Actes des neuvièmes journées nationales du

Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel

Laboratoire LIRMM – Université de Montpellier

13 au 16 juin 2017

Editeurs : Pierre-Etienne Moreau
Clémentine Nebut
Chouki Tibermacine

Table des matières

Préface

C'est avec grand plaisir que je vous accueille pour les neuvièmes journées nationales du GDR GPL (Génie de la Programmation et du Logiciel) au LIRMM et à l'Université de Montpellier.

Les missions principales du GDR GPL sont l'animation scientifique de la communauté et la promotion de nos disciplines, notamment en direction des jeunes chercheurs, mais également en direction des mondes académique et socio-économique. Cette animation scientifique est d'abord le fruit des efforts de nos groupes de travail, actions transverses et de l'Ecole des Jeunes Chercheurs en Programmation.

Le GDR GPL est maintenant dans sa neuvième année d'activité. Les journées nationales sont un temps fort de l'activité de notre GDR, l'occasion pour toute la communauté d'échanger et de s'enrichir des derniers travaux présentés. Plusieurs événements scientifiques sont co-localisés avec ces journées nationales : la 6ème édition de la Conférence en IngénieriE du Logiciel (CIEL 2017) ainsi que la 16ème édition de l'atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2017).

Ces journées sont une vitrine où chaque groupe de travail donne un aperçu de ses recherches. Un peu plus de vingt présentations ont ainsi été sélectionnées par les responsables des groupes de travail. Comme les années précédentes, nous avons demandé aux groupes de travail de nous proposer, en règle générale, des présentations qui avaient déjà fait l'objet d'une sélection dans une conférence nationale ou internationale ; ceci nous garantit la qualité du programme. Cette année, plusieurs sessions sont communes à plusieurs groupes de travail. C'est un signe intéressant d'ouverture, d'échange et de dynamisme.

Trois conférenciers nous ont fait l'honneur d'accepter notre invitation. Il s'agit d'Arnaud Gotlieb (Simula Research Laboratory), de Gérard Berry (Collège de France) et de Jordi Cabot (ICREA).

Le GDR GPL a à cœur de mettre à l'honneur les jeunes chercheurs. C'est pourquoi nous décernerons un prix de thèse pour la cinquième année consécutive. Nous aurons le plaisir de remettre le prix de thèse GPL à Jacques-Henri Jourdan pour sa thèse intitulée *Verasco : a Formally Verified C Static Analyzer*, ainsi que des accessits à Antoine Delignat-Lavaud pour sa thèse intitulée *On the security of authentification protocols for the Web*, à Maria Gomez pour sa thèse intitulée *Towards improving the quality of mobile apps by leveraging crowdsourced feedback* ainsi qu'à Jabier Martinez pour sa thèse intitulée *Mining Software Artifact Variants for Product Line Migration and Analysis*. Le jury chargé de sélectionner le lauréat a été présidé par Catherine Dubois, que je remercie tout particulièrement, ainsi que l'ensemble des membres du jury.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de ces journées nationales : les responsables de groupes de travail, les membres du comité de direction du GDR GPL et tout particulièrement le comité d'organisation de ces journées nationales présidé par Clémentine Nebut, Elisabeth Gréverie et Chouki Tibermacine. Je remercie chaleureusement l'ensemble des collègues Montpelliérains qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Pierre-Etienne Moreau
Directeur du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2017 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Pierre-Etienne Moreau (président), LORIA, Université de Lorraine

Yamine Ait Ameur, IRIT, ENSEEIHT
Nicolas Anquetil, CRIStAL, Université de Lille
Xavier Blanc, LaBRI, Université de Bordeaux, IUF
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Sandrine Blazy, IRISA, Université de Rennes
Florian Brandner, Télécom ParisTech
Eric Cariou, LIUPPA, Université de Pau et des pays de l'Adour
Khalil Drira, LAAS, CNRS
Catherine Dubois, Samovar, ENSIIE
Jean-Rémy Falleri, LABRI, ENSEIRB-MATMECA
Jean-Christophe Filliatre, LRI, CNRS
Aurélie Hurault, IRIT, ENSEEIHT
Laure Gonnord, LIP (ENS Lyon), Université Lyon 1
Akram Idani, LIG, Université Joseph Fourier
Claude Jard, AtlanSTIC, LINA, Université de Nantes
Nikolai Kosmatov, CEA-LIST
Régine Laleau, LACL, Université de Paris-Est Créteil
Yves Ledru, LIG, Université Joseph Fourier
Axel Legay, IRISA, Inria
Pascale Le Gall, MAS, Centrale Paris
Martin Monperrus, CRIStAL, Université de Lille
Sébastien Mosser, I3S, Université de Nice
Clémentine Nebut, LIRMM, Université de Montpellier
Flavio Oquendo, IRISA, Université de Rennes
Marc Pouzet, LIENS, IUF, ENS, Université Pierre et Marie Curie
Fabrice Rastello, INRIA, ENS Lyon
Olivier H. Roux, IRCCyN, Université de Nantes
Romain Rouvoy, CRIStAL, Université Lille 1
Camille Salinesi, CRI, Université Paris 1 Panthéon-Sorbone
Christelle Urtado, Mines d'Alès
Virginie Wiels, ONERA

Comité scientifique du GDR GPL

Franck Barbier (LIUPPA, Pau)
Pierre Casteran (LABRI, Bordeaux)
Pierre Cointe (LINA, Nantes)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Laurence Duchien (CRIStAL, Lille)
Stéphane Ducasse (INRIA, Lille)
Marie-Claude Gaudel (LRI, Orsay)
Jean-Louis Giavitto (IRCAMS, Paris)
Yann-Gaël Guéhéneuc (Polytech, Montréal)
Gaétan Hains (LACL, Créteil)
Nicolas Halbwachs (Verimag, Grenoble)
Olivier Hermant (Mines Paris)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christel Seguin (ONERA, Toulouse)

Comité d'organisation

Clémentine Nebut, Université de Montpellier
Elisabeth Gréverie, Cellule Communication, LIRMM
Chouki Tibermacine, Université de Montpellier

Abdelhak-Djamel Seriai, université de Montpellier
Christelle Urtado, école des mines d'Alès à Nîmes
Seza Adjoyan, LIRMM
Jessie Carbonnel, LIRMM
Anthony Ferrand, LIRMM, Pradeo
Yohan Frederick, LIRMM, Pradeo
Sahar Kallel, LIRMM
Alexandre Le Borgne, LGI2P
Frédéric Verdier, LIRMM, Acelys

Conférenciers invités

Testing Robotic Systems : A New Battlefield

Auteur : Arnaud Gotlieb (Simula Research Laboratory)

Résumé :
Industrial robotics is a field which evolves very fast. Nowadays, industrial robots are uncaged and can collaborate with humans and cooperate with other robots in order to perform more and more safety-critical tasks. However, understand how to control and test single- or multi-arm robots and their ability to interact safely with humans is challenging as their control systems have become very complex and their precise behaviors are not formally specified. My talk will address the challenges of testing industrial robots and will show a couple of examples where artificial intelligence techniques have been used to ease the automation of some parts of their testing processes. The presented work has been realized in the context of Certus, the Norwegian research-based innovation center dedicated to Software Validation and Verification.

Biographie :
Arnaud Gotlieb is a French research scientist in Computer Science, currently leader of the Certus centre which is the research-based innovation centre on Software Verification & Validation hosted at Simula Research Laboratory, Norway. Dr. Gotlieb's expertise is on the application of constraint solving to software testing. At the beginning of his career, he spent 7 years in Industry working for Dassault Electronics, Thomson-CSF and Thales, then joint INRIA, the French National Institute on Computer Science and Automatisms in 2002 where he developed constraint-based testing in a group specialized in static analysis and software verification and contributed to several projects aiming at improving static analysis, testing and verification of critical embedded software in civil and military avionics. He was the coordinator of the ANR CAVERN project (2008-2011) that explored the capabilities of constraint programming for program verification and participated to several large European Project proposals. Since Oct. 2011, he has joint the Simula Research Laboratory in Norway as a senior research scientist. Arnaud Gotlieb has co-authored more than eighty publications in international conferences and journals and he is the main architect of several constraint-based testing tools. He has served in the program committees of the IEEE Int. Conf. on Software Testing, Validation and Verification (ICST) from 2008 to 2015 and many other conferences such as ISSRE, ICSE-SEIP, TAP, QSIC and more recently CP and IJCAI. He has co-chaired the technical program of QSIC-13, the SEIP track of ICSE-14, the Testing and Verification track of CP-16 and CP-17. He has initiated the Constraints in Software Testing, Verification and Analysis (CSTVA) workshop series and was its main organizer during the first editions. He has successfully co-supervised more than ten PhD students and was heading the Software Engineering department of Simula from 2012 until 2015. He completed his PhD on automatic test data generation using constraint logic programming techniques in 2000 at the University of Nice-Sophia Antipolis and got habilitated (HDR) in Dec. 2011 from University of Rennes, France.

À la chasse aux bugs

Auteur : Gérard Berry (Collège de France)

Résumé :
L'informatique est un combat permanent entre deux exacts opposés : nous-mêmes les spécifieurs et programmeurs, imaginatifs mais lents et peu rigoureux, et les microprocesseurs, stupides mais hyper-rapides et totalement rigoureux. Le trajet de l'idée au programme qui marche est donc intrinsèquement difficile, et il doit être parfaitement maîtrisé pour éviter les bugs sournois qui résultent toujours d'une défaillance de l'homme et pas de la machine. Pour mieux comprendre leur nature profonde, nous analyserons certains bugs célèbres ou moins connus, dans lesquels l'ordinateur a amplifié des micro-erreurs en désastres ou encore provoqué des failles de sécurité facilement exploitables. Nous analyserons ensuite les méthodes qui permettent de chasser efficacement les bugs, le test classique s'avérant par nature insuffisant : s'il permet de trouver des erreurs, il est inopérant pour montrer leur absence. Nous verrons que le test aléatoire bien dirigé peut donner des résultats étonnants, par exemple en exhibant des centaines d'erreurs dans les compilateurs C actuellement disponibles. Nous présenterons ensuite les méthodes formelles qui permettent de démontrer automatiquement l'absence de certaines erreurs dans les circuits et programmes, voir même de prouver la correction totale du comportement comme pour l'arithmétique du Pentium chez Intel ou le compilateur C CompCert développé par Xavier Leroy et son équipe à l'Inria. Nous verrons d'autre cas où ces techniques commencent à s'appliquer, des noyaux de systèmes d'exploitation aux protocoles de sécurité. Nous discuterons enfin la question des mauvais fonctionnements des algorithmes d'apprentissage, qui ne sont pas de même nature que les bugs et demandent de nouvelles analyses.

Biographie :
Ancien élève de l'École polytechnique, ingénieur général du Corps des mines, membre de l'Académie des sciences, de l'Académie des technologies et de l'Academia Europaea, Gérard Berry a été chercheur à l'École des mines de Paris et à l'INRIA de 1970 à 2000, Directeur scientifique de la société Esterel Technologies de 2001 à 2009 puis Directeur scientifique Inria et président de la Commission d'évaluation de cet institut de 2009 à 2012. Il tient la chaire Informatique et sciences numériques au Collège de France depuis 2012, après y avoir tenu deux chaires annuelles en 2007-2008 et 2009-2010. Sa contribution scientifique concerne quatre sujets principaux : le traitement formel des langages de programmation et leurs relations avec la logique mathématique, la programmation parallèle et temps réel, la conception assistée par ordinateur de circuits intégrés, et la vérification formelle des programmes et circuits. Il est le créateur du langage de programmation Esterel. Il a reçu la Médaille d'or du CNRS en 2014.

Lightweight model-driven engineering

Auteur : Jordi Cabot (ICREA)

Résumé :
Model-driven engineering pretended to be the silver bullet for software engineering. And it failed, with many quickly dismissing it. But MDE is far from dead. If anything, it's lurking in the dark.

In this talk, I will try to turn non-believers into new adepts for the MDE cult (in fact, you're all modelers, whether you accept this universal truth or not !). I will also give MDE followers new and powerful arguments to continue their evangelization based on a new "lightweight" version of MDE.

This lightweight alternative combines a more pragmatic approach of applying MDE in practice, the development of new MDE tools targeting non-technical users (e.g. citizen developers interested in exploiting open data & big data initiatives) and the use of cognification techniques to multiply the effectiveness of MDE.

Biographie :
Jordi Cabot is an ICREA Research Professor at Internet Interdisciplinary Institute (Open University of Catalonia) where he leads the SOM (Systems, Software and Models) Lab. Previously, he was leader of the AtlanMod team, an INRIA and LINA research group at École des Mines de Nantes (France), a post-doctoral fellow at the University of Toronto and a visiting scholar at the Politecnico di Milano.

His research interests include software and systems modeling, model-driven and web engineering, formal verification and social aspects of software engineering. Apart from his scientific publications, he writes and blogs about all these topics in his Modeling Languages portal (http ://modeling-languages.com).

Session commune aux groupes de travail Compilation et LaHMA

Compilation — Langages et Modèles de Haut-niveau pour la programmation parallèle, distribuée, de grilles de calcul et Applications

Une caractérisation impérative à la ASM des algorithmes BSP

Frédéric GAVA (LACL)

Session Compilation/Lahma 1 du GDR GPL, juin 2017

Résumé

Nous tenterons de répondre formellement à la question : qu'est-ce qu'un algorithme BSP ? Pour ce faire, nous étendrons à BSP la définition axiomatique des algorithme séquentiels de Gurevich et ses "abstract state machines" (ASM).

Puis nous donnerons une équivalence algorithmique entre ces ASM -BSP et un mini langage impératif via une simulation pas-à-pas.

Weak memory using event structures

Simon Castellan (LIP)

Session Compilation/Lahma 1 du GDR GPL, juin 2017

Résumé

We propose a novel approach to model architectures featuring relaxed memory behaviours. The approach used here takes inspiration from standard denotational tools for semantics of concurrency and game semantics to give a denotational model based on event structures of an idealized assembly language.

The approach is neatly decomposed into a thread semantics and a storage semantics : the first one dealing with intra-thread causality (program order) and the second one dealing with inter-thread causalities (memory order). The two are then combined by taking the synchronized product.

In this talk, we demonstrate this technology on the architecture TSO, and build a model for a simple assembly language.

Profilage d'applications hétérogènes de haut niveau.

Mathias Bourgoin (LIFO)

Session Compilation/Lahma 1 du GDR GPL, juin 2017

Résumé

Les systèmes hétérogènes (combinant plusieurs types d'unités de calcul) sont maintenant très répandus. Soigneusement utilisés, ils permettent une augmentation de performances impressionnante. Cependant, ils exigent généralement que les développeurs combinent plusieurs modèles de programmation, à travers des langages et des outils très complexes. Les programmes hétérogènes sont ainsi difficiles à concevoir et à déboguer. L'écriture de programmes hétérogènes corrects est difficile. Obtenir de bonnes performances l'est encore plus.

Pour aider les développeurs, de nombreuses solutions de haut niveau ont été développées. La plupart d'entre elles génèrent une partie du code qui sera effectivement exécuté et isolent le programmeur des détails de bas niveau. Elles aident à créer des programmes efficaces et plus sûrs, mais laissent les programmeurs confus face à des bogues complexes ou lorsqu'ils tentent d'optimiser leur application.

Dans cette présentation, je décrirai notre solution pour le profilage d'applications hétérogènes portables décrites avec SPOC (Stream Processing with OCaml). Pour cela, je présenterai d'abord rapidement SPOC : une bibliothèque portable pour le langage OCaml associée à un DSL permettant de générer du code GPGPU à l'exécution. Je détaillerai ensuite notre solution de profilage qui se concentre sur deux points principaux : - rester portable et offrir des informations pertinentes, quelle que soit la combinaison d'architectures présentes dans le système hétérogène - offrir au programmeur des informations utiles facilement repliables au code écrit à l'origine et non au code généré par SPOC.

Analyse statique de programmes MPI par interprétation abstraite

Vincent BOTBOL (LIP6)

Session Compilation/Lahma 2 du GDR GPL, juin 2017

Résumé

Dans le but d'analyser statiquement des systèmes distribués, nous présentons un modèle de concurrence dont les états de programmes sont représentés par des langages reconnus par des automates symboliques. Nous utilisons ces automates (lattice automata) en tant que domaine abstrait pour notre analyse. La fonction de transition de notre système raisonne sur ces automates en réécrivant le langage reconnu à l'aide d'un ensemble de règles de réécritures encodant la sémantique des instructions de notre programme.

