaquant initié dispose déjà d'un accès au système. Nos travaux dans ce cadre visent l'utilisation conjointe de modèles graphiques UML et de spécifications formelles en B pour la modélisation et la validation de politiques de contrôles d'accès. Nous avons développé l'outil GenISIS (<http://genisis.forge.imag.fr>) qui permet d'extraire des scénarios malicieux à partir d'une spécification fonctionnelle du SI et une description de sa politique de contrôle d'accès. Pour ce faire, nous mettons en œuvre une technique de recherche symbolique, avec retour en arrière et basée sur le solveur de contraintes de ProB.

1 Introduction

Plusieurs vulnérabilités donnant lieu à des attaques en systèmes d'information sont dues à la logique même de la politique de sécurité plutôt qu'aux mécanismes d'implémentation (authentification, chiffrement, protocoles réseau, etc). Les utilisateurs qui exploitent ces failles sont souvent des utilisateurs de confiance ayant des droits d'usage légitimes du système. En effet, même si une politique de contrôle d'accès semble structurellement correcte de par l'affectation des utilisateurs aux bons rôles avec des permissions clairement identifiées, certaines attaques (dites attaques internes ou attaques d'initié) sont le fruit de l'exécution d'une séquence d'opérations autorisées. Par exemple, Jérôme Kerviel grâce à sa très bonne connaissance des procédures de contrôle interne, parvint à masquer l'importance et le risque élevé des opérations qu'il avait faites sur le marché en introduisant dans le système informatique de la Société générale des opérations inverses fictives les compensant. Dans ce type d'attaques, l'attaquant tente d'abuser de ses droits pour contourner les contraintes d'autorisation l'empêchant de réaliser certaines opérations critiques. Ces contraintes dépendent de l'état fonctionnel du SI, et

Extraction de scénarios malicieux

peuvent devenir vraies suite à des modifications de cet état fonctionnel au travers de l'exécution d'une séquence d'opérations autorisées.

Dans nos travaux, nous nous servons des modèles UML et SecureUML [4] pour représenter le modèle fonctionnel du SI et la politique de contrôle d'accès. Ces modèles graphiques permettent d'avoir une vue structurante et compréhensible du SI. Nous traduisons ensuite les divers modèles en spécifications formelles B au moyen de la plateforme B4MSecure¹ [3]. Le modèle formel résultant spécifie le comportement de l'application grâce à des opérations prédéfinies générées automatiquement. Ce modèle peut être enrichi par des invariants et des opérations additionnelles et a vocation à être formellement validé par des outils comme l'animateur ProB ou l'AtelierB. Ces spécifications B constituent un socle intéressant et utile pour la recherche de scénarios malicieux donnant lieu à des attaques d'initié. Pour ce faire, nous avons développé l'outil GenISIS (<http://genesis.forge.imag.fr>) qui met en œuvre une technique de recherche symbolique inverse basée sur le solveur de contraintes de ProB. Cet article résume les fondements de notre outil tant du point de vue théorique que pratique.

2 Exemple

2.1 Modèle fonctionnel

Nous considérons un exemple simple, inspiré de [1] et dédié à la gestion de comptes bancaires. Le modèle fonctionnel de la figure 1 est constitué d'une classe "Customer" représentant les clients de la banque et d'une classe "Account" représentant les comptes associés à ces clients. Les exigences au niveau fonctionnel peuvent être résumées par :

- Un compte est associé à au plus un client ;
- Un client enregistré dans le système dispose d'au moins un compte ;
- Les attributs "balance", "overdraft" et "IBAN" sont des attributs mono-valués et obligatoires qui représentent respectivement le solde du compte, le découvert autorisé et le numéro de compte ;
- Le solde d'un compte ne doit pas être inférieur au découvert autorisé
- Les transactions bancaires sont réalisées au moyen des opérations "transferFunds", "withdrawCash" et "depositFunds" qui permettent d'effectuer des transferts, des retraits ou des dépôts d'argent sur un compte bancaire.

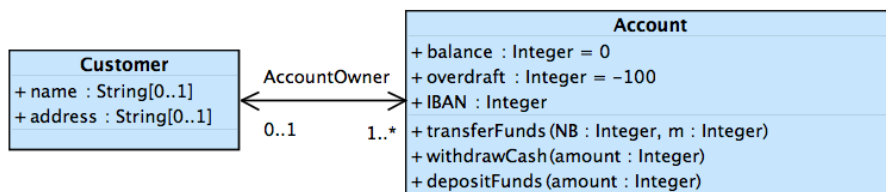


FIG. 1 – Diagramme de classes du modèle fonctionnel

1. La plateforme est disponible à l'adresse suivante : <http://b4msecure.imag.fr>

2.2 Modèle de contrôle d'accès

La gestion des droits d'accès est spécifiée au moyen de diagrammes SecureUML. La figure 2 présente les différents rôles de l'application ainsi que l'affectation des utilisateurs à ces rôles. Le rôle "AccountManager" définit le conseiller financier, et le rôle "CustomerUser" définit le client. Dans cet exemple, Bob est un conseiller financier, et Paul est un client. La contrainte DSD (Dynamic Separation of Duties) liant les rôles "Customer" et "AccountManager" indique un conflit entre ces deux rôles, et désigne le fait qu'un utilisateur ne peut pas se connecter à la fois en tant que client et conseiller financier.

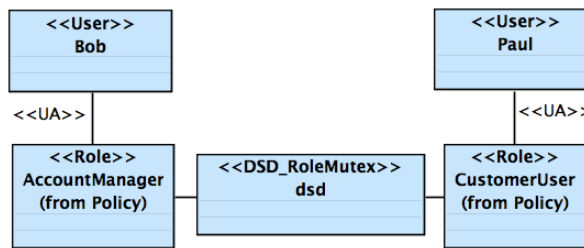


FIG. 2 – Affectation des utilisateurs aux rôles

La figure 3 représente les permissions associées aux divers rôles selon une politique RBAC [2] (Role Based Access Control). Les exigences de contrôle d'accès peuvent être résumées par :

- Le client peut lire ses données (permission "CustomerUserPerm1") et effectuer des transferts ou des retraits d'argent (permission "CustomerUserPerm2")
- Le conseiller financier peut réaliser toutes les opérations sur la classe "Customer" (création, suppression, modification, etc), mais ses droits d'accès sur la classe "Account" sont limités à la création et au dépôt d'argent).

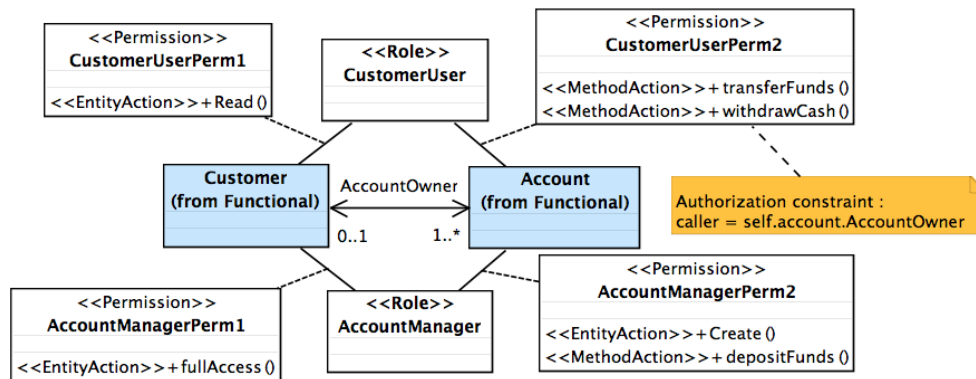


FIG. 3 – Politique de contrôle d'accès

Ce modèle considère une contrainte d'autorisation indiquant que les opérations "transferFunds" et "withdrawCash" sont autorisées uniquement pour le détenteur du compte sur lequel ces opérations sont invoquées.

Extraction de scénarios malicieux

3 Recherche d'attaques internes

3.1 Scénarios malicieux

Ayant une politique de contrôle d'accès conforme au modèle RBAC et impliquant des contraintes d'autorisation, notre intention est de chercher des scénarios permettant à un utilisateur de contourner ces contraintes en vue de réaliser les opérations correspondantes. Le principe de ces scénarios consiste à faire évoluer l'état fonctionnel du système pour qu'une personne non autorisée soit finalement autorisée à effectuer une action qui compromet l'un des objectifs de sécurité. Nous commençons par identifier une classe, la cible de sécurité, soumise à des contraintes d'autorisation. La figure 4 schématise ce type de scénario où O_c est une opération critique, U est un utilisateur malicieux ayant un ensemble de rôles R_u , et C est une contrainte d'autorisation. Dans ce scénario : (1) les états sont des états fonctionnels valides respectant les contraintes fonctionnelles, (2) les transitions correspondent à des appels d'opérations fonctionnelles, et (3) les étiquettes correspondent à des prémisses de contrôle d'accès désignant l'utilisateur malicieux, ses rôles, ainsi que la contrainte d'autorisation. Ces prémisses, doivent être vérifiées avant chaque transition.

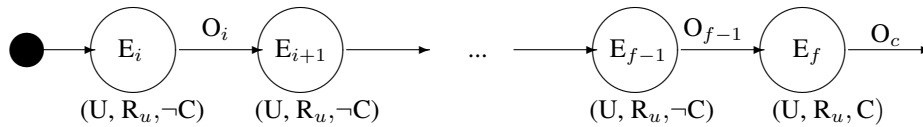


FIG. 4 – Définition d'un scénario malicieux

L'état final E_f est l'état critique car il permet le déclenchement de l'opération critique O_c . Dans cet état, l'utilisateur U dispose d'au moins un rôle dans l'ensemble de ses rôles R_u lui permettant d'exécuter O_c tout en vérifiant que la contrainte C est vraie. Dans les autres états du scénario la contrainte est fautive.

Dans notre exemple, notre cible de sécurité est donc la classe "Account" étant donné qu'une permission avec une contrainte d'autorisation lui est associée. Le modèle SecureUML, montre que Bob étant un conseiller financier, il n'a que le droit de création des clients et de leurs comptes, ainsi que le dépôt de fonds sur ces comptes. Cette vue statique ne lui donne ni l'autorisation de retirer de l'argent ni d'effectuer un transfert d'argent. Nous allons donc tenter de voir si un conseiller financier peut effectuer des transferts d'argent des comptes clients vers ses comptes personnels. La recherche d'attaques d'initié démarre à partir d'un état issu d'un scénario fonctionnel valide. Par exemple, dans l'état fonctionnel suivant, Paul est le seul client du système ; il dispose du compte `cpt1` dont le numéro IBAN est 111, le découvert autorisé est fixé à -100 euros et le solde est égal à 5000 euros.

$\begin{aligned} \text{Customer} &= \{\text{Paul}\} \wedge \text{Account} = \{\text{cpt1}\} \wedge \\ \text{AccountOwner} &= \{(\text{cpt1} \mapsto \text{Paul})\} \wedge \\ \text{Account_IBAN} &= \{(\text{cpt1} \mapsto 111)\} \wedge \\ \text{Account_overdraft} &= \{(\text{cpt1} \mapsto -100)\} \wedge \\ \text{Account_balance} &= \{(\text{cpt1} \mapsto 5000)\} \end{aligned}$

3.2 Spécifications B

Le modèle fonctionnel est traduit en B au moyen de la plateforme B4MSecure. Celle-ci produit toutes les opérations de base permettant de manipuler des instances du diagramme de classe et est enrichie par des opérations additionnelles ainsi que des invariants fonctionnels. Dans notre exemple nous considérons que Paul est la victime, et Bob l'attaquant. Ce dernier va devoir abuser de ses droits d'accès pour effectuer des opérations douteuses sur le compte cpt1. L'opération critique qui nous intéresse est l'opération `transferFunds` dont la spécification est la suivante :

```

Account__transferFunds(Instance, NB, m) ==
PRE
    Instance ∈ Account ∧ NB ∈ NAT ∧ m ∈ NAT1
    ∧ AccountOwner[{Instance}] ≠ ∅
    ∧ NB ∈ ran({Instance} ↦ Account__IBAN)
    ∧ AccountOwner[{Account__IBAN-1(NB)}] ≠ ∅
    ∧ Account__balance(Instance) - m ≥ Account__overdraft(Instance)
THEN
    Account__balance := {(Instance ↦ (Account__balance(Instance) - m))}
    ∪ {(Account__IBAN-1(NB) ↦ (Account__balance(Account__IBAN-1(NB)) + m))}
    ∪ {(Instance, Account__IBAN-1(accountNB)) ↦ Account__balance}
END;
    
```

FIG. 5 – Opération `transferFunds` de la classe `Account`

Cette opération permet un transfert d'argent d'un compte courant, vers un compte désigné par un numéro IBAN autre que le compte courant et vérifie que le découvert autorisé est bien respecté.

3.3 Technique de recherche d'attaques

La recherche d'attaque implantée dans l'outil GenISIS est une recherche symbolique avec retour en arrière qui part de l'état critique et qui tente d'atteindre, par résolution de contraintes, l'état initial. A chaque étape de la recherche l'outil dispose d'une description de l'état cible (E_c) et de l'état source (E_s), et tente pour chacune des opérations de la spécification d'exhiber une valuation de ses paramètres de telle sorte que si l'opération est déclenchable dans E_s alors elle atteint un état qui satisfait E_c .

Pour une opération donnée O_i de la forme $O_i(p_1, \dots, p_n)$, la contrainte que notre outil cherche à résoudre correspond à :

$$\{p_1, \dots, p_n \mid Invariant \wedge E_s \wedge Pre(O_i) \wedge (\neg[Action(O_i)] \neg E_c)\}$$

Cette contrainte détermine une valuation des paramètres de O_i de telle sorte que les propriétés de déclenchabilité et d'atteignabilité soient vraies :

- Déclenchabilité : cette propriété est désignée par le prédicat $Invariant \wedge E_s \wedge Pre(O_i)$ indiquant que la précondition de O_i et l'état source et l'invariant sont respectés
- Atteignabilité : cette propriété est désignée par $\neg[Action(O_i)] \neg E_c$ indiquant que le prédicat E_c peut être établi par les actions de O_i .

Extraction de scénarios malicieux

Lors de la première itération l'état cible correspond à l'état critique et est caractérisé par la précondition de l'opération critique, la contrainte d'autorisation et les conditions spécifiques à l'attaquant, et l'objet cible de sécurité. Quant à l'état source, il est caractérisé par la négation de l'état cible ($E_s == \neg E_c$). Par exemple, pour déclencher l'opération critique `transferFunds` sur le compte de Paul, les conditions suivantes doivent être satisfaites : le paramètre *Instance* doit être égal à `cpt1`, Bob doit être propriétaire du compte cible du transfert et doit satisfaire la contrainte d'autorisation $accountOwner(cpt1) = Bob$. Par conséquent, E_c sera défini par :

$$E_c == Pre(Account_transferFunds) \wedge Instance = cpt1 \wedge accountOwner(cpt1) = Bob \wedge AccountOwner[\{Account_IBAN^{-1}(NB)\}] = Bob$$

Cette première itération produit un ensemble d'opérations permettant d'atteindre l'état critique à partir d'un état symbolique compatible avec l'état initial : $E_i \Rightarrow E_s$. Ensuite, la recherche symbolique s'effectue à l'intérieur de l'état E_s en le partitionnant récursivement jusqu'à ce que l'état initial soit atteint. On calcule à chaque étape de la récursion, les états E'_s et E'_c pour chaque opération O trouvée lors de la première itération comme suit :

$$E'_c == E_s \wedge Pre(O)$$

$$E'_s == E_s \wedge \neg Pre(O)$$

La résolution de contrainte, pour ces deux états source et cible, permet d'exhiber pour l'opération O un ensemble d'opérations déclenchables à partir de $\neg Pre(O)$ et permettant d'atteindre un état déclenchant O tout en étant compatible avec E_s . Notre algorithme de recherche de scénarios procède récursivement de la même manière et s'arrête quand il trouve au moins une opération déclenchable à partir de l'état initial ($E_i \Rightarrow E'_c$) ou quand il aura tenté toutes les opérations sans atteindre l'état initial.

La mise en œuvre de cette technique dans GenISIS est réalisée en interaction avec le model-checker ProB. En effet, outre la possibilité d'animer des opérations, ProB offre un solveur de contraintes. Cette fonctionnalité nous est utile pour identifier les différentes valuations des paramètres satisfaisant les propriétés de déclenchabilité et d'atteignabilité.

3.4 Application

L'application de l'outil GenISIS sur notre exemple produit plusieurs scénarios fonctionnels partant de l'état initial et permettant d'atteindre l'état critique. Pour des raisons de place, nous nous contentons de présenter uniquement les deux scénarios suivants :

<ol style="list-style-type: none"> 1. Account_NEW(cpt2, 222) 2. Customer__AddAccountOwner(Paul, cpt2) 3. Customer__RemoveAccountOwner(Paul, cpt1) 4. Customer_NEW(Bob, {cpt1}) 5. Account_NEW(cpt3, 333) 6. Customer__AddAccountOwner(Bob, cpt3) 7. Account_transferFunds(cpt1, 333, 100) 	<ol style="list-style-type: none"> 1. Account_NEW(cpt2, 222) 2. Account_NEW(cpt3, 333) 3. Customer__AddAccountOwner(Paul, cpt2) 4. Customer__RemoveAccountOwner(Paul, cpt1) 5. Customer_NEW(Bob, {cpt1, cpt3}) 6. Account_transferFunds(cpt1, 333, 100)
--	---

Dans ces deux scénarios, deux comptes (cpt2 et cpt3) et un client (Bob) ont été créés. La création du compte cpt2 (`Account_NEW(cpt2, 222)`) et son affectation à Paul (`Customer__AddAccountOwner(Paul, cpt2)`), sont des pré-requis à l'opération qui détache le compte cpt1 de Paul (`Customer__RemoveAccountOwner(Paul, cpt1)`). En effet, la multiplicité 1..*

A. Radhouani et al.

du côté de la classe "Account" indique qu'un client doit être rattaché à au moins un compte. De même, la création du client Bob et d'un troisième compte cpt3 de telle sorte que Bob soit le propriétaire de cpt1 et cpt3 sont nécessaires pour la réalisation de l'opération Account_transferFunds dans l'état critique. Ces deux scénarios exhibés sur le modèle fonctionnel modifient l'état fonctionnel de telle sorte que la contrainte d'autorisation devienne vraie. Il suffit maintenant de vérifier que l'attaquant a bien le droit de réaliser toutes ces opérations. Dans les deux scénarios, mis à part l'appel à l'opération critique, Bob en tant que conseiller financier a bien les droits nécessaires. Toutefois, l'opération transferFunds ne peut être réalisée que par un client. Dans le scénario fonctionnel Bob s'est procuré ce rôle en ayant créé une instance nommée Bob de la classe "Customer". Notons que dans cet exemple, toute instance de la classe "Customer" correspond à un utilisateur ayant le rôle "CustomerUser". Vu la contrainte DSD indiquant un conflit des rôles "CustomerUser" et "AccountManager", Bob est contraint de se connecter à l'application en tant que conseiller financier pour réaliser la séquence malicieuse, et de se reconnecter en tant que client pour réaliser l'opération critique. Pour camoufler son attaque, il lui suffit ensuite de réaliser les opérations inverses pour remettre le système dans un état raisonnable : retrait d'argent sur le compte cpt3, réaffectation de cpt1 à Paul et suppression des comptes cpt2 et cpt3. Ces opérations sont exhibées en appliquant la même technique.

Dans le cadre dans cet article, nous nous sommes contentés de présenter une technique de recherche d'attaques impliquant un seul utilisateur, cependant, nous notons que dans sa version actuelle, GenISIS étend cette technique à la recherche de coalitions d'initiés. Vu que notre algorithme de recherche d'attaques se base sur le modèle fonctionnel, il peut exhiber des séquences pouvant être réalisées par plusieurs utilisateurs, mais également des séquences impliquant des opérations non autorisées.

4 Conclusion

Dans cet article nous mettons en œuvre une technique de recherche avec retour en arrière basée sur le solveur de contraintes de ProB dans le but d'exhiber des scénarios malicieux. Ces scénarios sont difficiles à identifier au vu des modèles statiques UML et SecureUML. Par exemple, le modèle de la figure 3 montre qu'un conseiller financier, ne peut que déposer de l'argent sur des comptes bancaires et n'est pas en mesure d'effectuer des retraits sur des comptes clients. La description dynamique du SI, favorisée par les spécifications B produites à partir de ces modèles, est très utile car elle permet une validation d'un point de vue comportemental.

L'outil GenISIS a été appliqué sur différentes études de cas et a donné lieu à des résultats très encourageants. L'utilisation de ProB purement en mode model-checking, trouve les scénarios discutés dans cet article après avoir exploré plus de 1500 états différents et 36000 transitions, en plus du fait que les scénarios exhibés contiennent du bruit (*e.g.* modification de l'adresse de Paul, etc). Ce nombre d'états important explorés par ProB sur un exemple simple peut devenir extrêmement problématique pour des spécifications de plus grande taille. La technique proposée permet d'une part de réduire considérablement l'espace de recherche en explorant uniquement les transitions pertinentes, et d'autre part de guider le model-checker vers la bonne direction. Notre outil a exhibé neuf scénarios au total dont sept sont interdits par le modèle de sécurité mais qui sont utiles pour donner une idée sur les opérations pouvant être source d'attaque. Sur la base de cet exemple, une vingtaine d'appels récursifs ont été suffisants pour exhiber ces scénarios.

Extraction de scénarios malicieux

Dans [5] nous avons proposé une technique complémentaire à celle présentée dans cet article mais basée sur la preuve de propriétés d'atteignabilité. Les techniques de résolution de contraintes et de preuve présentent chacune des avantages et des limitations. En effet, bien que la recherche d'attaques par la preuve développée dans [5] permette d'avoir une meilleure garantie quant à l'atteignabilité d'un état par une opération donnée, elle souffre des limitations classiques de la complexité de la preuve. Les expérimentations menées dans ce cadre ont montré que la plupart des preuves ne peuvent être déchargées automatiquement, et nécessitent l'interaction de l'analyste. Ceci reste fastidieux à mener dans le cas d'un nombre d'opérations accru ou de spécifications de grandes tailles comme celles décrivant des SI. La résolution de contraintes est une alternative intéressante car d'une part, cette technique est moins coûteuse, et d'autre part, quand l'espace d'états est borné, elle permet une prise de décision rapide. Cependant, cette technique nécessite de contraindre les domaines des variables d'état. Dans nos prochains travaux, nous songeons à combiner ces deux techniques pour pallier les limitations de l'une par les avantages de l'autre.

Remerciements. Les auteurs tiennent à remercier Abbes Rjab pour sa contribution pratique au développement de GenISIS.

Références

- [1] A. Bandara, H. Shinpei, J. Jurjens, H. Kaiya, A. Kubo, R. Laney, H. Mouratidis, A. Nhlabatsi, B. Nuseibeh, Y. Tahara, T. Tun, H. Washizaki, N. Yoshioka, and Y. Yu. Security patterns : Comparing modeling approaches. In *Software Engineering for Security Systems : Industrial and Research Perspectives*. IGI Global, 2010.
- [2] D.F. Ferraiolo, D.R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Computer Security Series. Artech House, 2003.
- [3] A. Idani and Y. Ledru. B for modeling secure information systems - the B4MSecure platform. In *ICFEM 2015*, volume 9407 of *LNCS*. Springer, 2015.
- [4] T. Lodderstedt, D. Basin, and J. Doser. SecureUML : A UML-Based Modeling Language for Model-Driven Security. In *UML 2002 - 5th International Conference on The Unified Modeling Language*, volume 2460 of *LNCS*, pages 426–441. Springer, 2002.
- [5] A. Radhouani, A. Idani, Y. Ledru, and N. Ben Rajeb. Symbolic search of insider attack scenarios from a formal information system modeling. *LNCS Transactions on Petri Nets and Other Models of Concurrency*, 10 :131–152, 2015.

Approche de spécification et validation formelles de politiques RBAC au niveau des processus métiers

Salim CHEHIDA
Université Oran1 Ahmed BenBella
Email:SalimChehida@yahoo.fr

Résumé

Ce travail présente une approche qui combine les langages UML et B pour la spécification et la validation de politiques RBAC au niveau des activités d'un processus métier. Notre approche commence par la spécification semi-formelle des règles de contrôle d'accès à l'aide de notre extension des diagrammes d'activité d'UML2. Les modèles d'activité sont ensuite traduits en une spécification formelle exprimée dans le langage B. La spécification B définit un cadre rigoureux favorisant la V&V de la politique RBAC.

MOTS-CLÉS— RBAC, Diagramme d'activité d'UML2, B, Animation, Preuve

1 Introduction

La spécification de politiques de contrôle d'accès constitue une étape cruciale dans le développement d'un système d'information. RBAC [2] est le modèle le plus répandu pour contrôler l'accès aux systèmes. Dans ce modèle, les droits d'accès sont centrés sur le concept de rôle qui représente une fonction dans le cadre d'une organisation. L'accès aux données est accordé à un utilisateur en fonction du rôle qui lui est associé. UML 2 [7] est une notation graphique largement adoptée pour l'analyse des besoins et la conception d'un système. Le concept de *Profil* permet l'extension des diagrammes d'UML pour spécifier un aspect particulier d'un système. Dans le cadre de notre étude, nous proposons le profil BAAC@UML (Business Activity Access Control with UML) qui étend les diagrammes d'activité d'UML2 pour qu'ils représentent le déroulement d'un processus métier en tenant compte d'une politique RBAC. Les extensions proposées ont été publiées dans [3].

B [1] est une méthode formelle basée sur des notations mathématiques. La génération des spécifications B à partir de modèles UML est considérée comme un moyen approprié qui permet d'utiliser conjointement UML et B dans une approche de développement rigoureuse. B4MSecure [4] est un outil dédié à la modélisation conjointe en UML et B des aspects fonctionnels d'un système ainsi que de politiques de contrôle d'accès. Nous utilisons cet outil pour traduire nos modèles d'activité en B. Les spécifications B générées servent de support pour la V&V de

la politique RBAC exprimée au niveau des activités d'un processus métier. Nous tirons profit de l'animateur *ProB* et du prouveur *AtelierB* dans les activités de V&V.

Afin d'illustrer notre travail, nous allons utiliser un exemple de système d'organisation de réunions. Ce système s'adresse à deux types d'utilisateurs : les *initiateurs* planifient des réunions et les *participants* répondent aux invitations des initiateurs. Il permet d'enregistrer les données des personnes, des invitations, des réunions et des propositions de changement, ainsi que les liens entre ces données.

2 Modélisation d'une politique RBAC

Notre profil introduit les concepts de *rôle* et *précondition de contrôle d'accès* pour garder l'accès des utilisateurs aux activités d'un processus métier. Les préconditions de contrôle d'accès expriment des contraintes contextuelles. Elles permettent de spécifier des politiques de contrôle d'accès qui dépendent de l'état fonctionnel du système (tel que les valeurs des attributs) et de son état sécuritaire (utilisateurs et rôles courants). Elles peuvent être spécifiées dans la vue statique au moyen du profil SecureUML et prendre la forme de gardes dans la vue dynamique de la politique RBAC exprimée par notre profil.

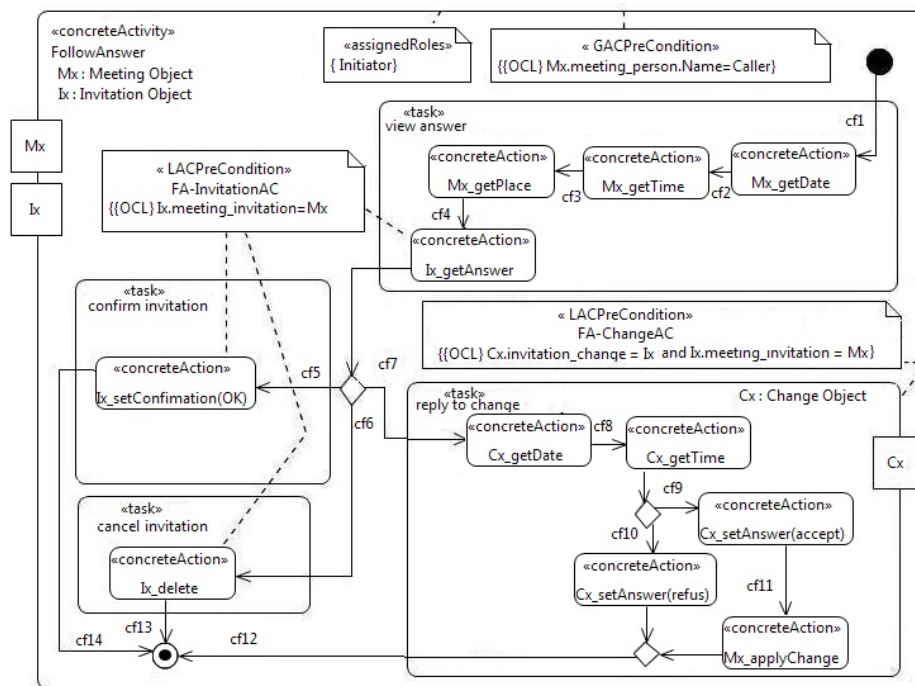


FIGURE 1 – Modèle BAAC@UML.

La figure 1 présente la spécification d'une activité métier de notre exemple par le profil BAAC@UML. L'initiateur consulte la réponse d'un invité à une réunion et suivant la réponse, il choisit de confirmer l'invitation, d'annuler l'invitation ou

de répondre à une proposition de changement de la réunion. Le diagramme spécifie le comportement de différentes tâches de l'activité par une coordination d'actions concrètes qui font référence à des opérations sur les objets des classes. La contrainte *assignedRoles* autorise les seuls utilisateurs assignés au rôle *Initiator* à exécuter l'activité. La précondition globale de contrôle d'accès (stéréotypée par *GACPreCondition*) garantit que l'utilisateur exécutant l'activité est le créateur de la réunion. Cette précondition s'applique à toutes les actions de l'activité. La précondition locale *FA_InvitationAC* garantit que le créateur de la réunion ne peut lire, modifier ou supprimer que les invitations de sa réunion. La précondition locale *FA_ChangeAC* garantit que le créateur de la réunion ne peut répondre qu'aux changements associés aux invitations de sa réunion.

3 Formalisation en B d'une politique RBAC

La figure 2 présente les modèles UML que nous utilisons, les machines B qui leur correspondent et les outils de validation utilisés. L'outil B4MSecure assure la partie A de la figure 2. Il permet la traduction en spécifications B des diagrammes de classes fonctionnels et des modèles SecureUML de spécification statique d'une politique RBAC. La machine *Functional* spécifie les différents éléments d'un diagramme de classes en B. La machine *RBAC*, issue du modèle SecureUML, associe à chaque opération de la machine *Functional*, une opération sécurisée spécifiant les utilisateurs autorisés à l'exécuter.

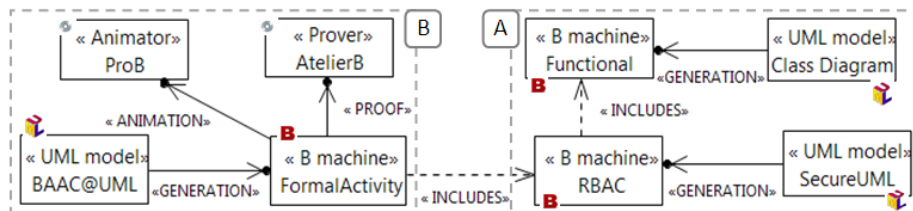


FIGURE 2 – Approche de formalisation.

Dans le cadre de notre travail, nous proposons une approche pour réaliser la partie B de la figure 2 qui permet de traduire les modèles BAAC@UML en une spécification B et d'utiliser cette spécification pour vérifier les contraintes de sécurité des modèles BAAC@UML et leur cohérence avec les modèles SecureUML. La machine *FormalActivity* issue du modèle BAAC@UML formalise les éléments fonctionnels (tâche, action concrète et flot de contrôle) et de sécurité (rôle et précondition de contrôle d'accès) de l'activité. Les contraintes *assignedRoles* et *GACPreCondition* sont traduites de l'OCL en B et exprimées comme invariants de la machine. Les contraintes locales (*LACPreCondition*) sont également transformées en B et exprimées comme préconditions des actions ou des tâches. Les opérations de la machine *FormalActivity* spécifient le comportement de l'activité en utilisant une seule opération par tâche. Cette dernière est représentée par une séquence de

substitutions qui fait appel aux opérations sécurisées (de la machine *RBAC*) correspondant aux actions concrètes. Nous avons également défini une machine *B* qui spécifie des opérations permettant de gérer l'enchaînement des actions.

4 V&V d'une politique RBAC

Le prouveur *AtelierB* nous a permis de vérifier que les contraintes *assigne-dRoles* et *GACPreCondition* (exprimées comme invariants) sont conservées dans l'exécution de l'activité. L'animateur *ProB* nous a permis de simuler l'exécution des scénarios d'activité par un utilisateur connecté à des rôles. L'animation permet d'évaluer les préconditions contextuelles et de vérifier que la politique RBAC est bien respectée au niveau des modèles d'activité. Les tests réalisés nous ont permis d'identifier et de corriger les erreurs dans les modèles BAAC@UML. Ils nous donnent une certaine garantie de correction de la politique RBAC exprimée au niveau des activités d'un processus métier malgré le fait qu'ils ne démontrent pas l'absence d'erreurs.

5 Conclusion

Plusieurs travaux, comme par exemple [5] et [6], ont proposé des profils qui étendent les diagrammes d'activité pour la spécification de politiques RBAC. Notre approche fournit un certain nombre d'avantages par rapport à ces travaux. Parmi ces avantages :

- La spécification des contraintes contextuelles d'autorisation.
- La séparation entre la spécification des processus métiers et la spécification de politiques de contrôle d'accès.
- La spécification graphique et formelle d'une politique RBAC au niveau des activités.
- La preuve et l'animation des spécifications *B* d'une politique RBAC.