Grâce à la méthodologie de l'interprétation abstraite, nous sommes capables d'établir une sur-approximation de l'espace d'atteignabilité d'états de programmes concurrents. L'utilisation modulaire de domaines abstraits numériques sous-jacents permet d'inférer des invariants entre les processus et donc de prouver, entre autre, des propriétés numériques séquentielles ou globales.

Nous conclurons par la présentation de notre prototype permettant une analyse de valeur d'un sous-ensemble de la bibliothèque de calcul parallèle MPI/C. Cet outil s'intégre en tant que plugin à la plateforme Frama-C dédiée à l'analyse de programmes C.

Code generation and Real-Time Analyses for Many-Core Architecture

Matthieu MOY (Verimag)

Session Compilation/Lahma 2 du GDR GPL, juin 2017

Résumé

Simple, single-core processors have been the main execution platform for critical real-time software for years. Such software can be handwritten in imperative languages, but code generation from higher-level languages such as the synchronous data-flow SCADE (industrial version of the Lustre language) are also used in production today.

Single-core processors are reaching their limits. Performance, or energy efficiency constraints are leading the industry towards multi-core or many-core including for critical systems. These architectures raise new major challenges for timing analysis.

In this talk, we present a code generation flow from SCADE that produces parallel and yet predictable code for Kalray MPPA architecture. The code generation scheme is designed together with a timing analysis that take into account interference between cores when they access the local memory.

This work is performed as part of the CAPACITES project : http ://capacites.minalogic.net/

Session du groupe de travail IDM

Ingénierie Dirigée par les Modèles

Impact de l'évolution dans un écosystème logiciel

Anne Etien

Pour beaucoup d'entreprises, leur richesse se trouvent dans leurs *logiciels patrimoniaux* qui existent souvent depuis plusieurs dizaines d'années et renferment une grande partie de la connaissance de l'entreprise, ses règles de gestions, son savoir faire... Au cours du temps, les besoins auxquels ces logiciels répondent ont évolué, de même que les technologies sur lesquels ils reposent, entrainant des modifications en leur sein même. Ces modifications ayant eu lieu après la livraison du logiciel sont considérées comme de la *maintenance*. Elles correspondent à plus de 80% du cycle de vie du logiciel et de son coût. Maintenir un logiciel est une activité complexe et nécessaire qui mérite d'être anticipé dès la phase de conception du logiciel. Des phases de remodularisation peut aussi être utiles pour réduire la complexité accumulée par des évolutions successives et fournir de nouvelles bases solides pour de futures évolutions. Ces différentes activités deviennent encore plus complexes dans le cas d'un écosystème logiciel, c'est-à-dire de logiciels, frameworks et bibliothèques qui interagissent ensemble.

Dans cet exposé, nous montrerons deux exemples d'utilisation de modèles pour aider à la gestion de l'évolution de logiciel. Plus précisément, dans une première partie, on s'intéresse à l'automatisation de transformations spécifiques. En effet, lors de restructuration de logiciels, il est fréquent d' effectuer plusieurs fois des séquences de transformation de code (par exemple, créer une classe, implémenter une interface donnée, puis surcharger une méthode) sur différentes entités du système (e.g. certaines classes dans une même hiérarchie ou dans un même paquetage). De par la nature répétitive de ces transformations, il est nécessaire d'automatiser leur support afin d'assurer que ces séquences de transformations sont appliquées de façon consistante sur la globalité du système. Une solution pour gérer des transformations systématiques de code est de permettre aux développeurs de composer leurs propres séquences de transformations de code. Ces séquences peuvent être définies manuellement par exemple en utilisant un langage pour spécifier chaque transformation ou elles peuvent être identifiées à partir d'un ou plusieurs exemples concrets fournis par le développeur. Nous affirmons que les approches existantes ne permettent pas de définir des séquences : (i) spécifiques au système sur lesquelles elles sont appliquées ; (ii) éventuellement complexes c'est-à-dire pas entièrement supportées par les outils de refactoring existant ; (iii) non localisées, c'est-à-dire plusieurs entités de code peuvent être impactées à chaque occurrence de la séquence ; et (iv) conscientes de possibles violations qui peuvent être introduites consécutivement à ces transformations. Nous proposons une approche outillée reposant sur l'utilisation de modèles pour aider le développeur dans la définition et l'application de transformations systématiques.

Dans une deuxième partie, on s'intéresse à la sélection de tests à relancer après un changement. Un partenaire industriel a fait appel à nous après avoir réalisé que les tests sont sous utilisés sur ses projets. En fait, sur certains projets l'execution de l'ensemble des tests peut prendre plusieurs heures. Il n'est donc pas possible pour le développeur de les relancer tous après chaque changement ou même plusieurs fois par jour. Nous proposons une approche outillée qui s'appuie sur une abstraction du code sous forme de modèles pour analyser statiquement aprés un changement les tests impactés par ce changement, qui nécessitent donc d'être relancés.

Nous avons mené une première expérience grandeur nature au sein de cette entreprise pour évaluer comment les développeurs testent. Les résultats de cette expérience nous servirons de point de com-

paraison. Nous menons actuellement une deuxième expérience au sein de la même entreprise pour voir si notre outil de sélection de test change le comportement des développeurs vis à vis des tests.

Ces deux travaux réalisés dans le cadre de thèses sont deux exemples de l'utilisation de modèles pour la gestion de l'evolution. Ils montrent que cette abstraction permet une indépendance vis à vis des logiciels, des domaines d'application et souvent des langages utilisés. La partie géneration des modèles par parsing du code ne sera pas abordée dans cet exposé. En revanche, c'est bien l'utilisation des modèles pour l'aide à l'évolution qui sera présentée.

Mots clés : métamodélisation, maintenance, test, évolution, remodularisation.

PrefetchML: a Framework for Prefetching and Caching Models

Gwendal Daniel
AtlanMod Team - Inria, Mines Nantes & Lina
4, Rue Alfred Kastler
Nantes, France
gwendal.daniel@inria.fr

Gerson Sunyé
AtlanMod Team - Inria, Mines Nantes & Lina
4, Rue Alfred Kastler
Nantes, France
gerson.sunye@inria.fr

Jordi Cabot
ICREA
UOC
Av. Carl Friedrich Gauss, 5
Castelldefels, Spain
jordi.cabot@icrea.cat

ABSTRACT

Prefetching and caching are well-known techniques integrated in database engines and file systems in order to speed-up data access. They have been studied for decades and have proven their efficiency to improve the performance of I/O intensive applications. Existing solutions do not fit well with scalable model persistence frameworks because the prefetcher operates at the data level, ignoring potential optimizations based on the information available at the metamodel level. Furthermore, prefetching components are common in relational databases but typically missing (or rather limited) in NoSQL databases, a common option for model storage nowadays. To overcome this situation we propose PrefetchML, a framework that executes prefetching and caching strategies over models. Our solution embeds a DSL to precisely configure the prefetching rules to follow. Our experiments show that PrefetchML provides a significant execution time speedup. Tool support is fully available online.

Keywords

Prefetching; MDE; DSL; Scalability; Persistence Framework; NoSQL

1. INTRODUCTION

Prefetching and caching are two well-known approaches to improve performance of applications that rely intensively on I/O accesses. Prefetching consists in bringing objects into memory before they are actually requested by the application to reduce performance issues due to the latency of I/O accesses. Fetched objects are then stored in memory to speed-up their (possible) access later on. In contrast, caching aims at speeding up the access by keeping in memory objects that have been already loaded.

Prefetching and caching have been part of database management systems and file systems for a long time and have proved their efficiency in several use cases [23, 25]. P. Cao et al. [6] showed that integrating prefetching and caching strategies dramatically improves the performance of I/O-intensive applications. In short, prefetching mechanisms works by adding load instructions (according to prefetching rules derived by static [16] or execution trace analysis [8]) into an existing program. Global policies, (e.g., LRU - least recently used, MRU - most recently used, etc.) control the cache contents.

Currently, there is lack of support for prefetching and caching at the model level. Given that model-driven engineering (MDE) is progressively adopted in the industry [15, 21] such support is required to raise the scalability of MDE tools dealing with large models where storing, editing, transforming, and querying operations are major issues [19, 28]. These large models typically appear in various engineering fields, such as civil engineering [1], automotive industry [4], product lines [24], and in software maintenance and evolution tasks such as reverse engineering [5].

Existing approaches have proposed scalable model persistence frameworks on top of SQL and NoSQL databases [11, 13, 17, 22]. These frameworks use lazy-loading techniques to load into main memory those parts of the model that need to be accessed. This helps dealing with large models that would otherwise not fit in memory but adds an execution time overhead due to the latency of I/O accesses to load model excerpts from the database, specially when executed in a distributed environment.

In this sense, this paper proposes a new prefetching and caching framework for models. We present *PrefetchML*, a domain specific language and execution engine, to specify prefetching and caching policies and execute them at runtime in order to optimize model access operations. This DSL allows designers to customize the prefetching rules to the specific needs of model manipulation scenarios, even providing several execution plans for different use cases. Our framework is built on top of the Eclipse Modeling Framework (EMF) infrastructure and therefore it is compatible with existing scalable model persistence approaches, regardless whether those backends also offer some kind of internal prefetching mechanism. A special version tailored to the NeoEMF/Graph [3] engine is also provided for further performance improvements. The empirical evaluation of PrefetchML highlights the significant time benefits it achieves.

The paper is organized as follows: Section 2 introduces further the background of prefetching and caching in the modeling ecosystem while Section 3 introduces the PrefetchML DSL. Section 4 describes the framework infrastructure

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

and its rule execution algorithm and Section 5 introduces the editor that allows the designer to define prefetching and caching rules, and the implementation of our tool and its integration with the EMF environment. Finally, Section 6 presents the benchmarks used to evaluate our prefetching tool and associated results. Section 7 ends the paper by summarizing the key points and presenting our future work.

2. STATE OF THE ART

Prefetching and caching techniques are common in relational and object databases [25] in order to improve query computation time. Their presence in NoSQL databases is much more limited, which is problematic due to the increasing popularity of this type of databases as model storage solution. Moreover, database-level prefetching and caching strategies do not provide fine-grained configuration of the elements to load according to a given usage scenario–such as model-to-model transformation, interactive editing, or model validation–and are often strongly connected to the data representation, making them hard to evolve and reuse.

Scalable modeling frameworks are built on top of relational or NoSQL databases to store and access large models [3,11]. These approaches are often based on lazy-loading strategies to optimize memory consumption by loading only the accessed objects from the database. While lazy-loading approaches have proven their efficiency in terms of memory consumption to load and query very large models [9,22], they perform a lot of fragmented queries on the database, thus adding a significant execution time overhead. For the reasons described above, these frameworks cannot benefit from database prefetching solutions nor they implement their own mechanism, with the partial exception of CDO [11] that provides some basic prefetching and caching capabilities[1]. For instance, CDO is able to bring into memory several objects of a list at the same time, or loading nested/related elements up to a given depth. Nevertheless, alternative prefetching rules cannot be defined to adapt model access to different contexts nor it is possible to define rules with complex prefetching conditions.

Hartmann et al. [14] propose a solution to tackle scalability issues in the context of models@run.time by splitting models into chunks that are distributed across multiple nodes in a cluster. A lazy-loading mechanism allows to virtually access the entire model from each node. However, to the best of our knowledge the proposed solution does not provide prefetching mechanism, which could improve the performance when remote chunks are retrieved and fetched among nodes.

Optimization of query execution has also been targeted by other approaches not relying on prefetching but using a variety of other techniques. EMF-IncQuery [4] is an incremental evaluation engine that computes graph patterns over an EMF model. It relies on an adaptation of the RETE algorithm, and results of the queries are cached and incrementally updated using the EMF notification framework. While EMF-IncQuery can be seen as an efficient EMF cache, it does not aim to provide prefetching support, and cache management cannot be tuned by the designer. Hawk [2] is a model indexer framework that provides a query API. It stores models in an index and allows to query them using the EOL [18] query language. While Hawk provides an efficient backend-independent query language, it does not allow the definition of prefetching plans for the indexed models.

In summary, we believe no existing solution provides the following desired characteristics of an efficient and configurable prefetching and caching solution for models:

1. Ability to define/execute prefetching rules independently of the database backend.
2. Ability to define/execute prefetching rules transparently from the persistence framework layered on top of the database backend.
3. A prefetching language expressive enough to define rules involving conditions at the type and instance level (i.e. loading all instances of a class A that are linked to a specific object of a class B).
4. A context-dependent prefetching language allowing the definition of alternative prefetching and caching plans for specific use cases.
5. A readable prefetching language enabling designers to easily tune the prefetching and caching rules.

In the following, we present PrefetchML, our prefetching and caching framework that tackles these challenges.

3. THE PREFETCHML DSL

PrefetchML is a DSL that describes prefetching and caching rules over models. Rules are triggered when an event satisfying a particular condition is received. These events can be the initial model loading, an access to a specific model element, the setting of a value or the deletion of a model element. Event conditions are expressed using OCL guards.

Loading instructions are also defined in OCL. The set of elements to be loaded as a response to an event are characterized by means of OCL expressions that navigate the model and select the elements to fetch and store in the cache. Not only loading requests can be defined, the language also provides an additional construct to control the cache content by removing cache elements when a certain event is received. Using OCL helps us to be independent of any specific persistence framework.

Prefetching and caching rules are organized in plans, that are sets of rules that should be used together to optimize a specific usage scenario for the model since different kinds of model accesses may require different prefetching strategies. For example, a good strategy for an interactive model browsing scenario is to fetch and cache the containment structure of the model, whereas for a complex query execution scenario it is better to have a plan that fit the specific navigation path of the query.

Beyond a set of prefetching rules, each plan defines a cache that can be parametrized, and a caching policy that manages the life-cycle of the cached elements.

In what follows, we first introduce a running example and then we formalize the abstract and concrete syntax of the PrefetchML DSL. Next Section will introduce how these rules are executed as part of the prefetching engine.

3.1 Running Example

In order to better illustrate the features of PrefetchML, we introduce a simple example model. Figure 1 shows a small excerpt of the *Java* metamodel provided by MoDisco [5].

[1]https://wiki.eclipse.org/CDO/Tweaking_Performance

A *Java* program is described in terms of *Packages* that are named containers that group *ClassDeclarations* through their *ownedElements* reference. A *ClassDeclaration* contains a *name* and a set of *BodyDeclarations*. *BodyDeclarations* are also named, and its *visibility* is described by a single *Modifier*. *ClassDeclarations* maintain a reference to their *CompilationUnit* (the physical file that stores the source code of the class). This *CompilationUnit* has a *name*, a list of *Comments*, and a list of *imported ClassDeclarations* (corresponding to the `import` clauses in Java programs).

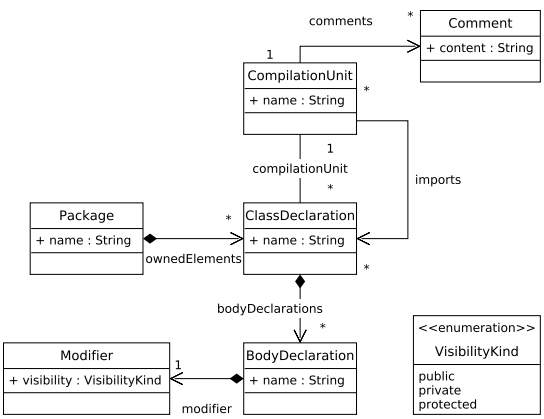


Figure 1: Excerpt of Java Metamodel

Listing 1 presents three sample OCL queries that can be computed over an instance of the previous metamodel: the first one returns the *Package* elements that do not contain any *ClassDeclaration* through their *ownedElements* reference. The second one returns from a given *ClassDeclaration* all its contained *BodyDeclarations* that have a private *Modifier*, and the third one returns from a *ClassDeclaration* a sequence containing the *return Comment* elements in the *ClassDeclarations* that are imported by the *CompilationUnit* associated to the current element.

```
context Package
def : isAnEmptyPackage : Boolean =
    self.ownedElements→isEmpty()

context ClassDeclaration
def : privateBodyDeclarations : Sequence(
    BodyDeclaration) =
    self.bodyDeclarations
      →select(bd | bd.modifier = VisibilityKind::
          Private)

context ClassDeclaration
def : importedComments : Sequence(Comment) =
    self.compilationUnit.imports.compilationUnit.
        comments
      →select(c | c.content.contains('@return')))
```

Listing 1: Sample OCL Query

3.2 Abstract Syntax

This section describes the main concepts of PrefetchML focusing on the different types of rules it offers and how they can be combined to create a complete prefetch specification.