Références

- [1] Jean-Raymond Abrial. *The B-book : assigning programs to meanings*. Cambridge University Press, 1996.
- [2] ANSI. Role Based Access Control. *American national standard for information technology*, 359(2004) :1–47, 2004.
- [3] S. Chehida, A. Idani, Y. Ledru, and M.K Rahmouni. Extensions du diagramme d'activité pour contrôler l'accès au SI. In *INFORSID*, Biarritz, 2015.
- [4] A. Idani and Y. Ledru. B for Modeling Secure Information Systems - the B4MSecure platform . In *ICFEM*, volume 4907 of *LNCS*. Springer, 2015.
- [5] J. Jurjens. *Secure Systems Development with UML*. Springer-Verlag, Berlin, Heidelberg, 2004.

- [6] Mark Strembeck and Jan Mendling. Modeling process-related RBAC models with extended UML activity models. *IST*, 53(5), 2011.
- [7] UML2. *Unified Modeling Language : Superstructure(version 2.4)*. Object Management Group, 2011.

Session doctorant : Inférence et analyse de propriétés dans les protocoles contrôle-commande

Emmanuel Perrier
Université Grenoble Alpes – LIG, France
Emmanuel.Perrier@imag.fr

1 Contexte

Les réseaux de contrôle-commande SCADA (pour *Supervisory Control And Data Acquisition*) sont un ensemble de systèmes qui visent à suivre et régir un procédé industriel ; ils sont présents dans un grand nombre de secteurs industriels dont la génération et distribution d'énergie (électrique, pétrole, nucléaire, . . .), la distribution d'eau et le transport par exemple.

Ces systèmes se composent notamment d'automates programmables (PLC pour *Programmable Logic Controller*) — qui assurent la liaison entre le monde physique et le monde informatique —, d'interfaces homme-machine, de boucles de contrôle et d'outils de diagnostic et de maintenance.

Les architectures SCADA tendent de plus en plus à s'ouvrir vers les réseaux informatiques de gestion de l'entreprise, voir vers l'Internet, comme dans le cas de l'architecture de quatrième génération dite *Internet of Things* ; ce qui est aussi rendu possible grâce à l'émergence de nouveaux protocoles de communication basés sur Ethernet tel que le standard OPC-UA. Toutefois, cette ouverture sur les systèmes informatiques traditionnels rend les systèmes SCADA plus vulnérables à des cyber-attaques et l'enjeu stratégique qu'ils peuvent représenter motive des attaques très ciblées et complexes comme l'ont été *Stuxnet*, *Duqu* et *Flame*.

Les systèmes de détection d'intrusion (*Intrusion Detection System* ou *IDS*) se fondent sur plusieurs approches pour détecter des activités anormales. La première, basée sur des signatures détecte des motifs de trafic connus pour être symptomatiques d'une intrusion. C'est la méthode la plus employée par les IDS conçus pour les systèmes informatiques traditionnels. La seconde est une approche comportementale qui consiste dans un premier temps à définir un modèle légitime du système et, dans un second temps, à comparer en phase de production le comportement observé du système avec ce modèle.

Il est désormais bien admis [Dem13; Che+07; EMM15] que les caractéristiques particulières des systèmes de contrôle-commande (architecture hiérarchique, trafic principalement périodique, simplicité des PLC, . . .) permettent l'utilisation d'IDS comportementaux dont l'efficacité est supérieure aux IDS par signature.

L'objectif des travaux de cette thèse est d'offrir des méthodes et outils permettant une modélisation fine des systèmes de contrôle-commandes dans le but d'améliorer la détection d'intrusions comportementales.

2 Techniques

Classification du trafic réseau La classification du trafic réseau vise à identifier la nature du trafic (web, voip, échange de fichier, etc.), le protocole employé (http, bittorrent, ssh, etc.) ou l'application qui le génère ; elle est souvent utilisée dans le cadre de la qualité de service ou de la sécurité par filtrage du trafic.

Les approches de classification peuvent se baser sur le port réseau utilisé, le contenu des paquets, les caractéristiques du flux ou sur le comportement de l'hôte suivant les objectifs d'identification (nature, protocole ou application).

Dans le cadre des réseaux SCADA où les hôtes et les protocoles employés sont bien définis, seule la classification applicative représente un réel intérêt pour modéliser puis identifier un hôte. Cette méthode peut-être employée par exemple pour opérer une discrimination de serveur sur la base de l'observation d'un trafic chiffré TLS [Kor12] grâce à la modélisation du processus d'initialisation de session en chaîne de Markov.

Inférence de boîte-noire Dans le domaine du test par modèle, l'inférence de boîte-noire permet de caractériser un système dont le comportement est inconnu. On distingue deux grandes approches pour l'inférence de boîte-noire : l'approche passive et l'approche active ; la première se contente d'observer les interactions entre le système ciblé et l'environnement, la seconde approche sollicite directement le système. Elle consiste à générer des sollicitations pour le système cible, construire un nouveau modèle à partir des réactions observées et réitérer tant que des différences entre le comportement de la cible et le modèle construit sont observées (voir 1).

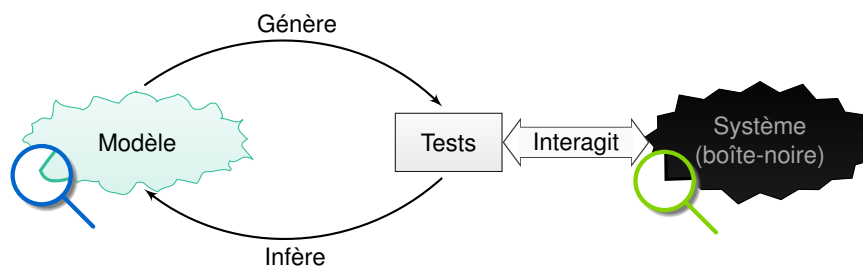


Figure 1: Principe de l'inférence active de boîte-noire

La précision de l'approche passive est fortement dépendante de la quantité et de la diversité des événements observés, tandis que l'approche active permet généralement d'obtenir un modèle plus complet et plus rapidement, mais nécessite d'avoir

une connaissance minimale du vocabulaire du système cible.

Plusieurs types d'automates existent pour représenter un système ; le choix d'un type d'automate puis d'un algorithme d'apprentissage adéquat est un facteur important pour la réussite de la modélisation.

L'inférence de boîte-noire est encore peu appliquée dans les réseaux de contrôle-commande, mais a été appliquée dans [Sme12] et donne des résultats prometteurs.

3 Orientation des travaux de thèse

Les travaux de cette thèse¹ se placent dans le contexte des réseaux de contrôle-commande et visent à l'amélioration des méthodes de classification du trafic et d'inférence de boîte-noire.

On s'intéressera dans un premier temps à l'identification des modèles et des algorithmes pertinents pour l'inférence des hôtes que l'on retrouve habituellement au sein de ces systèmes.

Dans un second temps, on vise à associer les approches passives et actives afin de permettre une modélisation plus complète des systèmes où les interactions qu'il est possible de générer sont limitées pour des raisons de sûreté par exemple.

Enfin, on étudiera la possibilité d'inférer un système en concurrence avec d'autres hôtes en adaptant les algorithmes de construction d'automate.

References

- [ARA] ARAMIS. *Projet Aramis*. Programme d'Investissement d'Avenir n° P3342-146798 - FSN AAP Sécurité Numérique n° 3. URL: <http://aramis.minalogic.net/pages/projet.html>.
- [Che+07] Steven Cheung et al. "Using model-based intrusion detection for SCADA networks". In: *Proceedings of the SCADA security scientific symposium*. Vol. 46. 2007, pp. 1–12.
- [Dem13] Thomas Demongeot. "Détection d'intrusion pour les systèmes industriels". In: *C&ESAR 2013*. Rennes, France, Nov. 20, 2013.
- [EMM15] Emmanuele Zambo, Marco Caselli, and Magnus Almgren. *Network-driven analysis tools*. Projet CRISALIS. Aug. 2015.
- [Kor12] Maciej Korczyński. "Classifying Application Flows and Intrusion Detection in Internet Traffic". Université de Grenoble, Nov. 2012.
- [Sme12] Wouter Smeenk. "Applying Automata Learning to Complex Industrial Software". Master thesis. Radboud University Nijmegen, Sept. 2012.

¹Thèse financée dans le cadre du projet Aramis [ARA]

Formalisation d'une Approche Compositionnelle de Patrons de Propriétés

Djamila Baroudi¹, Philippe Dhaussy², Nait Bahloul Safia³,

¹ UMAB, Université de Mostaganem, Algérie,
baroudi.djamila@univ-mosta.dz

² Lab-STICC, UMR CNRS 6285 ENSTA Bretagne, France,
philippe.dhaussy@ensta-bretagne.fr

³ Lab-LITIO, Université Oran 1, Ahmed BENBELLA, Algérie
bahloul.safia@univ-oran.dz

Abstract

Pour faciliter et encadrer l'expression des propriétés formelles, des alternatives aux logiques temporelles, telles que LTL ou CTL, ont été proposées, au prix de la réduction de l'expressivité. Celles-ci proposent des langages d'expression de propriétés basés sur des patrons de définition. Dans le but d'étendre l'expressivité des patrons proposés, nous avons conçu un langage de type DSL (*Domain Specific Language*) nommé CDL (*Context Description Language*). Ce langage permet une expression de propriétés de sûreté basée sur une extension des patrons proposés par Dwyer et al. Les propriétés sont traduites en automates observateurs et exploités par l'explorateur de modèles OBP (*Observer-Based Prover*). Pour pouvoir valider formellement la transformation des propriétés, nous avons formalisé, avec l'outil COQ, la sémantique des patrons dans une approche compositionnelle. **Mots Clés.** Patrons de propriétés, observateurs, sémantique, composition, OBP/CDL, COQ.

1 Introduction

Un défi bien connu dans le domaine des méthodes formelles est d'améliorer leur intégration dans les processus industriels de développement logiciel. En particulier, les approches de vérification formelle de type *model – checking* nécessitent que les exigences à vérifier sur des modèles de logiciels soient exprimées, sous la forme de propriétés formelles, dans des formalismes faciles d'accès et compréhensibles pour les ingénieurs.

Une difficulté de mise en œuvre d'une technique par *model-checking* est d'exprimer les exigences à vérifier de façon aisée. Les artefacts produits à travers les activités du processus de développement industriel (exigences, spécification, modèles de conception) ne sont pas directement exploitables par les outils de vérification. En particulier, les exigences sont en général disponibles sous une forme textuelle qui les rendent inexploitable sans une ré-écriture préalable plus formelle. Traditionnellement, les techniques de *model – checking* implique de pouvoir exprimer les exigences à l'aide de formalismes de type logique temporelle comme par exemple LTL ou CTL. Bien que ces langages aient une grande expressivité et permettent une expression rigoureuse, ils ne sont pas faciles à utiliser par des ingénieurs lors de leurs activités

de vérification. En effet, dans un contexte industriel, une exigence peut référencer de nombreux états et événements, liés à l'exécution du modèle ou de l'environnement, et est dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification.

Une autre méthode, pour l'expression d'exigences et popularisée par le langage LUSTRE, repose sur la notion d'observateur [1]. Un observateur est un automate ou un programme transformant une propriété d'invariance (*toujours p*) en une vérification d'accessibilité (*la situation not p est elle accessible ?*). Si l'expression d'une propriété d'invariance est effectivement plus simple sous la forme d'un automate observateur, l'écriture d'un tel automate peut se révéler malgré tout complexe dans le cas d'une propriété elle-même complexe. Un automate observateur réaliste peut rapidement dépasser la vingtaine d'états. Il est donc nécessaire de faciliter l'expression des exigences avec des langages adéquats, permettant d'encadrer l'expression des propriétés au prix de la réduction de l'expressivité. De nombreux auteurs ont fait ce constat et certains [2, 3, 4] ont proposé de formuler les propriétés à l'aide de patrons de définition. Un patron est une structure syntaxique textuelle qui permet un mode d'expression d'une propriété plus proche des langages que les ingénieurs ont l'habitude de manipuler. Chaque propriété, basée sur un patron, peut être associée à un *scope* qui précise les instants de l'exécution où la propriété est sensée être vérifiée (*Globally, Before, After, Between, After/Unless*). Dywer et al. [2] fournissent une sémantique formelle des combinaisons offertes par les patrons, par transformation en une formule logique exprimée dans différents langages comme LTL, CTL.

Dans le cadre d'expérimentations industrielles, nous avons souhaité étendre l'expressivité de ces patrons pour pouvoir exprimer plus aisément certaines exigences qui référencent un grand nombre d'événements. Pour cela, nous avons proposé un langage prototype, de type DSL (*Domain Specific Language*), nommé CDL (*Context Description Language*) [5].

2 Motivations

CDL reprend les principaux concepts proposés par [2, 3, 4] et permet l'expression de propriétés temporisée de sûreté, d'invariance ou vivacité bornée, basée sur une extension des patrons proposés. Les patrons CDL¹ permettent d'exprimer, comme [2], des propriétés temporisées de réponse (*Response*), de prérequis (*Precedence*), d'absence et d'existence. Les propriétés font référence à des prédicats et des événements détectables tels que des envois ou réceptions de valeurs entre processus du modèle, des changements d'état de processus ou de variables. Ces patrons (ou formes) de base ont été enrichis par des éléments optionnels (*options*) (*Pre-arity, Post-arity, Immediacy, Precedency, Nullity, Repeatability*) à l'aide d'annotations dans l'esprit de [3]. Des opérateurs *an* et *all* précisent respectivement si un événement ou tous les événements, ordonnés (*ordered*) ou non (*combined*), d'un ensemble d'événements sont concernés par la propriété comme proposé par [6].

Pour mettre en œuvre la vérification de propriétés, notre outillage OBP (*Observer-Based Prover*) transforme les propriétés en automates observateurs non intrusifs dotés d'états de rejet (*reject*). Un observateur est construit de manière à encoder une propriété logique [7] et a pour rôle d'observer, partiellement et de manière non intrusive, les événements significatifs survenant dans le modèle du système à valider. Lors de la simulation, il est composé, de manière synchrone,

¹L'outil OBP et la documentation CDL sont accessibles librement sur : <http://www.obpedl.org>.

au modèle à observer. La vérification de propriété consiste alors en une analyse d'accessibilité des états de rejet sur le graphe d'exploration qui résulte de la composition du modèle à valider et des observateurs.

Dans un travail précédent, une sémantique, dite de référence, des patrons CDL a été décrite et formalisée avec l'assistant COQ sous la forme de systèmes de transitions² représentant les automates observateurs qui sont générés par OBP. Nous précisons, en section 4, les principes adoptés pour la description de la sémantique de référence. Pour pouvoir maintenant implanter et valider les transformations de propriétés en des automates observateurs, nous souhaitons exprimer la sémantique des patrons CDL avec plus de facilité, c'est à dire avec une approche compositionnelle qui permet d'exprimer leur sémantique de manière extensible. Le but est de pouvoir, d'une part, construire, à l'aide d'opérateurs de composition, des observateurs complexes à partir de patrons élémentaires et d'éléments optionnels et, d'autre part, en déduire les algorithmes corrects de génération des automates pour notre outil OBP. Nous souhaitons disposer ainsi d'une base formelle pour enrichir l'expressivité des patrons et pouvoir valider formellement les transformations. Le travail, présenté ici, décrit les principes de cette approche. Le but est d'apporter ensuite la preuve de la préservation de la sémantique suite à la composition. Cette preuve sera basée, d'une part, sur une preuve de bisimulation entre les automates encodant la sémantique et le résultats de la composition patrons-options que nous avons implantée en COQ. D'autre part, une preuve par récurrence permettra de prouver que les opérateurs de composition préservent la sémantique lorsque la complexité des propriétés augmente.

3 Un exemple de patron de propriétés CDL

En guise d'illustration, nous présentons un exemple simple de propriété ($P1$) basé sur un patron de type *Response* de la forme : $all [e_1, e_2] \text{ leads-to } [0..5[all [r_1, r_2]$. Celui-ci définit une réponse (clause *Post*) constituée d'un ensemble d'évènements détectables $[r_1, r_2]$ qui doit avoir lieu suite à une action (clause *Pre*) décrite par un ensemble d'évènements $[e_1, e_2]$. La réponse doit intervenir dans un délai $[0..5[$, relatif au temps de simulation (ou d'exploration) du modèle. En examinant de plus près cette propriété, nous constatons qu'elle peut révéler plusieurs détails cachés qui nécessitent notre attention pour établir sa sémantique précise. Par exemple, si l'évènement e_1 est détecté plusieurs fois, l'ensemble d'évènements $[r_1, r_2]$ doivent-ils être considérés de manière répétitive ? ou encore : Un évènement autre que e_1, e_2, r_1 , ou r_2 peut-il se produire après e_1 et avant $[r_1, r_2]$? Dans [3], les auteurs abordent ces questions en étendant la notion de patron par l'ajout d'options évoquées précédemment.

Nous avons été inspirés par ce travail, en reprenant les options de [3], et en étendant la notion d'ensemble d'évènements (ordonné ou non). Les différentes formes d'options prévues dans CDL, avec leur variantes, sont les suivantes : *Pre-arity* (resp. *Post-arity*) détermine si un évènement e peut être détecté une fois (*exactly one occ e*) ou plusieurs fois (*one or more occ e*) dans la clause *Pre* (resp. *Post*). Dans ces clauses, des ensembles d'évènements peuvent être précisés pour spécifier si un seul évènement (*an*) de l'ensemble est à considérer ou tous les évènements (*all*) de l'ensemble, ordonnés (*ordered*) ou non (*combined*). *Immediacy* détermine si un évènement, autre que ceux présents dans les clauses *Pre* et *Post* peut survenir (*eventually*) ou

²Le code COQ (environ 1300 lignes de Gallina) de cette sémantique est disponible sous <http://www.obpcdl.org>.

non (*immediately*) avant un évènement de la clause *Post*. *Immediacy* précise aussi le délai attendu de la réponse dans le cas de propriétés temporisées. *Nullity* détermine si un évènement de la clause *Pre* doit arriver (*e must occur*) ou non (*e may never occur*) dans le délai spécifié par *Immediacy*. *Precedency* détermine si un évènement de la clause *Post* peut survenir (*one of list_{post} may occur before the first one of list_{pre}*) ou non (*one of list_{post} cannot occur before the first one of list_{pre}*) avant un évènement de la clause *Pre*. *list_{pre}* (resp. *list_{post}*) désigne ici une liste d'évènements pouvant potentiellement survenir dans la clause *Pre* (resp. *Post*). [3] considèrent l'option *Repeatability* avec deux valeurs possibles : *true* et *false*. Nous considérons ici que, pour des propriétés de sûreté, seule l'option à *true* est pertinente. La figure 1 illustre la propriété précédente *P1* sous la forme d'un patron *Response* exprimé avec le langage CDL. Dans cet exemple (figure 1.a), la clause *Pre* (resp. *Post*) référence deux évènements e_1, e_2 (resp. une liste d'évènements [r_1, r_2]). La figure 1.b montre l'observateur qui est généré par l'outil OBP.

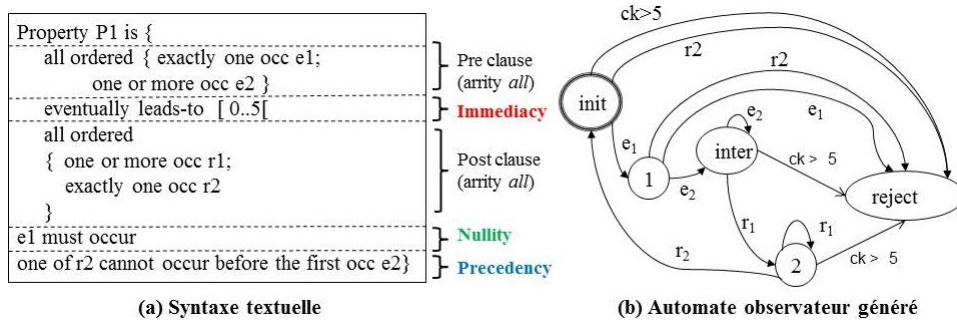


Figure 1: *P1* : un exemple de propriété CDL de type Réponse

4 L'expression de la sémantique des propriétés CDL

Une sémantique des patrons CDL a été exprimée suivant des principes que nous décrivons dans la suite. Mais notons, tout d'abord, que compte tenu des différentes options et du nombre d'évènements pouvant intervenir dans les clauses *Pre* et *Post* d'un patron, le nombre de combinaisons potentielles, dans l'expression de ce dernier, peut être très important. Par exemple, pour un patron de type *Response* qui référence un évènement dans chaque clause, *Pre* et *Post*, il existe 8 combinaisons possibles. Si le nombre d'évènements, dans chaque clause, *Pre* et *Post*, est de 2 (resp. 3 et 4), le nombre de combinaisons de patrons et d'options est de 96 (resp. 192 et 384), moins un certain nombre de combinaisons à supprimer car non significatives.

Pour décrire la sémantique, dite de référence, nous avons d'abord décrit, par un automate, toutes les combinaisons (*Pre*, *Post* et options) disponibles et pour des complexités réduites, c'est à dire, pour un nombre d'évènements, inférieur à 3, dans les clauses *Pre* et *Post*. A partir de la sémantique de ces patrons de base, nous avons défini, par récurrence, la sémantique des patrons obtenus lorsque le nombre d'évènements pris en compte dans les clauses *Pre* ou *Post* augmente. Par exemple, dans le cas de la propriété *P1*, figure 1, l'ajout d'un évènement dans la clause *Pre* ou *Post* donne lieu à un nouvel automate observateur. Nous pouvons donc définir, de manière inductive sous la forme d'une fonction, le calcul qui permet de générer le nouvel automate pour chaque combinaison de base, c'est à dire pour les 96 combinaisons de base pour un nombre de 2 évènements dans chaque clause *Pre* et *Post*. Ce raisonnement par induction nous

permet en théorie d'établir formellement la sémantique de base des patrons CDL pour n'importe quel nombre d'évènements référencés dans les clauses *Pre* et *Post*.

Mais ce mode d'expression de la sémantique nous pose problème pour la raison suivante. Dans le cas où nous voudrions apporter une variante à la sémantique d'un patron et de ces options, nous devons modifier l'automate de base et la fonction inductive associés à chaque combinaison. Compte tenue de cette difficulté, nous optons pour une démarche compositionnelle et donc extensible pour exprimer la sémantique des patrons. L'idée est (1) d'exprimer la sémantique de référence de toutes les combinaisons CDL possibles (*Pre*, *Post* et options) pour des complexités réduites (nombre d'évènements maximum égale à 2 dans *Pre* et *Post*), et les fonctions pour la définition par induction de la sémantique, (2) d'identifier un ensemble d'opérateurs élémentaires de composition, pour construire les automates observateurs à partir de la sémantique des éléments de base *Pre*, *Post* et des options, (3) de démontrer la bisimulation du résultat de la composition avec la sémantique de référence pour un niveau de complexité réduite, et enfin (4) de démontrer par récurrence que la sémantique est préservée par la composition lorsque la complexité des propriétés augmente. Nous détaillons cette idée dans la suite du papier.

5 L'approche compositionnelle

Nous reprenons l'idée développée par [8] qui proposent d'exprimer la sémantique des patrons de [2] par une composition d'automates de Büchi décrivant séparément les patrons et les *scopes*. Mais nous l'adaptons pour l'appliquer à la composition de patrons de base (*Pre* et *Post*) et d'options CDL. Nous ne considérons le *scope* qu'avec uniquement le type *Globally*, qui précise que la propriété exprimée doit tenir pendant toute l'exploration du modèle à valider. Nous présentons ici les règles de composition entre les patrons de base et les options. Pour cela, nous présentons les automates décrivant la sémantique des éléments *Pre*, *Post* et des options constituant un patron CDL.

Sémantique des clauses *Pre* et *Post*. Les formes de bases (*Pre* et *Post*) des patrons sont décrits par des automates comme illustré figures 2.a et 2.b pour la propriété *P1*. L'état *init* représente l'état initial. L'état *inter* est le nœud de connexion entre les clauses *Pre* et *Post*. Les transitions bouclant sur les états représentent les arités du type *one or more*. La transition e_1 (figure 2.a) menant à l'état *reject* représente l'arité du type *exactly one*.

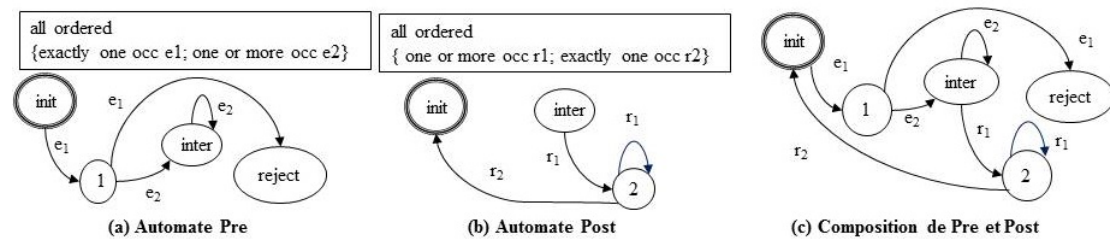


Figure 2: Automates pour les clauses *Pre* et *Post* de la propriété *P1* et leur composition.

Sémantique des options. Les options *Immediacy*, *Precedency*, et *Nullity* sont également décrits par des automates comme illustré figure 3 pour un patron de type *Response*. La figure 3.a

illustre un automate représentant l'option *Immediacy* dans le cas de la variante *immediately* (variante notée *Immediacy = true*). Le caractère '*' représente des événements d'une liste autres que ceux apparaissant dans les clauses *Pre* et *Post*. L'idée est de pouvoir spécifier des événements dont l'occurrence met en erreur (*reject*) la propriété. Cet automate référence aussi la contrainte d'horloge qui peut faire passer l'automate dans l'état *reject*. L'étiquette "** or ck > 5*" signifie l'occurrence * ou le dépassement du délai 5.

La figure 3.b illustre un automate représentant l'option *Immediacy* dans le cas de la variante *eventually* (variante notée *Immediacy = false*). La figure 3.c illustre un automate représentant l'option *Nullity* dans le cas de la variante *may never occur* (variante notée *Nullity = true*). *not a* représente l'absence de l'évènement *a*. Elle permet de spécifier l'évènement (ici *a*) qui est à prendre en compte dans l'option. Pour l'instant, cet évènement est unique mais cette spécification pourrait être étendue dans une version ultérieure à une liste d'évènements. La figure 3.d illustre un automate représentant l'option *Nullity* dans le cas de la variante *must occur* (variante notée *Nullity = false*). La figure 3.e illustre un automate représentant l'option *Precedency* dans le cas de la variante *may occur before* (variante notée *Precedency = true*). Cette option spécifie l'évènement r_2 qui peut être accepté avant e_2 . La figure 3.f illustre un automate représentant l'option *Precedency* dans le cas de la variante *cannot occur before* (variante notée *Precedency = false*). Cette option spécifie l'évènement r_2 qui met en erreur la propriété en cas d'occurrence de e_2 .

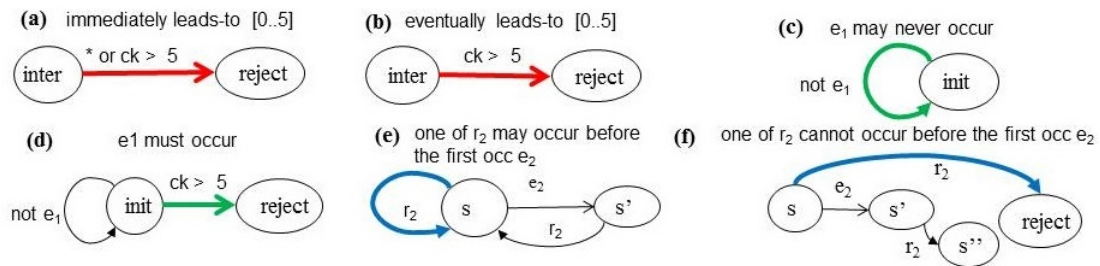


Figure 3: Automates pour les options applicables au patron *Response* de la propriété *P1*.

Dans chaque automate représentant une option (figure 3), nous illustrons en gras une transition particulière qui participera à l'automate généré par les règles de composition décrite ci-dessous.

Règles informelles de la composition. Le principe de la composition repose sur plusieurs compositions élémentaires. Ces opérations de composition reviennent à l'union des transitions et des nœuds des différents automates *Pre* et *Post* et l'ajout de transitions selon les options *Nullity*, *Immediacy* et *Precedency*. Nous illustrons ce principe sur l'exemple des clauses *Pre* et *Post* de l'exemple figure 2 et pour les options *Immediacy = false*, *Nullity = false* et *Precedency = false* illustrées respectivement dans les figures 3.b, 3.d et 3.f. Nous expliquons, ci-dessous, les règles appliquées, pour les options choisies. Le résultat est illustré figure 1.b.

La composition *Pre* et *Post* est construite par une union des états et des transitions des automates *Pre* et *post*. Nous nommons A_p le résultat de cette composition et qui, pour l'exemple, est illustrée figure 2.c. La composition de *Nullity* avec A_p est construite de la manière suivante.

Pour chaque état s de Pre , $inter$ et $reject$ exclus, précédent l'état source³ de la transition e_1 (c'est à dire comportant l'étiquette e_1), une transition t , étiquetée avec la contrainte contenue dans l'option *immediacy*, est ajoutée à A_p telle que $src(t) = s$ et $tgt(t) = reject$. Dans le cas de l'exemple, la transition (*init*, $ck > 5$, *reject*) est ajoutée à A_p . Nous nommons A_n le résultat de cette composition. La composition de *Immediacy* avec A_n est construite de la manière suivante. Pour chaque état s de $Post$, excepté *init*, une transition, étiquetée avec la contrainte contenue dans l'option *immediacy*, est ajoutée à A_n telle que $src(t) = s$ et $tgt(t) = reject$. Dans le cas de l'exemple, les transitions (*inter*, $ck > 5$, *reject*) et (2 , $ck > 5$, *reject*) sont ajoutées à A_n . Nous nommons A_i le résultat de cette composition. La composition de *Precedency* avec A_i est construite de la manière suivante. Pour chaque état s de Pre , excepté *inter* et *reject*, qui n'ont pas en entrée la transition e_2 , une transition t , étiquetée avec l'évènement r_2 , est ajoutée à A_p telle que $src(t) = s$ et $tgt(t) = reject$. Dans le cas de l'exemple, les transitions (*init*, r_2 , *reject*) et (1 , r_2 , *reject*) sont ajoutées à A_i . Nous nommons A_c le résultat de cette composition. Le résultat final A_c des opérations décrites précédemment est illustrée figure 1.b. Notons que la règle de composition des clauses Pre et $Post$ s'applique la première, et ensuite, dans le désordre, les règles associées à *Immediacy*, *Nullity* et *Precedency*.

Les règles de composition ont été implantées sous COQ pour l'instant uniquement pour les patrons de type *Response*. Les règles génèrent la description des systèmes de transitions représentant les propriétés spécifiées. Dans la suite de notre travail, nous devons poursuivre par l'implantation des preuves qui démontrent, d'une part, la bisimulation entre les systèmes de transitions générés par les opérateurs de composition avec les systèmes de transitions décrivant la sémantique des patrons, pour un niveau de complexité donné. D'autre part, la préservation de la sémantique, lors de l'augmentation de la complexité des patrons, doit être démontrée.

Pour démontrer la bisimulation, nous implantons les fonctions nécessaires à la vérification des règles définissant la bisimulation entre 2 systèmes de transitions. Pour démontrer la preuve de préservation par récurrence, le principe est le suivant. Soit $\mathcal{C}(k_{pre}, k_{post})$, représentant un niveau de complexité pour une propriété spécifiée avec les clauses Pre et $Post$ référençant respectivement k_{pre} et k_{post} évènements. Soit $\mathcal{C}(k_{pre} + 1, k_{post})$ (resp. $\mathcal{C}(k_{pre}, k_{post} + 1)$), représentant un niveau de complexité pour cette propriété avec la clause Pre (resp. $Post$) référençant $k_{pre} + 1$ (resp. $k_{post} + 1$) évènements. Nous devons démontrer que l'ajout de l'évènement supplémentaire, soit dans Pre , soit dans $Post$, donnera lieu, lors de la composition, à la génération de toutes les transitions qui auraient été ajoutées dans les systèmes de transitions de la sémantique de référence. Ce qui revient à démontrer que la bisimulation est préservée. Cette démonstration doit s'effectuer pour tous les types de patrons (*Response*, *Precedence*, *Absence* et *Existence*) et toutes les combinaisons avec les options. Dans la suite de notre travail, nous implanterons les règles démontrant cette préservation.

6 Conclusion et travaux futurs

Une sémantique des patrons CDL a été présentée, basée sur une approche compositionnelle. Nous formalisons la composition d'éléments de base constitutifs des patrons de propriétés

³On note $src(t)$ (resp. $tgt(t)$): état source (resp. cible) de la transition t .

avec des éléments optionnels qui viennent enrichir les patrons. Des règles de composition, implantées en COQ pour les patrons de type *Response*, permettent de générer des systèmes de transitions bisimilaires aux systèmes de transitions de référence, décrivant la sémantique des patrons. L'implantation reste à être finalisée pour les autres patrons de type *Precedence*, *Absence* et *Existence*. Les fonctions COQ prouvant la bisimulation sont encore à valider, Aussi, les fonctions permettant d'exécuter la preuve par récurrence sont encore à implanter. Ce travail nous permet de disposer d'un cadre formel de construction de propriétés plus complexes à partir d'éléments de base et pouvoir valider formellement les transformations. Par la suite, nous pourrions donc continuer à concevoir des extensions de CDL en rajoutant des options supplémentaires. Le principe de construction d'un observateur par composition peut aussi s'étendre à la composition de plusieurs observateurs afin de pouvoir exprimer des propriétés plus complexes basées sur les propriétés élémentaires. Compte tenu du type d'exigences que nous avons à traiter dans certaines applications industrielles, notre objectif est d'expérimenter ces extensions pour mener des actions de validation formelle sur les modèles industriels.