Figure 2 depicts the metamodel corresponding to the abstract syntax of the PrefetchML language. A *PrefetchSpecification* is a top-level container that *imports* several *Metamodels*. These metamodels represent the domain on which prefetching and caching rules are described, and are defined by their *Unified Resource Identifier (URI)*.

The imported *Metamodels* concepts (classes, references, attributes) are used in prefetching *Plans*, which are named entities that group rules that are applied in a given execution context. A *Plan* can be the *default* plan to execute in a *PrefetchSpecification* if no execution information is provided.

Each *Plan* contains a *CacheStrategy*, which represents the information about the cache policy the prefetcher applies to keep loaded objects into memory. Currently, available cache strategies are *LRUCache* (Least Recently Used) and *MRUCache* (Most Recently Used). These *Caches* define two parameters: the maximum number of objects they can store (*size*), and the number of elements to free when it is full (*chunkSize*). In addition, a *CacheStrategy* can contain a *tryFirst OCL expression*. This expression is used to customize the default cache replacement strategy: it returns a set of model elements that should be removed from the cache if it is full, overriding the selected caching policy.

Plans also contain the core components of the PrefetchML language: *PrefetchingRules* that describe tracked model events and the loading and caching instructions. We distinguish two kinds of *PrefetchingRules*:

- *StartingRules* that are prefetching instructions triggered only when the prefetching plan is loaded
- *ObjectRules* that are triggered when an element satisfying a given condition is accessed, deleted, or updated.

ObjectRules can be categorized in three different types: *Access* rules, that are triggered when a particular model element is accessed, *Set* rules that correspond to the setting of an attribute or a reference, and *Delete* rules, that are triggered when an element is deleted or simply removed from its parent. When to fire the trigger is also controlled by the *sourceContext* class, that represents the type of the elements that could trigger the rule. This is combined with the *sourceExpression* (i.e. the guard for the event) to decide whether an object matches the rule.

All kinds of *PrefetchingRules* contain a *targetExpression*, that represents the elements to load when the rule is triggered. This expression is an *OCLExpression* that navigates the model and returns the elements to load and cache. Note that if *self* is used as the *targetExpression* of an *AccessRule* the framework will behave as a standard cache, keeping in memory the accessed element without fetching any additional object.

It is also possible to define *removeExpressions* in *PrefetchingRules*. When a *removeExpression* is evaluated, the prefetcher marks as free all the elements it returns from its cache. Each *removeExpression* is associated to a *removeContext Class*, that represents the context of the OCL expression. *remove* expressions can be coupled with the *tryFirst* expression contained in the *CacheStrategy* to tune the default replacement policy of the cache.

3.3 Concrete Syntax

We introduce now the concrete syntax of the PrefetchML language, which is derived from the abstract syntax metamodel presented in Figure 2. Listing 2 presents the grammar

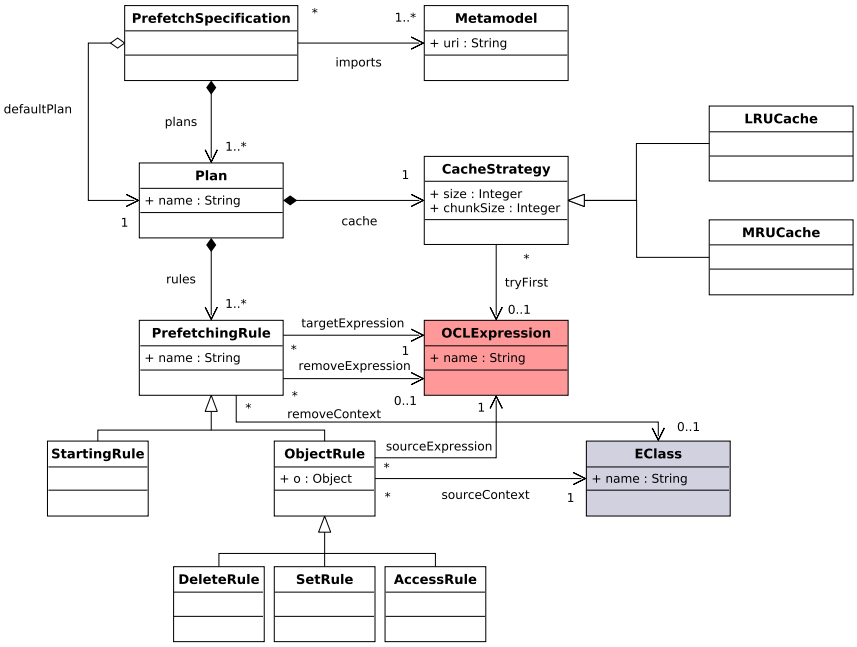


Figure 2: Prefetch Abstract Syntax Metamodel

of the PrefetchML language expressed using XText [12], an EBNF-based language used to specify grammars and generate an associated toolkit containing a metamodel of the language, a parser, and a basic editor. The grammar defines the keywords associated with the constructs presented in the PrefetchML metamodel. Note that *OCLExpressions* are parsed as *Strings*, the model representation of the queries presented in Figure 2 is computed by parsing it using the Eclipse MDT OCL toolkit[2]

```
grammar fr.inria.atlanmod.Prefetching
with org.eclipse.xtext.common.Terminals
import "http://www.inria.fr/atlanmod/Prefetching"

PrefetchSpecification:
  metamodel=Metamodel
  plans+=Plan+
;

Metamodel:
  'import' nsURI=STRING
;

Plan:
  'plan' name=ID (default?='default')? '{'
    cache=CacheStrategy
    rules+=(StartingRule | AccessRule)*
  '}'
;

CacheStrategy:
  (LRUCache{LRUCache} | MRUCache{MRUCache})
    (properties=CacheProperties)? ('when full
        remove' tryFirstExp=OCLExpression)?
;

LRUCache:
  'use cache' 'LRU'
;

MRUCache:
  'use cache' 'MRU'
;

CacheProperties:
  '[' 'size'size=INT ('chunk'chunk=INT)? ']'
;

PrefetchingRule:
  (StartingRule | AccessRule | DeleteRule |
      SetRule)
;

StartingRule:
  'rule' name=ID ':' 'on starting'
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

AccessRule:
  'rule' name=ID ':' 'on access'
  'type'sourceType=ClassifierExpression (
      sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

DeleteRule:
  'rule' name=ID ':' 'on delete'
  'type'sourceType=ClassifierExpression (
      sourcePatternExp=OCLExpression)?
  'fetch' targetPatternExp=OCLExpression
  ('remove' 'type'removeType=ClassifierExpression
      removePatternExp=OCLExpression)?
;

SetRule:
  'rule' name=ID ':' 'on set'
```

[2]http://www.eclipse.org/modeling/mdt/?project=ocl

```
    'type' sourceType=ClassifierExpression (
        sourcePatternExp=OCLExpression)?
    'fetch' targetPatternExp=OCLExpression
    ('remove ' 'type' removeType=ClassifierExpression
        removePatternExp=OCLExpression)?
;

OCLExpression: STRING ;

ClassifierExpression: ID;
```

Listing 2: PrefetchML Language Grammar

Listing 3 provides an example of a *PrefetchSpecification* written in PrefetchML. To continue with our running example, the listing displays prefetching and caching rules suitable for the queries expressed in the running example (Listing 1).

The *PrefetchSpecification* imports the Java *Metamodel* (line 1). This *PrefetchSpecification* contains a *Plan* named *samplePlan* that uses a *LRUCache* that can contain up to 100 elements and removes them by chunks of 10 (line 4). It is also composed of three *PrefetchingRules*: the first one, `r1` (5-6), is a starting rule that is executed when the plan is activated, and loads and caches all the *Package* classes. The rule `r2` (7-8) is an access rule that corresponds to the prefetching and caching actions associated to the query *PrivateBodyDeclarations*. It is triggered when a *ClassDeclaration* is accessed, and loads and caches all the *BodyDeclarations* and *Modifiers* it contains. The rule `r3` (9-11) corresponds to the query *ImportedComments*: it is also triggered when a *ClassDeclaration* is accessed, and loads the associated *CompilationUnit*, and the *Comment contents* of its *imported ClassDeclarations*. The rule also defines a `remove` expression, that removes all the *Package* elements from the cache when the load instruction is completed.

```
1  import "http://www.example.org/Java"
2
3  plan samplePlan {
4    use cache LRU[size=100,chunk=10]
5    rule r1 : on starting fetch
6      Package.allInstances()
7    rule r2 : on access type ClassDeclaration fetch
8      self.bodyDeclarations.modifier
9    rule r3 : on access type ClassDeclaration fetch
10     self.compilationUnit.imports.compilationUnit.
           comments.content
11   remove type Package
12 }
```

Listing 3: Sample Prefetching Plan

4. PREFETCHML FRAMEWORK INFRASTRUCTURE

In this Section we present the infrastructure of the PrefetchML framework and its integration in the modeling ecosystem (details on its integration on specific modeling frameworks are provided in the next section). We also detail how prefetching rules are handled and executed using the running example presented in the previous Section.

4.1 Architecture

Figure 3 shows the integration of the **PrefetchML** framework in a typical modeling framework infrastructure: grey nodes represent standard model access components: a model-based tool accesses a model through a modeling API, which delegates to a persistence framework in charge of handling the physical storage of the model (for example in XML files, or in a database).

In contrast, the PrefetchML framework (white nodes) receives events from the modeling framework. When the events trigger a prefetching rule, it delegates the actual computation to its **Model Connector**. This component interacts with the modeling framework to retrieve the requested object, typically by translating the OCL expressions in the prefetching rules into lower level calls to the framework API. Section 5 discusses two specific implementations of this component.

The PrefetchML framework also intercepts model elements accesses, in order to search first in its **Cache** component if the requested objects are already available. If the cache contains the requested information, it is returned to the modeling framework, bypassing the persistence framework and improving execution time.

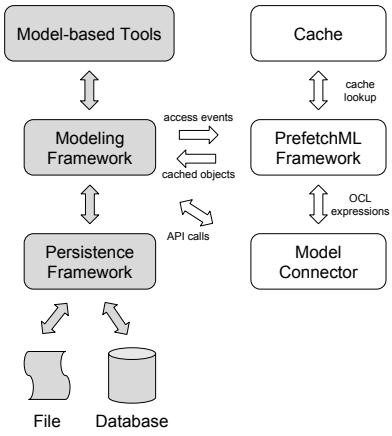


Figure 3: PrefetchML Integration in MDE Ecosystem

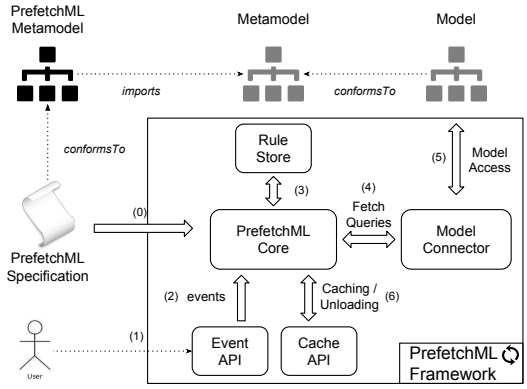


Figure 4: Prefetch Framework Infrastructure

Figure 4 describes the internal structure of the PrefetchML Framework. As explained in Section 3, a PrefetchML specifi-

cation *conforms to* the PrefetchML metamodel. This specification *imports* also the metamodel/s for which we are building the prefetching plans.

The **Core** component of the PrefetchML framework is in charge of loading, parsing and storing these *PrefetchML specifications* and then use them to find and retrieve the prefetching / caching rules associated with an incoming event, and, when necessary, execute them. This component also contains the internal *cache* that retains fetched model elements in memory. The **Rule Store** is a data structure that stores all the object rules (access, update, delete) contained in the input *PrefetchML description*. The **Model Connector** component is in charge of the translation and the execution of *OCLExpressions* in the prefetching rules. This connector can work at the modeling framework level, meaning that it executes fetch queries using the modeling API itself, or at the database level, translating directly OCL expressions into database queries.

The **CacheAPI** component gives access to the cache contents to client applications. It allows manual caching and unloading operations, and provides configuration facilities. This API is an abstraction layer that unifies access to the different cache types that can be instantiated by the Core component. Note that in our architecture we integrated prefetching and caching solutions in the sense that the core component manages its own cache, where only prefetched elements are stored. While this may result in keeping in the cache objects that are not going to be recurrently used, using a LRU cache strategy allows the framework to get rid off them when memory is needed. In addition, the grammar allows to define a minimal cache that would act only as a storage mechanism for the immediate prefetched objects.

The **EventAPI** is the component that is in charge of receiving events from the client application. It provides an API to send access, delete, and update events. These events are defined at the object level, and contain contextual information of their encapsulated model element, such as its identifier, the reference or attribute that is accessed, and the index of the accessed element. These informations are then used by the Core Component to find the rules that match the event.

In particular, when an object event is sent to the PrefetchML framework (1), the *Event API* handles it and forwards it to the *Core Component*, which is in charge of triggering the associated prefetching and caching rule. To do that, the *Core Component* searches in the *Rule Store* the rules that corresponds to the event and the object that triggered it (3). Each *OCLExpression* in the rules is translated into fetch queries sent to the *Model Connector* (4), which is in charge of the actual query computation over the model (5). Query results are handled back by the *PrefetchML Core*, which is in charge of caching them and freeing the cache from previously stored objects.

As prefetching operations can be expensive to compute, the PrefetchML Framework runs in the background, and contains a pool of working threads that performs the fetch operations in parallel of the application execution. Model elements are cached asynchronously and available to the client application through the *CacheAPI*. Prefetching queries are automatically aborted if they take too much time and/or if their results are not relevant (according to the number of cache hits) in order to keep the PrefetchML Framework synchronized with the client application, e.g. preventing it from loading elements that are not needed anymore.

The PrefetchML framework infrastructure is not tailored to a particular data representation and can be plugged in any kind of model persistence framework that stores models conforming to the Ecore metamodel and provides an API rich enough to evaluate OCL queries. This includes for example EMF storage implementations such as XMI, but also scalable persistence layers built on top of the EMF, like NeoEMF [13], CDO [11], and Morsa [22].

4.2 Rule Processing

We now look at the PrefetchML engine from a dynamic point of view. Figure 5 presents the sequence diagram associated with the initialization of the PrefetchML framework. When initializing, the prefetcher starts by loading the *PrefetchDescription* to execute (1). To do so, it iterates through the set of plans and stores the rules in the *RuleStore* according to their type (2). In the example provided in Listing 3 this process saves in the store the rules `r2` and `r3`, both associated with the *ClassDeclaration* type. Then, the framework creates the cache (3) instance corresponding to the active prefetching plan (or the default one if no active plan is provided). This creates the `LRU` cache of the example, setting its `size` to 100 and its `chunkSize` to 10.

Next, the PrefetchML framework iterates over the *StartingRules* of the description and computes their *targetExpression* using the *Model Connector* (4). Via this component, the OCL expression is evaluated (in the example the target expression is `Package.allInstances()`) and the resulting elements are returned to the *Core* component (5) that creates the associated identifying keys (6) and stores them in the cache (7). Note that starting rules are not stored in the Rule Store, because they are executed only once when the plan is activated, and are no longer needed afterwards.

Once this initial step has been performed, the framework awaits object events. Figure 6 shows the sequence diagram presenting how the PrefetchML handles incoming events. When an object event is received (8), it is encapsulated into a working task which contains contextual information of the event (object accessed, feature navigated, and index of the accessed feature) and asynchronously sent to the prefetcher (9) that searches in the *RuleStore* the object rules that have the same type as the event (10). In the example, if a *ClassDeclaration* element is accessed, the prefetcher searches associated rules and returns `r2` and `r3`. As for the diagram above, the next calls involve the execution of the target expressions for the matched rules and saving the retrieved objects in the cache for future calls. Finally, the framework evaluates the remove OCL expressions (17) and frees the matching objects from the memory. In the example, this last step removes from the cache all the instances of the *Package* type.

5. TOOL SUPPORT

In this Section we present the tool support for the PrefetchML framework. It is composed of two main components: a language editor (Section 5.1) that supports the definition of prefetching and caching rules, and a execution engine with two different integration options: the EMF API and the NeoEMF/Graph persistence framework (Sections 5.2 and 5.3). The presented components are part of a set of open source Eclipse plugins available at https://github.com/atlanmod/Prefetching_Caching_DSL.

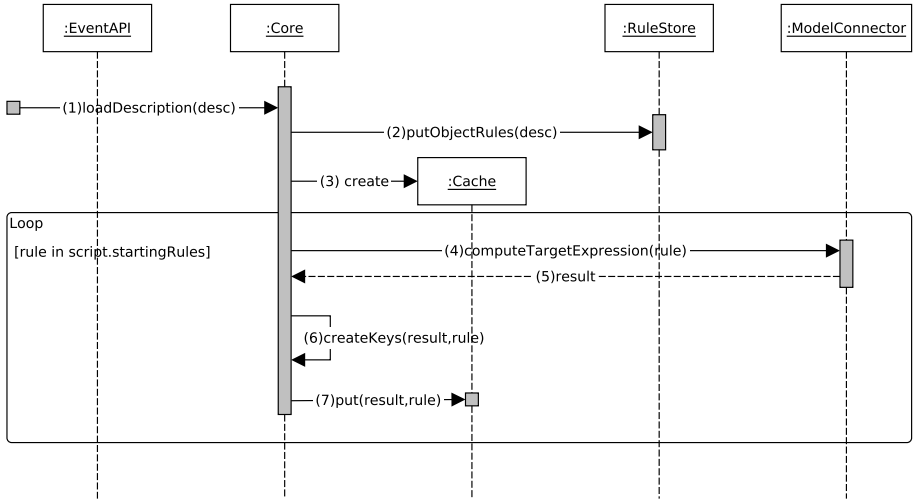


Figure 5: PrefetchML Initialization Sequence Diagram

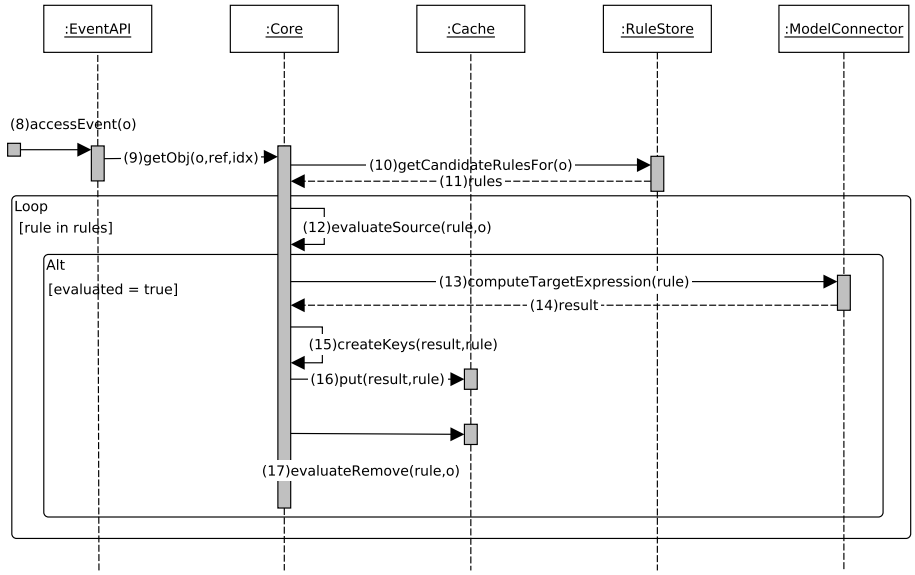


Figure 6: PrefetchML Event Handling Sequence Diagram

5.1 Language Editor

The PrefetchML language editor is an Eclipse-based editor that allows the creation and the definition of prefetching and caching rules. It is partly generated from the XText grammar presented in Section 3.3 and defines utility helpers to validate and navigate the imported metamodel. The editor supports navigation auto-completion by inspecting imported metamodels, and visual validation of prefetching and caching rules by checking reference and attribute existence.

Figure 7 shows an example of the PrefetchML editor that contains the prefetching and caching plan defined in the running example of Section 3.

5.2 EMF Integration

Figure 8 shows the integration of PrefetchML within the EMF framework. Note that only two components must

```
article_sample.prefetch
import "http://www.eclipse.org/MoDisco/Java/0.2.incubation/java-neoemf"

plan SamplePlan {
    use cache LRU[size 100 chunk 10]
    rule r1 : on starting
        fetch Package.allInstances()
    rule r2 : on access type ClassDeclaration
        fetch self.ownedElements.bodyDeclarations.modifier
    rule r3 : on access type ClassDeclaration
        fetch self.compilationUnit.imports.compilationUnit
            .comments.content
        remove type Package
}
```

Figure 7: PrefetchML Rule Editor

be adapted (light grey boxes). The rest are either generic PrefetchML components or standard EMF modules.

In particular, dark grey boxes represent the standard EMF-based model access architecture: an *EMF-based* tool accesses the model elements through the *EMF API*, that delegates the calls to the *PersistenceFramework* of choice (XMI, CDO, NeoEMF,...), which is finally responsible for the model storage.

The two added/adapted components are:

- An *Interceptor* that wraps the EMF API and captures the calls (1) to the EMF API (such as `eGet`, `eSet`, or `eUnset`). EMF calls are then transformed into *EventAPI* calls (2) by deriving the appropriate event object from the EMF API call. For example, an `eGet` call will be translated into the *accessEvent* method call (8) in Figure 6. Once the event has been processed, the *Interceptor* also searches in the cache the requested elements as indicated by the Model Connector (3). If they are available in the cache, they are directly returned to the EMF-based tool. Otherwise, the Interceptor passes on the control to the EMF API to continue the normal process.

- An *EMF Model Connector* that translates the OCL expressions in the prefetching and caching rules into lower-level EMF API calls. The results of those queries are stored in the cache, ready for the Interceptor to request them when necessary.

This integration makes event creation and cache accesses totally transparent to the client application. In addition, it does not make any assumptions about the mechanism used to store the models, and therefore, it can be plugged on top of any EMF-based persistence solution.

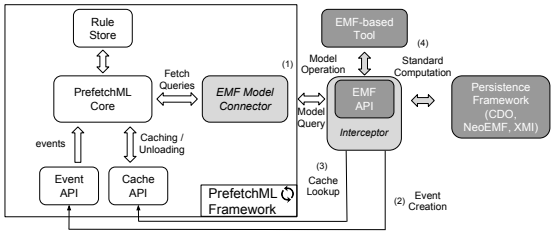


Figure 8: Overview of EMF-Based Prefetcher

5.3 NeoEMF/Graph Integration

To take advantage of the query facilities of graph databases (a proven good alternative to store large models) and make sure PrefetchML optimizes as much as possible the rule execution time in this context, we designed a Model Connector dedicated to NeoEMF/Graph, a persistence solution that stores EMF models into graph databases. Note that, PrefetchML can work with NeoEMF without this dedicated support by processing calls through the EMF API as explained in the previous section. Still, offering a native support allows for better optimizations.

NeoEMF/Graph is a scalable model persistence framework built on top of the EMF that aims at handling large models in graph databases [3]. It relies on the Blueprints API [26], which aims to unify graph database accesses through a common interface. Blueprints is the basis of a stack of tools that stores and serializes graphs, and provides a powerful query language called Gremlin [27]. NeoEMF/Graph relies on a *lazy-loading* mechanism that allows the manipulation of large models in a reduced amount of memory by loading only accessed objects.

The prefetcher implementation integrated in NeoEMF/-Graph uses the same mechanisms as the standard EMF one: it defines an *Interceptor* that captures the calls to the EMF API, and a dedicated *Graph Connector*. While the EMF Connector computes loading instructions at the EMF API level, the *Graph Connector* performs a direct translation from OCL into Gremlin, and delegates the computation to the database, enabling back-end optimizations such as uses of indexes, or query optimizers. The *Graph Connector* caches the results of the queries (i.e. database *vertices*) instead of the EMF objects, limiting execution overhead implied by object reifications. Since this implementation does not rely on the EMF API, it is able to evaluate queries significantly faster than the standard EMF prefetcher (as shown in our experimental results in Section 6), thus improving the throughput of the prefetching rule computation. Database vertices are reified into EMF objects when they are accessed from the cache, limiting the initial execution overhead implied by unnecessary reifications.

6. EVALUATION

In this Section, we evaluate the performance of our PrefetchML Framework by comparing the performance of executing a set of OCL queries when (i) no prefetching is used, (ii) EMF-based prefetching is active, or (iii) NeoEMF/Graph dedicated prefetching is active. In all three cases, the back-end to store the models to be queried is NeoEMF/Graph. This allows to test all three combinations with the same base configuration. To have a better overview of the performance gains, each prefetching mechanism is tested on two different models and with two cache sizes. We also repeat the process using a good and a bad prefetching plan. The latter aims to evaluate whether non-expert users choosing a wrong prefetching plan (e.g. one that for instance prefetches objects that will never be used since they are not involved in any of the queries / operations in the scenario) could harm tool efficiency a lot. Each query is executed twice on each combination to evaluate the benefit of the cache of PrefetchML in subsequent query executions.

Note that we do not compare our solution with existing tools that can be related to our one because we could not envision a fair comparison scenario. For instance, Moogle [20] is a model search approach that creates an index to retrieve

full models from a repository, where our solution aims to improve performances of queries at the model level. IncQuery [4] is also not considered as a direct competitor because it does not provide a prefetch mechanism. In addition, IncQuery was primarily designed to execute queries against models already in the memory which is a different scenario with a different trade-off.

Experiments are executed on a computer running Fedora 20 64 bits. Relevant hardware elements are: an Intel Core I7 processor (2.7 GHz), 16 GB of DDR3 SDRAM (1600 MHz) and a SSD hard-disk. Experiments are executed on Eclipse 4.5.2 (Mars) running Java SE Runtime Environment 1.7. To run our queries, we set the Java virtual machine parameters `-server` and `-XX:+UseConcMarkSweepGC` that are recommended in the Neo4j documentation.

6.1 Benchmark Presentation

The experiments are run over two large models automatically constructed by the MoDisco [5] Java Discoverer, which is a reverse engineering tool that computes low-level models from Java code. The two models used in the experiments are the result of applying MoDisco discovery tool over two Java projects: the MoDisco plugin itself, and the Java Development Tools (JDT) core plugin. Resulting models contain respectively 80 664 and 1 557 006 elements, and associated XMI files are respectively 20 MB and 420 MB large.

As sample queries on those models to use in the experiment we choose three query excerpts extracted from real MoDisco software modernization use cases:

- **BlockStatement**: To access all statements contained in a block
- **TypeToUnit**: To access a type declaration and navigate to all its imports and comments
- **ClassToUnit**: To extend the TypeToUnit query by navigating the body and field declarations of the input class

The first query performs a simple reference navigation from the input *Block* element and retrieves all its *Statements*. The second query navigates multiple references from the input *Type* element in order to retrieve the *Imports* and *Comments* contained in its *CompilationUnit*, and the third query extends it by adding filtering using the `select` expression, and by navigating the *BodyDeclarations* of the input *Class* element in order to collect the declared variables and fields[3].

Good prefetching plans have been created by inspecting the navigation path of the queries. The context type of each expression constitutes the source of *AccessRules*, and navigations are mapped to target patterns. *Bad* plans contain two types of prefetching and caching instructions: the first ones have a source pattern that is never matched during the execution (and thus should never be triggered), and the second ones are matched but loads and caches objects that are not used in the query.

The queries have been executed using two different cache configurations. The first one is a large MRU cache that can contain up to 20% of the input model (*C1*), and the second is a smaller MRU cache that can store up to 10% of the input model (*C2*). We choose this cache replacement policy according to Chou and DeWitt [7] who state MRU is the best replacement algorithm when a file is being accessed in a looping sequential reference pattern. In addition, we compare execution time of the queries when they are executed for the first time and after a warm-up execution to consider the impact of the cache on the performance.

[3]Details of the queries can be found at https://github.com/atlanmod/Prefetching_Caching_DSL

Table 1: Experimental Set Details (MoDisco)

Query	#Input	#Traversed	#Res
BlockStatement	1837	4688	2851
TypeToUnit	348	1895	1409
ClassToUnit	166	3393	2953

Table 2: Experimental Set Details (JDT)

Query	#Input	#Traversed	#Res
BlockStatement	58484	199228	140744
TypeToUnit	1663	16387	13496
ClassToUnit	1347	48961	41925

Queries are evaluated over all the instances of the models that conform to the context of the query. In order to give an idea of the complexity of the queries, we present in Tables 1 and 2 the number of input elements for each query (**#Input**), the number of traversed element during the query computation (**#Traversed**) and the size of the result set for each model (**#Res**).

6.2 Results

Table 3 presents the average execution time (in milliseconds) of 100 executions of the presented queries using Eclipse MDT OCL over the JDT and MoDisco models stored in the NeoEMF/Graph persistence framework. Columns are grouped according to the kind of prefetching that has been used. For each group we show the time when using the good plan with first cache size, the good plan with the second cache size and the bad plan with the first cache.

Each cell shows the execution time in milliseconds of the query the first time it is executed (Cold Execution). In this configuration the cache is initially empty, and benefits of prefetching depend only on the accuracy of the plan (to maximize the cache hits) and the complexity of the prefetching instructions (the more complex they are the more time the background process has to advance on the prefetching of the next objects to access). The second result shows the execution time of a second execution of the query (Warmed Execution), when part of the loaded elements has been cached during the first computation.

The correctness of query results has been validated by comparing the results of the different configurations with the ones of the queries executed without any prefetching enabled.

6.3 Discussion

The main conclusions we can draw from these results (Table 3) are

- EMF-based prefetcher improves the execution time of first time computations of queries that perform complex and multiple navigations for both JDT and MoDisco models (*ClassToUnit* query). However, when the query is simple such as *BlockStatement* or only contains in-

Table 3: Query Execution Time in milliseconds (Cold Execution / Warmed Execution)

Model	OCL Query	No Pref.	EMF Pref.			Graph Pref.		
			C1 (20%)	C2 (10%)	Inv	C1 (20%)	C2 (10%)	Inv
MoDisco	BlockStatement	2057/696	2193/169	2145/187	2134/722	1687/238	1688/233	2155/734
	TypeToUnit	1999/598	2065/83	2072/97	2045/615	1337/155	1418/165	2038/643
	ClassToUnit	2703/722	2588/169	2618/188	2798/758	1616/218	1664/233	2787/753
JDT	BlockStatement	15170/8521	16235/318	16700/3044	16180/8868	11688/1288	12446/4256	16234/8638
	TypeToUnit	6624/2267	6822/269	6768/336	6832/2347	5270/685	5338/685	6877/2331
	ClassToUnit	11637/5421	10780/229	10526/250	11946/5687	7716/873	7817/884	11789/5556

dependent navigations such as *TypeToUnit*, the EMF prefetcher results in a small execution overhead since the prefetch takes time to execute and with simple queries it cannot save time by fetching elements in the background while the query is processed.

- EMF-based prefetcher drastically improves the performance of the second execution of the query: an important part of the navigated objects is contained in the cache, limiting the database overhead.
- NeoEMF-based prefetcher is faster than the EMF one on the first execution because queries can benefit from the database query optimizations (such as indexes), to quickly prefetch objects to be used in the query when initial parts of the query are still being executed, i.e. the prefetcher is able to run faster than the computed query. This increases the number of cache hits in a cold setup (and thus the execution time)
- NeoEMF-based prefetcher is slower than the EMF-based one on later executions because it stores in the cache the vertices corresponding to the requested objects and not the objects themselves, therefore extra time is needed to reify those objects using a low-level query framework such as the Mogwaï [10]
- Wrong prefetcher plans are not dangerous. Prefetching does not add a significant execution time overhead and therefore results are in the same order of magnitude as when there is no prefetching at all.
- Too small caches reduce the benefits of Prefetching since we waste time checking for the existence of many objects that due to the cache size are not there any longer generating a lot of cache misses. Nevertheless, even with a small cache we improve efficiency after the initial object load.