References

- [1] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, pages 83–96, Twente, June 1993. Workshops in Computing, Springer Verlag.
- [2] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *21st Int. Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.
- [3] R. Smith, G.S. Avrunin, L. Clarke, and L. Osterweil. Propel: An approach supporting property elucidation. In *24th Int. Conf. on Software Engineering (ICSE02), St Louis, MO, USA*, pages 11–21. ACM Press, 2002.
- [4] S. Konrad and B. Cheng. Real-time specification patterns. In *27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA*, 2005.
- [5] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Adv. Software Engineering*, 2012:547157:1–547157:13, 2012.
- [6] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra Van Der Stappen. Model checking for managers. In *SPIN*, pages 92–107, 1999.
- [7] L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen. The power of reachability testing for timed automata. In *Proc. 18th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'98), Chennai, India, Dec. 1998*, volume 1530, pages 245–256. Springer, 1998.
- [8] S. Taha, J. Julliand, F. Dadeau, K. C. Castillos, and B. Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. In *Formal Aspects of Computing*, volume 27, 2015.

Formalisation des interactions asynchrones

Florent Chevrou

IRIT/Acadie – Université de Toulouse

Résumé

Dans les systèmes distribués, la communication asynchrone se divise en de multiples variantes. Ces modèles de communication, caractérisés par les propriétés d'ordre qu'ils engendrent sur la délivrance des messages qu'ils acheminent, jouent un rôle crucial dans la compatibilité de compositions de pairs. Nous spécifions ces modèles de manière uniforme avec TLA^+ ainsi que Event-B. Cette approche illustre les différences entre des modèles bien souvent confondus. En terme de conservation de la compatibilité, il est intéressant de connaître quels modèles sont interchangeable. Cela permet de substituer un modèle par un autre à la spécification plus simple pour faciliter des preuves formelles de cas particuliers. Nous établissons notamment une hiérarchie de raffinements. Elle est prouvée partiellement avec le TLA^+ Proof System et étendue à d'autres approches de modélisation sous Event-B.

1 Introduction

Construire des systèmes par assemblage de composants distribués est un principe courant duquel émane le problème de la compatibilité des différents pairs qui communiquent entre eux. Le modèle qui régit ces interactions agit directement sur les propriétés globales du système. Nous nous intéressons aux interactions asynchrones point à point qui se divisent en de multiples modèles de communication modélisés avec TLA^+ [Lam02] et Event-B [Abr10] présentés section 2 ainsi qu'à leur relations étudiées section 3.

2 Modèles de communication

En communication asynchrone, les événements d'envoi et de réception d'un message ne sont pas simultanés (contrairement à la communication synchrone). Les modèles de communication sont caractérisés par l'ordre de délivrance des messages. Nous proposons trois approches de spécification différentes. La première est basée sur une traduction de ces contraintes d'ordonnancement sous forme de propriétés sur les **exécutions distribuées** (séquences d'événements de communication). La seconde approche, opérationnelle, consiste en des implantations *uniformes* (pour faciliter l'étude formelle) et *distribuées* basées sur des **histoires de**

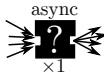
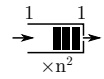
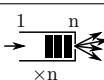
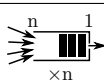
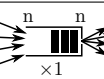

Modèle	Spécification	Implantation concrète
M_{async}	Asynchrone. Aucun ordre n'est imposé sur la délivrance des messages.	Un multi-ensemble duquel les messages sont extraits de façon arbitraire. 
M_{1-1}	Messages d'un couple émetteur/récepteurs délivrés dans l'ordre d'émission.	Une file de messages pour chaque couple de pairs. 
M_{causal}	Messages délivrés en suivant la causalité de leur émission.	Histoires causales ou matrices d'horloges logiques.
M_{1-n}	Messages d'un même pair délivré dans l'ordre d'émission.	Une file de messages sortants pour chaque processus. Retrait instantané. 
M_{n-1}	Messages d'un pair donné délivrés dans l'ordre global d'émission.	Une file de messages entrants pour chaque processus. Dépôt instantané. 
M_{n-n}	Messages délivrés dans l'ordre global d'émission.	Une file unique dans laquelle tous les messages transitent. 
M_{RSC}	Messages délivrés instantanément après leur émission.	Un tampon unique d'un message. 

TABLE 1 – Description des modèles de communication

messages : chaque message en transit transporte l'ensemble de ceux dont il dépend. Cette dernière approche est utilisée dans une chaîne d'outils de vérification automatique avec TLA^+ (*model checking*) de compatibilité sur des cas particuliers de composition de paires [CHQ15]. Enfin, nous proposons des *implantations concrètes* (parfois non distribuées) basées sur des structures de données usuelles (files, compteurs, ensembles). Les modèles sont récapitulés dans le tableau 1.

3 Raffinements

Connaître les rapports entre les modèles permet d'identifier les cas où ils sont substituables en conservant les propriétés de compatibilité qu'ils induisent (pour faciliter par exemple une preuve en utilisant le modèle le plus commode). Établir les relations de raffinement est la première étape, essentielle pour cet objectif, que nous avons réalisée. Les résultats obtenus sont synthétisés sur le diagramme Figure 1. Une hiérarchie se dessine de la frontière avec le monde synchrone (M_{RSC})

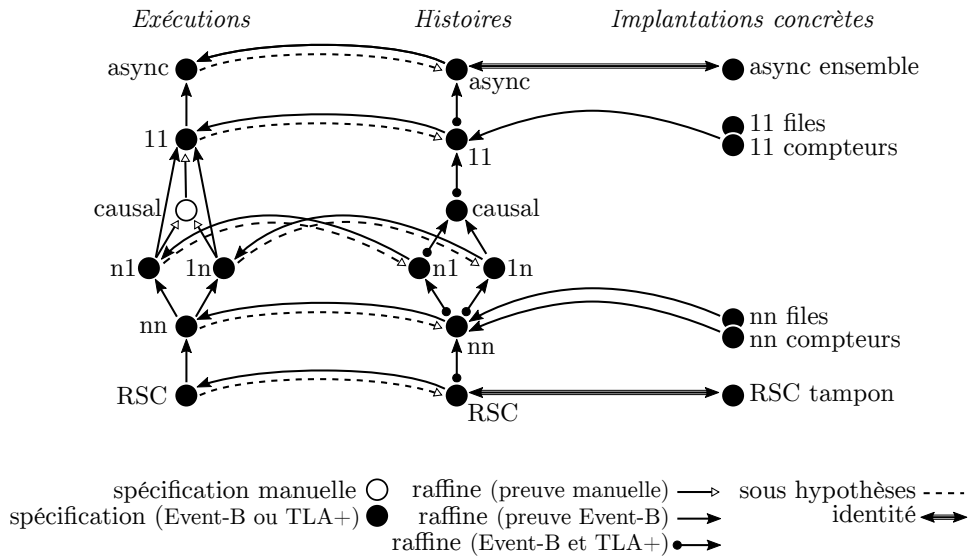


FIGURE 1 – Relations entre les modèles de communication

à l'asynchronisme pur (M_{async}). Nous l'avons prouvée formellement à l'aide de TLA⁺ [CHMQ16] et Event-B et représentée verticalement sur le diagramme pour chaque approche de spécification. Nous avons montré que les implantations uniformes avec histoires génèrent des exécutions distribuées qui respectent les propriétés d'ordre qui spécifient les modèles. Sous hypothèse (réception de tout message), nous prouvons (manuellement) qu'elles génèrent toutes les exécutions légales. Nous travaillons à étendre ces liens, représentés horizontalement sur le diagramme, aux implantations concrètes. La spécification de l'ordre causal et les preuves associées dans les outils formels se révèle délicate (pastille blanche sur le diagramme). Enfin, le passage à certaines implantations concrètes qui nécessitent des structures de données spécifiques complexifie les preuves de raffinement.

4 Conclusion

Nous avons établi des cadres formels et unifiés de spécification de modèles de communication asynchrone et prouvé la plupart des relations de raffinement qui lient ces modèles et ces approches de spécification. Nos résultats obtenus et attendus complètent des travaux existants sur la hiérarchisation de modèles de communication : nous considérons des modèles supplémentaires par rapport à [CBMT96] et ne nous restreignons pas à des spécifications de systèmes sans indéterminisme [EMR97]. Nous travaillons à compléter les liens manquants, notamment autour du modèle causal, et établir ainsi les équivalences de modèles vis-à-vis de la conservation de propriétés stables (ex. terminaison) sur les systèmes distribués.

Références

- [Abr10] Jean-Raymond Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
- [CBMT96] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4) :173–191, February 1996.
- [CHMQ16] Florent Chevrou, Aurélie Hurault, Philippe Mauran, and Philippe Quéinnec. Mechanized refinement of communication models with TLA+. In *ABZ 2016*, 2016.
- [CHQ15] Florent Chevrou, Aurélie Hurault, and Philippe Quéinnec. Automated verification of asynchronous communicating systems with TLA+. *Electronic Communications of the EASST (PostProceedings of the 15th International Workshop on Automated Verification of Critical Systems)*, 72, 2015.
- [EMR97] A. Engels, S. Mauw, and M.A. Reniers. A hierarchy of communication models for message sequence charts. In *Science of Computer Programming*, pages 75–90. Chapman & Hall, 1997.
- [Lam02] Leslie Lamport. *Specifying Systems*. Addison Wesley, 2002.

Démonstration d'outils : SIMPA

Simpa Infers Models Pretty Automatically

Catherine Oriat Roland Groz
Emmanuel Perrier

Université Grenoble Alpes – LIG, France

{Emmanuel.Perrier, Roland.Groz, Catherine.Oriat}@imag.fr

Résumé

SIMPA (Simpa Infers Models Pretty Automatically) est une plateforme d'inférence de modèles de composants logiciels sous forme d'automates développée dans l'équipe VASCO du LIG. Elle a été créée initialement dans le cadre du projet européen SPaCIoS, dont l'objectif était la découverte de vulnérabilités Web. Le modèle peut en effet alimenter des outils de vérification ou de test etc.

L'outil permet d'inférer un modèle comportemental d'un système qui peut être par exemple un composant logiciel ou un service Web. Il s'agit d'inférence *active* en *boîte noire*, qui s'effectue en testant le système en lui fournissant des entrées et en observant les sorties.

L'outil permet de générer différents types d'automates (machine de Mealy, machines d'états finies étendues) à l'aide de plusieurs algorithmes.

L'interface entre SIMPA avec des systèmes réels se fait au moyen de drivers. Simpa intègre également un outil de génération de drivers Web qui automatise en grande partie l'inférence d'applications Web.

1 Contexte

Les activités rigoureuses de validation et vérification sont généralement basées sur des *modèles* du logiciel. Mais il est fréquent que l'on n'en dispose pas, soit parce que ces modèles n'ont jamais été écrits, soit parce que l'évolution ou la maintenance du logiciel a fait que les modèles écrits au départ ne sont plus cohérents avec le code.

L'*inférence* de modèles vise à générer automatiquement un modèle à du système. On distingue l'inférence *passive*, pour laquelle un ensemble d'observations est donné au départ, et l'inférence active, pour laquelle les données d'entrées du système sont choisies au fur et à mesure de l'inférence en fonctions des sorties observées. On s'intéresse ici à l'inférence active de modèles pour des systèmes à entrées/sorties, où le logiciel est considéré comme une *boîte noire* dont on ne connaît au départ que les entrées possibles. En particulier, il n'est pas nécessaire de disposer du code source ou binaire du logiciel, ce qui permet de construire des modèles également pour des systèmes accessibles uniquement à travers un réseau, tel que des applications Web.

2 Plateforme SIMPA

SIMPA (Simpa Infers Models Pretty Automatically) est une plateforme d'inférence de modèles créée initialement par Karim Hossen lors de sa thèse [4] dans le cadre du projet européen SPaCIoS¹, dont le but était la détection de vulnérabilités dans les applications Web.

1. <http://spacios.eu>

La figure 1 montre l'architecture de la plateforme SIMPA. L'outil permet d'inférer différents types d'automates, en particulier des *machines de Mealy* i.e. des automates à entrées et sorties, et des machines d'états finies étendues à l'aide d'entrées paramétrées et de gardes (EFSM).

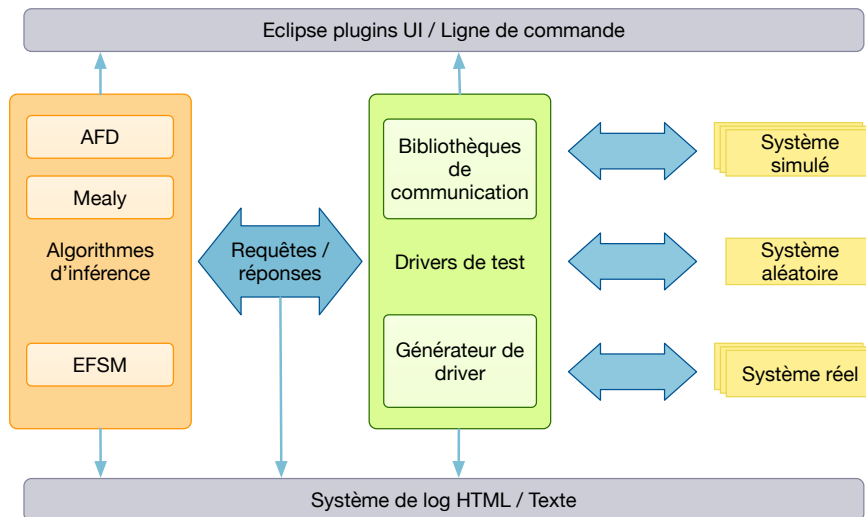


FIGURE 1 – Architecture de SIMPA [4]

Algorithmes d'inférence En inférence active, l'algorithme fondateur est l'algorithme L^* d'Angluin [2], qui permet d'inférer des automates reconnaisseurs. Cet algorithme est basé sur deux types de requêtes : les requêtes *d'appartenance*, qui correspondent à un test d'une séquence sur le logiciel, et les requêtes *d'équivalence*, qui permettent de déterminer si l'automate inféré est équivalent au système analysé, et fournit un *contre-exemple* si ce n'est pas le cas.

SIMPA implémente différents algorithmes d'inférence :

- l'algorithme Lm^* [9], extension pour les machines de Mealy de l'algorithme L^* ;
- l'algorithme *Z-Quotient* [7], basé sur une représentation arborescente des entrées/sorties, plus performant que les algorithmes basés sur des tables comme L^* ou Lm^* ;
- l'algorithme d'inférence sur les EFSM [6, 8], qui permet d'inférer des systèmes où les entrées sont paramétrées, et qui peuvent avoir des variables internes ; en outre il prend en compte des concepts importants pour la modélisation de protocoles cryptographiques, comme les *nonces* et les identificateurs de session ;
- l'algorithme *NoReset* [3], qui permet d'inférer des machines de Mealy à partir de systèmes qu'il n'est pas possible de réinitialiser (tous les autres algorithmes supposent qu'on peut réinitialiser la boîte noire).

SIMPA contient également différents algorithmes de traitement des contre-exemples.

Écriture de drivers Les algorithmes d'inférence interagissent avec le système à inférer par l'intermédiaire de composants d'interfaçage appelés *drivers* dans la terminologie de SIMPA (cf. figure 1). D'autres logiciels de test à base de modèles ou d'inférences parlent aussi d'adaptateurs (ou "mappers" ou "wrappers"). Dans le cas d'un système quelconque, cette interface entre SIMPA et ce système doit être écrite à la main. Dans le cas des applications Web, SIMPA fournit un générateur de drivers, basé sur un collecteur (*crawler*), qui permet de déterminer automatiquement les entrées possibles du système par un parcours de l'ensemble des pages html qui le composent.

Autres utilitaires À des fins d'évaluation des algorithmes, SIMPA inclut également des générateurs d'automates (aléatoires, ou semi-aléatoires). On peut aussi fournir des systèmes dont on a déjà un modèle de comportement. Il fournit, pour ces systèmes, les drivers adéquats.

Données sur l'outil SIMPA SIMPA est écrit en Java, et comporte plus de 200 classes Java et plus de 30000 lignes de code. L'architecture de l'outil permet d'étendre assez facilement celui-ci à d'autres types d'automates ou d'autres algorithmes ainsi que de s'appliquer à différents types de système pour lesquels il est possible de créer un driver.

Outils similaires Parmi les plateformes d'inférence active, on peut citer *LearnLib* [5], développé à l'Université de Dortmund. L'outil *Tomte* [1], développé à l'Université de Nijmegen, permet de réaliser automatiquement l'interface entre le système à inférer et LearnLib.

3 Résultats

Les différents algorithmes d'inférence de SIMPA ont été testés sur différents systèmes : des automates simulés, des automates aléatoires, ainsi que des systèmes réels.

À titre d'exemple, nous illustrons les modèles inférés pour deux serveurs SIP (téléphonie sur Internet) accédés par le réseau : iptel.org et SIP2SIP (figure 2). Le protocole SIP est un protocole standard de télécommunication qui permet d'établir, de modifier et de terminer des sessions multimédias. Ce protocole comporte 14 types de requêtes. Dans notre expérimentation, nous avons considéré uniquement les 4 requêtes les plus utilisées : *Register*, *Invite*, *Ack* et *Bye*.

Les deux automates ont été inférés en utilisant l'algorithme Z-Quotient, avec 26 requêtes et un contre-exemple (*Invite Invite*).

Bien que les deux serveurs implémentent le même protocole, on peut remarquer un traitement différent pour une suite de quatre *Invite*. De même, un appel à *Bye* dans l'état initial (*S0*) est traité différemment par les deux systèmes.

Notre outil permet donc de découvrir des différences subtiles entre différentes implémentations d'un même protocole, ce qui peut servir par exemple à vérifier l'interopérabilité de plusieurs composants d'un système.

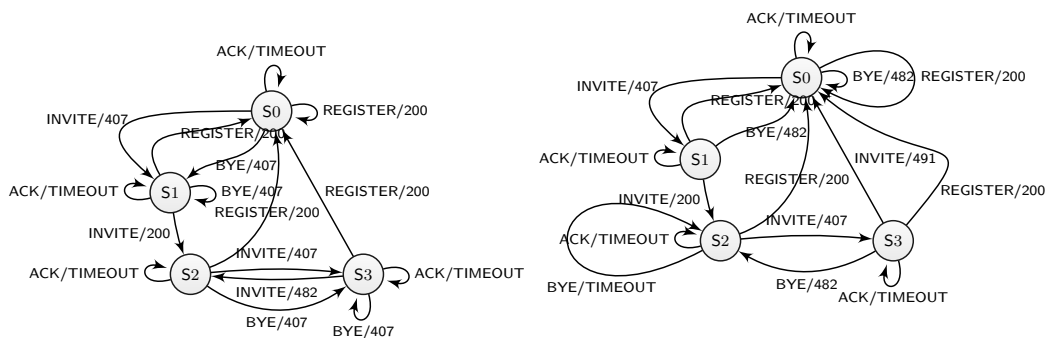


FIGURE 2 – Automates inférés par les deux implémentations du protocole SIP

4 Conclusion

La plateforme SIMPA permet d'inférer des modèles du comportement de systèmes réels ou simulés à l'aide de différents algorithmes d'inférence active, dans la lignée de l'algorithme L^*

d'Angluin. SIMPA implémente en particulier différents traitements des contre-exemples. Dans le cas des applications Web, SIMPA fournit également des drivers permettant de faire l'interface entre les algorithmes d'inférence et le système réel.

L'outil nous permet de réaliser des expérimentations à plusieurs niveaux. Il nous permet de tester et comparer différents algorithmes d'inférences, à l'aide par exemple de modèles aléatoires ou semi-aléatoires. Il nous permet aussi d'obtenir des modèles sous forme d'automates simples ou étendus avec des paramètres et des données pour des systèmes logiciels qui peuvent être accédés à distance à travers un réseau, ou être accédés via une API locale. Il nous permet également de tester des algorithmes de recherche de vulnérabilités basés sur de l'inférence active de modèles.

Références

- [1] F. Aarts, F. Heidarian, H. Kuppens, P. Olsen, and F.W. Vaandrager. Automata learning through counterexample-guided abstraction refinement. In *Proceedings 18th International Symposium on Formal Methods (FM 2012)*, LNCS 7436, pages 10–27, aug 2012.
- [2] Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2 :87–106, 1987.
- [3] Roland Groz, Adenilso Simao, Alexandre Petrenko, and Catherine Oriat. Inferring finite state machines without reset using state identification sequences. In *Proceedings of the International Conference on Testing Software and Systems, ICTSS 2015*, Dubai, nov 2015.
- [4] Karim Hossen. *Inférence automatique de modèles d'applications Web et protocoles pour la détection de vulnérabilités*. PhD thesis, Université de Grenoble, France, dec 2014.
- [5] Malte Isberner, Falk Howar, and Bernhard Steffen. The open-source LearnLib - A framework for active automata learning. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 487–495, 2015.
- [6] K. Li, R. Groz, and M. Shahbaz. Integration testing of distributed components based on learning parameterized I/O models. In *FORTE2006 Paris*, LNCS 4229, pages 436–450, Paris, september 2006.
- [7] Alexandre Petrenko, Keqin Li, Roland Groz, Karim Hossen, and Catherine Oriat. Inferring approximated models for systems engineering. In *15th IEEE International Symposium on High Assurance Systems Engineering (HASE 2014)*, pages 249–253, Miami, Florida, USA, jan 2014.
- [8] SPaCIoS European Project. Deliverable D2.2.1, Method for Assessing and Retrieving Models. <http://spacios.eu/deliverables/spacios-d2.2.1.pdf>, 2013.
- [9] Muzammil Shahbaz and Roland Groz. Inferring Mealy machines. In *Formal Methods 2009*, pages 207–222, Eindhoven, Netherland, November 2009.

(Démon) DynIBEX: une boîte à outils pour la vérification des systèmes cyber-physiques *

Julien Alexandre dit Sandretto et Alexandre Chapoutot

U2IS, ENSTA ParisTech, Université Paris-Saclay, 828 bd des Maréchaux,
91762 Palaiseau Cedex

22 avril 2016

Résumé

Les systèmes cyber-physiques requièrent des méthodes de vérification nouvelles. En particulier, des preuves de propriétés fonctionnelles nécessitent la prise en compte de l'environnement physique qui évolue de manière continue dans le temps, souvent décrit par des équations différentielles. Outre des méthodes d'intégration numérique, il est également nécessaire de pouvoir calculer avec des ensembles de valeurs afin de prouver des propriétés pour toutes les exécutions. DynIBEX est une bibliothèque libre qui offre un ensemble de méthodes permettant l'étude des systèmes cyber-physiques en fournissant des méthodes d'intégration ensembliste combinées avec des méthodes de résolution de contraintes intervalles.

1 Introduction

Les systèmes cyber-physiques sont des systèmes où des éléments informatiques potentiellement distribués collaborent pour le contrôle et la commande d'entités physiques. La conception de la partie informatique de ces systèmes nécessite des outils nouveaux afin de prendre en compte la relation forte avec le monde physique. En particulier, la preuve de propriétés fonctionnelles de la partie informatique ne peut pas se faire en indépendance de la connaissance du monde physique.

Nous proposons une bibliothèque libre et open-source, nommée DynIBEX¹, contenant un ensemble de briques élémentaires pour analyser et étudier des systèmes cyber-physiques. Cette bibliothèque est une extension de la bibliothèque IBEX²

* Cette recherche a bénéficié du support de la "Chaire Ingénierie des Systèmes Complexes - École polytechnique, THALES, DGA, FX, DASSAULT AVIATION, DCNS Research, ENSTA ParisTech, Télécom ParisTech, Fondation ParisTech, FDO ENSTA"

1. <http://perso.ensta-paristech.fr/~chapoutot/dynibex/>

2. <http://www.ibex-lib.org>

[4] de résolution de problèmes à contraintes sur les réels. Cette extension ajoute des méthodes de résolution d'équations différentielles ordinaires [1] et des équations algébro-différentielles [2] d'index 1. Autrement dit, DynIBEX est capable de résoudre des problèmes de la forme

$$\dot{\mathbf{y}}(t) = f(\mathbf{y}(t), \mathbf{p}, \mathbf{u}),$$

i.e., des équations différentielles ordinaires avec des paramètres \mathbf{p} à valeurs bornées et un contrôle \mathbf{u} . Également, il est capable de résoudre des problèmes différentiels de la forme

$$\begin{cases} \dot{\mathbf{y}} = f(\mathbf{y}, \mathbf{x}, \mathbf{p}, \mathbf{u}) \\ 0 = g(\mathbf{y}, \mathbf{x}, \mathbf{q}) \end{cases},$$

i.e., des équations algébro-différentielles d'index 1 avec des paramètres \mathbf{p} et \mathbf{q} à valeurs bornées et un contrôle \mathbf{u} . A noter que ces problèmes sont d'une complexité plus importante que les équations différentielles ordinaires car la fonction \mathbf{x} , inconnue, est définie implicitement. Ce genre d'équations est très répandues dans les outils comme Modelica.

La bibliothèque DynIBEX peut être utilisée dans différents contextes. D'une part, elle peut être utilisée comme un moteur de simulation. D'autre part, le couplage d'un solveur de contraintes sur les réels et de méthodes de résolution d'équations différentielles peut être utilisé pour analyser ou vérifier les systèmes cyber-physiques. Par exemple, cette bibliothèque a été utilisée pour concevoir des algorithmes de contrôle dans [3].

2 Principales fonctionnalités

La bibliothèque DynIBEX repose sur un moteur de calculs intervalles dans lequel il existe différentes façons de représenter des ensembles (intervalles et zonotopes) et de les manipuler (arithmétique, union, etc.). De plus, les méthodes standards issues de l'analyse par intervalles [6] y sont présentes comme le pavage [4].

Nous présentons les principales caractéristiques de DynIBEX au travers d'un exemple dans la suite de cette section.

L'ajout de méthodes de résolution numériques ensemblistes pour des équations différentielles permet de pouvoir modéliser et simuler des systèmes cyber-physiques. Nous donnons un exemple de code source DynIBEX dans le listing 1. Celui-ci représente la dynamique non-linéaire de la vitesse d'une voiture prenant en compte une incertitude bornée sur sa masse (entre les lignes 22 et 24). A cette dynamique s'ajoute un régulateur de vitesse suivant un algorithme de type proportionnel-intégral (PI) aux lignes 18 et 19. Le moteur de simulation est appelé aux lignes 27 et 28. Cet exemple représente donc un système hybride avec un contrôleur discret et une dynamique continue.

Le moteur de simulation de DynIBEX permet de calculer des tubes (i.e., des ensembles de trajectoires) pour les systèmes cyber-physiques. L'étude de propriétés

Listing 1 – Exemple de programme DynIBEX : régulation en boucle fermée, contrôleur PI, de la vitesse d'une voiture avec incertitude sur la masse

```

1  int main(){
2
3  const int n = 1;
4  Variable y(n);
5
6  IntervalVector state(n);
7  state[0] = 0.0;
8
9  double t = 0;
10 const double sampling = 0.005;
11 Affine2 integral(0.0);
12
13 while (t < 5.0) {
14     Affine2 goal(10.0);
15     Affine2 error = goal - state[0];
16
17     // Controleur PI
18     integral = integral + sampling * error;
19     Affine2 u = 1400.0 * error + 70.0 * integral;
20
21     // Dynamique d'une voiture avec incertitude sur sa masse
22     Function ydot(y, (u.itv() - 50.0 * y[0] - 0.4 * y[0] * y[0])
23                 / Interval(990, 1010));
24     ivp_ode vdp = ivp_ode(ydot, 0.0, state);
25
26     // Integration numerique ensembliste
27     simulation simu = simulation(&vdp, sampling, HEUN, 1e-5);
28     simu.run_simulation();
29
30     // Verification de surete
31     IntervalVector safe(n);
32     safe[0] = Interval(0.0, 11.0);
33     bool flag = simu.stayed_in(safe);
34     if (!flag) {
35         std::cerr << "Error_safety_violation" << std::endl;
36     }
37
38     // Mise a jour du temps et des etats
39     state = simu.get_last_aff();
40     t += sampling;
41 }
42
43 return 0;
44 }
    
```

des systèmes cyber-physiques est alors possible en vérifiant que ces tubes respectent certaines propriétés. Par exemple, aux lignes 31 et 36, nous vérifions que les valeurs minimale et maximale de la vitesse de la voiture régulée sont comprises entre 0 et 11 pour tous les instants du temps. Un ensemble d'opérations similaires à la fonction `stayed_in` existe dans la bibliothèque DynIBEX pour vérifier, par exemple, que les trajectoires n'atteignent pas une zone dangereuse ou ne sortent pas d'une certaine zone.

Pour terminer, nous mentionnons qu'il est également possible d'associer des contraintes intervalles aux équations différentielles dans la bibliothèque DynIBEX profitant ainsi du moteur de résolution de contraintes de la bibliothèque IBEX.

3 Conclusion

La bibliothèque DynIBEX est une première brique générique pour étudier et analyser les systèmes cyber-physiques. Cependant, encore beaucoup de travail reste à effectuer pour entre autre

- mieux modéliser et simuler les systèmes cyber-physiques, i.e., avec des conditions de transition dépendantes des états des équations différentielles ;
- avoir un outil de vérification formelle libre du type “bounded model-checking” (BMC) en intégrant cette bibliothèque avec des moteurs SMT. Bien que ce genre d'outils existe déjà, par exemple dReal [5], nous pensons que de nouvelles perspectives s'ouvrent pour obtenir des algorithmes plus efficaces grâce à notre combinaison, en une seule bibliothèque, du moteur de résolution de contraintes et du moteur de résolution d'équations différentielles alors que dReal utilise deux outils distincts.

Références

- [1] J. Alexandre dit Sandretto and A. Chapoutot. Validated explicit and implicit Runge-Kutta methods. under submission, 2015.
- [2] J. Alexandre dit Sandretto and A. Chapoutot. Validated simulation of differential algebraic equations with Runge-Kutta methods. under submission, 2015.
- [3] J. Alexandre dit Sandretto, A. Chapoutot, and O. Mullier. Tuning PI controller in non-linear uncertain closed-loop systems with interval analysis.
- [4] Gilles Chabert and Luc Jaulin. Contractor programming. *Artificial Intelligence*, 173(11) :1079–1100, 2009.
- [5] S. Gao, S. Kong, and E. M. Clarke. dReal : An SMT Solver for Nonlinear Theories over the Reals. In *CADE*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013.
- [6] L. Jaulin, M. Kieffer, O. Didrit, and E. Walter. *Applied Interval Analysis*. Springer, 2001.

Préservation de la cohérence des transformations topologiques et géométriques

Valentin Gauthier¹, Thomas Bellet², Hakim Belhaouari¹ et Agnès Arnould¹

¹ Université de Poitiers, XLim, ² École Centrale Paris, MICS

Résumé

Dans des travaux antérieurs, nous avons proposé un formalisme basé sur les transformations de graphes pour définir des opérations géométriques à base topologique. Une règle de transformation définit une opération sur une orbite topologique, telle que la triangulation d'une face, ou la translation d'une composante connexe. Elle définit les nouveaux plongements (dont la géométrie) à l'aide d'expressions de plongement. Des conditions syntaxiques garantissent la préservation de la cohérence topologique des objets d'une part et de leurs plongements d'autre part. Cet article présente l'utilisation conjointe des orbites topologiques et des expressions de plongement. Il étend les conditions syntaxiques pour garantir simultanément la cohérence de la topologie et des plongements.

Mots clés : transformation de graphes, modélisation géométrique à base topologique, conditions syntaxiques, préservation de la cohérence.

1 Introduction

Les modelleurs géométriques permettent de construire des objets complexes pour des applications variées (résistance des matériaux, écoulement de fluides, simulation chirurgicale, reconstruction du sous-sol...). Pour cela, ils s'appuient sur des modèles à base topologique décrivant les objets modélisés par leur structure topologique (leur décomposition en sommets, arêtes, faces, volumes, etc.) et leurs plongements (valeurs géométriques et physiques - par exemple les coordonnées des sommets, les couleurs des faces, la densité des volumes). Les opérations que subissent les objets lors d'une construction ou d'une simulation doivent préserver la cohérence des objets.

Des travaux antérieurs ont proposé l'utilisation des transformations de graphes pour définir les opérations sur les objets (*cf.* [Pou09] et [Bel12]). Ce formalisme offre la possibilité de définir de façon graphique et concise des opérations parfois complexes, rendant possible le prototypage rapide de nouvelles opérations. Il permet également de définir des conditions syntaxiques garantissant par construction la préservation de la cohérence des objets modélisés. Cet article présente les conditions à vérifier lorsque les opérations modifient à la fois la structure topologique et les plongements des objets.

Après une présentation du modèle des objets en section 2, nous présentons en section 3 les transformations topologiques et les conditions pour préserver la cohérence topologique. Symétriquement, la section 4 présente les transformations des plongements et les conditions associées. Nous combinons les deux aspects (topologie et plongement) dans la section 5 et donnons les conditions nécessaires à la préservation de la cohérence. Enfin, nous concluons dans la section 6.