To summarize our results, the PrefetchML framework is an interesting solution to improve execution time of model queries over EMF models. The gains in terms of execution time are positive, but results also show that the EMF prefetcher is not able to provide first-time improvement for each kind of query, and additional information has to be taken into account to provide an optimal prefetching strategy, such as the reuse of navigated elements inside a query, or the size of the cache.

7. CONCLUSIONS AND FUTURE WORK

We presented the PrefetchML DSL, an event-based language that describes prefetching and caching rules over models. Prefetching rules are defined at the metamodel level and allow designers to describe the event conditions to activate the prefetch, the objects to prefetch, and the customization of the cache policy. Since OCL is used to write the rule conditions, PrefetchML definitions are independent from the underlying persistence backend and storage mechanism.

Rules are grouped into plans and several plans can be loaded/unloaded for the same model, to represent fetching and caching instructions specially suited for a given specific usage scenario. Some automation/guidelines could be added to help on defining a good plan for a specific use-case in order to make the approach more user-friendly. However, our experiments have shown that even if users choose a bad plan the overhead is really small. The execution framework has been implemented on top of the EMF as well as NeoEMF/Graph, and results of the experiments show a significant execution time improvement compared to non-prefetching use cases.

PrefetchML satisfies all the requirements listed in Section 2. Prefetching and caching rules are defined using a high-level DSL embedding the OCL, hiding the underlying database used to store the model (1). The EMF integration also provides a generic way to define prefetching rules for every EMF-based persistence framework (2), like NeoEMF and CDO. Note that an implementation tailored to NeoEMF is also provided to enhance performance. Prefetching rules are defined at the metamodel level, but the expressiveness of OCL allows to refer to specific subset of model elements if needed (3). In Section 3 we presented the grammar of the language, and emphasized that several plans can be created to optimize different usage scenario (4). Finally, the PrefetchML DSL is a readable language that eases designers' task on writing and updating their prefetching and caching plan (5). Since the rules are defined at the metamodel level, created plans do not contain low-level details that would make plan definition and maintenance difficult.

As future work we plan to work on the automatic generation of PrefetchML scripts based on static analysis of available queries and transformations for the metamodel we are trying to optimize. Another information source to come up with prefetching plans is the dynamic discovery of frequent access patterns at the model level (e.g. adapting process mining techniques). This is a second direction we plan to explore since it could automatically enhance existing applications working on those models even if we do not have access to their source code and/or no prefetching plans have been created for them. Adding an adaptive behavior to PrefetchML may also allows to detect if a plan is relevant for a given scenario, and switch-on/off specific rules according to the context of the execution.

8. REFERENCES

[1] S. Azhar. Building information modeling (BIM): Trends, benefits, risks, and challenges for the AEC industry. *Leadership and Management in Engineering*, pages 241–252, 2011.

[2] K. Barmpis and D. Kolovos. Hawk: Towards a scalable model indexing architecture. In *Proc. of BigMDE'13*, pages 6–9. ACM, 2013.

[3] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay. Neo4EMF, a Scalable Persistence Layer for EMF Models. In *Proc. of the 10th ECMFA*, pages 230–241. Springer, 2014.

[4] G. Bergmann, Á. Horváth, I. Ráth, D. Varró, A. Balogh, Z. Balogh, and A. Ökrös. Incremental evaluation of model queries over EMF models. In *Proc. of the 13th MoDELS Conference*, pages 76–90. Springer, 2010.

[5] H. Bruneliere, J. Cabot, G. Dupé, and F. Madiot. MoDisco: A model driven reverse engineering framework. *IST*, pages 1012 – 1032, 2014.

[6] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A study of integrated prefetching and caching strategies. *ACM SIGMETRICS Performance Evaluation Review*, pages 188–197, 1995.

[7] H.-T. Chou and D. J. DeWitt. An evaluation of buffer management strategies for relational database systems. *Algorithmica*, pages 311–336, 1986.

[8] K. M. Curewitz, P. Krishnan, and J. S. Vitter. Practical prefetching via data compression. In *ACM SIGMOD Record*, pages 257–266. ACM, 1993.

[9] G. Daniel, G. Sunyé, A. Benelallam, and M. Tisi. Improving memory efficiency for processing large-scale models. In *Proc. of BigMDE'14*, pages 31–39. CEUR Workshop Proceedings, 2014.

[10] G. Daniel, G. Sunyé, and J. Cabot. Mogwaï: a framework to handle complex queries on large models. In *Proc. of the 10th RCIS Conference*. IEEE, 2016.

[11] Eclipse Foundation. The CDO Model Repository (CDO), 2016. URL: http://www.eclipse.org/cdo/.

[12] M. Eysholdt and H. Behrens. Xtext: implement your language faster than the quick and dirty way. In *Proc. of OOPSLA'10*, pages 307–309, New York, NY, USA, 2010. ACM.

[13] A. Gómez, G. Sunyé, M. Tisi, and J. Cabot. Map-based transparent persistence for very large models. In *Proc. of the 18th FASE Conference*. Springer, 2015.

[14] T. Hartmann, A. Moawad, F. Fouquet, G. Nain, J. Klein, and Y. Le Traon. Stream my models: reactive peer-to-peer distributed models@ run. time. In *Proc. of the 18th MoDELS Conference*, pages 80–89. IEEE, 2015.

[15] J. Hutchinson, M. Rouncefield, and J. Whittle. Model-driven engineering practices in industry. In *Proc of the 33rd ICSE*, pages 633–642. IEEE, 2011.

[16] A. C. Klaiber and H. M. Levy. An architecture for software-controlled data prefetching. In *ACM SIGARCH Computer Architecture News*, pages 43–53. ACM, 1991.

[17] M. Koegel and J. Helming. EMFStore: a model repository for EMF models. In *Proc. of the 32nd ICSE*, pages 307–308. ACM, 2010.

[18] D. S. Kolovos, R. F. Paige, and F. A. Polack. The epsilon object language (EOL). In *Proc. of the 2nd ECMDA-FA*, pages 128–142. Springer, 2006.

[19] D. S. Kolovos, L. M. Rose, N. Matragkas, R. F. Paige, E. Guerra, J. S. Cuadrado, J. De Lara, I. Ráth, D. Varró, M. Tisi, et al. A research roadmap towards achieving scalability in model driven engineering. In *Proc. of BigMDE'13*, pages 1–10. ACM, 2013.

[20] D. Lucrédio, R. P. d. M. Fortes, and J. Whittle. Moogle: A model search engine. In *Proc. of the 11th MoDELS Conference*, pages 296–310. Springer, 2008.

[21] P. Mohagheghi, M. A. Fernandez, J. A. Martell, M. Fritzsche, and W. Gilani. MDE adoption in industry: challenges and success criteria. In *Proc. of Workshops at MoDELS 2008*, pages 54–59. Springer, 2009.

[22] J. E. Pagán, J. S. Cuadrado, and J. G. Molina. Morsa: A scalable approach for persisting and accessing large models. In *Proc. of the 14th MoDELS Conference*, pages 77–92. Springer, 2011.

[23] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. *Informed prefetching and caching*. ACM, 1995.

[24] R. Pohjonen and J.-P. Tolvanen. Automated production of family members: Lessons learned. In *Proc. of PLEES'02*, pages 49–57. IESE, 2002.

[25] A. J. Smith. Sequentiality and prefetching in database systems. *TODS*, pages 223–247, 1978.

[26] Tinkerpop. Blueprints API, 2016. URL: blueprints.tinkerpop.com.

[27] Tinkerpop. The Gremlin Language, 2016. URL: gremlin.tinkerpop.com.

[28] J. Warmer and A. Kleppe. Building a flexible software factory using partial domain specific models. In *Proc. of the 6th DSM Workshop*, pages 15–22. University of Jyvaskyla, 2006.

Ingénierie itérative des langages : Expliciter l'implicite

Marc Pantel, Arnaud Dieumegard, Andres Toom, Xavier Crégut

Les modèles et les programmes contiennent souvent des informations implicites issues du domaine métier des concepteurs. En effet, les langages de modélisation et de programmation exploités par ces concepteurs ne permettent souvent pas d'exprimer explicitement ces informations en dehors des annotations et commentaires en langage naturel. Cela est souvent vrai pour la sémantique de ces langages qui n'est décrite formellement que sous la forme du code sources des outils exploitant ces langages. Pour rendre explicite les connaissances métiers, les langages utilisés par les concepteurs doivent être étendus pour fournir l'expressivité nécessaire. Pour permettre ces extensions, il faut souvent expliciter leur sémantique. Nous proposons une définition itérative des langages pour expliciter d'abord la syntaxe et sémantique du langage support puis les différentes extensions nécessaires de manière itératives. Nous proposons de donner les éléments nécessaires aux concepteurs dans le langage support pour qu'il puisse étendre les langages de la manière la plus appropriée à l'expression de leurs connaissances. Cette présentation repose sur l'exemple du langage de spécification de librairies de blocs (Block Library Specification Language) qui permet d'expliciter la sémantique de langages de flots de données tel Simulink et permet aux concepteurs d'enrichir le langage par itération.

Session commune au groupe de travail MTV^2 et à AFADL

Méthodes de test pour la validation et la vérification — Approches Formelles dans l'Assistance au Développement de Logiciels

Domestiquer la variété des critères de test avec le langage HTOL et l'outil LTest *

Michaël Marcozzi, Sébastien Bardin,
Nikolai Kosmatov, Virgile Prevosto et Mickaël Delahaye
CEA, LIST, Laboratoire de Sûreté des Logiciels, PC 174, 91191, Gif-sur-Yvette, France

Contexte. Automatiser le test en boîte blanche est un sujet majeur en ingénierie du logiciel. Au cours des années, de nombreux outils ont ainsi été développés pour supporter les différentes parties du processus de test. Ces outils se basent implicitement ou explicitement sur un critère de couverture de code pour guider l'automatisation. Le critère spécifie formellement quels sont les objectifs de test. Ceux-ci peuvent alors être utilisés automatiquement pour mesurer la qualité d'une suite de tests et pour piloter la génération de tests pertinents. Dans les domaines régulés, comme l'aéronautique, tester le logiciel selon un critère de test précis peut être une exigence normative stricte. Dans les autres domaines, l'utilisation de critères de test performants est reconnue comme une bonne pratique de développement et comme un ingrédient clé pour le développement dirigé par le test.

Problème. Des dizaines de critères de couverture de code ont été proposés dans la littérature, de la couverture de branches au test par mutation, offrant notamment des ratios différents entre l'effort et la minutie du test. Cependant, ces critères ont toujours été vus comme des guides d'automatisation très différents, si bien que la plupart des outils de test sont limités à un petit sous-ensemble prédéfini de critères. En conséquence, la grande variété et la profonde sophistication des critères académiques de test est à peine exploitée dans l'industrie.

Objectif et défis. L'objectif des deux articles résumés ici est de combler le fossé entre le large corpus de travail académique sur les critères de couverture de code d'une part, et leur usage limité dans l'industrie d'autre part. En particulier, le premier article [1] vise à proposer un mécanisme de spécification unificateur pour ces critères, permettant une séparation claire entre déclaration précise des objectifs de test d'une part, et automatisation du test guidée par ces objectifs d'autre part. Cette approche est ambitieuse car le mécanisme proposé doit à la fois être bien défini, suffisamment expressif pour encoder les objectifs de test de la plupart des critères existants et suffisamment souple pour servir de base à l'automatisation. L'objectif du second article [2] est de monter comment ce mécanisme de spécification peut être techniquement placé au cœur d'un outil d'automatisation de test, qui ne soit plus limité à un sous-ensemble de critères, mais bien proprement générique et uni-

*Ce travail a été partiellement financé par l'ANR (grant ANR-12-INSE-0002).

versel. L'outil doit pouvoir automatiser tous les aspects du test (génération de tests, mesure de couverture, détection d'objectifs incouvrables) pour du code réel.

Proposition. Nous introduisons HTOL (Hyperlabel Test Objective Language), un langage de spécification générique pour les objectifs de test en boîte blanche. Les hyperlabels de HTOL sont une extension des labels, développés précédemment par notre équipe au sein de l'outil de test LTest. Bien que capables d'encoder une grande variété de critères, les labels sont trop limités en terme d'expressivité, et l'utilisation d'hyperlabels est nécessaire pour exprimer des critères importants, comme MCDC. De plus, les hyperlabels sont suffisants pour exprimer tous les critères de la littérature (sauf mutations fortes), mais également d'autres objectifs de test intéressants, comme vérifier d'importantes propriétés de sécurité logicielle. Enfin, en comparaison aux approches existantes, les hyperlabels proposent un point d'équilibre entre généricité, spécialisation aux critères de couverture et capacité d'automatisation. L'outil LTest est mis à jour afin de générer des tests, mesurer la couverture ou détecter l'incouvrabilité pour des objectifs de couverture spécifiés avec HTOL, et plus seulement à l'aide de labels. L'efficacité et la capacité de passage à l'échelle de l'outil sont vérifiées sur des programmes open-source standards.

Contributions. Les cinq principales contributions des articles présentés ici sont :
1. Une nouvelle taxonomie des critères de test, orthogonale aux classifications existantes, car sémantique et basée sur la nature des contraintes d'accessibilité sous-jacente aux critères classés. Une représentation visuelle de cette classification est proposée : le cube des critères de couverture.
2. Une syntaxe et une sémantique formelle pour le langage HTOL, qui introduisent des opérateurs permettant de combiner les labels, étendant ainsi leur expressivité des critères à l'origine du cube à l'entièreté des critères portés par celui-ci.
3. Une présentation à la fois pédagogique et exhaustive de comment l'entièreté des critères standards (sauf mutations fortes) peuvent être encodés à l'aide de HTOL.
4. Un rapport sur les avancées techniques réalisées dans LTest pour offrir le support de HTOL et une présentation des nouvelles APIs de l'outil (open source).
5. Des expériences montrant l'efficacité et le passage à l'échelle de LTest, avec des suites de 10000 cas de test sur des programmes comme OpenSSL et SQLite.

Impact potentiel. Les hyperlabels sont une *lingua franca* pour définir, étendre et comparer des critères de test d'une façon clairement documentée. Ils sont aussi un langage de spécification pour écrire des outils de test universels, extensibles et interopérables, comme l'outil LTest, qui rend la variété et la sophistication des critères de test facilement utilisables dans un cadre industriel.

Ce résumé court synthétise les deux articles suivants:

[1] Generic and Effective Specification of Structural Test Objectives

[2] Taming Coverage Criteria Heterogeneity with LTest

acceptés à *10th IEEE Int. Conf. on Software Testing, Verification and Validation*

Un Data-Store pour la Génération de Cas de Test

Loukmen Regainia, Cédric Bouhours et Sébastien Salva
LIMOS - UMR CNRS 6158, Auvergne University, France
Email : [prenom.nom]@uca.fr

Résumé

De nos jours, un grand nombre de documents publiques de sécurité sont disponibles. Ces documents, souvent complexes, exposent les développeurs à la difficulté de choisir des solutions de sécurité appropriées d'une application tout au long de son cycle de vie. Dans cet article, nous proposons une méthodologie, basée sur l'acquisition et l'intégration des données afin de construire un data-store permettant de classifier des patrons de sécurité et des attaques et de générer des Arbres d'attaques de défenses. Nous présentons également une méthodologie qui permet l'extension de ce data-store dans le but de générer automatiquement des cas de test.
Mots clés : Patrons de sécurité ; Patrons d'attaque ; Cas de test ; Arbres d'attaque et de défense

1 Introduction

La conception et l'implémentation d'une application sécurisée est une tâche difficile. D'une part la sécurité dès la phase de conception n'est pas une compétence courante dans les équipes de développement, et d'autre part la détection des problèmes de sécurité aux dernières étapes de la vie de l'application est compliquée à la vue de la difficulté pour couvrir l'ensemble des vulnérabilités connues.