2 Objets géométriques à base topologique

Le modèle topologique utilisé est celui des cartes généralisées [Lie94]. Cette structure topologique est issue de la décomposition des objets en cellules topologiques de dimensions décroissantes (sommets, arêtes, faces, volumes, etc.). Par exemple l'objet 2D en Figure 1a se décompose en faces (cellules de dimension 2) reliées entre elles par des arcs de dimension 2 (*cf.* Figure 1b). Puis les faces se décomposent en arêtes reliées par des arcs de dimension 1 (*cf.* Figure 1c). Enfin, la décomposition des arêtes en

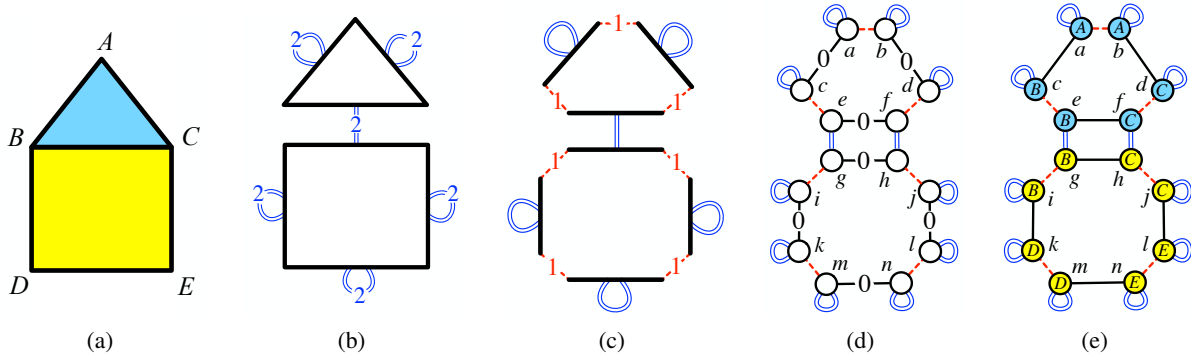


FIGURE 1 – Décomposition d'un objet géométrique 2D

sommets reliés par des arcs de dimension 0 (cf. Figure 1d) donne les nœuds du graphe. Ainsi les graphes manipulés ont leurs arcs étiquetés sur $[0, n]$ où n est la dimension topologique des objets représentés. Les objets pris pour exemple dans cet article étant de dimension 2, leurs arcs seront étiquetés sur $[0, 2]$ et seront représentés par le code couleur de la Figure 1.

Les cellules topologiques des objets sont définis par des sous-graphes appelés orbites :

Orbite : pour o un sous-ensemble de $[0, n]$ et v un nœud d'un graphe G de dimension n , l'orbite $\langle o \rangle(v)$ est le sous-graphe de G contenant v , les nœuds de G atteignables par des arcs étiquetés sur o , ainsi que ces arcs. On dit que $\langle o \rangle(v)$ est de type $\langle o \rangle$, ou est une $\langle o \rangle$ -orbite.

Par exemple sur la Figure 2, le sommet B est défini par le sous-graphe atteignable à partir du nœud e et des arcs de dimensions 1 et 2, c'est-à-dire l'orbite $\langle 1, 2 \rangle(e)$ de la Figure 2a. De même, l'arête BC est l'orbite $\langle 0, 2 \rangle(e)$ de la Figure 2b, et la face ABC est l'orbite $\langle 0, 1 \rangle(e)$ de la Figure 2c. La notion d'orbite n'est pas limitée aux cellules. Par exemples les Figures 2d et 2e présentent respectivement une demi-arête et une composante connexe.

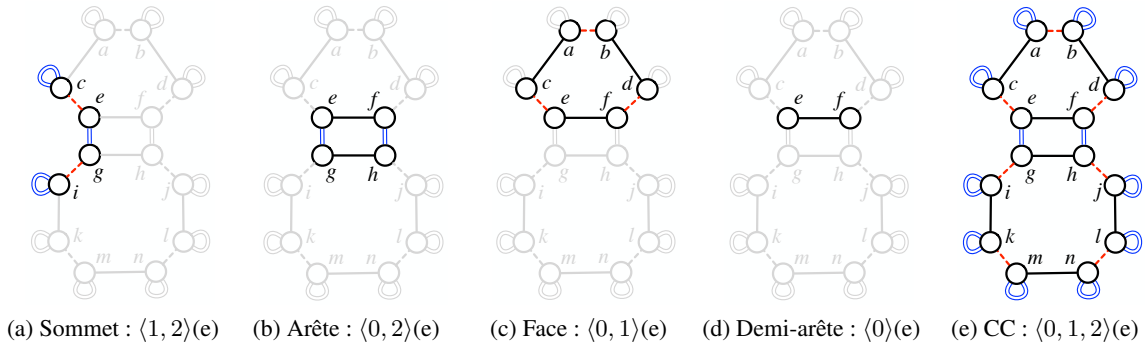


FIGURE 2 – Orbites incidentes à e

Les plongements sont pour leur part représentés par les étiquettes des nœuds du graphe. Par exemple, Figure 1e, les nœuds sont étiquetés par des points géométriques (A, B, C, D et E) et des couleurs (bleu et jaune). Chaque plongement se caractérise par le type de l'information (ici un point 2D et une couleur) et le type des orbites support du plongement (ici les sommets de type orbite $\langle 1, 2 \rangle$ ou $\langle 1, 2 \rangle$ -orbites et les faces de type orbite $\langle 0, 1 \rangle$ ou $\langle 0, 1 \rangle$ -orbites).

La cohérence d'un objet de dimension n est définie par les contraintes suivantes :

- **Non-orientation** : un objet est un graphe non orienté.
- **Arcs adjacents** : tout nœud est la source de $n + 1$ arcs étiquetés respectivement de 0 à n .
- **Cycle** : pour toutes dimensions i et j telles que $0 \leq i \leq i + 2 \leq j \leq n$, tout nœud est la source d'un cycle¹ étiqueté par iji .

1. Un arc ω a un nœud source $s(\omega)$, un nœud cible $t(\omega)$, et une étiquette $\alpha(\omega)$. Un chemin $\omega_1\omega_2\omega_3\omega_4$ est une suite d'arcs tel que $t(\omega_i) = s(\omega_{i+1})$ et est étiqueté par $\alpha(\omega_1)\alpha(\omega_2)\alpha(\omega_3)\alpha(\omega_4)$. Un cycle $\omega_1\omega_2\omega_3\omega_4$ est un chemin tel que $s(\omega_1) = t(\omega_4)$.

— **Plongements** : pour chaque plongement de support $\langle o \rangle$, tous les nœuds d'une même $\langle o \rangle$ -orbite sont étiquetés par la même valeur de plongement.

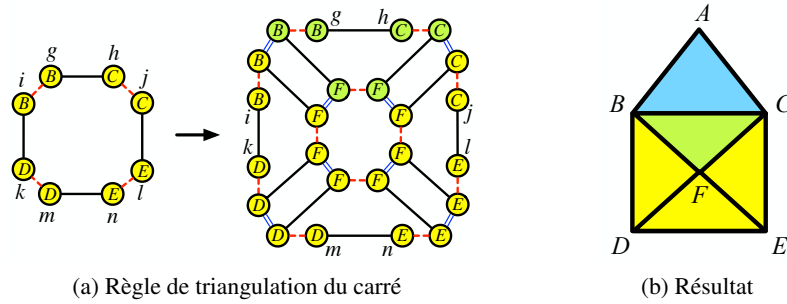


FIGURE 3 – Opération de triangulation

Les règles de transformation de graphe [EEPT06] permettent de définir des opérations sur les objets. Une règle de transformation de graphe $L \rightarrow R$ se compose de deux parties : le membre gauche L spécifie le motif filtré et le membre droit R spécifie le motif réécrit. Par exemple, la règle de la Figure 3a définit la triangulation du carré de l'objet de la Figure 1 dont le résultat est représenté en Figure 3b. Intuitivement, cette opération subdivise le carré en quatre triangles en introduisant un sommet F au barycentre de la face d'origine, et en coloriant les triangles créés en fonction des faces voisines (le triangle BCF est colorié en vert par mélange avec le bleu du triangle ABC alors que les autres, sans voisins, restent jaunes).

Cependant, une telle règle de transformation ne définit qu'une application donnée d'une opération de modélisation (dans le cas présent, la triangulation d'une face carrée jaune aux points B, C, D, E). Pour permettre une définition générique des opérations, les transformations de graphes permettent l'utilisation de variables [Hof05]. Dans les deux sections suivantes, nous présentons deux types de variables respectivement dédiées aux modifications de la topologie et des plongements.

3 Transformation topologique

Les variables topologiques permettent de définir génériquement les transformations topologiques. Elles sont basées sur la notion d'orbite. Par exemple, la triangulation d'une face quel que soit son nombre d'arêtes peut être définie par la règle avec variable topologique de la Figure 4a. Le membre gauche de la règle, effectuant le filtrage, possède désormais un seul nœud nommé v étiqueté par le type orbite face $\langle 0, 1 \rangle$. Ce nœud sera instancié par une face de l'objet à transformer, quelle qu'elle soit (par exemple un triangle ou un carré, cf. Figure 4b).

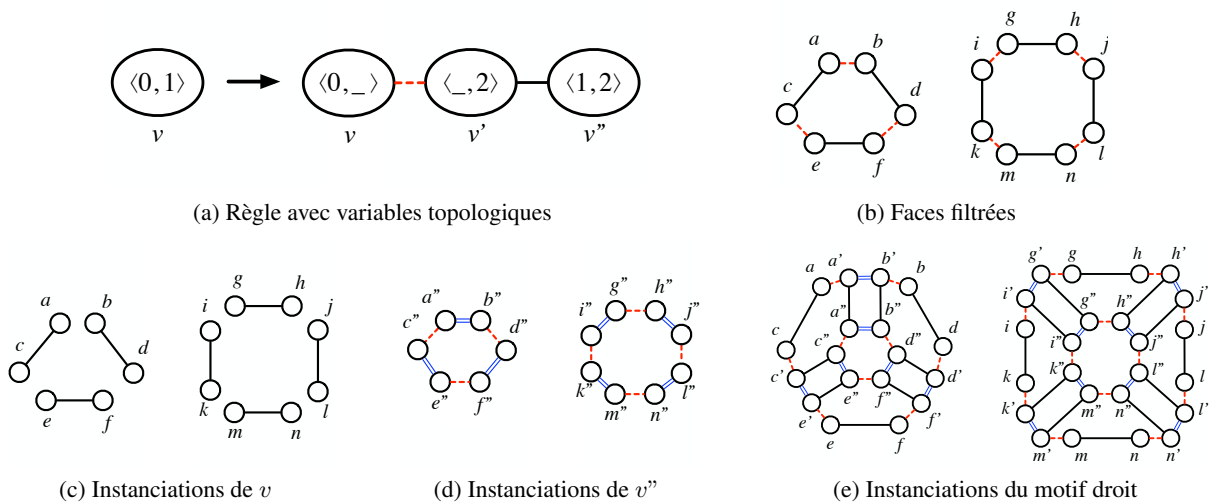


FIGURE 4 – Règle de triangulation

Dans le membre droit, chaque nœud abstrait une copie de l'orbite filtrée à renommage/suppression

des arcs près. Par exemple, le nœud v étiqueté par $\langle 0, _ \rangle$ signifie que les arcs de dimension 1 sont supprimés (notés ' $_$ ') dans l'instanciation (cf. Figure 4c) et donc que les arêtes seront séparées. De façon similaire, le nœud v'' est étiqueté par $\langle 1, 2 \rangle$. Par conséquent, tous les arcs de dimension 0 sont renommés en arcs de dimension 1, et tous les arcs de dimension 1 sont renommés en arcs de dimension 2, ce qui construit le sommet dual de la face triangulée, cf. Figure 4d. Enfin, notons que les arcs du motif droit permettent de relier deux à deux les copies instances des nœuds. Ainsi, la Figure 4e contient bien la face triangulée quelle que soit sa forme.

Afin de garantir la préservation de la cohérence topologique des objets lors de l'application d'une règle, des conditions syntaxiques sont définies dans [Pou09]. Par exemple, pour garantir la contrainte d'arcs adjacents, chaque nœud ajouté doit posséder toutes les dimensions. En particulier, le nœud v'' ajouté par la règle de la Figure 4a possède bien les trois dimensions 0, 1 et 2. La dimension 0 est portée par l'arc entre v' et v'' et relie dans l'objet les instances de v' à celles de v'' . Les dimensions 1 et 2 sont portées par le nœud lui-même et relient les instances de v'' entre elles par renommage des arcs filtrés.

4 Transformation des plongements

De façon similaire, une méthode est introduite dans [Bel12] pour calculer les nouveaux plongements dans une règle pour tous les objets transformés. Elle consiste à utiliser les nœuds de la partie gauche de la règle comme variables pour construire les expressions des nouveaux plongements. Ces expressions utilisent conjointement des opérateurs sur la structure topologique et sur les plongements.

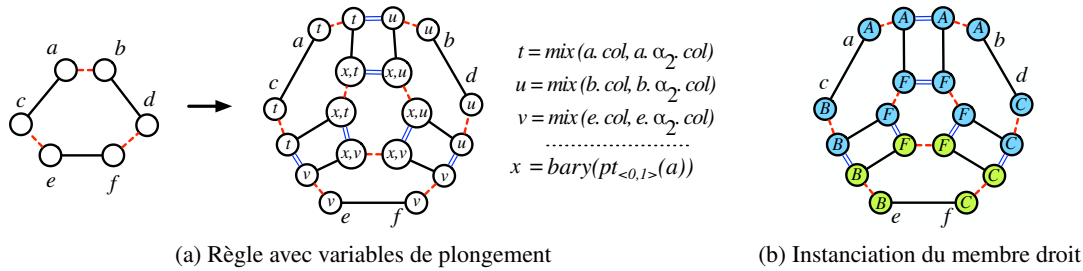


FIGURE 5 – Triangulation avec variables de plongement

Par exemple, dans la règle de la Figure 5a, la couleur du nœud a est réécrite à l'aide de l'expression de plongement $t = \text{mix}(a.col, a.\alpha_2.col)$. Outre l'accès aux plongements (dont $a.col$ qui permet d'accéder à la couleur du nœud a), les expressions contiennent différents opérateurs propres à la topologie, comme les opérateurs de voisinage ou les collectes. Par exemple, $a.\alpha_2$ permet l'accès au nœud voisin de a par l'arc de dimension 2 (unique grâce à la contrainte d'arc incident). Ou encore, $pt_{\langle 0,1 \rangle}(a)$ permet de collecter le multi-ensemble des plongements point de l'orbite face $\langle 0, 1 \rangle(a)$.

Les expressions de plongement permettent aussi de manipuler des opérateurs définis sur les types des plongements de l'utilisateur. Par exemple, mix représente un opérateur du type couleur qui permet de mélanger deux couleurs. Cet opérateur est ici utilisé par exemple sur a pour calculer sa nouvelle couleur comme le mélange de la sienne et celle de sa face voisine. De même, l'opérateur bary permet de calculer le barycentre d'un ensemble de points, ici celui des points de la face incidente à a .

Les différentes expressions de plongements sont évaluées lors de l'instanciation de la règle avec variables de plongement (cf. Figure 5a), sur le triangle ABC de l'objet de la Figure 1. La Figure 5b présente l'instance du membre droit de la règle. Ainsi l'expression v sur la face incidente à e , a été évaluée en vert qui est le mélange de la couleur jaune de e et de la couleur bleu de son voisin de dimension 2, g . De même, l'expression t est évaluée en jaune car a est son propre voisin de dimension 2, et donc le jaune est mélangé avec lui-même.

Des conditions syntaxiques de préservation de la cohérence des plongements sont également définies dans [Bel12]. Par exemple, tous les nœuds d'une même orbite support de plongement doivent porter la même expression de plongement. Sur la Figure 5a par exemple, tous les nœuds de l'orbite face $\langle 0, 1 \rangle(e)$

portent bien la même expression de plongement couleur v . De même, ces conditions interdisent la redéfinition d'une valeur de plongement si toute son orbite support n'est pas filtrée. Dans le cas présent, la règle respecte cette condition pour la redéfinition de la couleur car toute la face est filtrée.

Montrons maintenant comment ces variables s'utilisent avec les variables topologiques.

5 Combiner les transformations de topologies et de plongements

Les variables topologiques et les variables de plongement peuvent être utilisées simultanément. Pour cela une instanciation en deux étapes est introduite dans [Bel12]. La première consiste à instancier les variables topologiques, et la deuxième à instancier les variables de plongement.

Par exemple, la Figure 6a illustre la règle de triangulation avec les deux types de variables. La première étape d'instanciation topologique, se déroule comme nous l'avons illustré en Figure 4, en ajoutant les expressions de plongement dans les nœuds (cf. Figure 6b). Ainsi par exemple, l'expression du plongement point $x = \text{bary}(pt_{\langle 0,1 \rangle}(v))$, portée par le nœud v'' de la règle de la Figure 6a, s'instancie en six expressions sur les six nœuds instances de v sur la Figure 6b. De même, l'expression du plongement couleur $u = \text{mix}(v.col, v.\alpha_2.col)$, portée par les trois nœuds v, v', v'' de la règle de la Figure 6a, s'instancie en six expressions pour les six nœuds instances de v et leurs copies instances de v' et v'' .

L'instanciation des variables de plongement décrite en section 4, appliquée à la règle de la Figure 6b, produit exactement la règle instanciée en Figure 5b. Ainsi son application sur le triangle bleu de l'objet de la Figure 1a produit l'objet de la Figure 6c.

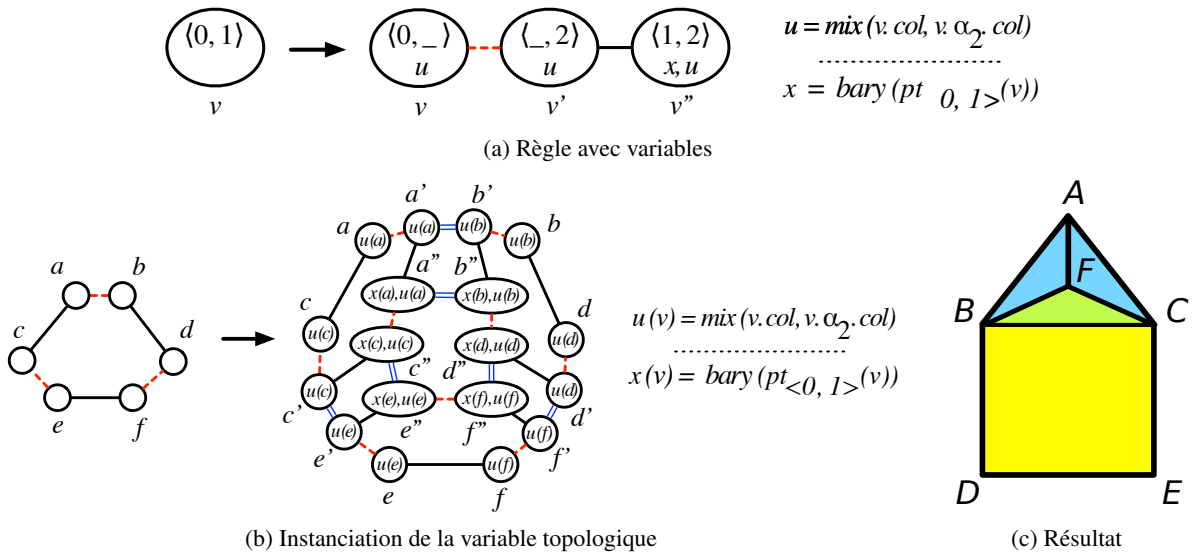


FIGURE 6 – Triangulation avec variables topologiques et variables de plongement

Les conditions syntaxiques de préservation de la cohérence topologique présentées en section 3, peuvent être utilisées telles quelles en présence simultanée de variables topologiques et de plongement. En effet, l'instanciation des expressions de plongement ne modifie pas la structure topologique.

Au contraire, les conditions syntaxiques de préservation des plongements présentées en section 4, ne peuvent être utilisées. En particulier une même expression de plongement dans la règle avec des variables topologiques, correspond à plusieurs expressions de plongement dans la règle instanciée. Par exemple, Figure 6b, les nœuds a et c portent deux expressions de couleurs différentes $u(a)$ et $u(c)$, alors qu'ils appartiennent à la même orbite face support du plongement couleur.

L'objectif de cet article est d'introduire des conditions de préservation des plongements sur les règles qui combinent variables topologiques et les variables de plongement. Ces conditions assurent la vérification des conditions syntaxiques introduites section 4 sur les règles avec variables de plongement, et donc la préservation de la cohérence des plongements lors de l'application des règles. Une équivalence à orbite près a été introduite dans [Bel12]. En particulier, dans notre exemple en Figure 6b, $u(a) \equiv_{\langle 0 \rangle} u(c)$, et

donc a et c portent la même couleur dans l'objet de la Figure 6c. Dans le présent article nous poursuivons cette approche. Nous définissons la notion de stabilité d'une expression le long d'une orbite sur un nœud, qui définit les conditions syntaxiques recherchées.

Reprenons l'exemple de la Figure 6, l'expression v est stable le long de $\langle 0, 1 \rangle$ sur le nœud v , car à gauche ce dernier est étiqueté par $\langle 0, 1 \rangle$ (tous les nœuds filtrés par v appartiendront ainsi à la même orbite $\langle 0, 1 \rangle$). Donc, $v.\alpha_2$ est stable le long de $\langle 0 \rangle$ sur le nœud v , car le 0202 vérifie la condition de cycle (deux instances de v , voisins de dimension 0, ont leurs voisins de dimension 2 respectifs liés par un arc de dimension 0). Donc, $v.\alpha_2.col$ est stable le long de $\langle 0 \rangle$ sur le nœud v , car $\langle 0 \rangle$ est une sous-orbite du support $\langle 0, 1 \rangle$ du plongement col (deux instances de v , voisins de dimension 0, ont leurs voisins de dimension 2 respectifs de la même couleur).

Si on poursuit le raisonnement, on peut déduire que $mix(v.col, v.\alpha_2.col)$ est stable le long de $\langle 0 \rangle$ sur le nœud v . Ce qui est l'une des conditions syntaxiques de préservation de la cohérence du plongement pour la règle de la Figure 6a. Elle garantit que a et c portent deux expressions $\langle 0 \rangle$ -équivalente dans la règle de la Figure 6b et que ces deux nœuds portent la même couleur dans l'objet transformé de la Figure 6c.

6 Conclusion

Dans cet article, nous avons présenté l'extension des conditions de préservation de la cohérence des plongements en cas de règles qui combinent variables topologiques et variables de plongement. Cette extension permettra la définition plus sûre et plus rapide d'opérations géométriques qui modifient simultanément la topologie et les plongements des objets (dont la géométrie).

Actuellement la bibliothèque Jerboa [BALGB14] permet déjà d'utiliser les variables topologiques et de plongement simultanément. Cependant, les expressions de plongement sont programmées directement dans le langage de programmation sous-jacent (Java ou C++), et seules la préservation de la cohérence topologique est vérifiée. Plus précisément, actuellement Jerboa ne vérifient pas les conditions syntaxiques de préservation des plongements présentés en section 5. Dans ce cas la règle instanciée porte différentes valeurs de plongement pour une même orbite de plongement. Lors de l'application, Jerboa calcule aléatoirement l'une de ces valeurs pour mettre à jour le plongement partagé de l'objet transformé. L'objet construit est donc cohérent, mais il ne correspond généralement pas au résultat attendu de l'opération.

La future version de Jerboa disposera d'un véritable langage d'expressions qui simplifiera l'écriture des règles et de leurs plongements. Surtout, elle vérifiera l'ensemble des conditions syntaxiques de préservation de la cohérence topologique et de plongement, ce qui facilitera la mise au point des règles.

Références

- [BALGB14] H. Belhaouari, A. Arnould, P. Le Gall, and T. Bellet. JERBOA : A graph transformation library for topology-based geometric modeling. In *7th International Conference on Graph Transformation (ICGT 2014)*, volume 8571 of *LNCS*, York, UK, July 2014. Springer.
- [Bel12] Thomas Bellet. *Transformations de graphes pour la modélisation géométrique à base topologique*. PhD thesis, 2012. 2012POIT2261.
- [EEPT06] H. Ehrig, K. Ehrig, U. Prange, and G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science. An EATCS Series)*. Springer, Secaucus, NJ, USA, 2006.
- [Hof05] Berthold Hoffmann. Graph transformation with variables. In *Formal Methods in Software and System Modeling (Festschrift for Hartmut Ehrig on the Occasion of his 60th Birthday)*, volume 3393 of *Lecture Notes in Computer Science*, pages 101–115. Springer-Verlag Heidelberg, 2005.
- [Lie94] P. Lienhardt. N-dimensional generalised combinatorial maps and cellular quasimanifolds. *International Journal of Computational Geometry and Applications*, 1994.
- [Pou09] Mathieu Poudret. *Graphs transformations for the topological operations of the geometric modeling - Application to the study of the Golgi apparatus dynamics*. PhD thesis, University of Evry-Val d'Essonne, October 2009.

Algèbre linéaire pour invariants polynomiaux

Steven de Oliveira¹, Saddek Bensalem², Virgile Prevosto¹

1 : CEA, List ; 2 : Université Grenoble Alpes

Résumé

Nous présentons dans ce papier une nouvelle technique de génération d'invariants dans le contexte d'un certain type de boucles polynomiales. Notre méthode a l'avantage d'être plus rapide que les méthodes existantes pour des boucles équivalentes et plus simple à implanter car elle repose sur des algorithmes d'algèbre linéaire de complexité polynomiale. Un outil implémentant cette méthode est en cours de développement dans Frama-C [2], une plateforme open-source, extensible et collaborative dédiée à l'analyse de programmes C.

1 Introduction

La phase de développement d'un programme requiert plus que la simple connaissance de la syntaxe d'un langage mais aussi de sa sémantique, parfois complexe. Aussi il est largement admis que la vérification en est une étape importante. Les techniques de vérification déductive permettent d'établir formellement les propriétés fonctionnelles d'un programme. Cependant, elles nécessitent un effort important de la part de l'utilisateur, qui doit entre autres choses fournir des invariants (propriétés qui restent vraies à chaque tour) pour chacune des boucles présentes dans le programme. De ce fait, de nombreux travaux visent à automatiser la découverte de certains de ces invariants. C'est dans cette optique que nous avons développé une nouvelle technique de génération d'invariants de boucle dans le contexte d'un certain type d'affectations définies dans [5]. Notre méthode a l'avantage d'être plus rapide que les méthodes existantes pour des boucles équivalentes et est plus simple à implanter car elle repose sur des algorithmes d'algèbre linéaire de complexité polynomiale. Un outil implantant cette procédure est en cours de développement dans Frama-C [2], une plateforme open-source, extensible et collaborative dédiée à l'analyse de programmes C.

Structure de l'article Cet article sera centré sur l'analyse de l'exemple en figure 1, sur lequel nous allons appliquer en détail chaque étape de notre méthode jusqu'à la génération de l'invariant propre à cette boucle. L'affectation de x et de y est simultanée. Notre méthode est composée de deux opérations distinctes :

```

(x, y) := (0, 0);
while (y < N) do
    (x, y) := (x + y*y, y + 1);
done
    
```

Figure 1: Programme calculant la somme de carrés successifs. Cette boucle admet l'invariant $-6.x + y - 3.y^2 + 2y^3 = 0$ pour tout N .

1. la transformation des affectations polynomiales de la boucle en affectations affines (Section 2.1);
2. la génération d'invariants sur la nouvelle boucle affine (Section 2.2).

Nous présenterons ensuite notre procédure pour prendre en compte les conditions présentes dans les boucles dans la section 2.3.

Remarque Toutes les propriétés présentées, notamment la complétude et la complexité de notre méthode ont été prouvées. Par souci de place, elles ne sont pas présentées dans cet article et feront partie d'une version plus longue.

2 Génération automatique d'invariants polynomiaux

2.1 Linéarisation de boucles polynomiales

2.1.1 Réduction

Nous allons chercher à trouver une affectation affine dont le comportement est équivalent à celui de l'affectation P de la boucle, avec $P(x, y) = (x + y^2, y + 1)$. Nous allons avoir besoin de nous débarrasser du monôme y^2 qui est la seule opération non-linéaire. Plutôt que de considérer y^2 comme un calcul à effectuer, on peut chercher à exprimer l'évolution de sa valeur après l'affectation comme proposé par [4]. Cela nous conduit à remarquer que si $y' = y + 1$, alors $(y^2)' = y^2 + 2y + 1$. Notons que l'évolution de y^2 s'exprime comme une combinaison affine de y^2 et de y . Posons donc une variable y_2 que l'on initialise à y^2 avant l'exécution de la boucle : elle représentera le monôme y^2 le long de l'exécution de la boucle. Dès lors $f(x, y, y_2) = (x + y_2, y + 1, y_2 + 2y + 1)$. f est une application affine et a exactement le même comportement que P sur x et y . Nous avons ainsi réduit l'analyse de l'affectation polynomiale P à celle de f , plus simple.

Remarque Notre exemple a la particularité d'être linéarisable mais ce n'est pas le cas pour toutes les affectations polynomiales. Par exemple si l'on essaie pour l'application $P(x) = x^2$ d'exprimer le monôme x^2 en fonction de x , il sera nécessaire d'exprimer x^4 , x^8 , etc. Nous devons ainsi nous restreindre aux affectations

qui ne transforment pas polynomialement une variable elle-même. Cette classe d'affectation a déjà été définie dans [5] :

Définition 1 Soit $g \in \mathbb{Q}[X]^m$ une affectation polynomiale. g est soluble s'il existe une partition de X en sous-vecteurs de variables $x = w_1 \uplus \dots \uplus w_k$ telle que $\forall j. 1 \leq j \leq k$ on a

$$g_{w_j}(x) = M_j w_j^T + P_j(w_1, \dots, w_{j-1})$$

En d'autres termes, une affectation soluble transforme une variable :

- soit de manière affine;
- soit par une transformation affine de variables plus un polynôme d'autres variables qui sont elles-mêmes soit transformées linéairement, soit par une affectation soluble *indépendante*.

Sur l'exemple précédent, on peut prendre $w_1 = (y)$, $P_1 = 1$, $w_2 = (x)$ et $P_2(y) = y^2$. Dans ce cas, on remarque que $(x', y', P_2(y')) = f(x, y, P_2(y))$. Plus généralement, la propriété suivante est vraie :

Propriété 1 Pour toute affectation soluble g , il existe une application affine f et un polynôme P tels que $X' = g(X) \Rightarrow (X', P(X')) = f(X, P(X))$.

2.1.2 Élévation

L'invariant que nous cherchons est de degré 3, c'est à dire qu'il est composé de monômes de degré maximal 3. Notre méthode de génération d'invariant sur les affectations linéaires ne prend en compte que les variables utilisées, donc si nous souhaitons trouver des invariants de degré supérieur à 2, l'étape de réduction n'est pas suffisante car on n'y exprime que l'évolution d'un monôme de degré 2. Nous devons continuer d'appliquer la méthode décrite précédemment pour exprimer de nouveaux monômes de degré *plus élevé*. Dans notre cas, nous aurons besoin de considérer dans notre analyse des monômes de degré 3 et moins, c'est à dire xy et y^3 . Considérer x^2 n'est pas nécessaire car l'évolution de ce monôme nécessite une expression de y^4 , un monôme de degré supérieur à 3. C'est dans le cas de l'élévation d'une application affine que l'on aura le plus de variables, dont une borne supérieure est donnée dans [4].

Propriété 2 Toute affectation soluble g utilisant n variables est linéarisable par une affectation utilisant au plus $\binom{n+d}{d}$ nouvelles variables, où d est le degré du polynôme utilisé dans la linéarisation. Pour un nombre de variables donné, on a donc besoin d'une quantité polynomiale de nouvelles variables.

2.2 Génération d'invariants

Nous présentons dans cette partie une technique de génération d'invariants pour les boucles affines, qui étend naturellement l'étape de linéarisation. Par la suite, nous considérerons une application affine comme une application linéaire à laquelle on ajoute une variable $\mathbb{1}$ toujours égale à 1. Les constantes k rendant une application non-linéaire seront alors remplacées par $k.\mathbb{1}$.

Informellement, un invariant est une formule logique telle que

1. elle est valide à l'état initial de la boucle ;
2. après chaque itération de la boucle, elle reste valide.

Nous nous intéressons d'abord à la recherche de *semi-invariants*, des formules qui satisfont uniquement le second critère sous la forme d'une combinaison linéaire sur X . Dans ce cadre, une telle formule est une forme linéaire φ telle que :

$$\varphi(X) = 0 \Rightarrow \varphi(f(X)) = 0 \quad (1)$$

Par l'algèbre linéaire, l'égalité suivante est toujours vraie

$$\varphi(f(X)) = (f^*(\varphi))(X) \quad (2)$$

où f^* est l'application duale de f . Si φ est un vecteur propre de f^* (i.e. il existe λ tel que $f^*(\varphi) = \lambda\varphi$), l'équation (1) est automatiquement vraie.

Dans notre cas, f^* peut être calculée en transposant la matrice représentant f . Par souci de simplicité, nous allons ignorer le monôme xy car il n'intervient pas dans l'expression de l'invariant que nous cherchons.