Afin d'aider les équipes de développement à gérer la sécurité des applications dès les premières phases du cycle de vie, les concepteurs ont à leur disposition la notion de patrons de sécurité [1] qui sont des solutions communautaires génériques aux problèmes récurrents de sécurité permettant de prévenir partiellement ou complètement des attaques. La nature abstraite et le nombre croissant des patrons de sécurité permet de couvrir une myriade de problématiques de sécurité indépendamment des contextes d'application. Néanmoins, cette représentation abstraite et le nombre de patrons disponibles rend le choix difficile.

Dans cet article, nous présentons premièrement une classification des patrons de sécurité pour faciliter leurs choix face aux attaques. Cette classification prend en compte plusieurs critères de qualité [2] : la Non-ambiguïté, vu que nous présentons en détail la démarche de classification . Elle prend également en compte la Navigabilité, en utilisant les relations inter-patrons. Puis, Nous présentons une méthode des arbres d'attaques et de défense (ADTree), permettant de présenter les actions d'attaque et de défense,

à partir de la classification obtenue. Nous présentons finalement une méthodologie de génération de squelettes de cas de test permettant de tester si une application est vulnérable aux attaques, et si des patrons de sécurité sont présents dans une application en inspectant la présence des éléments caractérisants la bonne intégration des patrons (section “conséquences” du descriptif des patrons).

Ce papier est structuré comme suit : dans la Section 2, nous présentons les notions de sécurité utilisées pour générer la classification. Ensuite, nous présentons la méthode associant les attaques et les patrons de sécurité ainsi que la génération des ADTree dans la Section 3. Puis nous présentons l'extension de notre démarche dans le but de produire des squelettes de cas de test en Section 4. Enfin, nous concluons et nous proposons quelque perspectives en Section 5.

2 Contexte

Dans le contexte de nos travaux, nous utilisons différentes notions de sécurité telles que les notions des patrons de sécurité, des principes de sécurité [4], des patrons d'attaque [5] et des arbres d'attaques et de défense [3]. Nous en présentons quelque unes dans cette section. Un patron de sécurité est une solution générique à un problème récurrent de sécurité, caractérisée par un ensemble de propriétés structurelles et comportementales à utiliser dès la phase de conception. Un patron de sécurité peut être représenté textuellement et/ou avec des diagrammes UML [1]. Il est également caractérisé par un ensemble de points forts qui décrivent les sous propriétés du patron. Un exemple de patron de sécurité, "Authorisation Enforcer", vise à fournir un mécanisme d'autorisation et de gestion de droits à une application. Il centralise le mécanisme d'autorisation et le découple de la logique fonctionnelle de l'application [6].

Un patron d'attaque est une description des éléments et des techniques utilisées pour attaquer une application, ainsi que les challenges auxquels les concepteurs doivent faire face afin de protéger leur application. La base CAPEC (Common Attack Pattern Enumeration and Classification) offre une documentation des patrons d'attaque. Dans la base CAPEC, un patron d'attaque est composé d'un ensemble de sections, à savoir, les impacts et les conséquences de l'attaque, les contremesures, etc [5]. Pour construire notre data-store, nous nous sommes concentré sur la section *“Attack Execution Flow”*, qui décrit les étapes de l'attaque ainsi que les techniques, les indicateurs et les mesures de sécurité associées à chaque étape. De plus, nous nous sommes intéressés à la section "Attack Prerequisites" et aux ressources nécessaires pour effectuer l'attaque (section "Resources Required").

Parmi les méthodes qui permettent la modélisation des attaques, les arbres d'attaque et de défense (Attack Defense Tree “ADTree”)[3] représentent les mesures qu'un attaquant peut prendre pour attaquer un système ainsi que les défenses qui peuvent être utilisées pour protéger le système. Ce sont des arbres étiquetés où chaque nœud permet de présenter une action d'attaque ou de défense. Chaque nœud peut être raffiné par des nœuds-enfants qui peuvent être associés par des opérateurs “AND”, “OR” et “SAND”. Ce dernier est un opérateur conjonctif qui impose une relation d'ordre entre les actions.

En plus de l'aspect visuel d'un ADTree, il est fournie avec une algèbre qui permet de décrire formellement les étapes d'une attaque ainsi que les défenses associées avec des ADTerms. Un ADTree peut être exprimé avec un ensemble d'ADTerms qui permet de présenter deux types de rôles attaquant (opponent "o") et défenseur (proponent "p") ainsi que trois type d'opérateurs Conjonction ($\wedge^{o/p}$), Disjonction ($\vee^{o/p}$) et Conjonction ordonnée "SAND" ($\overrightarrow{\wedge}^{p/o}$) d'actions de défense ou d'attaque (o/p). La relation entre un nœud père et un nœud fils peut être une relation de *raffinements* si les deux nœuds sont du même type et une relation de *contre-mesure* sinon, on dénote cette dernière par $c^o(a, b)$ entre deux nœuds a, b.

3 La classification des patrons de sécurité et la génération d'ADTree

Les documents présentant des attaques ou des patrons de sécurité sont souvent décrits de façon informelle avec des niveaux différents d'abstraction. Ceci rend la liaison directe entre un patron de sécurité et une attaque souvent ambiguë et sujette à des imprécisions. Dans le but de réduire l'ambiguïté et l'imprécision, nous proposons une méthode qui a pour but la construction d'un data-store qui rassemble les attaques, issues de la base CAPEC (*Common Attack Patterns Enumeration and Classification*)[5], les patrons de sécurité, les principes de sécurité, et les relations inter-patrons issus d'un catalogue de patrons [1]. Ce data-store permet la génération d'arbres d'attaques et de défenses (ADTree) pour chaque attaque. La méthode, illustrée sur la figure 1.A, se déroule en quatre étapes.

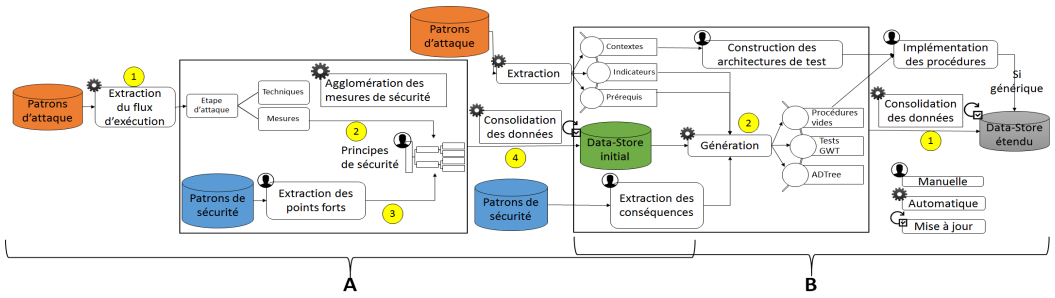


Figure 1 – Méthodologie

1- Extraction du flux d'exécution des attaques : depuis la base des patrons d'attaques CAPEC nous effectuons une extraction automatique de l'ensemble ordonné des étapes de chaque attaque, qui est défini dans la section *"Attack Execution Flow"* de la documentation des attaques. Cette section décrit les étapes que l'attaquant doit effectuer dans l'ordre ainsi que les différentes techniques qu'il doit exécuter pour satisfaire une étape, les indicateurs de succès de l'étape ainsi que des mesures de sécurité qui doivent protéger l'application. Nous avons extrait automatiquement 209 étapes différentes pour 215 attaques sachant que les attaques peuvent partager des étapes.

2- Extraction des principes de sécurité de chaque étape de l'attaque : chaque étape d'attaque est caractérisée par un ensemble de mesures de sécurité *"Security Controls"* (Détectives, Préventives et Correctives). Nous en avons extrait un ensemble de 217. Chaque mesure suit un principe élémentaire de sécurité (Autorisation, Défense en profondeur, Audit, etc) dont l'extraction manuelle peut être fastidieuse et rendre l'approche difficile à refaire. Nous avons utilisée une méthode issue du Data-mining avec l'outil KHCoder [1] pour agglomérer les mesures de sécurité selon leur appartenance au même ensemble de principes de sécurité. Les 217 mesures de sécurité ont été assemblées dans 21 groupes (clusters) cloisonnés. Cette méthode d'analyse de texte s'appuie sur trois notions statistiques :

- L'utilisation de "Stanford POS tagger" afin d'ordonner les mots clés *(log, input, credentials, etc.)* par rapport à leur fréquence dans le texte ainsi que leur type *(verbe, nom, adjectif)* ;
- Compte tenu des fréquences des mots clés, une matrice de distances entre les mesures de sécurité est établie moyennant la méthode "Jaccard". La distance entre deux mesures de sécurité est minimisée si les deux mesures contiennent plus de mots clés en commun ;
- Sur la base de la matrice de distances obtenue, nous avons groupé hiérarchiquement les mesures de sécurité moyennant la méthode "Ward" [7]. L'avantage de cette méthode est qu'elle privilégie la construction de niveaux de groupes étape par étape au lieu de la construction de grands groupes, qui peuvent couvrir beaucoup de principes de sécurité. La méthode "Ward" est une méthode supervisée imposant que le nombre de groupes soit choisi manuellement.

Concrètement, nous avons choisi le nombre de groupes, après une série de tests, de telle façon que les éléments d'un même groupe traitent de la même problématique de sécurité. basés sur un ensemble de travaux traitant de la notion de principes de sécurité [8, 9, 4], nous avons opté pour une organisation hiérarchique de 66 principes de sécurité afin d'améliorer leur compréhension et présenter les relations inter-principes.

Les éléments d'un groupe dans la classification obtenue sont proches dans le sens où ils traitent de la même problématique de sécurité. Cela facilite l'extraction manuelle des principes de sécurité traités par chaque groupe de mesures de sécurité. Nous avons ensuite lié chaque groupe de mesures de sécurité aux principes de sécurité qu'ils couvrent.

3- Association des principes de sécurité et des patrons de sécurité : Un patron de sécurité est caractérisé par un ensemble de propriétés appelées points forts définissant les raisons pour lesquelles un patron est une solution adaptée à un problème [10]. Les patrons de sécurité peuvent partager des points forts. Chaque point fort est caractérisé par un concept de sécurité que nous lions à un principe de sécurité.

4- Consolidation des données, Extraction de la classification et Génération des ADTree : afin de consolider les données, nous avons développé une série de flux de données sous "Talend" [2] qui alimentent le data-store suivant les associations établies dans les étapes précédentes.

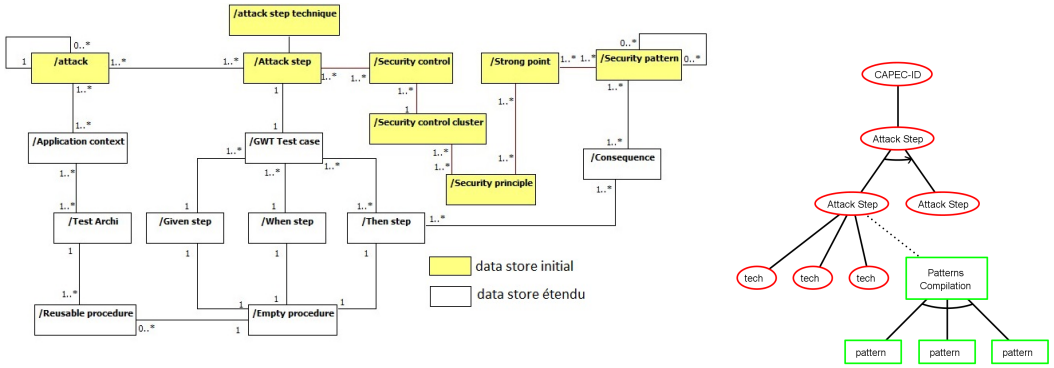
1. https ://sourceforge.net/projects/khc/
2. http ://fr.talend.com/

Le data-store obtenu contient l'ensemble des données nécessaires pour l'extraction des différents types de relations et de classifications. Le data-store exprime les liens entre les étapes des attaques et les principes de sécurité, les liens entre l'ensemble des patrons de sécurité et les principes de sécurité, ainsi que les relations inter-patrons de sécurité. Par conséquent, le data-store permet d'extraire, pour chaque attaque :

- les informations concernant l'attaque (identifiant, nom, présentation) ;
- l'ensemble ordonné des étapes et des sous-étapes ;
- les techniques qui permettent d'implémenter chaque étape ;
- les patrons de sécurité liés à chaque étape ainsi que les relations inter patrons ;

Nous avons utilisé ZOHO Reports [3] afin de rendre automatique l'extraction et la visualisation de la classification depuis le data-store sous la forme d'un tableau de bord interactif, ceci est disponible en ligne [11].

Dans le but de faciliter l'utilisation et l'analyse des ADTree, nous avons également développé un outil qui génère un ADtree pour chaque attaque présentée dans le data-store, la figure 2b illustre la forme générale des ADTree générés. Cet outil lit les informations concernant les étapes, les techniques, et les patrons de sécurité depuis le data-store et les présente sous la forme visuelle d'ADTree. Pour ce faire, ces informations sont traduites en un fichier XML que l'utilisateur peut le charger dans l'outil ADTool [4] qui permet l'affichage et l'analyse visuelle d'une attaque.



(a) data-store (b) exemple d'ADTree

Figure 2 – La structure des data-stores

4 L'extension du data-store pour le test

En plus de la classification des patrons de sécurité par attaques et la génération des ADTree, nous avons étendu le data-store dans le but de générer un cas de test Given, When, Then (GWT) pour chaque étape d'attaque. Ces cas de test ont pour objectif

3. https ://www.zoho.eu/reports/ ?src=zoho
4. http ://satoss.uni.lu/members/piotr/adtool/

de tester si une application est vulnérable a une attaque, et si on peut observer les conséquences de patron de sécurité depuis une application.

Un cas de test *GWT* est ainsi composé d'une section *Given*, exprimant les prérequis et les éléments à préparer pour effectuer l'attaque, d'une section *When* présentant les actions à effectuer pour concrétiser l'attaque et d'une ou plusieurs sections *Then* présentant les conditions de succès de l'attaque (verdicts).

Nous avons associé les patrons de sécurité à des conséquences (qui peuvent être partagées entre patrons). Chaque conséquence est aussi associée à une section Then, permettant de vérifier si la conséquence est observable ou non depuis l'application sous test.

Chaque section Given, When, Then est liée à une procédure qui implémente la section du test.

Nous avons généré automatiquement l'ensemble de ces éléments en intégrant les données dans le data-store puis en générant les cas de test et les procédures commentées selon la méthode, illustrée dans la figure 1.B, décrite ci-après :

4.1 Data-store étendu

Depuis la base CAPEC nous avons extrait automatiquement pour chaque attaque, contenu dans la data-store précédant, l'ensemble d'éléments qu'il faut satisfaire préalablement à l'exécution de l'attaque. Les sections *"Attack Prerequisites"* et *"Resources Required"* du CAPEC décrivent ce qu'il faut préparer pour chaque attaque. Nous avons également, depuis la section *"Attack Step Techniques"* de chaque étape, extrait automatiquement les différentes façons dont une étape peut être implémentée. Ainsi que l'ensemble d'éléments qui permettent de décider si l'attaque est bien réussie depuis les sections *"Indicators"* et *"Outcomes"* de chaque étape. Ensuite, nous avons extrait les différents **contextes** dans lesquels une attaque est applicable depuis la partie "Environments" de chaque technique d'étape d'attaque. Chaque contexte implique une **architecture de test** différente regroupant un ensemble d'outils permettant de tester les application de ce contexte.

La présence d'un patron de sécurité dans une application peut être résumée par un ensemble de conséquences sur sa structure et son comportement. Cet ensemble d'informations, présenté dans la section *"Consequences"* de la documentation des patrons de sécurité. Nous avons extrait manuellement ces informations depuis la documentation des patrons de sécurité.

Nous avons généré un cas de test GWT pour chaque étape d'attaque ainsi qu'une procédure pour chaque section Given, When Then. Chacune de ces procédures est complété par des directives, sous forme de commentaires. Les directives fournies sont issues du data-store et sont réparties comme suit :

— pour les procédures liées aux sections Given, les directives sont extraites depuis les pré-requis et les ressources nécessaires de l'attaque ;
— pour les procédures When, les directives sont extraites depuis les techniques de chaque étape ;
— pour les procédures Then, on distingue deux cas

1. pour les assertions de succès de l'étape, les directives sont issues des indicateurs de succès de chaque étape ;
2. pour les assertions d'observations de présence des patrons de sécurité, les directives sont issues des informations concernant les conséquences des patrons de sécurité ;

Finalement, nous avons consolidé l'ensemble de ces éléments dans un data-store qui contient les informations nécessaires permettant, en plus de la classification des patrons de sécurité et des attaques, la générations des cas de test comme illustré en figure 1.