Nous avons alors $f^*(x, y, y_2, y_3, \mathbb{1}) = (x, y + 2.y_2 + 3.y_3, x + y_2 + 3.y_3, y_3, y + y_2 + y_3 + \mathbb{1}, y + y_2 + y_3 + \mathbb{1})$. f^* admet la valeur propre 1. L'espace propre de f^* associé à 1 est généré par les vecteurs $e_1 = (-6, 1, -3, 2, 0)^T$ et $e_2 = (0, 0, 0, 0, 1)^T$. On trouve ainsi que $F_{k_1, k_2} = (k_1 \cdot (-6.x + y - 3.y_2 + 2.y_3) + k_2.\mathbb{1} = 0)$ est un semi-invariant, avec $k_1, k_2 \in \mathbb{Q}$. En écrivant $k = -\frac{k_2}{k_1}$ et en remplaçant $\mathbb{1}$ par 1, on peut ré-écrire la formule précédente par

$$F_k = (-6.x + y - 3.y_2 + 2.y_3 = k)$$

Le paramètre peut être inféré par l'état initial de la boucle. Si par exemple la boucle commence avec $(x = 0, y = 0)$, alors $-6.x + y - 3.y_2 + 2.y_3 = 0$ et donc F_0 est un invariant de la boucle. Plus généralement, l'union des espaces propres de f^* est toujours un ensemble de semi-invariants pour l'affectation f .

Complexité La transposition de matrices et la recherche des vecteurs propres sont des problèmes polynomiaux en la taille de la matrice, donc polynomiaux en le nombre de variables. Dans le cas d'une boucle linéarisée par la méthode de la section 2.1 qui génère un nombre polynomial de nouvelles variables, la composition des deux est bien polynomiale.

```

x = a, y = b, z = 0;
while (y > 1) {
  if (y%2 == 1)
  then
    (x, y, z) := (2x, (y-1)/2, x + z)
  else
    (x, y, z) := (2x, y/2, z) }
    
```

Figure 2: Calcul du produit de a et b (exemple de [5])

2.3 Boucles avec conditions

Une boucle peut contenir plusieurs conditions qui séparent le graphe de flot de contrôle en plusieurs chemins différents. Supposons qu'à chaque itération, la boucle peut choisir aléatoirement chaque condition. Dès lors, à chaque tour de boucle une des affectations va être effectuée, indépendamment des tours précédents. Prenons en exemple le programme de la figure 2 calculant le produit de x et y . Posons $X = (x, y, z, \mathbb{1})$. Même si on considère comme non déterministes les conditions du programme original, nous pouvons trouver des invariants utiles. Nous avons ici deux affectations : $f_1(X) = (2x, (y-1)/2, x+z, \mathbb{1})$ et $f_2(X) = (2x, y/2, z, \mathbb{1})$. L'ensemble des semi-invariants vrais sur les différents corps de boucle possibles est alors l'intersection des semi-invariants vrais sur chaque corps de boucle. Nous venons de voir que l'ensemble des invariants générés peut être représenté comme une union d'espaces vectoriels, leur intersection est donc également une union d'espaces vectoriels. L'élévation au degré 2 de f_1 et de f_2 retourne 10 vecteurs propres pour chacune des applications. Pour simplifier, nous allons nous concentrer sur les vecteurs propres associés à la valeur propre 1. Posons φ_e l'application linéaire induite par e .

f_1^* a 4 vecteurs propres $\{e_i\}_{i \in [1,4]}$ associés à 1 tels que

- $\varphi_{e_1}(X) = -x + xy;$
- $\varphi_{e_2}(X) = x + z;$
- $\varphi_{e_3}(X) = -2.xz + x^2 + z^2;$
- $\varphi_{e_4}(X) = \mathbb{1}.$

f_2^* a également 4 vecteurs propres $\{e'_i\}_{i \in [1,4]}$ associés à 1 tels que

- $\varphi_{e'_1}(X) = xy;$
- $\varphi_{e'_2}(X) = z;$
- $\varphi_{e'_3}(X) = z^2;$
- $\varphi_{e'_4}(X) = \mathbb{1}.$

Premièrement, on remarque que $e_4 = e'_4$. Ensuite, on peut voir que $e_1 + e_2 = e'_1 + e'_2$. Finalement, on remarque que $e_1 + e_2 + k.e_4 \in (\text{Vect}(\{e_i\}_{i \in [1,4]}) \cap \text{Vect}(\{e'_i\}_{i \in [1,4]}))$. C'est pourquoi $(\varphi_{e_1+e_2+k.e_4}(X) = 0)$ est un semi-invariant pour f_1 et f_2 , donc il l'est pour toute la boucle. En remplaçant $\varphi_{k.e_4}(X)$ par $k.\mathbb{1} = -k'$ et $\varphi_{e_1+e_2}(X)$ par $xy + z$, on trouve $xy + z = k'$. Initialement,

$xy + z = ab$, donc on sait que $xy + z = ab$ est un invariant de la boucle. Cet invariant est nécessaire pour prouver que ce programme multiplie bien x et y .

Remarque L'intersection de deux espaces vectoriel est équivalente au calcul du noyau de la matrice dont les colonnes sont les bases des deux espaces vectoriels. C'est donc également un calcul polynomial.

3 Conclusion

Ce nouvel algorithme de génération d'invariants pour boucle affine composé avec nos procédures de linéarisation et de gestion des conditions nous permet de générer des invariants non-triviaux sur des boucles complexes. Il est en réalité complet pour les affectations solubles. Notons également que notre algorithme bénéficie d'une complexité polynomiale qui contraste avec les techniques existantes qui sont soit incomplètes avec [1], soit exponentielles avec les bases de Gröbner [3, 5].

Notre attention se porte désormais sur l'étude des boucles affines admettant des vecteurs propres irrationnels qui ne permettent pas une représentation correcte des variables selon un modèle proche de celui d'un programme C. Nous travaillons sur un formalisme complet pour étendre la précédente méthode et développons plusieurs pistes pour faire face à ce problème.

Remerciements Nous remercions les relecteurs anonymes pour leur suggestions et remarques sur une première version de cet article.

Références

- [1] D. Cachera, T. Jensen, A. Jobin, and F. Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. *Science of Computer Programming*, 93:89–109, 2014.
- [2] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Framac: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, 2015.
- [3] L. Kovács. Aligator: A Mathematica package for invariant generation (system description). In *Automated Reasoning*, pages 275–282. Springer, 2008.
- [4] M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *ACM SIGPLAN Notices*, volume 39, pages 330–341. ACM, 2004.
- [5] E. Rodríguez-Carbonell and D. Kapur. Generating all polynomial invariants in simple loops. *Journal of Symbolic Computation*, 42(4):443–476, 2007.

Vérification formelle de programmes de génération de données structurées

Résumé des travaux de thèse de doctorat

Richard Genestier

FEMTO-ST UMR CNRS 6174 (UBFC/UFC/ENSM/UTBM)

Université de Franche-Comté, France

richard.genestier@femto-st.fr

Résumé

Ce document résume les travaux de recherche de ma thèse d'informatique intitulée "Vérification formelle de programmes de génération de données structurées", dirigée depuis septembre 2012 par O. Kouchnarenko, professeur et co-encadrée par A. Giorgetti, maître de conférences.

Les méthodes du génie logiciel, telles que la spécification formelle et la preuve déductive, rendent la conception de programmes plus rationnelle, augmentent leur qualité et accroissent la confiance du développeur dans la correction de sa production. En raison de leur fort coût, ces méthodes n'ont longtemps été appliquées que dans des domaines à fort enjeu de sûreté ou de sécurité. De nombreux progrès récents rendent ces pratiques vertueuses accessibles à d'autres domaines du développement logiciel.

Par différentes études de cas, ce travail de thèse évalue la faisabilité de démonstrations automatiques de programmes manipulant des tableaux d'entiers. Le cadre d'investigation choisi est le domaine de la combinatoire énumérative car il fournit de nombreux algorithmes dont la difficulté et la structure sont bien adaptées aux capacités des prouveurs actuels. Dans la combinatoire, les efforts de formalisation et de démonstration se sont concentrés sur certains objets, appelés *cartes*.

La combinatoire est la partie des mathématiques discrètes qui étudie des ensembles finis ou dénombrables d'objets appelés *structures combinatoires*. Les permutations de n objets distinguables et les permutations particulières, telles que les involutions, sont des exemples de structures combinatoires. La combinatoire énumérative propose des algorithmes pour générer ces structures et pour compter le nombre de structures composées d'un nombre donné d'éléments. Le lecteur désirent davantage de précisions peut par exemple se référer à [Sta97].

Les cartes sont des objets mathématiques qui apparaissent naturellement dans l'étude de nombreux problèmes, de nature mathématique ou physique. Une *carte (combinatoire) étiquetée* est un triplet (D, R, L) où D est un ensemble fini à $2n$ éléments (pour un entier naturel $n \geq 0$), appelés *étiquettes* ou *brins*, R est une permutation sur D et L est une involution sans point fixe sur D , telles que le groupe $\langle R, L \rangle$ engendré par R et L agit transitivement sur D [LZ04]. L'action transitive des permutations R et L correspond au fait que deux brins quelconques d'une carte peuvent être envoyés l'un sur l'autre par un nombre fini d'applications de R et L . Les combinaticiens s'intéressent plus particulièrement aux cartes enracinées. Combinatoirement, une *carte enracinée* est une classe d'équivalence de cartes étiquetées isomorphes par les isomorphismes renommant leurs brins mais préservant l'un d'entre eux, appelé *racine*.

Dans cette thèse, la spécification formelle, la validation par test et la vérification déductive sont appliquées à des implémentations d'algorithmes connus de combinatoire, mais également à des implémentations de nouveaux algorithmes. L'accent est mis sur les algorithmes qui facilitent la génération ou le dénombrement des cartes.

Les travaux de vérification déductive concernant le domaine des cartes combinatoires sont peu nombreux. Dans [DM07], C. Dubois et J.-M. Mota proposent une formalisation des cartes générales utilisant le formalisme B, très proche de leur définition mathématique à base de permutations et d'involutions. J.-F. Dufourd et F. Puitg ont proposé une spécification en ML des cartes combinatoires pointées [DP00]. En utilisant des hypercartes combinatoires, J.-F. Dufourd a établi en Coq [Coq] une preuve formelle du théorème de Jordan [Duf09]. On trouve également une formalisation avancée des hypercartes combinatoires dans le vaste projet aboutissant à la preuve formelle du théorème des quatre couleurs en Coq [Gon05].

L'objectif d'effectuer des preuves formelles automatiques sur des objets complexes comme les cartes enracinées est assez ambitieux. Une approche par étapes a donc été adoptée, d'une part pour en évaluer sa faisabilité, et d'autre part pour identifier les limites des outils de vérification automatique utilisés. Les algorithmes de génération présentés sont programmés en C et spécifiés en ANSI C Specification Language (ACSL) [BCF⁺13]. ACSL est un langage dédié à l'analyse statique de programmes C qui permet de spécifier formellement des contrats qui doivent être vérifiés par le programme. L'outil principal utilisé est la plateforme d'analyse de programmes C Frama-C [KKP⁺15] développée par le CEA LIST et INRIA Saclay, qui utilise le langage de spécification ACSL. Pour la vérification déductive, Frama-C utilise le greffon WP, qui implante le calcul de plus faible précondition pour des programmes C annotés en ACSL, et les solveurs SMT Alt-Ergo [Alt], CVC3 [BT07] et CVC4 [CVC]. Ces solveurs sont utilisés par WP pour déterminer si une formule du premier ordre est satisfaisable.

Les cartes combinatoires étant définies à partir de permutations, il paraissait intéressant de considérer la génération des permutations et de diverses structures combinatoires proches, encodables en C par un tableau d'entiers dit *structuré*, car satisfaisant une contrainte structurelle exprimée en logique du premier ordre. Avec A. Giorgetti et G. Petiot, nous avons porté notre attention sur les algorithmes d'énumération combinatoire qui génèrent séquentiellement toutes les tableaux structurés de même longueur, selon un ordre total prédéfini. Un tel algorithme est nommé *générateur séquentiel* lorsqu'il est composé de deux fonctions : la première fonction construit la plus petite structure de longueur donnée selon l'ordre prédéfini, et la seconde fonction modifie une structure quelconque pour construire la structure suivante de même longueur selon cet ordre. Nous présentons une approche uniforme pour la mise en œuvre rationnelle de ces générateurs séquentiels de tableaux structurés, implantés par des fonctions C. Nous considérons trois propriétés comportementales pour ces fonctions de génération. La propriété de *correction* affirme que les deux fonctions génèrent des tableaux satisfaisant leur contrainte structurelle. La propriété de *progression* affirme que la deuxième fonction génère un tableau supérieur à son tableau d'entrée selon l'ordre prédéfini. Cette propriété implique la terminaison des appels répétés à la seconde fonction. La propriété d'*exhaustivité* affirme que le générateur n'omet aucune solution. Les propriétés de correction et de progression sont vérifiées statiquement. Dans ce but, nous spécifions formellement le comportement des fonctions de génération de tableaux. Nous les annotons ensuite avec des invariants et variants de boucle de manière à ce que leurs contrats formels soient automatiquement prouvés. La propriété d'exhaustivité se traduit par le fait qu'il n'existe pas de tableau structuré entre deux tableaux structurés successifs générés. Nous n'avons pas identifié de fragment décidable de la théorie des tableaux incluant la condition de vérification de cette propriété, qui comporte une quantification (existentielle) sur un tableau. De fait, elle n'est pas déchargée par les prouveurs automatiques actuels. En conséquence, nous validons la propriété d'exhaustivité par comptage grâce à une génération exhaustive bornée des structures. Cette approche a débouché sur

la conception de patrons généraux d'aide à l'écriture de ces programmes spécifiés, de générateurs séquentiels génériques de tableaux structurés, et d'une bibliothèque de générateurs séquentiels vérifiés¹. Ce travail a fait l'objet de deux publications, correspondant à deux stades successifs de son avancement [GGP15a, GGP15b].

Il résulte de cette approche un certain nombre de constats. Afin de faciliter la démonstration automatique de la correction de ces fonctions C , ces dernières doivent utiliser un petit fragment du langage C , et leurs spécifications doivent pouvoir être exprimées en logique du premier ordre. Plus précisément, il convient de considérer au sein de ces fonctions des tableaux d'entiers alloués préalablement (pas de pointeur ni d'allocation dynamique), et uniquement des expressions d'arithmétique linéaire sur les entiers. Ensuite, les difficultés dans l'établissement des preuves sont liées à la correction et à la précision des invariants de boucles, qui ne doivent être ni trop faibles (ils ne permettent pas de prouver les postconditions attendues), ni trop forts (les prouveurs peinent alors à prouver leur préservation). De manière générale, lorsque la preuve de la correction d'une fonction présente une difficulté, ajouter des assertions ou décomposer cette fonction en sous-fonctions munies de contrats contenant des postconditions intermédiaires menant progressivement à l'établissement des postconditions finales aide grandement les prouveurs dans leur tâche.

Les cartes étiquetées étant des couples de permutations agissant transitivement sur un ensemble fini D , nous avons complété cette bibliothèque avec de nouveaux générateurs génériques de couples de structures transitives, puis nous les avons instanciés pour obtenir divers générateurs séquentiels de familles de cartes étiquetées. Cela permet de visualiser ces objets et de valider des conjectures sur ces cartes, par test exhaustif borné.

En complément de cette étude des cartes étiquetées, les cartes enracinées (non étiquetées) ont été considérées à travers l'une de leurs premières représentations par des mots. Avec J.-L. Baril, A. Giorgetti et A. Petrossian, nous nous sommes intéressés plus particulièrement aux cartes planaires, qui sont des cartes qu'on peut représenter par un graphe dessiné sur une sphère. Une carte planaire enracinée (non étiquetée) à $2n$ brins peut être représentée par un mot à $2n$ lettres issues d'un alphabet à 4 lettres. Pour tout mot π sur cet alphabet, appelé *motif*, nous définissons sur ces mots la relation d'équivalence suivante : deux mots de même longueur sont dits π -équivalents si et seulement s'ils contiennent le motif π aux mêmes positions. Pour quasiment tous les motifs de longueur au plus deux, nous avons déterminé dans [BGGP16] la fonction génératrice du nombre de classes de π -équivalence. La bijection entre mots et cartes planaires enracinées donne ainsi des résultats d'énumération pour les classes d'équivalence de cartes planaires enracinées. En suivant la méthodologie présentée dans [GGP15a, GGP15b], nous avons mis au point un générateur séquentiel de ces mots. Ce générateur a été utilisé pour compter le nombre de classes de π -équivalence des mots de petite longueur. Cette expérimentation a grandement facilité l'établissement des résultats de [BGGP16].

Avec A. Giorgetti et C. Dubois, nous nous sommes également intéressés à d'autres modes de génération, notamment des générations arborescentes, basées sur des constructions inductives de cartes combinatoires enracinées. Dans ce cadre, le choix d'une représentation formelle des cartes est primordial. En effet, selon ce choix, l'implémentation et la vérification des constructions inductives seront plus ou moins aisées et efficaces. Nous proposons à cet effet la notion de carte locale, codée par une seule permutation, ainsi que deux opérations sur les permutations adaptées au mode de construction inductive adopté. Nous avons formalisé cette notion, ces opérations et leur correction en Coq. Cette correction est validée par test exhaustif borné et aléatoire, avant d'être prouvée interactivement en Coq [DGG16].

1. Archive `enum.*.tar.gz` téléchargeable depuis la page <http://members.femto-st.fr/richard-genestier/en>.

Les deux opérations sur les permutations consistent à insérer un élément dans une permutation et à fusionner deux permutations. Nous les avons implémentées en C, nous avons spécifié leur correction en ACSL et avons prouvé cette correction automatiquement [GG16] avec Frama-C et son greffon WP.

Remerciements. L'auteur remercie O. Kouchnarenko, A. Giorgetti et les relecteurs pour leurs suggestions.

Références

- [Alt] The Alt-Ergo SMT solver. <http://alt-ergo.lri.fr>.
- [BCF⁺13] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2013. <http://frama-c.com/acsl.html>.
- [BGGP16] J.-L. Baril, R. Genestier, A. Giorgetti, and A. Petrossian. Rooted planar maps modulo some patterns. *Discrete Mathematics*, 339 :1199–1205, 2016.
- [BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
- [Coq] The Coq Proof Assistant. <http://coq.inria.fr>.
- [CVC] CVC4. <http://cvc4.cs.nyu.edu/web/>.
- [DGG16] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In *Tests and Proofs (TAP)*, 2016. To appear.
- [DM07] C. Dubois and J.-M. Mota. Geometric modeling with B : formal specification of generalized maps. *Journal of Scientific & Practical Computing Journal*, 1(2) :9–24, 2007.
- [DP00] J.-F. Dufourd and F. Puitg. Functional specification and prototyping with oriented combinatorial maps. *Computational Geometry*, 16(2) :129 – 156, 2000.
- [Duf09] J.-F. Dufourd. An intuitionistic proof of a discrete form of the Jordan curve theorem formalized in Coq with combinatorial hypermaps. *J. Autom. Reason.*, 43(1) :19–51, June 2009.
- [GG16] R. Genestier and A. Giorgetti. Spécification et vérification formelle d'opérations sur les permutations. In *Approches Formelles dans l'Assistance au Développement Logiciel (AFADL)*, 2016.
- [GGP15a] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In *Journées Francophones des Langages Applicatifs (JFLA)*, 2015. <https://hal.inria.fr/hal-01099135>.
- [GGP15b] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *Tests and Proofs (TAP)*, volume 9154 of *LNCS*, pages 109–128. Springer, Heidelberg, 2015.
- [Gon05] G. Gonthier. A computer checked proof of the Four Colour Theorem, 2005. <http://research.microsoft.com/gonthier/4colproof.pdf>.
- [KKP⁺15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. *Formal Aspects of Computing*, 27(3) :573–609, 2015.
- [LZ04] S. K. Lando and A. K. Zvonkin. *Graphs on Surfaces and Their Applications*. Springer, 2004.

- [Sta97] R. P. Stanley. *Enumerative Combinatorics*, volume 1. Cambridge University Press, Cambridge, England, 1997.

Spécification et vérification formelle d'opérations sur les permutations

Richard Genestier et Alain Giorgetti

FEMTO-ST, UMR CNRS 6174 (UBFC/UFC/ENSMM/UTBM)

Université de Franche-Comté, France

`firstname.lastname@femto-st.fr`

Résumé

Dans le cadre d'un projet plus vaste de formalisation de la combinatoire, nous présentons dans cet article deux opérations sur les permutations, leur implémentation en C sous forme de fonctions modifiant des tableaux d'entiers, et leur spécification formelle en ACSL. Après avoir spécifié des invariants de boucle adaptés, nous prouvons automatiquement que ces fonctions préservent les permutations, avec l'outil Frama-C et son greffon WP.

1 Introduction

Cette étude complète un travail de preuve interactive de la correction d'opérations de construction de cartes combinatoires [DGG16]. Une *carte (combinatoire)* est un triplet (D, R, L) où D est un ensemble fini à $2n$ éléments ($n \geq 0$), R est une permutation sur D et L est une involution sans point fixe sur D , telles que le groupe $\langle R, L \rangle$ engendré par R et L agit transitivement sur D . Dans [DGG16] nous formalisons en Coq [Coq] les notions de permutation et de carte combinatoire, deux opérations sur les permutations et deux opérations de construction de cartes combinatoires, qui utilisent les opérations sur les permutations. Techniquement, nous définissons d'abord ces quatre opérations sur des fonctions quelconques sur D . Puis nous démontrons formellement que ces quatre opérations préservent les permutations et que les deux opérations sur les cartes combinatoires préservent la condition de transitivité. Nous effectuons les démonstrations de manière interactive, à l'aide des tactiques de l'assistant de preuve Coq.

Dans cet article, nous étudions la question de l'automatisation des deux démonstrations portant sur les permutations. Dans ce but, nous représentons une permutation des n premiers entiers naturels par un tableau C de longueur n et nous implémentons les deux opérations sur les permutations par deux fonctions C sur ces tableaux d'entiers. Selon la démarche détaillée dans [GGP15a, GGP15b], nous spécifions en ANSI C Specification Language (ACSL) [BCF⁺13] que les deux fonctions C peuvent être restreintes aux tableaux qui représentent des permutations : si leurs tableaux en entrée représentent des permutations, alors il en est de même de leur tableau en sortie. On dit aussi que ces fonctions *préservent les permutations*. ACSL est un langage dédié à l'analyse statique de programmes C qui permet de spécifier formellement des contrats qui doivent être vérifiés par le programme. Nous utilisons la plateforme d'analyse de programmes C Frama-C [KKP⁺15] développée par le CEA LIST et INRIA Saclay, qui utilise le langage de spécification ACSL. Pour la vérification déductive, nous utilisons le greffon WP [BBCD15], qui implémente le calcul de plus faible précondition pour des

programmes C annotés en ACSL, et les solveurs SMT Alt-Ergo [Alt], CVC3 [BT07] et CVC4 [CVC]. Ces solveurs sont utilisés par WP pour déterminer si une formule du premier ordre est satisfaisable.

La partie 2 rappelle quelques notions sur les permutations utiles pour la suite. La partie 3 définit les deux opérations sur les permutations introduites dans [DGG16], puis décrit leur implémentation en C. Leur spécification en ACSL et leur vérification déductive sont détaillées dans la partie 4. La partie 5 conclut cette étude.

2 Permutations

Les permutations sur un ensemble fini forment une famille combinatoire élémentaire mais centrale, largement étudiée dans le domaine de la combinatoire énumérative. Nous proposons une formalisation machine d'opérations basiques sur les permutations, accompagnée de la vérification déductive de leur correction.

Une *permutation* est une bijection sur un ensemble fini, identifié ici à l'ensemble $\{0, \dots, n-1\}$ des n premiers entiers naturels. La *taille* d'une permutation est la cardinalité n de cet ensemble. L'image de $i \in \{0, \dots, n-1\}$ par la permutation p sur $\{0, \dots, n-1\}$ est notée $p(i)$. Appliquer la permutation p_1 puis la permutation p_2 revient à appliquer la permutation $p_1 p_2$ appelée *composition* ou *produit* de p_1 et p_2 . La composition est ici appliquée de gauche à droite, c'est-à-dire que $(p_1 p_2)(i) = p_2(p_1(i))$. L'ensemble des permutations sur l'ensemble $\{0, \dots, n-1\}$ muni de l'opération interne de composition est appelé *groupe symétrique* et noté S_n . On peut représenter toute permutation p de S_n sur deux lignes, par la matrice

$$\begin{pmatrix} 0 & 1 & \dots & i & \dots & n-1 \\ p(0) & p(1) & \dots & p(i) & \dots & p(n-1) \end{pmatrix}$$

ou sur une ligne par le mot $p(0) p(1) \dots p(n-1)$ sur l'alphabet $\{0, \dots, n-1\}$. La permutation p est codée en C par le tableau p de longueur n tel que $p[i] = p(i)$ pour $0 \leq i \leq n-1$, qu'on peut représenter par

$p[0]$	$p[1]$...	$p[i]$...	$p[n-1]$
--------	--------	-----	--------	-----	----------

. Ce tableau est appelé *représentation fonctionnelle* de p .

Un *cycle* d'ordre $k \leq n$ est une permutation $p \in S_n$ telle qu'il existe des éléments i_0, i_1, \dots, i_{k-1} distincts de $\{0, \dots, n-1\}$ tels que $p(i_0) = i_1, p(i_1) = i_2, \dots, p(i_{k-2}) = i_{k-1}, p(i_{k-1}) = i_0$ et $p(j) = j$ pour tous les autres éléments de $\{0, \dots, n-1\}$. Un cycle est représenté par sa *notation cyclique* $(i_0 i_1 \dots i_{k-1})$. On dit que deux cycles sont *disjoints* s'ils ne modifient pas les mêmes éléments. Toute permutation p de S_n peut être obtenue par produit de cycles disjoints de S_n , appelé *décomposition (cyclique)* de p . Parmi ces décompositions, la *décomposition canonique* de p est le produit de cycles dans lequel la notation de chaque cycle commence par son plus petit élément et les cycles sont écrits dans l'ordre croissant de leur plus petit élément.

Un point fixe x d'une permutation $p \in S_n$ est un élément x de $\{0, \dots, n-1\}$ tel que $p(x) = x$. Si x est un point fixe de la permutation p , toute décomposition cyclique de p contient le cycle (x) , qui ne comporte pas d'autre élément que x .

Exemple 1 La permutation $p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 0 & 4 & 2 \end{pmatrix} = 1\ 3\ 5\ 0\ 4\ 2$ est codée par le tableau C

1	3	5	0	4	2
---	---	---	---	---	---

. Sa décomposition canonique est $(0\ 1\ 3)\ (2\ 5)\ (4)$. L'entier 4 est son unique point fixe.

3 Opérations sur les permutations

Dans [DGG16] nous considérons des opérations de construction de cartes combinatoires encodées par des permutations. Pour définir ces opérations, il suffit de savoir insérer un élément dans un cycle d'une permutation et de fusionner deux permutations pour en obtenir une seule. Nous présentons dans les parties 3.1 et 3.2 les opérations d'insertion et de somme directe sur les permutations, ainsi que leurs implémentations par des fonctions C.

En toute rigueur, puisque ces fonctions C agissent sur des tableaux d'entiers C quelconques, cette implémentation généralise à toute fonction définie sur un intervalle d'entiers incluant 0 les deux opérations initialement définies uniquement sur les fonctions bijectives que sont les permutations. Nous ne revenons pas sur ce point dans la suite, et nous ne décrivons cette implémentation que dans le cas où chaque tableau C est la représentation fonctionnelle d'une permutation.

L'opération de composition de deux permutations a également été étudiée mais n'est pas présentée ici, car elle est plus classique et n'est pas utile pour définir les opérations sur les cartes.

3.1 Insertion dans une permutation

Soit p une permutation sur $\{0, \dots, n-1\}$, vue comme un produit de cycles disjoints, et i un entier naturel entre 0 et n inclus. On appelle *insertion* de i dans p , et on désigne par *insert*, l'opération qui ajoute à p le cycle (n) si $i = n$ et qui insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i \leq n-1$. La permutation qui résulte de cette opération est désignée par $insert(p, i)$.

Exemple 2 À partir de la permutation $p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2 \in S_5$, on peut définir 6 permutations de S_6 par application de l'opération *insert* avec $i \in \{0, \dots, 5\}$. Par exemple, $insert(p, 0)$ est la permutation $(0\ 1\ 3\ 5)\ (2\ 4) = 1\ 3\ 4\ 5\ 2\ 0$, $insert(p, 4)$ est la permutation $(0\ 1\ 3)\ (2\ 5\ 4) = 1\ 3\ 5\ 0\ 2\ 4$ et $insert(p, 5)$ est la permutation $(0\ 1\ 3)\ (2\ 4)\ (5) = 1\ 3\ 4\ 0\ 2\ 5$ qui ajoute le point fixe 5 à p .

Nous présentons dans le listing 1 deux implémentations de cette opération par une fonction C : l'une qui étend la permutation p en place, et l'autre qui duplique cette permutation. Ces deux fonctions utilisent la représentation fonctionnelle de la permutation p .

```

1 void insert_inplace(int p[], int i, int n) {
2     if (0 ≤ i ∧ i ≤ n) {
3         p[n] = i;
4         for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
5     }
6 }
7
8 void insert(int p[], int i, int q[], int n) {
9     if (0 ≤ i ∧ i ≤ n) {
10        q[n] = i;
11        for (int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
12    }
13 }
    
```

Listing 1 – Fonctions d'insertion en C.

La première fonction `insert_inplace` prend en paramètres une permutation p , sa taille n et un entier i . Si ces paramètres permettent d'insérer la valeur n dans la permutation p , c'est-à-dire si $0 \leq i \leq n$, le comportement de la fonction est différent selon que i vaut n ou pas. Si $i = n$, la valeur n insérée crée un point fixe dans p (ligne 3). Dans ce cas la boucle de la ligne 4 n'a aucun effet sur p . Si $i < n$, la valeur n est insérée avant la valeur i (ligne 3) et après la valeur $p^{-1}(i)$ dans un cycle de p , par la boucle de la ligne 4.

La seconde fonction `insert` présentée lignes 8-13 effectue la même opération mais en construisant une nouvelle permutation q de taille $n + 1$, ce qui nécessite une recopie des valeurs de la permutation p dans q (boucle ligne 11).

3.2 Somme directe de deux permutations

Nous présentons à présent une implémentation en C de l'opération de *somme directe* de deux permutations, bien connue en combinatoire [Kit11]. Cette opération permet de fusionner deux permutations par décalage des valeurs de l'une d'entre elles. Soit p une permutation sur $\{0, \dots, n - 1\}$ et k un entier naturel. On appelle *décalage* de p selon k la permutation p' sur $\{k, \dots, n + k - 1\}$ telle que $p'(i+k) = p(i)+k$. Informellement, l'opération \oplus de somme directe fusionne deux permutations $p_1 \in \{0, \dots, n_1 - 1\}$ et $p_2 \in \{0, \dots, n_2 - 1\}$ afin d'obtenir une permutation $p_1 \oplus p_2 \in \{0, \dots, n_1 + n_2 - 1\}$ dont la restriction sur $\{0, \dots, n_1 - 1\}$ est égale à p_1 et la restriction sur $\{n_1, \dots, n_1 + n_2 - 1\}$ est égale au décalage de p_2 selon n_1 . Formellement, la somme directe $p_1 \oplus p_2$ de deux permutations $p_1 \in \{0, \dots, n_1 - 1\}$ et $p_2 \in \{0, \dots, n_2 - 1\}$ est définie par $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$ et $(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$.

Exemple 3 *Considérons les deux permutations $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3$ et $p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4) \in S_5$, ainsi que la permutation vide notée \emptyset . Alors $p_1 \oplus \emptyset = \emptyset \oplus p_1 = p_1$, le décalage de p_2 selon 3 est la permutation $4\ 6\ 7\ 3\ 5 = (3\ 4\ 6)\ (5\ 7)$ sur $\{3, \dots, 7\}$, et $p_1 \oplus p_2 = 2\ 1\ 0\ 4\ 6\ 7\ 3\ 5 = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$.*

Une implémentation de la somme directe sous forme d'une fonction C `sum` est présentée dans le listing 2. La fonction `sum` prend en paramètres une permutation p_1 , sa taille n_1 , une permutation p_2 , sa taille n_2 , et construit une nouvelle permutation $p = p_1 \oplus p_2$, résultat de la somme directe de p_1 et p_2 . Grâce à une seule boucle, cette fonction recopie les valeurs de p_1 dans p (partie `p1[i]` des affectations de la boucle), effectue le décalage de la permutation p_2 selon n_1 , et le recopie dans p à la suite des valeurs de p_1 (partie `p2[i-n1]+n1` des affectations de la boucle).

```

1 void sum(int p1[], int p2[], int p[], int n1, int n2) {
2   for (int i = 0; i < n1+n2; i++) p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
3 }
```

Listing 2 – Somme directe de deux permutations en C.

4 Vérification déductive

Nous pouvons doter les fonctions `insert_inplace`, `insert` et `sum` de contrats et d'annotations, afin de prouver automatiquement qu'elles préservent les permutations.

Il existe différentes manières de caractériser une permutation. Par exemple, bien qu'une permutation soit définie comme une bijection sur un ensemble fini, il est suffisant de la définir comme une endofonction injective ou surjective, puisque sur un ensemble fini l'injectivité et la surjectivité s'impliquent mutuellement. Afin d'effectuer la vérification déductive des fonctions présentées dans la partie 3, nous spécifions une permutation à l'aide du prédicat ACSL `is_perm` présenté dans le listing 3. Ce prédicat caractérise une permutation appartenant à S_n comme une endofonction injective sur $\{0, \dots, n - 1\}$. La propriété caractéristique d'une fonction est exprimée par le prédicat `is_fct` et la propriété d'injectivité par le prédicat `is_linear`. Le prédicat `is_insert` (resp. `is_sum`) paraphrase les instructions de boucle de la fonction `insert` (resp. `sum`).


```

1 /*@ predicate is_fct(int *a, Z b, Z c) =  $\forall Z i; 0 \leq i < b \Rightarrow 0 \leq a[i] < c;$ 
2 @ predicate is_linear(int *a, Z n) =
3 @  $\forall Z j; 0 \leq j < n \Rightarrow \forall Z k; 0 \leq k < n \Rightarrow (j \neq k \Rightarrow a[j] \neq a[k]);$ 
4 @ predicate is_perm(int *a, Z n) = is_fct(a,n,n)  $\wedge$  is_linear(a,n);
5 @ predicate is_insert(int *p, int *p1, Z b, Z c, Z d) =
6 @  $\forall Z i; 0 \leq i < b \Rightarrow p[i] == ((p1[i] == c) ? d : p1[i]);$ 
7 @ predicate is_sum(int *p, int *p1, int *p2, Z b, Z c) =
8 @  $\forall Z i; 0 \leq i < c \Rightarrow p[i] == ((i < b) ? p1[i] : p2[i-b]+b);$  */

```

Listing 3 – Prédicats ACSL pour les fonctions insert et sum.

```

1 /*@ requires 0  $\leq$  i  $\leq$  n;
2 @ requires \valid(p+(0..n-1));
3 @ requires \valid(q+(0..n));
4 @ requires
5 @ \separated(q+(0..n),p+(0..n-1));
6 @ requires is_perm(p,n);
7 @ assigns q[0..n];
8 @ ensures is_perm(q,n+1); */
9
10 /*@ loop invariant 0  $\leq$  j  $\leq$  n;
11 @ loop invariant is_insert(q,p,j,i,n);
12 @ loop invariant is_fct(q,j,n+1);
13 @ loop invariant is_linear(q,j);
14 @ loop assigns j, q[0..n-1];
15 @ loop variant n-j; */

```

Listing 4 – Contrat et annotations de boucle de la fonction insert.

Le listing 4 présente le contrat (lignes 1-8) et les annotations de boucle (lignes 10-15) de la fonction `insert`. Les annotations de boucle sont à insérer entre les lignes 10 et 11 de la fonction `insert` présentée dans le listing 1. Cette fonction manipulant plusieurs tableaux, il est nécessaire de garantir que ces tableaux sont alloués dans des zones distinctes de la mémoire, à l'aide du prédicat natif `separated` (lignes 4-5). L'invariant ligne 11 spécifie le comportement de la boucle grâce au prédicat `is_insert`. De plus, les invariants de boucle des lignes 12-13 spécifient qu'à chaque itération de boucle, les propriétés caractéristiques d'une permutation sont satisfaites jusqu'à l'indice de boucle.

Le contrat et les annotations de boucle de la fonction `sum` suivent le même principe. Ils sont présentés dans le listing 5. Ces annotations de boucle sont à insérer entre les lignes 1 et 2 de la fonction `sum` présentée dans le listing 2. Les contrats de ces deux fonctions sont prouvés automatiquement, grâce aux invariants de boucle. Ces invariants étant des généralisations "naturelles" des postconditions qu'on veut démontrer, les preuves de ces deux fonctions ne présentent pas de difficulté importante.

```

1 /*@ requires n1  $\geq$  0  $\wedge$  n2  $\geq$  0;
2 @ requires \valid(p1+(0..n1-1));
3 @ requires \valid(p2+(0..n2-1));
4 @ requires \valid(p+(0..n1+n2-1));
5 @ requires \separated(p1+(0..n1-1),
6 @ p2+(0..n2-1),p+(0..n1+n2-1));
7 @ requires is_perm(p1,n1)  $\wedge$ 
8 @ is_perm(p2,n2);
9 @ assigns p[0..n1+n2-1];
10 @ ensures is_perm(p,n1+n2); */
11
12 /*@ loop invariant 0  $\leq$  i  $\leq$  n1+n2;
13 @ loop invariant is_sum(p,p1,p2,n1,i);
14 @ loop invariant is_fct(p,i,n1+n2);
15 @ loop invariant is_linear(p,i);
16 @ loop assigns i, p[0..n1+n2-1];
17 @ loop variant n1+n2-i; */

```

Listing 5 – Contrat et annotations de boucle de la fonction sum.

Le listing 6 montre la spécification complète de la fonction `insert_inplace`. Son contrat est plus simple que celui de la fonction `insert`, puisqu'il n'y a pas de tableau `q`. Ses annotations de boucle sont similaires à celles de la fonction `insert`, sauf l'invariant

```
loop invariant is_insert(q,p,j,i,n);
```

qui est remplacé par deux invariants qui relient les valeurs du tableau dans l'état courant (étiquette `Here`) et dans l'état initial (étiquette `Pre`). Le listing 7 définit les deux prédicats utilisés dans ces invariants. Chaque expression `e` en ACSL peut être écrite `\at(e, L)`, signifiant que l'expression `e` est évaluée dans l'état identifié par l'étiquette `L`. Les étiquettes `Pre` et `Here` sont prédéfinies. Le prédicat `is_insert_inplace` paraphrase le code de la boucle. L'invariant de boucle

`loop invariant is_insert_inplace{Pre,Here}(p, j, i, n);`
 l'utilise pour décrire l'évolution de la partie $p[0..j - 1]$ du tableau p , entre son indice 0 inclus et l'indice courant j de la boucle exclu. Le prédicat `is_eq_gt` spécifie qu'un suffixe d'un tableau n'est pas modifié entre les états étiquetés par L1 et L2. L'invariant de boucle

`loop invariant is_eq_gt{Pre,Here}(p, j, n);`
 l'utilise pour spécifier que, lors de l'exécution de la boucle, la partie $p[j..n - 1]$ du tableau p , de l'indice courant de boucle (inclus) à la fin du tableau, n'est pas modifiée.

```

1 /*@ requires 0 ≤ i ≤ n ∧ \valid(p+(0..n-1)) ∧ is_perm(p,n);
2   @ assigns p[0..n];
3   @ ensures is_perm(p,n+1); */
4 void insert_inplace(int p[], int i, int n) {
5     if (0 ≤ i ∧ i ≤ n) {
6         p[n] = i;
7         /*@ loop invariant 0 ≤ j ≤ n;
8           @ loop invariant is_insert_inplace{Pre,Here}(p, j, i, n);
9           @ loop invariant is_eq_gt{Pre,Here}(p, j, n);
10          @ loop invariant is_fct(p, j, n+1);
11          @ loop invariant is_linear(p, j);
12          @ loop assigns j, p[0..n-1];
13          @ loop variant n-j; */
14         for(int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
15     }
16 }
    
```

Listing 6 – Fonction `insert_inplace` annotée.

```

1 /*@ predicate is_insert_inplace(L1,L2)(int *p, Z b, Z c, Z d) =
2   @ ∀ Z k; 0 ≤ k < \at(b,L2) ⇒
3   @   \at(p[k],L2) == ((\at(p[k],L1) == c) ? d : \at(p[k],L1));
4   @ predicate is_eq_gt{L1,L2}(int *p, Z b, Z c) =
5   @   ∀ Z k; b ≤ k < c ⇒ \at(p[k],L2) == \at(p[k],L1); */
    
```

Listing 7 – Prédicats ACSL pour la fonction `insert_inplace`.

Grâce à ces invariants, la correction de la fonction `insert_inplace` est prouvée automatiquement par WP. La difficulté ici était de penser à spécifier que la boucle ne modifie pas la partie $p[j..n - 1]$ du tableau p .

5 Conclusion

Nous avons implémenté en C deux opérations sur les permutations, spécifié formellement leur comportement et démontré automatiquement que ces implémentations étaient conformes à leur spécification.

Pour les trois opérations (y compris la version `inplace`) le greffon WP de Framac-Neon-20140301 utilisant les solveurs Alt-Ergo 0.95.2, CVC3 2.4.1 et CVC4 1.3 démontre 54 obligations de preuve en moins d'une minute sur un PC Intel Core i5-3230M à 2,6 GHz sous Linux Ubuntu 14.04. La durée allouée à chaque solveur pour chaque obligation de preuve a été étendue de 10 secondes (valeur par défaut) à une minute.

L'utilisation de la vérification déductive automatique de programmes en combinatoire est pour l'instant peu répandue. Dans ce domaine, le travail le plus proche porte sur la fonction de calcul du conjugué d'une partition d'entiers, implémentée en C dans le logiciel SCHUR, spécifiée en ACSL et vérifiée avec Framac [BHMT10]. Un générateur de toutes les solutions du problème des n -reines est spécifié et vérifié avec l'outil Why3 [BFM⁺13]. La preuve formelle complète nécessite cependant des étapes interactives [Fil12].

Dans [DGG16] nous avons démontré, de manière interactive avec Coq, la correction des opérations sur les cartes qui utilisent les opérations sur les permutations présentées ici. Un défi serait d'étendre la présente étude avec une automatisation de ces démonstrations.

Remerciements. Les auteurs remercient J.-L. Baril et les relecteurs pour leurs suggestions.

Références

- [Alt] The Alt-Ergo SMT solver. <http://alt-ergo.lri.fr>.
- [BBCD15] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Plug-in Manual*, 2015. <http://frama-c.com/download/frama-c-wp-manual.pdf>.
- [BCF⁺13] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2013. <http://frama-c.com/acsl.html>.
- [BFM⁺13] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. The Why3 platform 0.81, 2013. <https://hal.inria.fr/hal-00822856>.
- [BHMT10] F. Butelle, F. Hivert, M. Mayero, and F. Toumazet. Formal proof of SCHUR conjugate function. In S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton, editors, *Artificial Intelligence and Symbolic Computation (AISC)*, volume 6167 of *LNCS*, pages 158–171. Springer, Heidelberg, 2010.
- [BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
- [Coq] The Coq Proof Assistant. <http://coq.inria.fr>.
- [CVC] CVC4. <http://cvc4.cs.nyu.edu/web/>.
- [DGG16] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In B. Aichernig and C. A. Furia, editors, *Tests and Proofs (TAP)*, volume **** of *LNCS*. Springer, Heidelberg, 2016. To appear.
- [Fil12] J.-C. Filliâtre. Verifying two lines of C with Why3 : An exercise in program verification. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 83–97. Springer, Heidelberg, 2012.
- [GGP15a] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In D. Baelde and J. Alglave, editors, *Journées Francophones des Langages Applicatifs (JFLA)*, 2015. <https://hal.inria.fr/hal-01099135>.
- [GGP15b] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In J. C. Blanchette and N. Kosmatov, editors, *Tests and Proofs (TAP)*, volume 9154 of *LNCS*, pages 109–128. Springer, Heidelberg, 2015.
- [Kit11] S. Kitaev. *Patterns in permutations and words*. Springer, Berlin, 2011.
- [KKP⁺15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. *Formal Aspects of Computing*, 27(3) :573–609, 2015.

Quelle confiance peut-on établir dans un système intelligent ?

Lydie du Bousquet^{1,2}, Masahide Nakamura³

¹Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France

²CNRS, LIG, F-38000 Grenoble, France

³Kobe University, Japon

e-mail : Lydie.du-Bousquet@imag.fr, masa-n@cs.kobe-u.ac.jp

Résumé

De plus en plus, nous sommes confrontés à des systèmes intelligents (“*smart systems*”). Ce sont des systèmes supposés apprendre de leur environnement pour apporter des réponses les plus pertinentes possibles pour leurs utilisateurs. Une caractéristique de ces systèmes est que leurs comportements évoluent avec le temps. Dans le projet CNRS-PICS *Safe-Smartness*, collaboration entre le LIG et l’université de Kobe, nous étudions comment exprimer les propriétés d’oracle quant à la qualité d’un système intelligent.

Introduction Les systèmes intelligents (*smart systems*) sont caractérisés par leur capacité à prédire ou à adapter leurs comportements à partir d’une analyse d’un ensemble de données. Ce type d’application se retrouve dans différents domaines (domotique, médical, transport, ...). Selon la *Royal Academy of Engineering*, on peut classer les systèmes intelligents en trois catégories¹. Ainsi, selon la façon dont les données sont collectées et utilisées, ces systèmes permettent :

- d’aider les concepteurs à produire une nouvelle version de l’application plus efficace,
- de calculer et de présenter des informations à l’opérateur humain pour qu’il puisse prendre des décisions pertinentes par rapport à la situation,
- de choisir les actions à effectuer en fonction de la situation et de les exécuter, sans intervention humaine.

En ce qui concerne les deux dernières catégories, la question de la qualité du système prend toute son importance. Un tel système ne doit pas induire ses utilisateurs en erreur et/ou ne doit pas exécuter des actions inappropriées. L’objet du projet CNRS-PICS *Safe-Smartness* est d’établir des méthodes de validation pour ce type de systèmes. Dans la suite, illustrons les difficultés sur la base d’un exemple.

Exemple Soit un système de climatisation programmable “intelligent”. L’utilisateur peut programmer une heure h et une température t . Le système doit alors faire

1. <http://www.raeng.org.uk/publications/reports/smart-infrastructure-the-future>

en sorte que la pièce considérée soit à la température t à l'heure h , tout en minimisant l'énergie nécessaire pour y parvenir. Pour cela, le système doit tenir compte de l'inertie thermique, qui est propre à chaque pièce. Le système est donc doté d'une partie "intelligente", qui enregistre un certain nombre de facteurs et "apprend" la vitesse à laquelle la pièce se tempère en fonction de ces facteurs. Après le temps d'apprentissage, le système doit choisir au mieux à quel moment démarrer pour atteindre la température t à l'heure fixée.

Propriétés attendues du système Un des premiers éléments à vérifier est donc que la pièce soit à la température t à l'heure h . Comme cela dépend de la pièce, il faut envisager de le faire à chaque exécution (*monitoring*). Pour autant, la non-satisfaction de cette propriété ne signifie pas forcément que le système est incorrect. Deux situations peuvent se produire. (a) L'utilisateur peut fixer un objectif impossible à atteindre ; par exemple demander à 7h55 que la pièce soit à 21 °C à 8h alors que la température actuelle est de 10 °C. (b) L'objectif peut aussi être atteignable en théorie, mais l'environnement réel ne pas le permettre ; par exemple, si une fenêtre est laissée ouverte en plein hiver.

Dans ces conditions, le système doit être en mesure de fournir une notification si l'objectif est jugé non-atteignable (e.g. 7h55 pour 8h et 11 °C d'écart). Il sera alors possible de valider si la notification est exacte (i.e. pertinente). La présence de faux-négatifs (notification "impossible" mais la température est atteinte dans les délais) et de faux-positifs (notification "possible" mais la température n'est finalement pas atteinte dans les délais) est un indicateur de qualité du système .

Le système doit aussi être en mesure d'indiquer si les caractéristiques de l'environnement ont changé (e.g. inertie différente si fenêtre ouverte). Cela est nécessaire pour faire la différence entre un dysfonctionnement du système et une anomalie ponctuelle d'usage.

Par ailleurs, on attend du système qu'il minimise l'énergie nécessaire pour atteindre l'objectif. Ici, on peut par exemple s'assurer que le système ne démarre pas trop tôt, c'est-à-dire qu'à partir du moment où le climatiseur commence à chauffer (resp. à refroidir) la pièce, il ne passe pas plusieurs fois en attente.

Mais ces propriétés ne permettent pas de s'assurer directement que le système *apprend* correctement. L'apprentissage est un processus progressif. Pour décider de la qualité de l'apprentissage, il faut observer les exécutions successives et s'assurer que l'objectif est atteint de plus en plus souvent, et/ou de mieux en mieux (ici, en consommant de moins en moins d'énergie). La difficulté ici est qu'il s'agit d'une tendance, qui peut se dégrader de temps à autre, en particulier si l'utilisateur change ses habitudes. Le fait que la tendance se dégrade pendant un temps ne signifie donc pas forcément que le système est incorrect, mais peut caractériser le fait qu'il entre dans une nouvelle étape d'adaptation.

Les outils habituels d'expression et d'évaluation de propriétés ne sont pas adaptés pour nos besoins. C'est pourquoi, l'objectif du projet *Safe-Smartness* est de proposer des moyens alternatifs pour exprimer et valider la qualité de l'apprentissage.

Génération systématique de scénarios d'attaques contre des systèmes industriels

Maxime Puys, Marie-Laure Potet, and Jean-Louis Roch

VERIMAG, Univ. Grenoble Alpes / Grenoble-INP, France
prénom.nom@imag.fr *

Résumé Les systèmes industriels (SCADA) sont la cible d'attaques informatiques depuis Stuxnet [4] en 2010. De part leur interaction avec le mode physique, leur protection est devenue une priorité pour les agences gouvernementales. Dans cet article, nous proposons une approche de modélisation d'attaquants dans un système industriel incluant la production automatique de scénarios d'attaques. Cette approche se focalise sur les capacités de l'attaquant et ses objectifs en fonction des protocoles de communication auxquels il fait face. La description de l'approche est illustrée à l'aide d'un exemple.

1 Introduction

Les systèmes industriels aussi appelés SCADA (Supervisory Control And Data Acquisition) sont la cible de nombreuses attaques informatiques depuis Stuxnet [4] in 2010. De part leur interaction avec le monde physique, ces systèmes peuvent représenter une réelle menace pour leur environnement. Suite à une récente augmentation de la fréquence des attaques, la protection des installations industrielles est devenue une priorité des agences gouvernementales. Ces systèmes diffèrent également de l'informatique de gestion du fait de leur très longue durée de vie, de leur difficulté à appliquer des correctifs et des protocoles souvent spécifiques utilisés.

État de l'art : La modélisation et la génération de scénarios d'attaques sont essentielles à la sécurité des systèmes industriels. En 2013, la norme IEC 62443-3-3 [2] détaille de façon très précise une méthode d'analyse de la sécurité informatique des installations industrielles. En 2015, Conchon *et al.* [1] proposent une approche se basant sur EBIOS. Toujours en 2015, Kriaa *et al.* [3] décrivent S-CUBE, une approche de modélisation de systèmes industriels faisant le pas entre la sécurité et la sûreté de fonctionnement. Leur article analyse un grand nombre d'approches de production de scénarios d'attaques et conclut notamment qu'aucune n'est automatisée ni facilement automatisable.

*. Ce travail a été partiellement financé par le LabEx PERSYVAL-Lab (ANR-11-LABX-0025) et le projet Programme Investissement d'Avenir FSN AAP Sécurité Numérique n° 3 ARA-MIS (P3342-146798).

Contributions : Nous proposons dans cet article une approche de modélisation d'attaquants basée à la fois sur l'infrastructure et les possibilités d'attaques dans un système industriel. Cette approche se base sur les composants d'une infrastructure et les canaux de communication entre ses composants [6]. Elle vise la production automatique de scénarios d'attaques. Nous nous focalisons sur l'attaquant en modélisant sa position dans l'infrastructure, ses objectifs d'attaques et les protocoles qu'il peut utiliser en tenant compte de leurs propriétés de sécurité. Enfin, à l'aide d'une approche systématique, nous générons l'ensemble des scénarios d'attaques pour lesquels un attaquant est capable de réaliser l'un de ses objectifs face à un protocole donné.

Plan : La section 2 détaille la modélisation des attaquants et la production des scénarios d'attaques à l'aide d'un un exemple. Ensuite, la section 3 décrit comment nous souhaitons inclure cette approche dans une approche *Model-Based Testing* plus globale. Enfin la section 4 conclut.

2 Approche de génération des scénarios d'attaques

Dans cette section, nous détaillons le formalisme utilisé dans notre modélisation des attaquants et comment exploiter cette modélisation pour générer des scénarios d'attaques. Cette approche est double. Elle propose dans un premier temps d'étudier les objectifs des attaquants et comment ils pourraient les réaliser (approche descendante) avant de les mettre dans un second temps face aux protections apportées par les protocoles de communication (approche ascendante).

2.1 Approche descendante

Nous commençons par définir l'ensemble des attaquants \mathcal{A} et l'ensemble des objectifs d'attaques \mathcal{O} . Ces ensembles sont liés par la relation \mathcal{R}_{Obj} entre un attaquant a et un objectif o tel que $\mathcal{R}_{Obj} \subseteq \mathcal{A} \times \mathcal{O}$ si a cherche à atteindre o fait partie de notre analyse de risque. Il va donc de soit que cette relation, fournie par le concepteur du modèle doit tenir compte de la position des attaquants dans l'architecture globale du système (ex. : un attaquant ne peut pas avoir pour objectif la modification d'un message s'il n'y a jamais accès).

Exemple :

Nous considérons l'infrastructure de communication en figure 1 où les attaquants (en couleur) sont $\mathcal{A} = \{Client_A, Routeur_A\}$ (un client et un routeur compromis) et les objectifs d'attaques (tirés de recommandations gouvernementales) considérés, sont $\mathcal{O} = \{VolId, ContAuth, Alte, Alte_C\}$ avec :

— $VolId$ = Vol d'identifiants,

- *ContAuth* = Contournement d'authentification,
- *Alte* = Altération d'un message,
- *Alte_C* = Altération ciblée d'un message.

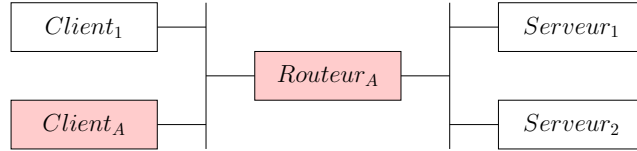


FIGURE 1: Exemple d'infrastructure

Les attaquants \mathcal{A} et les objectifs \mathcal{O} sont liés par la relation \mathcal{R}_{Obj} définie en Table 1, où un ✓ signifie que l'objectif o est retenu pour l'attaquant a .

\mathcal{R}_{Obj}	<i>VolId</i>	<i>ContAuth</i>	<i>Alte</i>	<i>Alte_C</i>
<i>Client_A</i>		✓		
<i>Routeur_A</i>	✓	✓	✓	✓

TABLE 1: Exemple d'objectifs retenus pour chaque attaquant

Nous définissons ensuite l'ensemble des vecteurs d'attaques \mathcal{V} et $Real \subseteq \mathcal{O} \times \mathbb{P}(\mathcal{V})$, la fonction entre un objectif et l'ensemble des parties de \mathcal{V} (ie. les combinaisons de vecteurs d'attaques). Ainsi, $Real(o)$ décrit la réalisation d'un objectif o à l'aide de vecteurs de \mathcal{V} . Un vecteur peut être vu comme les techniques, tactiques ou procédés pouvant servir à la réalisation d'une attaque [1].

Exemple :

Nous considérons ici les vecteurs $\mathcal{V} = \{Lire, Usurp, Mod, Rej\}$ avec :

- *Lire* = Lecture (et compréhension) d'un message
- *Usurp* = Usurpation d'une identité
- *Mod* = Modification d'un message
- *Rej* = Rejeu d'un message

La fonction $Real$ décrivant comment réaliser les objectifs d'attaque à l'aide des vecteurs d'attaques peut par exemple être :

- $Real(VolId) = \{\{Lire\}\}$
- $Real(ContAuth) = \{\{Usurp\}, \{Rej\}\}$
- $Real(Alte) = \{\{Mod\}\}$
- $Real(Alte_C) = \{\{Lire, Mod\}\}$

En particulier, *ContAuth* peut être réalisé en usurpant une identité **ou** en rejouant un message d'authentification (ex. : l'envoi d'un mot de passe). Tandis que *Alte_C* nécessite **à la fois** la capacité de modifier un message et d'en comprendre le contenu.

Ainsi, nous sommes en mesure de représenter les attaquants, leurs objectifs et comment ces objectifs peuvent être réalisés au moyen de vecteurs d'attaques. La section suivante décrit comment une analyse des propriétés de sécurité offertes par les protocoles vérifie si ces objectifs sont atteignables.

2.2 Approche ascendante

Dans un second temps, nous définissons l'ensemble des configurations des protocoles \mathcal{P} considérés pour l'analyse. Dans la suite de cet article nous considérerons chaque configuration comme un protocole différent. $Vect \subseteq \mathcal{P} \times \mathbb{P}(\mathcal{V})$ est l'ensemble des vecteurs d'attaques de \mathcal{V} accessibles à l'attaquant pour chaque protocole (ie. leurs faiblesses exploitables).

Exemple :

Pour les protocoles $\mathcal{C} = \{\text{MODBUS}, \text{FTP}, \text{FTP}_{Auth}, \text{OPC-UA}_{None}, \text{OPC-UA}_{Sign}, \text{OPC-UA}_{SignEnc}\}$, les capacités d'attaques $Vect$ sont définies en table 2, où un ✓ signifie que le vecteur d'attaque est accessible à l'attaquant pour ce protocole.

<i>Vect</i>	<i>Lire</i>	<i>Usurp</i>	<i>Mod</i>	<i>Rej</i>
MODBUS	✓	✓	✓	✓
FTP	✓	✓	✓	✓
FTP _{Auth}	✓		✓	✓
OPC-UA _{None}	✓	✓	✓	✓
OPC-UA _{Sign}	✓			
OPC-UA _{SignEnc}				

TABLE 2: Exemple de capacités d'attaques retenus pour chaque protocole

Les protocoles MODBUS, FTP et OPC-UA_{None} ne garantissent aucune sécurité et permettent donc tous les vecteurs d'attaques. Le protocole FTP_{Auth} ajoute une authentification à l'aide d'un mot de passe empêchant l'usurpation d'identité. Les protocoles OPC-UA_{Sign} et OPC-UA_{SignEnc} apportent des signatures cryptographiques et de l'estampillage aux messages, empêchant ainsi leur usurpation, modification ou rejeu. Enfin OPC-UA_{SignEnc} garantit également la confidentialité des communications.

Il est alors possible de déterminer si un objectif o est réalisable à l'aide de l'ensemble des vecteurs d'attaques pour un protocole p en vérifiant si : $\exists e \in \mathcal{R}_{Real}(o) \mid e \subseteq Vect(p)$. Alors, l'ensemble des scénarios d'attaques $\mathcal{S}_{a,p}$ pour un attaquant a et un protocole p est alors défini par :

$$\mathcal{S}_{a,p} = \{(o, e) \mid o \in \mathcal{O} \wedge e \subseteq \mathcal{R}_{Real}(o) \wedge e \subseteq Vect(p) \wedge (a, o) \in \mathcal{R}_{Obj}\}$$

L'ensemble des scénarios d'attaques à considérer dans le cadre d'une campagne de test est alors l'ensemble des $\mathcal{S}_{a,p}$, $\forall a \in \mathcal{A}$ et pour tous les protocoles de \mathcal{P} que pourraient utiliser chaque attaquants.

Exemple :

Dans notre exemple, si l'on suppose que $Client_A$ communique via FTP_{Auth} :

$$\mathcal{S}_{Client_A,FTP_{Auth}} = \{(ContAuth, Rej)\}$$

Cela s'explique par le fait que seul l'objectif $ContAuth$ est considéré pour $Client_A$ (table 1) et qu'il est réalisable avec au moins l'un des vecteurs d'attaques $Usurp$ ou Rej dont le dernier est offert par FTP_{Auth} . De même pour $Routeur_A$ qui est face à la fois à $OPC-UA_{None}$ et FTP_{Auth} :

$$\mathcal{S}_{Routeur_A,OPC-UA_{None}} = \{(VolId, Lire), (ContAuth, Usurp), (ContAuth, Rej), (Alte, Mod), (Alte_C, (Lire, Mod))\}$$

$$\mathcal{S}_{Routeur_A,FTP_{Auth}} = \{(VolId, Lire), (ContAuth, Rej), (Alte, Mod), (Alte_C, (Lire, Mod))\}$$

3 Méthodologie globale

Notre objectif est d'utiliser cette phase d'analyse dans une approche globale allant de la modélisation du système à la production automatique des paquets réseau implémentant et testant les attaques identifiées. Nous proposons donc une approche *Model-Based Testing* dans l'objectif de vérifier si les attaques trouvées par l'approche sont effectivement jouables sur une plate-forme, voire de quantifier leur plausibilité. La figure 2 illustre la méthodologie que nous voulons développer. L'approche part d'une architecture représentant les composants du systèmes, les canaux de communication et les protocoles (similaire à la figure 1) ; croisée avec des propriétés de sécurité spécifiant les objectifs des attaquants. Ensuite, l'analyse présentée en section 2 permet d'obtenir les vecteurs d'attaques à utiliser par des attaquants pour violer les propriétés de sécurité. Ces vecteurs sont ensuite concrétisés en paquets réseaux à l'aide d'une bibliothèque décrivant comment implémenter les vecteurs pour chaque protocole (ex. : comment modifier un paquet OPC-UA). Enfin, ces paquets sont instanciés, soit de manière aléatoire, soit en fonction de la logique applicative de la plate-forme. Cette approche pourrait se généraliser aux systèmes d'information, mais elle exploite néanmoins des propriétés souvent présentes dans les systèmes industriels telles que l'absence de réseaux dynamiques, simplifiant la représentation de l'infrastructure.

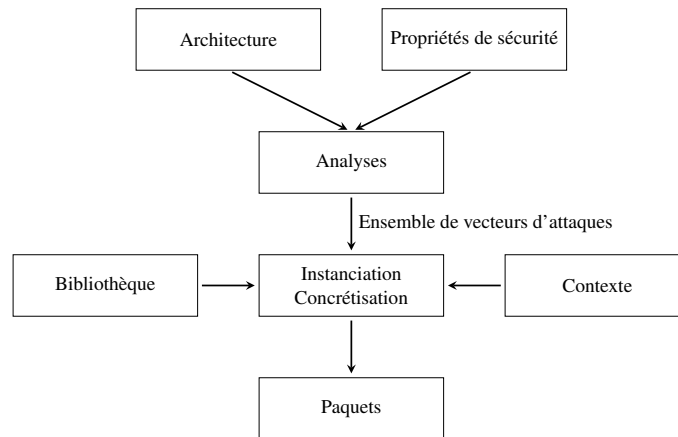


FIGURE 2: Méthodologie globale

4 Conclusion

En conclusion, nous proposons une approche de modélisation d'attaquants basée à la fois sur l'infrastructure d'un système industriel et des possibilités d'attaques contre celui-ci. Des scénarios d'attaques montrent comment un attaquant peut exploiter les faiblesses d'un protocole de communication pour satisfaire ses objectifs. Nous nous focalisons ici sur les possibilités de forger des attaques. À terme, nous souhaitons produire des attaques sur le fonctionnement du système, tenant donc compte du contenu des messages. L'analyse en elle-même pourrait être améliorée en considérant des attaques en plusieurs étapes. Par exemple, nous avons énoncé en section 2.2 que la configuration FTP_{Auth} ne permet pas le vecteur d'attaque $Usurp$ car elle utilise une authentification par mot de passe. Cependant, le vecteur d'attaque $Lire$ pourrait révéler ce mot de passe et ainsi donner l'accès en un second temps au vecteur $Usurp$. Ces attaques en plusieurs étapes, tenant alors compte de l'ordre des actions à exécuter, pourraient être décrites sous forme d'arbre d'attaques [5] (représentation classique pour modéliser des attaquants). Un autre axe pourrait être la prise en compte de plusieurs attaquants coopérant pour des objectifs communs.

Références

1. Sylvain Conchon and Jean Caire. Expression des besoins et identification des objectifs de résilience. *C&esar'15*, 2015.
2. ISA-62443-3-3. Security for industrial automation and control systems, part 3-3 : System security requirements and security levels, 2013.

3. S Kriaa, M Bouissou, and Y Laarouchi. A model based approach for SCADA safety and security joint modelling : S-Cube. In *IET System Safety and Cyber Security*. IET Digital Library, 2015.
4. Ralph Langner. Stuxnet : Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3) :49–51, 2011.
5. Bruce Schneier. Attack trees. *Dr. Dobbs's journal*, 24(12) :21–29, 1999.
6. Theodore J Williams. *A Reference Model for Computer Integrated Manufacturing (CIM) : A Description from the Viewpoint of Industrial Automation : Prepared by CIM Reference Model Committee International Purdue Workshop on Industrial Computer Systems*. Instrument Society of America, 1991.

MBeeTle - un outil pour la génération de tests à-la-volée à l'aide de modèles *

Julien Lorrain¹

julien.lorrain@femto-st.fr

Elizabeta Fournieret²

elizabeta.fournieret@smartesting.com

Frédéric Dadeau¹

frederic.dadeau@femto-st.fr

Bruno Legeard^{1,2}

bruno.legeard@femto-st.fr

¹Institut FEMTO-ST, Besançon, France

²Smartesting Solutions & Services, Besançon, France

Résumé

Le Model-Based Testing est une activité permettant, à partir de la modélisation du système sous test (SUT), de générer des tests pour la validation du SUT par rapport aux spécifications de ce système. Cet article présente MBeeTle, un outil de génération de test à-la-volée pour le MBT, ainsi que ses principes et ces stratégies de génération. Il permet la recherche des anomalies profondes et la validation du modèle de test en générant et exécutant des pas de test sur la base d'une approche aléatoire. MBeeTle est présenté avec une expérimentation sur la une norme d'interfaces utilisateurs et logicielles pour les systèmes d'exploitation (POSIX).

1 Introduction et contexte

Dans ce papier nous présentons MBeeTle, un outil de génération de test à-la-volée [1] basé sur le Model-Based Testing (MBT) [2] pour l'exploration des systèmes sous test. Le MBT est une activité permettant de générer des tests abstraits à partir de la modélisation du SUT qui nécessite l'utilisation d'une couche d'adaptation afin de les concrétiser pour pouvoir les exécuter sur le SUT. La catégorie principale de génération de test est la génération dite hors-ligne, dont le principe est de générer des scénarios de test abstrait. Pour pouvoir les exécuter automatiquement sur le SUT, il faut créer le lien entre les données abstraites du modèle et le SUT. C'est ici que la couche d'adaptation est nécessaire pour associer à chaque donnée abstraite les informations de concrétisation permettant leur exécution sur le SUT. Une autre catégorie de génération, implémentée dans l'outil MBeeTle, est la génération à-la-volée. Le principe est d'attendre le résultat du premier pas de test avant de générer les pas suivants. Il est donc nécessaire de disposer de la couche d'adaptation pour mettre en œuvre cette méthode. Cette approche permet de générer des tests fonctionnels longs (*i.e.* plusieurs centaines de pas de test) basés sur les comportements modélisés du SUT.