4.2 Génération des squelettes de cas test

Depuis le data-store que nous avons étendu, nous générons un ADTree pour chaque attaque donnée par un utilisateur.

De façon résumé, la génération des cas de test depuis le data-store s'effectue comme suit :

1. L'utilisateur fournit une liste d'attaques. Un outil génère un ADTree par attaque depuis le data-store ;
2. L'utilisateur choisis depuis chaque ADTree une conjonction de patrons de sécurité selon le contexte de l'application.
3. Les ADTree peuvent être exprimés formellement par des expressions appelées ADTerm. A partir des ADTree générés, les ADTerms sont des expressions sur des éléments de la forme $c^o(st, \wedge^p(sp_1, \ldots, sp_m))$ qui exprime st une étape d'attaque ainsi que $\wedge^p(sp_1, \ldots, sp_m))$ une conjonction de patrons de sécurité.
4. chaque BADSeq est automatiquement extrait depuis l'arbre et traduit en un cas de test GWT. Chaque cas de test présente une étape d'attaque avec une section Given, une section When et une section Then ainsi qu'une section Then pour chaque conséquence des patrons de sécurité lié à l'étape ;
5. Chacune des sections des cas de test GWT est implémenté par une procédure. Les squelettes de ces procédures sont générées automatiquement et complétées par des directives, sous formes de commentaires, afin d'aider le développeur dans l'implémentation. Avec le recours d'un ensemble d'outils de tests d'intrusion (*pen-testing tools*), un ensemble d'étapes *Given, When, Then* peut être pré-implémenté et rendu directement utilisable pour les concepteurs. Dans le contexte des applications web, nous en avons codés manuellement 28 cas de test ainsi que les verdicts sur la présence des conséquences de 11 patrons de sécurité. Ces cas de test peuvent être réutilisable de façon générique sur les applications web.
6. L'exécution des cas de test produit deux types de verdicts, si l'application est vulnérable aux attaques et si on peut observer des conséquences de patrons depuis l'application.

5 Conclusion

Nous avons présenté une classification associant les patrons de sécurité et les attaques. Nous avons également présenté la méthode de génération d'ADTree pour chaque attaque. Notre méthodologie a permis la construction d'un data-store contenant 215 attaques, 209 étapes, 217 mesures de sécurité et 448 techniques d'attaque, qui est disponible en ligne [11]. Nous avons également introduit une méthode de génération de squelettes de cas de test permettant de tester les attaques ainsi que les conséquences observables des patrons de sécurité et les directives permettant aux concepteurs de concrétiser ces tests. Les 209 étapes d'attaques générée automatiquement sont liée à un ensemble de 627 squelettes de procédures commentées avec des aidant le concepteur à les compléter. Parmis cet ensemble de 209 cas de test nous en avons complété 28 étapes avec 84 procédure que l'utilisateur peut directement utiliser dans le contexte des applications web.

Références

[1] K. Yskout, T. Heyman, R. Scandariato, and W. Joosen, "A system of security patterns," 2006.

[2] K. Alvi, Aleem and M. Zulkernine, "A Comparative Study of Software Security Pattern Classifications," *2012 Seventh International Conference on Availability, Reliability and Security*, pp. 582–589, 2012.

[3] B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer, "Attack–defense trees," *Journal of Logic and Computation*, p. exs029, 2012.

[4] J. Meier, "Web application security engineering," *Security & Privacy, IEEE*, vol. 4, no. 4, pp. 16–24, 2006.

[5] Mitre corporation, "Common attack pattern enumeration and classification, url :https ://capec.mitre.org/," 2015.

[6] C. Steel, *Core Security Patterns : Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.

[7] P. Willett, "Recent trends in hierarchic document clustering : a critical review," *Information Processing & Management*, vol. 24, no. 5, pp. 577–597, 1988.

[8] J. Viega and G. McGraw, *Building Secure Software : How to Avoid Security Problems the Right Way, Portable Documents*. Pearson Education, 2001.

[9] J. Scambray and E. Olson, *Improving Web Application Security*. 2003.

[10] C. Bouhours, *Détection, Explications et Restructuration de défauts de conception : les patrons abîmés*. PhD thesis, l'Université Toulouse III – Paul Sabatier, 2010.

[11] "Security pattern classification "http ://regainia.com/research/database.html"."

Un outil d'assistance à la construction de tests de modèles à composants et services

André, Pascal Ardourel, Gilles Mottu, Jean-Marie
Sunyé, Gerson

LS2N UMR CNRS 6004
Université de Nantes, IMT-Atlantique, Inria
Prenom.Nom@univ-nantes.fr

Dans l'article *"COSTOTest : A Tool for Building and Running Test Harness for Service-Based Component Models"* publié dans les proceedings de la conférence internationale *ISSTA 2016* [1] et présenté en session démonstration, nous décrivons comment l'outil COSTOTest nous assiste pour tester directement les modèles de composants logiciels basés sur les services. Ces travaux concernent la vérification de systèmes logiciels à base de composants et services (*Service-based Component* ou SBC) et exploitent l'ingénierie dirigée par les modèles (IDM).

Contexte Tester le plus tôt possible permet de réduire le coût du processus de vérification et de validation [2]. En IDM, c'est encore plus vrai et la correction des modèles est essentielle car ils sont le point de départ de transformations plus ou moins directes vers les modèles opérationnels et le code. Alors que les modèles servent de spécification dans les approches *Model-Based Testing* (MBT) pour générer des tests exécutés sur le code final, nous considérons la vérification de ces modèles qui peuvent être erronés [3]. Nous contribuons donc à la vérification des modèles en les testant directement (*Model Testing* MT). Ces tests sont complémentaires d'autres vérifications appliquées sur les modèles (preuves, model checking [4]). L'effort est porté sur la détection en amont des erreurs "métiers" (*platform independent*), coûteuses à corriger lorsqu'elles sont repérées tardivement et propagées dans différentes versions et implantations du système. De plus, le test de modèles permet de s'affranchir des erreurs spécifiques à l'implantation (*platform specific*), et réduit la complexité globale du test [5]. En exploitant l'IDM, les tests sur les modèles sont évolutifs car ils sont eux-mêmes des modèles. On bénéficie ainsi des mêmes outils de transformation de modèles que le système sous test.

Objectifs Nous ciblons les tests fonctionnels (unitaire, intégration, recette fonctionnelle hors IHM) pour des modèles à composants et services, ayant un niveau de description suffisamment précis et détaillé pour pouvoir exécuter les tests. Assurer la correction de ces modèles reste un défi. La vérification formelle permet de filtrer

une partie des modèles erronés mais elle ne suffit pas parce que les outils restent limités face à la combinatoire de langages expressifs. Nous considérons le test pour améliorer le niveau de correction. Notre objectif est de tester ces modèles à composants c'est-à-dire de concevoir des cas de tests, de les appliquer sur les modèles mis dans un contexte adéquat pour être exécutés et obtenir un verdict.

Processus Le processus part d'une intention de test (variables et oracles) et d'un modèle sous test. Nous considérons la construction de *harnais de test* unitaire ou d'intégration pour des systèmes à composants et services, qui permettent de passer des données de test, d'exécuter les tests, et de récupérer le verdict (succès ou échec du test). Pour réduire l'effort de construction du harnais de test, nous proposons une méthode qui guide le testeur dans le processus de conception des tests. Sachant que le testeur découvre l'application à tester et que les intentions de test restent informelles, il a besoin d'itérer jusqu'à ce que le niveau de précision soit atteint pour que le système soit testable. Il a besoin d'être assisté pour définir la forme véritable du test et déterminer la partie du système nécessaire au test. Définir concrètement l'oracle et comment trouver ou fournir les données de test est difficile : par exemple, pour atteindre une variable, il faut trouver les services qui la manipulent ou qui en dépendent. L'assistance à la construction est basée sur la détection d'incohérences et d'incomplétude entre le harnais et le modèle de test ainsi que sur des propositions générant les éléments manquants. Le programme de test est alors transformé vers une plateforme technique dédiée à l'exécution des tests.

Mise en œuvre La mise en œuvre est réalisée avec un outillage qui permet d'expérimenter l'approche. Il est mis en œuvre dans COSTO (*COmponent Study Toolbox*), la plate-forme Eclipse (`http://costo.univ-nantes.fr/`) dédiée aux modèles à composants écrits avec le DSL Kmelia. Nous avons développé un *framework* en Java qui donne un cadre d'exécution et d'animation pour les modèles de test (plateforme spécifique). L'approche est illustrée sur un cas d'étude simple, un système de *platoon* de véhicules.

Références

[1] Pascal André, Jean-Marie Mottu, and Gerson Sunyé. Costotest : a tool for building and running test harness for service-based component models (demo). In Andreas Zeller and Abhik Roychoudhury, editors, *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 437–440. ACM, 2016.

[2] Graeme Shanks, Elizabeth Tansley, and Ron Weber. Using ontology to validate conceptual models. *Commun. ACM*, 46(10) :85–89, October 2003.

[3] Martin Gogolla, Jørn Bohling, and Mark Richters. Validating uml and ocl models in use by automatic snapshot generation. *Software and Systems Modeling*, 4(4) :386–398, 2005.

[4] Pascal André, , Christian Attiogbé, and Jean-Marie Mottu. Combining techniques to verify service-based components. In *Proceedings of the International Workshop on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn, AMARETTO@MODELSWARD 2017, Porto, Portugal, February 19-21, 2017.*, pages 645–656, 2017.

[5] Marc Born, Ina Schieferdecker, Hans-gerhard Gross, and Pedro Santos. Model-driven development and testing - a case study. In *First European Workshop on MDA with Emphasis on Industrial Application*, pages 97–104. Twente Univ., 2004.

Model Based Testing : maximiser la couverture structurelle de tests fonctionnels

Yanjun SUN, Gérard MEMMI and Sylvie VIGNES
LTCI, Télécom ParisTech, Université Paris-Saclay, 75013, Paris, France

Le problème que nous avons traité dans l'article de QRS MVV [1] concerne le renforcement de la génération de suites de tests fonctionnels à exécuter sur un modèle de façon à assurer une très haute couverture structurelle du modèle.

Le domaine d'application concerne le prototypage virtuel d'un système de contrôle commande de centrale nucléaire. Le contrôle commande est constitué de plusieurs centaines de systèmes élémentaires (SE) en charge de la protection et de la supervision du procédé physique. Le gain espéré de la démarche est de raccourcir les délais de production des études en garantissant le haut niveau de qualité et en bénéficiant de façon complémentaire des approches formelles, des tests et des simulations. Le contexte du système existant initialement développé par des automaticiens a conduit assez naturellement au choix de modéliser en Esterel Scade.

Un modèle Scade est formellement représenté par un ensemble hiérarchisé de blocs connectés offrant chacun et donc globalement des interfaces bien définies en termes de ports d'entrée/sortie. Chaque bloc décrit un comportement par une Machine à états finis étendue (EFSM) ; les interfaces sont des paires de vecteurs d'entrée et de sortie contraintes par les connections entre blocs à chaque pas de temps. La relation de transition d'un bloc s'exprime par des expressions booléennes et arithmétiques sur des vecteurs d'entrée/sortie aux valeurs booléennes ou numériques. Le processus de génération de suites de tests présenté dans l'article part d'un ensemble de tests exécutés sur le modèle Scade dont on mesure le critère de couverture par exemple MC/DC. Ce modèle étant transformé en Lustre, le processus permet d'augmenter la couverture en utilisant le model checker GATeL pour générer des tests pour des branches non initialement atteintes.
Dans cet article, nous présentons les résultats de validation fonctionnelle d'un SE du Contrôle Commande SRI, Système de Refroississement Intermédiaire.

[1] Y. Sun, G. Memmi and S. Vignes, "Model-Based Testing Directed by Structural Coverage and Functional Requirements," 2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), Vienna, 2016, pp. 284-291.

Session du groupe de travail GLACE

Génie Logiciel pour les systèmes Cyber-physiquEs

InS3PECT : Ingénierie Système de Services Sécurisés Pour objEts ConnecTés

Auteur : Frédéric Mallet (I3S)

Résumé :
L'ingénierie de services sécurisés pour objets connectés nécessite la prise en compte de contraintes très diverses (autonomies énergétiques des objets, capacité de traitement, confidentialité des données véhiculées, sécurité, coût, ...). Les compétences requises sont multidisciplinaires. L'aspect sécurité est transversal et omniprésent dès lors qu'on rassemble des objets communicants dans des infrastructures ouvertes, à large échelle. Il s'agit ici d'aborder la modélisation de tels systèmes mais aussi leur vérification et la conformité des produits vis à vis du niveau d'exigence requis.

Le projet InS3PECT [1] a comme objectif de cartographier les acteurs et compétences et d'identifier les verrous technologiques et scientifiques, pour une approche système allant de la conception des objets et de leurs services jusqu'à leur déploiement et mise en œuvre dans des infrastructures dynamiques, largement distribuées.

Le consortium actuel est ouvert aux chercheurs intéressés par les problématiques associées et les verrous identifiés par le consortium (`https://www.i3s.unice.fr/ins3pect`)

1. PEPS CNRS-INS2I Thème Objets Communicants Algorithmes, Architectures et Applications

Un regard synchrone sur la bibliothèque standard de Simulink

Auteur : Marc Pouzet (UPMC)

Résumé :
Les modeleurs hybrides tels que Simulink sont aujourd'hui très utilisés pour concevoir et réaliser du logiciel de controle embarqué en interaction avec un environnement physique. Ils disposent d'une librairie de blocs à temps discret et à temps continu qui est utilisée partout. Cependant, ces blocs ne sont pas définis ou programmés en fonction de blocs élémentaires dont chacun aurait une définition précise mais ils sont implémentés directement sous forme de S-function (en C) et décrits de manière informelle seulement. Peut-on identifier un ensemble minimal et orthogonal de constructions qui soit suffisamment expressif pour définir l'ensemble des blocs de la librairie standard, dans un langage mathématiquement précis et compilable vers du code séquentiel efficace ? Dans cet exposé, nous montrerons qu'un ensemble conséquent de blocs de la bibliothèque standard de Simulink peut être programmé dans un langage formellement défini, purement fonctionnel, reposant sur une sémantique synchrone et combinant à la fois des équations de suites et des équations différentielles ordinaires (ODE). Certains blocs ne peuvent être exprimés et seront rejetés par typage parce qu'ils combinent des signaux discrets et continus sans discipline et conduisent à des modèles non déterministes et fragiles. L'expérience a été conduite avec deux langages, Zélus, une extension de Lustre avec des ODEs et le prototype industriel SCADE Hybrid. Ce dernier est une extension conservatrice de SCADE—les modèles existant à temps discret sont compilés sans modification—avec des constructions pour combiner des signaux à temps discret et à temps continu.

Monitoring the oceans with autonomous floats

Auteur : Guust Nolet (GeoAzur)

Résumé :
Seismographs around the world give us an important record of seismicity, and also serve to image temperature variations at depth using waves that traverse the interior of our planet. However, these instruments are invariably located on land. To fill in the gap, we developed a robot with a hydrophone that floats at depth in the oceans and surfaces whenever a strong earthquake signal is observed to transmit a seismogram in quasi-real time by satellite. It also records temperature and salinity. The latest version can handle up to eight different instruments. These may compete with each other, having different priorities of recording, storing and transmitting data. In this presentation I will summarize some of the results already obtained for seismology,and give an overview of other possible uses to monitor the oceanic environment, whichinclude acoustic monitoring, census of whales and dolphins, and geochemical measurements.