*Cet outil a en partie été réalisé lors des projets ANR ASTRID OSEP et ANR ASTRID maturation MBT_SEC.

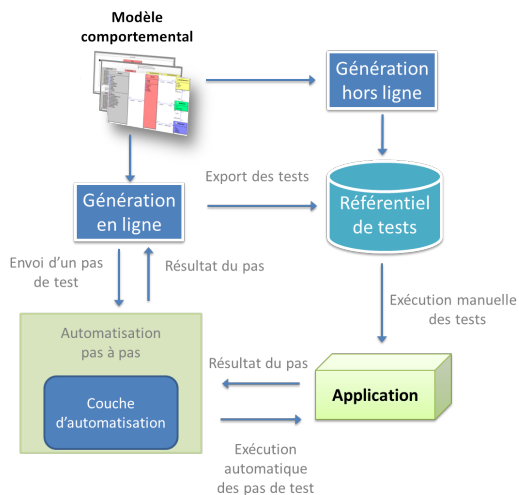


FIGURE 1 – Approche à-la-volée

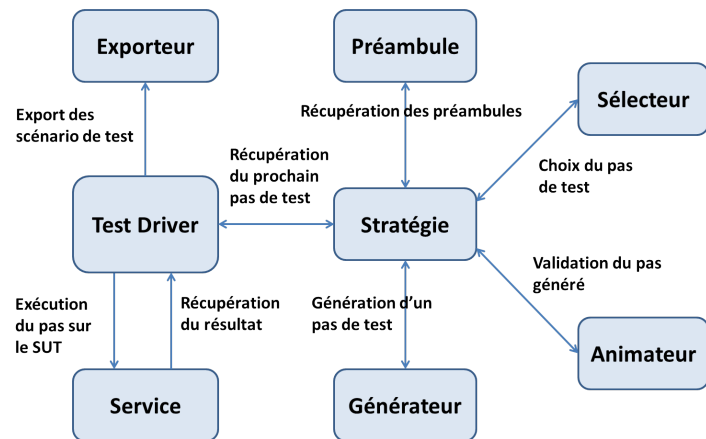


FIGURE 2 – Schéma de fonctionnement

MBeeTle est un outil de génération de tests à-la-volée qui génère puis exécute directement chaque pas de test sur le SUT. Il se base sur l'utilisation de modèles réalisés à l'aide de sous-ensembles des langages UML et OCL [3]. Il s'intègre dans une approche hors-ligne existante, représenté dans la figure 1 et réutilise des briques logicielles de l'outil Smartesting¹ CertifyIt [4] (un générateur de tests fonctionnels hors-ligne), notamment son animateur pour valider sur le modèle les différents pas de test générés par les algorithmes de MBeeTle.

2 L'outil MBeeTle

Le principe de MBeeTle est de générer des scénarios de test plus ou moins longs (plusieurs centaines de pas) à l'aide d'une combinaison d'algorithmes de génération de pas de test, de préambules et d'un ou plusieurs critères de sélection de tests nommés sélecteurs. Cette combinaison est appelé une *stratégie de génération*.

MBeeTle est composé de plusieurs modules, décrit dans la figure 2, ayant chacun un objectif dans la création d'un scénario de test à-la-volée. Le premier, le Test Driver, pilote et lie les autres modules. Les générateurs permettent, grâce à un tirage aléatoire, de générer des pas de test en se servant des informations du modèle. Les pas générés sont cohérents, *i.e.* utilisent des valeurs existantes dans le modèle, mais pas forcément valides par rapport à l'état courant du modèle. Un pas de test est valide si son animation sur le modèle ne rentre pas en contradiction avec les contraintes et/ou les pré-conditions présentes dans le modèle. C'est le rôle de l'animateur : si le pas est valide, il est envoyé au sélecteur. Si le sélecteur choisit le pas alors la stratégie le valide et le retourne au TestDriver qui, au travers du module de service, l'exécute et récupère le résultat de cette exécution. Cette suite d'actions est alors recommencée jusqu'à remplir les conditions d'arrêt de la campagne de génération à-la-volée.

L'interface utilisateur présenté figure 3 permet de configurer entièrement l'outil et d'afficher des informations de la campagne courante, comme le nombre de scénarios générés et la couverture des opérations et des comportements du modèle. Les deux algorithmes de génération de tests utilisés dans MBeeTle sont basés sur l'aléatoire. Le premier est purement pseudo-aléatoire et ne tient pas compte de ce qui a été généré précédemment. Afin de facili-

1. <http://www.smartesting.com/>

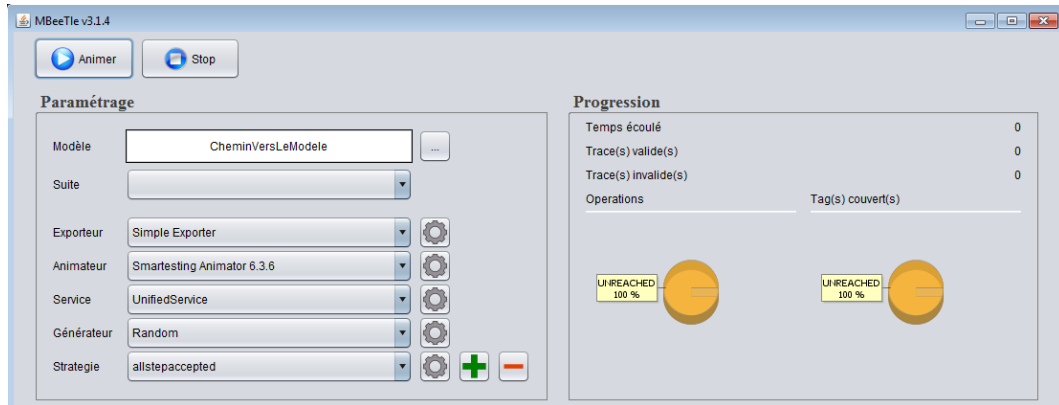


FIGURE 3 – L'interface de l'outil MBeeTle

ter le passage des comportements complexes, le second implémente une pondération sur les opérations et sur les paramètres.

Trois sélecteurs sont disponibles dans MBeeTle. Le premier est simple et utilise n'importe quel pas de test qui lui est soumis. Le second se base sur les comportements modélisés et optimise leur couverture en favorisant les pas de test qui augmentent la couverture des comportements. Le dernier explore les états du système dans des contextes plus divers que l'approche habituelle. Il se base sur les cas non-passants/passants, *i.e.* les pas qui provoquent un état d'erreur du système et les autres, le but étant de générer des séquences inhabituelles d'opérations qui peuvent provoquer de nouvelles anomalies.

Avant de lancer la génération l'outil peut importer des tests existants. Les bénéfices de cette ré-utilisation de tests sont multiples : augmentation de la diversité des contextes d'activation des comportements déjà testés en évitant les enchainements de comportements déjà couverts, diminution du temps de génération et passage des goulots d'étranglements du modèle (une difficulté inhérente aux approches aléatoire) et du coup augmentation de l'espace d'exploration. L'outil permet d'exporter les résultats sous différents formats.

3 Expérimentations

Le périmètre du standard POSIX² modélise la gestion de fichiers, d'utilisateurs et de droits pour les systèmes de type UNIX. Le modèle est composé de 19 classes, avec 24 opérations pour 2 442 lignes d'OCL et chaque interface de l'API est accessible à tout moment. Les tests hors-ligne ont été générés en 24 heures pour couvrir 88% du modèle. Nous avons choisi de faire des lancements/exécutions avec MBeeTle respectant un volume de 20 000 pas correspondant à celui des tests fonctionnels, tout en découpant les lancements en trois configurations types : 1000 scénarios de 20 pas de test, 250 scénarios de 80 pas de test et 100 scénarios de 200 pas de test. Pour chaque stratégie de génération on évalue la couverture des comportements, et le temps de génération de tests et d'exécution des pas de test. Par ailleurs nous avons généré, avec l'outil Pitest³, 1290 mutants qui nous ont servi pour une analyse mutationnelle. La génération hors-ligne tue 548 mutants avec 1484 tests, la génération à la volée en tue en moyenne 546 et la somme des deux approches en tue 630.

2. <https://standards.ieee.org/findstds/standard/1003.1-2008.html>

3. <http://pitest.org/>

De cette expérimentation ressort que MBeeTle permet de couvrir rapidement plus de la moitié des comportements du modèle (environ 16 minutes contre 3 heures pour la version hors-ligne). En réutilisation des tests hors-ligne générés, on augmente significativement la couverture du modèle. Par ailleurs MBeeTle est capable de générer rapidement un volume de pas de test conséquent (ici 20000 pas en 13,6 secondes en moyenne). De plus l'utilisation de cette approche a permis de détecter des mutants différents de ceux détectés par l'approche hors-ligne.

Par ailleurs, l'application de l'approche à-la-volée sur cette expérimentation montre que les artefacts de modélisation hors-ligne sont réutilisables avec un faible coût de main d'œuvre pour adapter le banc de test pour une utilisation pas à pas. Il en ressort également que MBeeTle peut être utilisé pour mettre au point le modèle en générant rapidement des scénarios de test.

4 Conclusion et perspectives

Nous avons présenté MBeeTle en association avec l'approche hors-ligne sur un modèle de taille réelle. Dans ce contexte il permet de compléter cette approche avec un faible coût d'adaptation et une réutilisation complète des ressources et des outils mis en place. Il permet d'augmenter l'espace d'exploration pour activer des comportements dans d'autres contextes. Par ailleurs sur un autre cas d'étude, une implémentation de la norme PKCS#11⁴, il a permis de montrer des erreurs dues à des fuites mémoires qui n'ont pas été trouvés par les outils classiques grâce à une utilisation intensive de l'outil⁵.

Pour la suite, nous prévoyons d'adapter cette approche à-la-volée pour en faire une approche de tests en-ligne où nous utiliserons le retour du SUT pour calculer le pas de test pour calculer le prochain. De part son aspect modulaire et ses stratégies paramétrables, des expérimentations pour de la génération en-ligne (utilisation du retour d'exécution des pas de test pour décider du prochain pas de test à générer) [5] sont envisageables. Par ailleurs pour améliorer son efficacité, de nouvelles stratégies seront ajoutées pour palier aux difficultés rencontrées, par exemple sur les modèles à goulots d'étranglements.

Références

- [1] J.-C. Fernandez, C. Jard, T. Jérón, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites," in *Computer Aided Verification*, pp. 348–359, Springer, 1996.
- [2] M. Utting and B. Legeard, *Practical model-based testing : a tools approach*. Morgan Kaufmann, 2010.
- [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*, pp. 95–104, 2007.
- [4] 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, (New York, NY, USA), pp. 45–48, ACM, 2008.
- [5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 75–84, IEEE, 2007.

4. <https://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm>

5. http://ucaat.etsi.org/2015/presentations/FEMTOST_FOURNERET.pdf

BINSEC : Plate-forme d'analyse de code binaire *

Adel Djoudi ¹ Robin David ^{1,3} Thanh Dinh Ta ² Josselin Feist ²

¹ CEA, LIST, 91191 Gif-sur-Yvette France, e-mail : prenom.nom@cea.fr

² Vérimag, Grenoble, France, e-mail : prenom.nom@imag.fr

³ Université de Lorraine, CNRS and Inria, LORIA, France, e-mail : prenom.nom@loria.fr

BINSEC est une plate-forme pour l'analyse formelle de code binaire, disponible en LGPL (<http://binsec.gforge.inria.fr>). La plate-forme repose sur une représentation intermédiaire (IR), les DBA ¹ [3, 5], bien adaptée aux analyses formelles (peu d'opérateurs, pas d'effets de bord).

La plate-forme propose quatre modules principaux : (1) un front-end incluant plusieurs méthodes de désassemblage syntaxique et une simplification de la représentation intermédiaire générée [5], (2) un simulateur de programmes DBA enrichi d'un modèle mémoire à régions bas-niveau [2], (3) un module d'analyse statique par interprétation abstraite, et (4) un module d'exécution symbolique dynamique (DSE) [4].

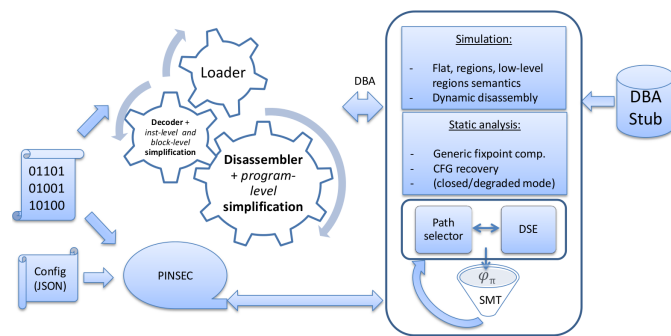


FIGURE 1 – Plate-forme Binsec

Front-end et simplification des DBA. Chaque instruction machine est traduite vers un bloc d'instructions DBA sémantiquement équivalent. Les analyses sémantiques se basent ensuite sur ce code DBA. BINSEC fournit actuellement un parseur pour le format de binaire ELF (PE en cours) et un décodeur pour l'architecture x86-32 (support de 380 instructions de base, ainsi que 100 instructions SIMD hors opérations flottantes). Grâce à un mécanisme de stubs formels, il est également possible de remplacer ou d'insérer un bloc DBA à une adresse spécifiée. Ceci est utile si on veut abstraire une partie du code (typiquement, une fonction de bibliothèque standard) ou si le code n'est pas disponible à une adresse donnée (analyse de parties de programmes avant l'édition de liens). Des algorithmes basiques de reconstruction du graphe de flot de contrôle (*linear sweep*, *recursive traversal*) sont implémentés avec la possibilité d'interagir avec les autres modules (notamment exécution symbolique et analyse statique) pour découvrir les cibles des sauts dynamiques. Un moteur de simplifications permet d'éliminer certaines opérations inutiles, typiquement les mises à jour systématiques de flags (75% de suppression dans nos expériences [5]).

Simulation. Le module de simulation supporte les nouvelles extensions des DBA [5], Il permet de simuler l'exécution des programmes désassemblés et la génération aléatoire

*La plate-forme BINSEC est un effort commun CEA - IRISA - LORIA - Université Grenoble Alpes, financé par l'ANR (bourse ANR-12-INSE-0002). Les résultats décrits ici ont été présentés à TACAS 2015 [5] et SANER 2016 [4]. Ils ont été encadrés par Sébastien Bardin, Jean-Yves Marion, Laurent Mounier et Marie-Laure Potet.

1. Dynamic Bit-vector Automata

de tests (*fuzzing*). Trois modèles mémoire peuvent être sélectionnés : le modèle plat (la mémoire comme tableau d'octets), le modèle à régions standard – à la CompCert [6], ou un modèle à régions bas-niveau [2]. L'adoption du modèle à région bas-niveau est motivée par sa capacité à permettre à la fois de profiter de l'abstraction offerte par le modèle à régions standard et de donner une sémantique définie aux programmes bas niveau.

Analyse statique. L'analyse statique permet d'inférer, à partir du code d'un programme et sans l'exécuter, des propriétés vraies pour toutes les exécutions de ce programme. L'analyse statique de programmes binaires permet d'analyser des exécutables dont on ne dispose pas du code source, mais aussi de compléter les analyses appliquées au niveau source avec des informations plus précises, car proches de la plateforme cible. Le module d'analyse statique de BINSEC offre un calcul générique de point fixe abstrait, paramétré par un domaine abstrait, afin de prototyper rapidement des analyses niveau binaire. L'implémentation actuelle permet une interaction avec le désassemblage syntaxique pour la résolution sûre des cibles des sauts dynamiques, ainsi qu'un mode d'analyse non-sûre (contrôlé par l'utilisateur) permettant par exemple de limiter les cibles des sauts dynamiques. L'interface est pour le moment limitée à des domaines non-relationnels mais en cours d'extension pour profiter des domaines relationnels.

Exécution symbolique. BINSEC/SE [4] est un moteur d'exécution symbolique dynamique, architecturé autour des trois composants principaux : le traceur, le noyau d'exécution symbolique et le sélecteur de chemins. Le traceur, appelé PINSEC, est articulé autour du framework Pin et permet de générer une trace pour des exécutables Linux et Windows. Un système interactif permet d'échanger des messages avec le noyau DSE afin de faciliter l'instrumentation. Le noyau d'exécution symbolique permet de produire le prédicat de chemin (ensemble de contraintes sur les entrées du programme) d'une trace à partir de la représentation intermédiaire des DBA, puis d'en tester la satisfiabilité via un solveur SMT. Enfin, le sélecteur de chemins permet de spécifier l'ordre dans lequel explorer les chemins du programme, suivant la méthode décrite dans [1].

Expérimentations. Le moteur d'exécution symbolique a été testé sur des programmes de type coreutils (Unix) et malware (Windows), ainsi que sur des challenges de type "crackme". Le moteur d'analyse statique a été testé sur des programmes issus des benchmarks Juliet/samate du NIST.

Références

- [1] Bardin, S., Baufreton, P., Cornuet, N., Herrmann, P., Labbé, S. : Binary-level Testing of Embedded Programs. In : QSIC 2013. IEEE, Los Alamitos (2013)
- [2] Blazy S., Besson F., Wilke P. : A Precise and Abstract Memory Model for C using Symbolic Values. In : APLAS 2014. Springer, Heidelberg (2014)
- [3] Bardin S. , Herrmann P., Leroux J., Ly O., Tabary R., Vincent A. : The BINCOA Framwork for Binary Code Analysis. In : CAV 2011. Springer, Heidelberg (2011)
- [4] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In : SANER '16. IEEE, 2016
- [5] A. Djoudi and S. Bardin. BINSEC : Binary code analysis with low-level regions. In : TACAS '15. Springer, 2015
- [6] Leroy, X., Appel, A.W., Blazy, S., Stewart, G. : The CompCert memory model. In : Program Logics for Certified Compilers. Cambridge University Press (2014)

Un processus de développement Event-B pour des applications distribuées

Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix et Mamoun Filali
Université de Sfax
IRIT CNRS UPS Université de Toulouse

Résumé

Nous présentons une méthodologie basée sur le raffinement manuel et automatique pour le développement d'applications distribuées correctes par construction. À partir d'un modèle Event-B, le processus étudié définit des étapes successives pour décomposer et ordonnancer les calculs associés aux événements et distribuer le code sur des composants. La spécification de ces deux étapes est faite au travers de deux langages dédiés. Enfin, une implémentation distribuée en BIP est générée. La correction du processus repose sur la correction des raffinements et de la traduction vers le code cible BIP.

1 Introduction

Les systèmes distribués demeurent difficiles à concevoir, construire et faire évoluer. Ceci est lié à la concurrence et au non déterminisme. Plusieurs formalismes tels que les algèbres de processus, ASM et TLA^+ sont utilisés pour modéliser et raisonner sur la correction des systèmes distribués. Un spécifieur utilisant ces formalismes est censé modéliser des mécanismes de bas niveau tels que canaux de communication et ordonnanceurs.

Dans cet article, nous proposons un support outillé d'aide à la conception système en utilisant une méthodologie basée sur des raffinements prouvés. Les systèmes considérés sont vus comme un ensemble d'acteurs en interaction. Les premiers raffinements fournissent une vue centralisée du système. Ils sont construits en prenant en compte progressivement les exigences du système. Ces raffinements sont exprimés à l'aide de machines abstraites décrites en Event-B [2]. Ensuite, nous proposons des schémas de raffinements destinés à prendre en compte la nature distribuée du système étudié. Ces schémas de raffinements sont guidés par l'utilisateur et les machines Event-B associées sont automatiquement générées. En conséquence, on obtient un ensemble de machines qui interagissent, dont la composition est prouvée correcte et conforme avec le niveau abstrait. Le système peut ensuite être exécuté sur une plateforme distribuée via une traduction en BIP [3]. Remarquons que notre objectif n'est pas d'automatiser complètement le processus de distribution, mais de l'assister. Tout en restant modeste, la différence est

similaire à celle entre un vérificateur de modèle où la preuve d'un jugement est automatique et un assistant de preuve où l'utilisateur doit composer des stratégies de base afin de résoudre son but. De même qu'un assistant de preuve aide à construire la preuve d'un but, notre objectif est d'aider à l'élaboration par raffinements d'un modèle distribué.

Event-B et BIP admettent une sémantique basée sur les systèmes de transitions étiquetées. Ceci favorise leur couplage. Event-B est utilisée pour la spécification et la décomposition formelles des systèmes distribués. BIP est utilisée pour leur implantation et leur déploiement sur une plateforme à mémoire répartie. Le passage d'Event-B vers BIP s'appuie sur des raffinements manuels et automatiques. Les raffinements manuels horizontaux permettent l'introduction progressive de propriétés du futur système. Les raffinements manuels verticaux permettent l'obtention de modèles Event-B traduisibles vers BIP : détermination d'actions et concrétisation des données. Quant aux raffinements automatiques, nous en avons élaboré deux sortes : l'une est appelée *fragmentation* (voir section 3) et l'autre *distribution* (voir section 4). Ces deux sortes de raffinement sont guidées par le spécifieur via deux langages dédiés (DSL). Cette démarche peut être comparée aux travaux portant sur la génération automatique de code source à partir de spécifications formelles. [4] propose un générateur de code efficace séquentiel en utilisant un sous-ensemble B0 impératif de B. Le raffinement automatique de machines B est également possible grâce à l'outil Bart [5]. Cependant, il ne concerne pas les modèles Event-B et est destiné aux raffinements de modèles B vers le sous-ensemble B0. Concernant Event-B, plusieurs générateurs de code source ont été proposés [6, 8]. Il est possible de générer du code Ada parallèle. Cette dernière cible est cependant plus restrictive que BIP puisqu'elle n'offre que de la synchronisation binaire.

Le processus de développement correct par construction de systèmes distribués préconisé combine les raffinements manuels et automatiques. Il se termine par la génération de code BIP. Dans la suite, après une introduction aux formalismes Event-B et BIP, nous présentons les deux transformations par raffinement (la fragmentation et la distribution), puis la génération de code BIP. Ces étapes seront illustrées sur l'exemple de l'hôtel¹.

Le passage d'Event-B vers BIP se fait en trois étapes : fragmentation, distribution et génération de code (voir Figure 1).

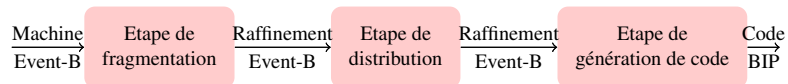


FIGURE 1 – Etapes du processus

1. Le texte complet est accessible via le lien <https://dl.dropboxusercontent.com/u/98832434/hotelrefinements.html>.

2 Event-B et BIP

2.1 Event-B

Event-B permet de décrire le comportement d'un système par une chaîne de raffinements de machines abstraites. Une machine contient des événements, éventuellement paramétrés, faisant évoluer l'état de la machine tout en préservant un invariant. Le non-déterminisme s'exprime dans le corps des événements grâce aux actions ($:\in$ et $:\!$) ou via leurs paramètres introduits dans la clause `ANY` et contraints par les propriétés spécifiées dans la clause `WHERE`. Dans une optique de décomposition formelle d'une spécification centralisée en plusieurs composants Event-B, les contraintes d'un paramètre d'un événement partagé doivent être résolues localement. Nous adressons ce problème ultérieurement.

Récemment, deux opérations ont été introduites dans Event-B : la composition et la décomposition [7]. Elles ont pour but d'introduire la notion de composant dans la démarche de raffinement. Ces deux opérations sont reliées par la propriété suivante : la composition des sous-composants produits par décomposition d'un modèle doit raffiner ce modèle. Deux variantes ont été identifiées : par variables partagées et par événements partagés. La première est adaptée aux systèmes à mémoire partagée, tandis que la deuxième est plutôt adaptée aux systèmes distribués. Dans ce travail, nous nous intéressons à la composition/décomposition par événements partagés. La plateforme Rodin offre un outil interactif [7] sous forme d'un plugin permettant la composition/décomposition par événements partagés. Dans la section 4, nous situerons notre plugin de distribution vis-à-vis de ce plugin.

2.2 BIP

Le modèle de composant BIP comporte trois couches : *Behavior*, *Interaction* et *Priority*. La couche *Behavior* permet de décrire les aspects **comportementaux** des composants atomiques tandis que les couches *Interaction* et *Priority* décrivent les aspects **architecturaux** d'un système. Les contraintes de synchronisation entre les composants BIP sont exprimées par des interactions regroupées au sein de la construction `connector` de BIP tandis que les contraintes d'ordonnement des interactions sont exprimées grâce au concept *Priority* de BIP. En BIP, un composant atomique englobe des données, des ports et un comportement. Les données (`data`) sont typées. Les ports (`port`) donnent accès à des données et constituent l'**interface** du composant. Le comportement est un ensemble de transitions définies par un port, une garde et une fonction de mise à jour des données.

3 La fragmentation

Cette étape, dite de fragmentation, prend en entrée un modèle Event-B quelconque (voir figure 1). Elle a pour but de réduire le non-déterminisme lié au calcul des paramètres locaux d'un événement. L'ordre de calcul des paramètres est

décrit par le spécifieur à l'aide d'un DSL (voir listing 1). En se basant sur cette description et le modèle Event-B abstrait, l'étape de fragmentation génère un modèle Event-B. Ce raffinement automatique s'appuie sur des règles simples assurant l'obtention d'un raffinement : introduction d'un nouvel événement convergent, raffinement d'un événement par plusieurs (one to many), introduction d'une nouvelle variable, renforcement de garde, renforcement d'invariant et instanciation d'un paramètre local d'un événement par une variable d'état.

Plugin de fragmentation. La transformation de fragmentation est implémentée par un plugin Rodin. Elle génère un raffinement de la machine en question à partir d'une *spécification de la fragmentation*. Celle-ci comporte des déclarations donnant pour un paramètre p d'un événement ev les paramètres p_i dont il dépend et les gardes g_i le spécifiant. Une valeur initiale v est requise pour des besoins de typage.

event ev **when** $p_1 \dots p_n$ **parameter** p **init** v **with** $g_1 \dots g_m$

Exemple d'illustration. Dans l'exemple de l'hôtel, l'événement `register` a trois paramètres : g, r, c . Nous spécifions que le paramètre client g doit être calculé en premier. Une chambre r est alors choisie et la carte d'accès c est enfin générée. La fragmentation de l'événement `register` est ainsi spécifiée :

```

splitting hotel_splitted refines hotel
events
  event register
    parameter  $g$  init  $g_0$  with  $tg$  //  $tg$  ne dépend pas de  $r, c$ 
    when  $g$  parameter  $r$  init  $r_0$  with  $tr$   $g_1$  // déclenché après le calcul de  $g$ 
    when  $g$   $r$  parameter  $c$  init  $c_0$  with  $tc$   $g_2$   $g_3$  // déclenché après  $g, r$ 
  end
    
```

Listing 1 – Spécification de la fragmentation

La fragmentation produit une machine dont le schéma général est le suivant :

```

machine generated refines input_machine
variables
   $ev\_p$   $ev\_p\_computed$  // témoin et statut pour param.  $p$  de l'événement  $ev$ 
invariants
  @ $ev\_gi$   $ev\_p\_computed = TRUE \Rightarrow gi$  // ou  $p$  est remplacé par  $ev\_p$ 
variant // compteur des paramètres restant à calculer
  {FALSE  $\mapsto$  1, TRUE  $\mapsto$  0}( $ev\_p\_computed$ ) + ...
events
  event INITIALISATION extends INITIALISATION
  then
    @ $ev\_p$   $ev\_p := v$ 
    @ $ev\_p\_comp$   $ev\_p\_computed := FALSE$ 
  end
  convergent event compute_ $ev\_p$  // calcule le paramètre  $p$  de  $ev$ 
  any  $p$  where
    @ $gi$   $gi$  // garde spécifiant  $p$ 
    @ $pi$   $ev\_pi\_computed = TRUE$  // paramètres dont  $p$  dépend ont été calculés
    @ $p$   $ev\_p\_computed = FALSE$  //  $p$  reste à calculer
  then
    @ $a$   $ev\_p := p$ 
    @ $computed$   $ev\_p\_computed := TRUE$  // décroissance du variant
    
```

```

end
event ev refines ev
when
  @p_comp ev_p_computed = TRUE
with
  @p p = ev_p // le paramètre p de l'événement hérité est raffiné en ev_p
then
  // remplacer p par ev_p dans les actions de l'événement raffiné
end
end

```

Listing 2 – Machine générée pour le raffinement de fragmentation

En fait, cette machine implante les contraintes d'ordonnancement par l'introduction d'une variable booléenne, *computed state*, par paramètre. L'invariant de la machine est étendu par les propriétés des variables introduites. Si une variable a été calculée (*ev_p_computed = TRUE*), sa spécification, donnée par sa garde *gi*, est alors satisfaite. Lorsque tous les paramètres d'un événement ont été calculés, l'événement en question peut être activé. Enfin, la progression du calcul des paramètres requis est assurée par un variant défini comme le nombre de paramètres restant à calculer.

4 La distribution

Après l'étape de fragmentation (voir figure 1), l'étape de gestion de la distribution prend en compte les particularités de l'architecture cible. Il s'agit ici de composants BIP synchronisés par des connecteurs n-aires et supposés ne réaliser que des transferts de données (pas de traitement interne dans un connecteur). À l'instar de l'étape de fragmentation, l'étape de distribution prend en entrée un modèle abstrait et une spécification de distribution. Celle-ci est décrite par le spécifieur à l'aide d'un DSL. Elle introduit la configuration retenue : les noms des sous-composants. De même, elle répartit les variables et éventuellement les gardes sur les sous-composants. Les variables référencées par une garde doivent être localisées sur un même sous-composant. Autrement, des copies *faiblement cohérentes* de ses variables seront automatiquement ajoutées. Elles sont mises à jour par des événements convergents ordonnancés avant l'événement accédant aux copies. De même, une action est réalisée par le composant (supposé unique) sur lequel les variables modifiées sont localisées. Les variables lues par une action peuvent être distantes. Les valeurs de ces variables seront transmises lors de la synchronisation. Notons que la gestion des copies faiblement cohérentes et des variables distantes sont spécifiques à notre proposition. Ceci peut être vu comme une extension du plugin de *décomposition par événement partagé* [7]. Pour y parvenir, nous introduisons une phase de pré-traitement.

La phase de pré-traitement. Cette phase prend en entrée une machine abstraite, des identifiants de sous-composants et la répartition de ces variables sur des sous-composants. Elle produit un raffinement introduisant des copies des variables dis-

tantes lues par les gardes, les événements anticipés mettant à jour ces variables, et des paramètres recevant les valeurs des variables distantes lues par les actions. Nous supposons dans ce qui suit que les variables v_i sont localisées sur les composants C_i . Les événements convergents sont partagés par les origines (C_i) et destinations (C_j) des variables lues par les gardes. Les raffinements des événements hérités sont partagés par les origines (C_i) des copies (sur C_j) et par les composants (C_k). Ces derniers comportent les variables directement lues par les actions.

```

machine generated refines input_machine
variables
  vi // variables héritées, sur Ci
  Cj_vi // copie sur Cj de vi (utilisée par une garde sur Cj)
  vi_fresh // true si vi a été copié, sur Ci
invariants
  @Cj_vi_f vi_fresh = TRUE ⇒ Cj_vi = vi // la copie est à jour
variant
  {FALSE ↦ 1, TRUE ↦ 0}(vi_fresh) + ...
events
  convergent event share_vi // partagé par Ci et Cj
  any local_vi where
    @g vi_fresh = FALSE // sur Ci
    @l local_vi = vi // sur Ci
  then
    @to_Cj Cj_vi := local_vi // sur Cj
    @done vi_fresh := TRUE // sur Ci
  end
  event ev refines ev // partagé par Ci, Cj, Ck
  any local_vk where
    @vj_access local_vk = vk // sur Ck, accès direct par les actions
    @vi_fresh vi_fresh = TRUE // sur Ci, la copie sur Cj est à jour
    @g [vi := Cj_vi]g // sur Cj, garde héritée avec accès aux copies
  then
    @a vj := [vi := Cj_vi; vk := local_vk]e // sur Cj, synchro avec Ck
  end
end
    
```

Listing 3 – Pré-traitement

La phase de projection. Cette phase construit une machine pour chaque sous-composant, et reprend le plugin de décomposition par événement partagé [7]. Les sites de projection des gardes et actions sont indiqués dans le listing 3. Notons que les invariants faisant référence à des variables distantes sont naturellement écartés.

5 La génération de code BIP

L'étape de génération de code BIP prend en entrée les composants Event-B issus de l'étape de distribution. Elle génère les éléments suivants :

Types de port. Pour chaque événement de chaque sous-composant nous générons un type de port. Les variables associées à un type de port donné sont issues des variables référencées par l'événement correspondant.

Types de connecteur. Pour chaque événement faisant référence à des variables de plusieurs composants, nous générons un type de connecteur.

Squelette de sous-composants. Pour chaque sous-composant, nous générons un composant atomique BIP. Il comporte les variables du sous-composant ainsi que les variables distantes appartenant à d'autres sous-composants, les instances des types de ports associés à ce sous-composant et, pour chaque événement, une transition synchronisée sur l'instance du port correspondant.

Le composant composite. Il regroupe une instance de chaque sous-composant et connecteur. Chaque instance de connecteur admet une instance de port appartenant aux instances de sous-composants définies précédemment.

6 Conclusion

Nous avons présenté une méthode de développement de systèmes distribués corrects par construction. À partir d'une machine Event-B représentant un modèle centralisé, nous appliquons deux types de raffinements automatiques (la fragmentation et la distribution) dont les paramètres sont déclarés par deux langages dédiés. Enfin, un modèle BIP exécutable sur une architecture répartie est produit. La correction de cette méthode repose sur la correction des étapes de raffinement et de la traduction finale en code BIP.

Nous envisageons maintenant d'améliorer l'outillage de ce processus. Les transformations de modèles Event-B sont réalisées via `xtext` et `xtend` [1]. L'étape suivante sera la finalisation du générateur BIP.

Références

- [1] Language engineering for everyone ! <https://eclipse.org/Xtext>. Acc : 16-01-06.
- [2] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.
- [3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3) :41–48, 2011.
- [4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable translator of B specifications to embedded c programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCIS*, pages 94–113. Springer, 2003.
- [5] Clearsy. Bart (b automatic refinement tool). http://tools.clearsy.com/wp-content/uploads/sites/8/resources/BART_GUI_User_Manual.pdf.
- [6] A. Edmunds and M. Butler. Tasking Event-B : An extension to Event-B for generating concurrent code. Event Dates : 2nd April 2011, February 2011.
- [7] R. Silva and M. Butler. Shared event composition/decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates : 29 November - 1 December 2010.
- [8] N. K. Singh. Eb2all : An automatic code generation tool. In *Using Event-B for Critical Device Software Systems*, pages 105–141. Springer London, 2013.

Projet ANR BINSEC

analyse formelle de code binaire pour la sécurité

Sébastien Bardin (coordinateur)

prenom.nom@cea.fr

Airbus Group, CEA LIST, IRISA, LORIA, Université Grenoble Alpes

BINSEC : BINary-code analysis for SECurity

- Projet ANR INS, appel 2012
- Axes 1 (sécurité) et 2 (génie logiciel)
- Projet de recherche fondamentale sur 4 ans (2013-2017)
- Sujet : Techniques formelles d'analyse de sécurité au niveau binaire

Contexte, objectifs et défis. L'objectif général du projet BINSEC est de combler le fossé actuel entre le succès des méthodes formelles pour la sûreté logicielle (niveau source ou modèle) et les besoins des analyses de sécurité niveau binaire (analyse de vulnérabilité, analyse de malware), pour l'instant peu outillées (approches syntaxiques). Pour les malware, nous nous concentrons sur les problèmes de camouflage et d'obscurcissement (*obfuscation*). Pour les vulnérabilités, nous nous concentrons sur la découverte de bugs et sur la distinction entre bugs bénins et bugs exploitables.

Les défis principaux viennent de la structure même du code binaire (données et contrôle bas niveau), de la complexité des architectures modernes, du manque de compréhension claire de certaines propriétés considérées (obscurcissement, exploitabilité), de la présence d'un attaquant et enfin de la volonté de considérer des programmes non critiques (robustesse et passage à l'échelle).

Bien qu'a priori distincts, les domaines que nous attaquons partagent des problèmes communs, par exemple le manque de sémantique claire et unifiée, la reconstruction précise du flot de contrôle (même avec obscurcissement), le besoin de raisonnement bas niveau précis, etc. Le projet est ainsi architecturé autour de quelques problématiques générales (sémantique, analyses de base) sur lesquelles se basent les travaux applicatifs. Les résultats sont en partie intégrés dans la plate-forme open-source BINSEC.

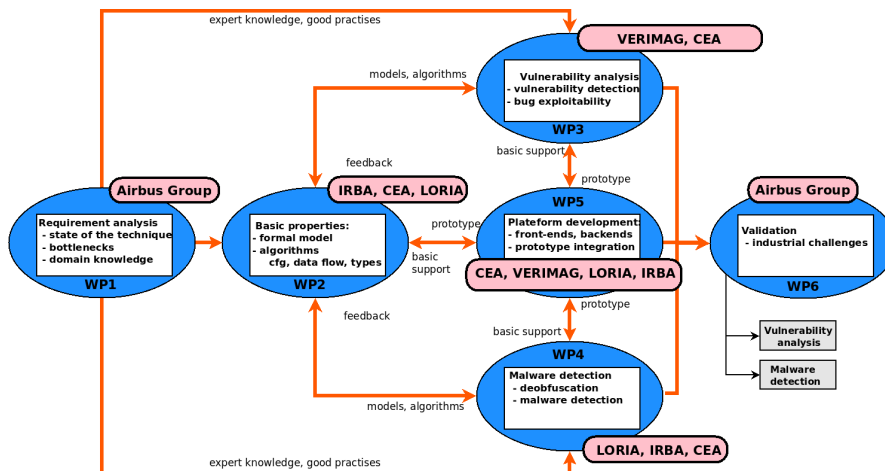


FIGURE 1 – Aperçu du projet

Quelques résultats. Les résultats du projet jusqu'à ce jour incluent : une représentation intermédiaire formelle et concise pour l'analyse de code binaire [5], basée entre autre sur un nouveau modèle mémoire à régions "bas niveau" [1, 2] raffinant le modèle usuel "à la CompCert" [7]; une technique originale et un prototype de détection de "use-after-free" sur code exécutable [6]; une technique et un prototype de désassemblage de code obscurci par auto-modification et chevauchement d'instruction [3]; une plate-forme open-source d'analyse de code binaire [5] proposant des moteurs originaux d'analyse statique (reconstruction sûre de graphe de contrôle) et d'exécution symbolique [4] (exploration partielle précise). La plate-forme sera disponible à partir de juin 2016 (<http://binsec.gforge.inria.fr>) .

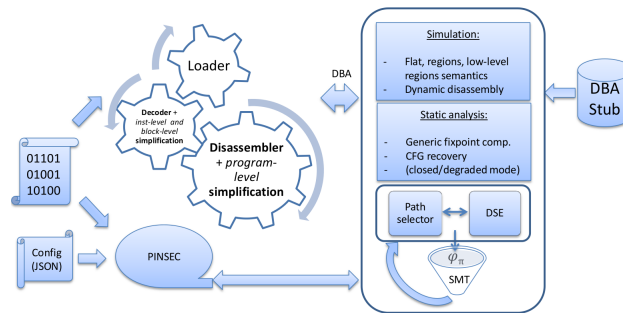


FIGURE 2 – Plate-forme Binsec

Participants. Sébastien Bardin, Frédéric Besson, Sandrine Blazy, Guillaume Bonfante, Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Colas Le Guernic, Jean-Yves Marion, Laurent Mounier, Marie-Laure Potet, Than Dinh Ta, Franck Védrine, Pierre Wilke, Sara Zennou.

Références

- [1] Blazy S., Besson F., Wilke P. : A Precise and Abstract Memory Model for C using Symbolic Values. In : APLAS 2014. Springer, Heidelberg (2014)
- [2] Besson F., Blazy S., Wilke P. : A Concrete Memory Model for CompCert. In : ITP 2015. Springer, Heidelberg (2015)
- [3] G. Bonfante, J. Fernandez, JY. Marion, B. Rouxel, F. Sabatier, A. Thierry : Co-Disasm : Concatic disassembly of self-modifying binaries with overlapping. In : CCS 2015. ACM (2015)
- [4] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In : SANER '16. IEEE, 2016
- [5] A. Djoudi and S. Bardin. BINSEC : Binary code analysis with low-level regions. In : TACAS '15. Springer, 2015
- [6] J. Feist, L. Mounier and M.L. Potet : Statically detecting Use-After-Free on Binary Code. Journal of Computer Virology and Hacking Techniques, 2014.
- [7] Leroy, X., Appel, A.W., Blazy, S., Stewart, G. : The CompCert memory model. In : Program Logics for Certified Compilers. Cambridge University Press (2014)

Combiner des diagrammes d'état étendus et la méthode B pour faciliter la validation de systèmes industriels

Thomas Fayolle
 LACL, Université Paris Est
 GRIL, Université de Sherbrooke
 Ikos Consulting, Levallois Perret

11 mars 2016

1 Introduction

Dans le cadre industriel, les systèmes sont développés par des ingénieurs systèmes spécialisés en partenariat avec des ingénieurs métier. Lorsque le système est critique, l'utilisation de méthodes formelles est fortement recommandée pour son développement. Les méthodes formelles utilisent des notations mathématiques parfois complexes, qui ne facilitent pas le dialogue entre ceux qui connaissent le fonctionnement du système et ceux qui connaissent le langage pour l'exprimer. Pour faciliter ce dialogue, une solution peut être d'utiliser un langage de spécification graphique.

Les ASTD [2](Algebraic State Transition Diagram) sont un langage de spécification qui combine les diagrammes d'état [5] et des opérateurs d'algèbre de processus [4]. Ils permettent de décrire graphiquement le comportement d'un système. Ils peuvent donc apporter une solution pour la validation de spécifications. Cependant, ils ne permettent de modéliser que l'ordonnancement des opérations. Pour décrire l'effet de ces opérations sur les données du système, nous proposons de les combiner au langage B.

2 Méthodologie

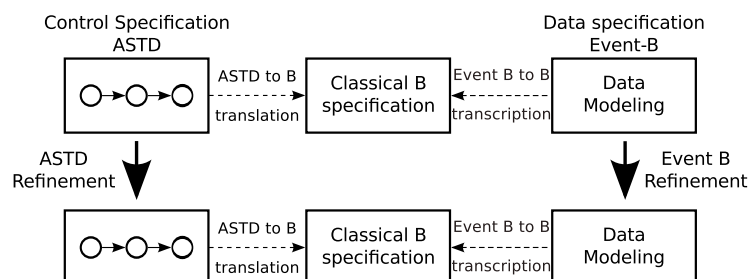


Figure 1: Méthodologie de la spécification

Notre méthode de spécification combine le langage ASTD et le langage Event-B. Cette méthode est résumée à la figure 1. Le système est modélisé en deux parties. Son comportement dynamique est décrit dans la première partie en utilisant les ASTD (spécification à gauche sur la figure 1). La deuxième partie en Event-B décrit les modifications apportées au modèle de données (spécification à droite sur la figure 1). Elle contient un événement pour chaque label de la spécification ASTD. Lorsque la transition est effectuée, le modèle de données est modifié en suivant la spécification de l'événement Event-B. La spécification en B classique (partie centrale de la figure 1) permet d'assurer la cohérence du système.

2.1 La spécification ASTD

La spécification ASTD modélise l'ordonnement des opérations en utilisant des notations graphiques des diagrammes d'état et des opérateurs des algèbres de processus. La spécification graphique facilite la validation tout en restant formelle. L'ajout des opérateurs des algèbres de processus permet de modéliser des systèmes dans lesquels plusieurs processus fonctionnent en parallèle, comme c'est le cas dans beaucoup de systèmes industriels.

2.2 La spécification Event-B

La spécification ASTD modélise l'ordre des opérations, tandis que la spécification Event-B modélise les effets de chaque opération sur le modèle de données. Elle contient un événement pour chaque label de transition de la spécification ASTD. Les invariants Event-B traduisent les propriétés statiques, notamment les propriétés de sûreté. Les outils de la plateforme RODIN ¹ permettent de générer et de prouver les obligations de preuve associées à la préservation des invariants. Au contraire du raffinement de la méthode B, le raffinement de la méthode Event-B permet l'ajout de nouveaux événements. Pour coller au raffinement des ASTD, c'est ce langage qui a été choisi pour la spécification de la partie donnée.

2.3 Spécification B

Pour un niveau de spécification, la modélisation en B classique contient deux machines. La première est une traduction de la spécification ASTD, la seconde est une transcription de la spécification Event-B.

Un outil effectue la traduction automatique de la spécification ASTD. Cette traduction est effectuée suivant les règles définies dans [6]. Le mécanisme de la traduction peut être résumé de la manière suivante. Les états des ASTD sont encodés par une variable d'état B. Une opération B est écrite pour chaque label de la spécification ASTD. La précondition de cette opération vérifie que les variables d'état sont bien dans l'état initial de la transition. La postcondition met à jour les variables d'état de manière à ce qu'elles correspondent à l'état final de la transition.

De plus, nous souhaitons que les variables du modèle de données soient modifiées lors de chaque transition. Puisqu'il n'est pas possible d'appeler un événement Event-B dans une spécification en B classique, la spécification Event-B est traduite en B. Les variables d'état et les invariants de typage sont conservés. Les événements sont réécrits en opération, la garde devient une précondition et la postcondition reste identique.

¹<http://event-b.org>

Regrouper les deux spécifications en une seule nous permet de vérifier la cohérence globale du système (c'est-à-dire un niveau de spécification horizontal sur la figure 1). En effet, les propriétés statiques de la spécification Event-B ne sont vérifiées que si les événements sont exécutés quand leurs gardes sont vraies. La génération des obligations de preuves de la méthode B oblige à vérifier que les préconditions des opérations appelées sont vraies au moment de l'appel. La preuve de ces obligations garantit que les événements du modèle de données sont toujours exécutés quand leur garde est vraie. Pour prouver ces obligations, il est nécessaire d'ajouter des invariants qui lient les variables du modèle de données aux états de l'ASTD.

2.4 Raffinement

Le raffinement du système est effectué en parallèle sur les deux modèles. La spécification de l'ordonnancement des opérations est effectuée en suivant la définition du raffinement pour les ASTD. Cette définition (donnée dans [3]) permet de garantir la préservation des traces. Le raffinement du modèle de données suit la définition du raffinement Event-B.

3 Conclusion

La méthode présentée dans cet article permet de modéliser graphiquement un système critique ce qui facilite la validation du système dans un domaine industriel. La méthode a été testée sur un exemple dans le cadre de la modélisation d'un système ferroviaire [1].

La modélisation de ce cas a permis de voir l'efficacité de la méthode. Cependant, la preuve de la cohérence globale du système s'est montrée fastidieuse. Des travaux sont en cours pour simplifier les règles automatiques de traduction et définir des règles et des outils de raffinement pour les ASTD.

References

- [1] T. Fayolle. Specifying a Train System Using ASTD and the B Method. Technical report, 2014. <http://www.lacl.fr/~tfayolle>.
- [2] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. Saint-Denis. Extending Statecharts with Process Algebra Operators. *Innovation in System Software Engineering*, Volume 4, Number 3:285–292, 2008.
- [3] M. Frappier, F. Gervais, R. Laleau, and J. Milhau. Refinement patterns for ASTDs. *Formal Aspects of Computing*, pages 1–23, 2012.
- [4] M. Frappier and R. St-Denis. EB3 : an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.
- [5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.
- [6] J. Milhau. *A formal integration of access control policies into information systems*. Theses, Université Paris-Est ; Université de Sherbrooke, December 2011.

Vers un développement formel non incrémental

Thi-Kim-Dung Pham^{1,3}, Catherine Dubois², Nicole Levy¹

1. Conservatoire National des Arts et Métiers, lab. Cedric, Paris

2. ENSIIE, lab. Samovar, Évry

3. University of Engineering and Technology, Vietnam National University, Hanoi

Résumé

Modularité, généricité, héritage sont des mécanismes qui facilitent le développement et la vérification formelle de logiciels corrects par construction en permettant de réutiliser des spécifications, du code et/ou des preuves. Cependant les lignes de produits exploitent d'autres techniques de réutilisation ou de modification graduelle. Les méthodes formelles permettant la production de code correct par construction (en B ou FoCaLiZe par exemple) ne sont pas bien adaptées à la variabilité telle qu'elle apparaît dans les lignes de produits. Nous proposons d'approcher ce problème par la définition d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits permettant de spécifier, implanter et prouver. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Cet article illustre par l'exemple une des constructions offertes par GFML ainsi que sa traduction en FoCaLiZe.

1 Introduction

Nous aimerions construire des programmes corrects par construction, c'est-à-dire qui répondent au cahier des charges, sont cohérents et complets. Il est effectivement possible de le faire à l'aide de certaines méthodes qui utilisent des langages formels tels que B [1] ou FoCaLiZe [7]. Il faut alors définir les propriétés que le programme doit vérifier et utiliser des outils de preuve pour prouver que l'on a bien suivi toutes les étapes de la méthode.

Un des principaux problèmes de ces méthodes réside dans la difficulté à les appliquer. On pourrait espérer que les méthodes de réutilisation facilitent le développement. Il existe plusieurs manières de réutiliser un programme. Par exemple, les méthodes orientées objets qui proposent de compléter ou de modifier, par héritage, une classe d'objets. Le langage FoCaLiZe possède l'héritage et permet ainsi de réutiliser des spécifications, des programmes et des preuves.

Mais ce n'est pas suffisant. On aimerait réutiliser une fonctionnalité d'un programme mais pour l'utiliser dans un contexte différent. Il existe des mécanismes permettant de décrire les propriétés d'un contexte, indispensables pour un programme, sous la forme de paramètres de ce programme. En FoCaLiZe, les composants s'appellent des espèces (*species*) et peuvent être paramétrés par des espèces représentant des objets ou des propriétés du contexte requis.

Mais ce n'est pas encore suffisant. L'utilisation de la paramétrisation se révèle assez difficile et restrictive.

En fait, pour réutiliser un programme (ou une partie d'un programme), il faut savoir comment il a été construit et connaître les décisions prises lors de son développement.

Le problème qui se pose est l'expression des savoirs-faire et leur réutilisation. La réalité, c'est qu'il est bien difficile de décrire ou de formaliser ce savoir-faire. Mais c'est un des avantages des approches lignes de produits [4] qui permettent de décrire la succession des décisions à prendre et d'associer à chaque décision une solution concrète sous la forme d'un morceau de code (*asset*). En général, les décisions prises sont constructives : il s'agit d'ajouter une information, une fonctionnalité, une propriété, etc. Dans ce cas, la composition des morceaux peut être définie simplement.

Mais qu'en est-il lorsque la décision prise à une étape est de modifier l'étape précédente et non seulement de rajouter quelque chose ? Comment formaliser des méthodes de réutilisation non incrémentales ?

Il existe des approches telle que celle de Thomas Thüm [13, 12] qui utilise les lignes de produits pour développer des programmes Java selon une approche par contrats à l'aide de pré et post-conditions. Il utilise le langage JML pour écrire les contrats et les traduit ensuite en Coq pour les vérifier. C'est de là qu'est venue l'idée d'introduire un langage dédié pour décrire la construction des modifications à apporter au code.

Notre proposition est de faciliter de telles étapes de développement, permettant de modifier une étape précédente, c'est-à-dire de modifier le code préalablement défini, et prouvé. L'idée est d'introduire un langage d'expression de ces modifications et un compilateur réalisant la modification à opérer et générant le code. Le langage que nous proposons s'appelle GFML. Couplée à une approche lignes de produits, notre démarche permet de décrire des savoirs-faire qui ne sont pas purement incrémentaux mais peuvent passer par des étapes de remise en cause de développements préalables, tout en bénéficiant de l'approche de développement de code correct par construction.

Nous avons choisi d'illustrer notre démarche dans le cadre proposé par FoCaLiZe. En effet, il permet de définir à la fois spécifications, code et preuves, donc tous les artefacts associés à une caractéristique d'une ligne de produits. D'autre part, FoCaLiZe admet pour seul mécanisme de raffinement, l'enrichissement par héritage, ce qui laisse beaucoup de liberté.

La suite de cet article est structurée de la manière suivante. Dans un premier temps nous allons étudier les constructions existantes en FoCaLiZe qui permettent de suivre un développement formel incrémental. Nous signalerons aussi les limites de ce modèle. Nous montrons à l'aide d'un exemple de compte bancaire un exemple de modification souhaitée, à savoir le renforcement d'une fonctionnalité tout en conservant au maximum les preuves déjà faites. Cette modification peut être exprimée en FoCaLiZe à l'aide de l'utilisation conjointe de l'héritage et de la paramétrisation. Nous présentons ensuite l'expression de cette modification dans notre langage GFML et sa traduction en FoCaLiZe. Un avantage de cette approche est le fait qu'elle s'adapte très bien aux lignes de produits puisque la composition des morceaux associés à chaque étape entraîne une suite de modifications et non seulement des enrichissements. L'approche générale est présentée dans [8], nous détaillons ici, sur un exemple, une des constructions offertes par GFML et sa compilation en FoCaLiZe.

2 Le modèle FoCaLiZe

L'environnement FoCaLiZe (anciennement FoCal) [7, 6] permet le développement de programmes certifiés pas à pas, de la spécification à l'implantation. Cet environnement propose un langage également nommé FoCaLiZe et des outils d'analyse de code,

de preuve (Zenon) et des compilateurs vers Ocaml, Coq et Dedukti [5]. Le langage FoCaLiZe permet d'écrire des propriétés en logique du premier ordre, des signatures et des fonctions dans un style fonctionnel et enfin des preuves dans un style déclaratif. Ce langage offre également des mécanismes inspirés par la programmation orientée objets, comme l'héritage, la liaison retardée et la redéfinition afin de faciliter la modularisation et la réutilisation. FoCaLiZe permet d'hériter de spécifications, de code mais aussi de preuves [9]. Un mécanisme de paramétrisation vient compléter l'ensemble.

Les constructions de FoCaLiZe [3] permettent d'ajouter à une espèce, une signature, une définition, une propriété, une preuve, de définir une représentation concrète pour les objets manipulés, de redéfinir une fonction. Dans ce dernier cas, le calcul des dépendances fait par le compilateur détermine si les preuves déjà effectuées concernant la fonction redéfinie doivent être refaites ou non. Ces primitives assurent une construction incrémentale des spécifications, des programmes et des preuves. Dans le modèle actuel de FoCaLiZe, par héritage, les spécifications d'une fonction redéfinie sont héritées, elles peuvent être complétées par de nouvelles propriétés mais elles ne peuvent pas être *redéfinies* (en restreignant par exemple la précondition comme dans la section suivante), voire supprimées comme on le désirerait par exemple pour développer les fonctionnalités du logiciel en mode dégradé. La redéfinition mise en œuvre dans FoCaLiZe impose également de conserver l'arité d'une fonction, il n'y a donc pas de possibilité de surcharge. Enfin, la représentation d'un type (mot-clé `representation` dans les exemples de la section 3), une fois définie dans un composant *C*, ne peut être redéfinie dans les composants qui héritent de *C*. Ceci conduit à un choix tardif de cette représentation. Mais dans le cadre d'un développement non incrémental, on peut être amené à modifier la représentation choisie, par exemple en ajoutant des attributs à la manière des langages orientés objets. Une étude similaire a été proposée dans le cadre de la programmation orientée *feature* et de la programmation par contrats par Thüm et al. dans [11]. Relativement à la classification proposée dans cet article, le modèle actuel de FoCaLiZe, en cas d'héritage, met en œuvre le raffinement de spécifications par sous-typage. Avec l'exemple ci-dessous, nous nous rapprochons du raffinement dit explicite et du raffinement par surcharge de contrats.

3 Raffinement d'une propriété

Nous utilisons dans la suite l'exemple (inspiré de [13]) de la ligne de produits concernant la gestion de comptes bancaires représentée graphiquement (par son diagramme de caractéristiques ou *features*) à la figure 1. La feature racine, BankAccount (ou BA), définit le fonctionnement de base d'un compte : calculer le solde d'un compte, débiter et créditer un compte. Le débit n'est autorisé que si le compte reste créditeur d'une certaine somme (ici `over`). A ces fonctionnalités élémentaires peuvent être ajoutées, de manière optionnelle, les fonctionnalités dénotées par les features DailyLimit (DL), LowLimit (LL) et Currency. DL introduit une limitation pour les retraits. Dans la suite, seuls BA et DL sont utilisés.

A chaque feature de l'arbre est associé un fichier FoCaLiZe contenant des spécifications, du code et des preuves. Les artefacts associés au feature racine, BA, définissent le noyau minimal. Le fichier FoCaLiZe associé contient trois espèces données ci-dessous, `BA_spec0`, `BA_spec` et `BA_impl`, les deux premières contiennent la spécification et la dernière, l'implantation des fonctions ainsi que les preuves des propriétés énoncées dans les autres espèces. Comme nous le verrons plus tard, la séparation de la spécification en deux espèces facilitera la définition des autres features.

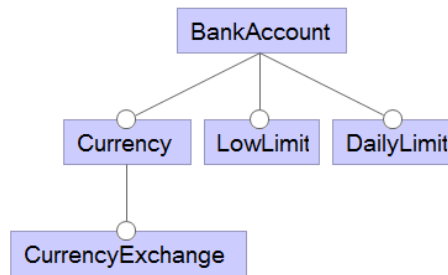


FIGURE 1 – Ligne de produits des comptes bancaires

```

species BA_spec0 =
signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;
property ba_over : all b: Self, balance(b) >= over;
end ;;

species BA_spec =
inherit BA_spec0;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
  balance(update(b, a)) = balance(b) + a;
end ;;

species BA_impl =
inherit BA_spec;
representation = int;
let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;
proof of ba_update = by definition of update, balance ;
proof of ba_over = assumed ;
end ;;
    
```

Dans `BA_spec0` sont introduites les signatures de deux fonctions, `balance` et `update` et une constante `over`. La fonction `balance` permet de consulter le solde du compte passé en argument, `update` permet de créditer/débiter le compte passé en argument, le résultat est le nouveau compte. La propriété/spécification `ba_over` est une propriété invariante, elle indique que tout compte doit avoir un solde supérieur ou égal à `over`. Cette propriété doit être vérifiée dans tout produit dérivé de cette ligne de produits. La deuxième espèce hérite des signatures et des propriétés de la première espèce et ajoute une nouvelle propriété spécifiant la mise à jour d'un compte, en particulier les conditions de succès de cette opération. Enfin, l'espèce `BA_impl` réalise la spécification. Elle précise la représentation des comptes bancaires. Ici, un compte est représenté par son solde, soit un entier. Et donc l'accès au solde d'un compte (la fonction `balance`) est l'identité. La preuve de la propriété `ba_update` énoncée précédemment est faite très simplement par le prouveur automatique Zenon [2]. Il suffit de lui indiquer d'utiliser les définitions des fonctions `update` et `balance`. La preuve de la propriété `ba_over` est, quant à elle, admise. En fait, elle ne peut être faite directement mais elle doit être remplacée par la preuve que cette propriété est préservée par

chacune des fonctions (dont le résultat a le type `Self`) de l'espèce [10].

La caractéristique DL permet de définir les fonctionnalités des comptes bancaires avec une limite de retrait. Ainsi on ajoute une nouvelle constante `limit_withdraw`. Cette extension demande une redéfinition de la fonction `update`, ce que permet le langage FoCaLiZe. En cas de retrait (argument négatif), la fonction va vérifier que l'on est bien en deçà de la limite de retrait. Cependant la propriété `ba_update` n'est plus vraie dans le contexte de DL. Nous avons ici une propriété plus restreinte. Nous définissons deux propriétés par renforcement des pré-conditions de `ba_update`, ce qui revient à spécifier séparément les actions créditer et débiter. Nous appelons ce schéma, un raffinement de propriété : il se limite à la redéfinition d'une propriété en ajoutant une prémisse. Du fait de cette modification de comportement, on peut définir la spécification de DL en héritant de `BA_spec0` mais on ne peut hériter de `BA_spec` car la propriété `ba_update` doit être adaptée. C'est la raison pour laquelle nous avons scindé en deux espèces la spécification de BA.

L'idée est de réutiliser le code existant et les preuves existantes. Le `super` tel qu'on le trouve en Java (ou `original` en programmation par aspects) n'existe pas en FoCaLiZe, il va donc falloir le simuler en utilisant héritage et paramétrisation.

La caractéristique DL est donc définie elle aussi en trois espèces : la première introduit la limite de retrait et hérite des spécifications réutilisables de BA, i.e. l'espèce `BA_spec0`, la deuxième définit les propriétés obtenues par raffinement et enfin la dernière contient les définitions des fonctions et les preuves des propriétés.

```

species DL_spec0 =
inherit BA_spec;
signature limit_withdraw : nat;
end ;;

species DL_spec =
inherit DL_spec0;
property ba_update_ref1 : all b:Self, all a: int,
  a >= 0 -> balance(b) + a >= over ->
  balance(update(b, a)) = balance(b) + a;

property ba_update_ref2 : all b:Self, all a: int,
  a < 0 -> (0 - a) <= limit_withdraw ->
  balance(b) + a >= over ->
  balance(update(b, a)) = balance(b) + a;
end ;;

species DL_impl (M is BA_impl) =
inherit DL_spec;
representation = M;
let balance(b) = M!balance (b) ;
let over = M!over ;

proof of ba_over =
  by definition of balance, over property M!ba_over ;
let update(b,a) =
  if a < 0 then
    if (0 - a) <= limit_withdraw then M!update (b, a) else b
  else M!update(b, a);

proof of ba_update_ref1 =

```

```

    by definition of update, balance, over
      property M!ba_update int_ge_le_eq ;

proof of ba_update_ref2 =
  by definition of update, balance, over
    property M!ba_update ;
end ;;

```

L'espèce `DL_impl` a été écrite de manière à réutiliser autant que possible le code et les preuves déjà faites dans le cadre de la caractéristique `BA`.

4 De GFML à FoCaLiZe

L'approche que nous proposons consiste à décrire à l'aide du langage GFML, les modifications à réaliser, c'est-à-dire les seules informations qui différencient `DL` de `BA`. Le module GFML correspondant à `DL`, décrit ci-dessous, permet de générer automatiquement les espèces de `DL`, soient `DL_spec0`, `DL_spec` et `DL_impl`. On remarquera que les preuves sont faites dans le format accepté par FoCaLiZe (cette partie est en fait recopiée textuellement par le compilateur GFML vers FoCaLiZe). On peut également noter que la syntaxe de GFML est largement inspirée de celle de FoCaLiZe.

```

fmodule DL from BA

signature limit_withdraw : nat;

property ba_update_ref1 refines BA!ba_update
extends premise (a >= 0);
property ba_update_ref2 refines BA!ba_update
extends a < 0 ^ -a <= limit_withdraw;

let update(b,a) =
if a < 0
  then if -a <= limit_withdraw then BA!update (b, a) else b
  else BA!update(b, a);

proof of ba_update_ref1 =
{* focalizeproof = by definition of update, balance, over
  property M!ba_update, int_ge_le_eq ; *}
proof of ba_update_ref2 =
{* focalizeproof = by definition of update, balance, over
  property M!ba_update ; *}
end ;;

```

La caractéristique racine `BA` est également décrite à l'aide d'un module GFML, détaillé ci-dessous, qui rassemble les signatures, définitions et preuves. La compilation de ce module en FoCaLiZe produit exactement les trois espèces `BA_spec0`, `BA_spec` et `BA_impl`.

```

fmodule BA

signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;

```

```

invariant property ba_over : all b: Self, balance(b) >= over;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;

representation = int;

let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;

proof of ba_update =
  { * focalizeproof = by definition of update, balance ; * }
proof of ba_over = assumed ;
end ;;

```

5 Conclusion

Dans cet article, nous avons présenté une approche de la construction non incrémentale de logiciels à l'aide d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits et permettant de spécifier, implanter et prouver les étapes de développement par ajout ou modification. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Nous avons illustré notre approche par l'exemple en ne présentant qu'une des constructions offertes par GFML et avons explicité sa traduction en FoCaLiZe.

GFML est un langage formel dont la sémantique est donnée par sa traduction en FoCaLiZe. Le compilateur de GFML vers FoCaLiZe est en cours de réalisation, en même temps que la définition de différents opérateurs de développement liés à l'approche lignes de produits. En plus de l'exemple donné d'ajout d'une pré-condition, citons l'ajout ou la modification d'un paramètre d'une fonction qui va avoir un impact partout où est appelée cette fonction. Les modifications calculées ont un impact sur les preuves déjà réalisées. L'opérateur décrit le travail à réaliser pour obtenir une nouvelle version prouvée du logiciel. Ainsi, l'ambition de GFML est de proposer aux concepteurs de lignes de produits, un langage permettant de décrire les modifications à apporter à une première version d'une caractéristique pour prendre en compte une nouvelle idée. Les constructions de GFML représentent des décisions et le compilateur les prend en compte pour générer le code correspondant.

Remerciements. Nous remercions François Pessaux pour son aide lors du développement du traducteur.

Références

- [1] J. Abrial. On constructing large software systems. In J. van Leeuwen, editor, *Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain*, volume A-12 of *IFIP Transactions*, pages 103–112. North-Holland, 1992.
- [2] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007*,

- Proceedings*, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.
- [3] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de doctorat, Université Paris 6, 2000.
 - [4] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.
 - [5] Dedukti. <http://dedukti.gforge.inria.fr/>.
 - [6] C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. In H. Loidl, editor, *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004*, volume 5 of *Trends in Functional Programming*, pages 33–48, 2006.
 - [7] FoCaLiZe. <http://focalize.inria.fr/>.
 - [8] T.-K.-Z. Pham, C. Dubois, and N. Levy. Towards correct-by-construction product variants of a software product line : Gfml, a formal language for feature modules. In *Proceedings 6th Workshop on Formal Methods and Analysis in SPL Engineering*, London, UK, 11 April 2015, volume 182 of *EPTCS*, pages 44–55, 2015.
 - [9] V. Prevosto and D. Doligez. Algorithms and proofs inheritance in the FOC language. *J. Autom. Reasoning*, 29(3-4) :337–363, 2002.
 - [10] R. Rioboo. Invariants for the focal language. *Ann. Math. Artif. Intell.*, 56(3-4) :273–296, 2009.
 - [11] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying design by contract to feature-oriented programming. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2012.
 - [12] T. Thüm. Product-line verification with feature-oriented contracts. In M. Pezzè and M. Harman, editors, *Int'l Symposium on Software Testing and Analysis, ISSA '13, Lugano, Switzerland, July 15-20, 2013*, pages 374–377. ACM, 2013.
 - [13] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST'11)*, pages 270–277. IEEE Computer Society, 2011.