Session commune au groupe de travail LTP et à AFADL

Langages, Types et Preuves — Approches Formelles dans l'Assistance au Développement de Logiciels

ELIOM : Un langage ML pour la programmation Web sans tiers

Gabriel RADANNE [a], Jérôme VOUILLON [a,b,c], et Vincent BALAT [a,b]

[a]IRIF UMR 8243 CNRS, Univ Paris Diderot, Sorbonne Paris Cité
[b]BeSport
[c]CNRS

Résumé

ELIOM est un dialecte d'OCAML pour la programmation Web qui permet, à l'aide d'annotations syntaxiques, de déclarer code client et code serveur dans un même fichier. Ceci permet de construire une application complète comme un unique programme distribué dans lequel il est possible de définir des widgets faciles à composer avec des comportements à la fois client et serveur. Notre langage expose également un modèle de communication simple et sûre via un typage statique fort. ELIOM correspond aux spécificités de la programmation Web en permettant de mélanger code client et serveur tout en maintenant une communication unidirectionnelle et efficace. Le langage ELIOM est suffisamment minimaliste pour être implémenté comme extension d'un langage existant, et suffisamment expressif pour implémenter la plupart des idiomatismes du Web.

Les applications Web traditionnelles sont composées de différents tiers : les pages Web sont écrites en HTML et CSS, ces pages peuvent être produites par un langage quelconque : PHP, Ruby, C++, ... Le comportement dynamique est contrôlé par un langage client tel que JAVASCRIPT. La manière usuelle de composer ces différents tiers est d'écrire un programme client et un programme serveur distincts. Le programmeur doit alors respecter une interface commune entre les deux programmes. Cette contrainte n'est généralement pas vérifiée automatiquement et doit être respectée par le programmeur. Ceci est évidement sujet à erreurs et est la cause de nombreux bugs dans les applications Web.

L'un des buts des frameworks modernes de programmation Web client-serveur est d'offrir la possibilité de créer des pages Web dynamiques d'une manière *composable*. Le programmeur devrait pouvoir définir *sur le serveur* une fonction qui créer un fragment de page, ainsi que le comportement associé *du coté client*. Les langages de programmation sans tiers ont pour but de résoudre ces problèmes de modularité en permettant au programmeur de mêler librement code client et serveur. Pour la plupart de ces langages, deux parts sont extraites d'un programme : une part est exécutée sur le serveur tandis que l'autre est compilée vers JAVASCRIPT et exécutée sur le client. Ce paradigme de programmation permet de mélan-

ger code client et serveur librement tout en fournissant de forte garanties sur les communications client-serveur ainsi qu'une notion très fine de composition.

ELIOM est un dialecte d'OCAML pour la programmation Web sans tiers qui supporte des interactions client-server statiquement typées et composables. ELIOM fait parti du projet OCSIGEN qui inclus également le compilateur JS_OF_OCAML, un serveur Web et diverses librairies pour la programmation Web. Les librairies OCSIGEN sont implémentées en utilisant un langage minimal qui permet d'exprimer toutes les fonctionnalités nécessaires pour la programmation Web sans tiers. Ce langage est bâti sur un certain nombre de concepts.

Composition ELIOM permet de construire des composants indépendants et réutilisables qui peuvent être assemblés facilement. Il permet de définir et de manipuler *sur le serveur*, comme valeurs de première classe, des fragments de code qui seront exécutés *sur le client*. Ceci permet de construire des widgets réutilisables qui capturent à la fois un comportement serveur et un comportement client.

Typage statique ELIOM introduit un nouveau système de type qui permet une construction modulaire des programmes client-serveur tout en préservant les garanties du typage statique et la capacité d'abstraction. Ceci garantit, via le système de type, que le code client n'est pas utilisé dans le code serveur (et inversement) et assure que les communications sont correctes.

Communication explicites Les communications entre le serveur et le client sont explicites dans ELIOM. Ceci permet au programmeur de raisonner sur l'exécution du programme et le comportement résultant. Le programmeur peut, par exemple, s'assurer que certaines données ne quittent pas le serveur ou le client, ou choisir la quantité de communication ainsi que l'endroit ou les calculs sont effectués.

Modèle d'exécution efficace ELIOM repose sur un nouveau modèle d'exécution efficace pour les communications client-serveur qui évite un aller-retour répété. Ce modèle est simple et prévisible, ce qui est particulièrement important dans le cadre d'un langage impur tel que ML.

Dans cette présentation, nous exposerons notre travail sur la définition, formalisation et implémentation du langage ELIOM. La formalisation du langage d'expression à été présentée dans Radanne et al. [2016]. Un système de module est présenté dans Radanne and Vouillon [2017], en cours de soumission.

Références

G. Radanne and J. Vouillon. Tierless Modules. Submitted to ICFP 2017, Mar. 2017. URL https://hal.archives-ouvertes.fr/hal-01485362.

G. Radanne, J. Vouillon, and V. Balat. Eliom : A core ML language for tierless web programming. In A. Igarashi, editor, *Programming Languages and Systems - 14th Asian Symposium, APLAS 2016, Hanoi, Vietnam, November 21-23, 2016, Proceedings*, volume 10017 of *Lecture Notes in Computer Science*, pages 377–397, 2016. doi : 10.1007/978-3-319-47958-3_20. URL http://dx.doi.org/10.1007/978-3-319-47958-3_20.

Preuve formelle du théorème de Lax–Milgram

Sylvie Boldo [1] François Clément [2] Florian Faissole [1]
Vincent Martin [3] Micaela Mayero [4]

1 Introduction

La méthode des éléments finis est fréquemment utilisée pour résoudre numériquement des équations aux dérivées partielles qui interviennent par exemple en physique ou en biologie [4]. Pour augmenter la confiance dans les programmes qui l'implémentent, on doit commencer par la formalisation des notions mathématiques nécessaires à sa correction, à l'aide par exemple de preuves formelles. Le théorème de Lax–Milgram est l'un de ces résultats fondamentaux. Il permet, sous certaines hypothèses de coercivité et de complétude, de montrer l'existence et l'unicité de la solution de certains problèmes aux limites et de leur version discrète. Cet article présente une formalisation complète du théorème de Lax–Milgram dans l'assistant de preuves Coq, et utilise plus particulièrement la bibliothèque Coquelicot pour l'analyse réelle [2]. Des résultats mathématiques variés sont requis, provenant de l'algèbre linéaire, la géométrie et l'analyse fonctionnelle des espaces de Hilbert. Ce travail repose sur une preuve papier détaillée [3] et est plus précisément décrit dans [1].

2 Espaces de Hilbert

On étend, à l'aide du mécanisme de structures canoniques [5], la hiérarchie de la bibliothèque Coquelicot pour définir les espaces préhilbertiens et hilbertiens. Un espace préhilbertien est un module équipé d'un produit scalaire (avec les propriétés usuelles). On prouve qu'un espace préhilbertien est un module normé, dont la norme se dérive du produit scalaire. Un espace de Hilbert est un espace préhilbertien complet. Dans ces espaces, on définit des notions géométriques comme le projeté orthogonal d'un vecteur sur un sous-espace ou le complémentaire orthogonal d'un sous-espace et on prouve formellement les résultats associés (lemmes d'existence ou de caractérisation).

On formalise les espaces d'applications linéaires, puis on vérifie l'équivalence entre plusieurs définitions de la continuité de telles fonctions. L'une d'elle est la finitude de la norme d'opérateur $|||f|||$ d'une fonction f de E dans F, modules normés. On définit ainsi $\texttt{clm}(E,F)$ l'espace des applications linéaires continues et on prouve que la norme d'opérateur lui confère la structure de module normé (le cas $F = \mathbb{R}$ est utilisé dans la preuve du théorème de Lax–Milgram et correspond au dual topologique E' de E) :

```
Record clm := Clm {                    (* type dépendant *)
  m:> E->F ;      (* la fonction, avec une coercion pour usage direct *)
  Lf: is_linear_mapping m;             (* preuve de linéarité *)
  Cf: is_finite (operator_norm m) }.   (* preuve de continuité *)
```

1. Inria, LRI, Université Paris-Sud & CNRS, Université Paris-Saclay, France
2. Inria, 2 rue Simone Iff, CS 42112, FR-75589 Paris cedex 12, France
3. LMAC, UTC, BP 20529, FR-60205 Compiègne, France
4. LIPN, Université Paris 13, CNRS UMR 703, Villetaneuse, F-93430, France

3 Preuves des théorèmes de Riesz–Fréchet et Lax–Milgram

Le théorème de Riesz–Fréchet est un résultat intermédiaire pour la preuve du théorème de Lax–Milgram. Il s'agit d'un résultat de représentation, qui identifie, à l'aide du produit scalaire, une fonction f de E' à un unique vecteur de l'espace hilbertien E. Ce théorème se démontre avec les lemmes de caractérisations du projeté orthogonal, le complémentaire orthogonal de $ker(f)$ et la complétude de E. Il est nécessaire de distinguer le cas où f est identiquement nulle, ce qui n'est pas décidable. Ainsi, on ajoute des hypothèses de décidabilité.

Le théorème de Lax–Milgram s'énonce comme suit :

Théorème (Lax–Milgram). *Soit* $E : Hilbert$, $f \in E'$, $0 < \alpha$, $\varphi : E \to \texttt{Prop}$ *un sous-module complet. Soit* a *une forme bilinéaire sur* E*, bornée et* α*-coercive.*
On suppose $\forall f \in E',\ decidable(\exists u \in E, u \in \ker(f)\ \wedge\ \neg\neg\varphi(u)) \wedge$
$(\forall u \in E, \forall \varepsilon \in \mathbb{R}_+, decidable(\exists w \in E, \varphi(w) \wedge \|u - w\| < \varepsilon))$.
Alors : $\exists! u \in E, \neg\neg\varphi(u) \wedge \forall v \in E, \neg\neg\varphi(v) \Longrightarrow f(v) = a(u, v) \wedge \|u\|_E \leq \frac{1}{\alpha} \cdot |||f|||_\varphi$.

Une simplification calculatoire et deux applications du théorème de Riesz–Fréchet permettent de se ramener à chercher l'unique point fixe d'une fonction contractante bien choisie. On applique alors un théorème de point fixe de Banach. On utilise les axiomes de $ProofIrrelevance$ et de $FunctionalExtensionality$ dans les preuves d'analyse fonctionnelle. Ceci étant, on ne fait pas appel à la logique classique et afin d'identifier certains points critiques, on préfère ajouter des hypothèses de décidabilité lorsque nécessaire ainsi que quelques doubles négations.

4 Conclusion

On obtient une preuve formelle complète du théorème de Lax–Milgram en Coq. Le développement Coq comporte environ 7 000 lignes de code, tout comme la preuve papier LaTeX d'environ 50 pages. L'usage de Coq et de Coquelicot pose des écueils non présents dans les preuves papier, comme la représentation des sous-espaces ou l'usage de filtres dans les raisonnements topologiques. Cette formalisation est la première étape d'un travail de certification autour de la méthode des éléments finis, qui vise à prouver des programmes l'implémentant.

Références

[1] S. Boldo, F. Clément, F. Faissole, V. Martin, and M. Mayero. A Coq Formal Proof of the Lax–Milgram theorem. In *6th Conference on Certified Programs and Proofs*, Paris, 2017.

[2] S. Boldo, C. Lelay, and G. Melquiond. Coquelicot : A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1) :41–62, 2015.

[3] F. Clément and V. Martin. The Lax-Milgram Theorem. A detailed proof to be formalized in Coq. Research Report RR-8934, Inria Paris, July 2016.

[4] A. Ern and J-L. Guermond. *Theory and practice of finite elements*, volume 159 of *Applied Mathematical Sciences*. Springer-Verlag, New York, 2004.

[5] A. Mahboubi and E. Tassi. Canonical structures for the working coq user. In *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*, pages 19–34, 2013.

Temporary Read-Only Permissions for Separation Logic (Extended Abstract)

Arthur Charguéraud[1,2] and François Pottier[1]

[1] Inria, Paris, France*
[2] ICube – CNRS, Université de Strasbourg, France

Abstract. We present an extension of Separation Logic with a general mechanism for temporarily converting any assertion (or "permission") to a read-only form. No accounting is required: our read-only permissions can be freely duplicated and discarded. We argue that, in circumstances where mutable data structures are temporarily accessed only for reading, our read-only permissions enable more concise specifications and proofs. The metatheory of our proposal is verified in Coq.

Separation Logic [2] offers a natural and effective framework for proving the correctness of imperative programs that manipulate the heap. However, practice shows some limitations of the specification language. Consider the following example, which describes the specification of the concatenation of two arrays.

$$\{a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L_1 +\!\!+ L_2) \mathbin{\star} a_1 \rightsquigarrow \mathsf{Array}\, L_1 \mathbin{\star} a_2 \rightsquigarrow \mathsf{Array}\, L_2\}$$

Above, $a \rightsquigarrow \mathsf{Array}\, L$ asserts the existence (and unique ownership) of an array at address a whose content is given by the list L. A separating conjunction $\star$ is used in the precondition to require that a_1 and a_2 be disjoint arrays. Its use in the postcondition guarantees that a_3 is disjoint with a_1 and a_2. In this specification, the fact that the arrays a_1 and a_2 are unaffected must be explicitly stated as part of the postcondition, making the specification seem verbose.

We wish to extend the specification language of Separation Logic so as to be able to specify the same function more concisely, as follows.

$$\{\mathsf{RO}(a_1 \rightsquigarrow \mathsf{Array}\, L_1) \mathbin{\star} \mathsf{RO}(a_2 \rightsquigarrow \mathsf{Array}\, L_2)\}$$
$$(\texttt{concat}\, a_1\, a_2)$$
$$\{\lambda a_3.\ a_3 \rightsquigarrow \mathsf{Array}\,(L_1 +\!\!+ L_2)\}$$

At first sight, it may seem that the notation "RO", which stands for "read-only", could be interpreted as syntactic sugar for repeating part of the precondition in the postcondition. However, we wish to assign RO a much stronger

* This research was partly supported by the French National Research Agency (ANR) under the grant ANR-15-CE25-0008.

interpretation. First, we wish an assertion of the form $\mathsf{RO}(H)$ to prevent writing the heap fragment described by H, whereas the syntactic sugar interpretation only guarantees that this heap fragment satisfies H upon entry and upon exit. Second, we wish to allow aliasing of read-only arguments, whereas traditional Separation Logic requires a separate specification to handle the case where the two arguments are aliased. Third, we wish to save the user the need the trouble of proving that H is still satisfied upon exit: since a read-only heap fragment cannot be mutated, its properties cannot be falsified.

Fractional permissions [1] offer a partial solution. Let $a \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L$ denote a fraction α of the ownership of the array. When α is 1, this assertion is equivalent to $a \rightsquigarrow \mathsf{Array}\, L$. When α is less than 1, this assertion grants read-only access. Fractional permissions can be split or merged using the following rule:

$$a \overset{\alpha+\beta}{\rightsquigarrow} \mathsf{Array}\, L \;=\; a \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L \star a \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L \qquad \text{with } 0 < \alpha, \beta \leqslant 1.$$

With fractional permissions, the concatenation function may be specified as:

$$\begin{array}{l} \forall \alpha\beta.\ \{a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \star a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2\} \\ \quad (\mathtt{concat}\, a_1\, a_2) \\ \quad \{\lambda a_3.\ a_1 \overset{\alpha}{\rightsquigarrow} \mathsf{Array}\, L_1 \star a_2 \overset{\beta}{\rightsquigarrow} \mathsf{Array}\, L_2 \star a_3 \overset{1}{\rightsquigarrow} \mathsf{Array}\, (L_1 +\!\!+ L_2)\} \end{array}$$

This specification allows a_1 and a_2 to be aliases. Nevertheless, fractional permissions suffer from several limitations. First, they require explicit quantification over the fractions α and β, and may involve arithmetic operations when splitting and merging permissions. Second, they require careful accounting: if a single nonzero fraction is lost, a full read/write permission can never be recovered. Third, this approach lacks generality because scaling $\frac{1}{2}H$ cannot be defined for an arbitrary assertion H.

We propose an extension of the logic with a modality "RO" that applies to any assertion H. The assertion $\mathsf{RO}(H)$ is duplicable, allowing read-only arguments to be aliased. Moreover, an assertion $\mathsf{RO}(H)$ appears only in preconditions, never in postconditions. Thus, compared with traditional Separation Logic, our specifications are more concise, and the number of proof obligations for establishing postconditions is reduced.

Read-only permissions are introduced by a generalization of the frame rule:

$$\frac{\{H \star \mathsf{RO}(H')\}\ t\ \{Q\} \qquad \mathsf{no\text{-}ro\text{-}in}\ H'}{\{H \star H'\}\ t\ \{Q \star H'\}}\ \textsc{frame-ro}$$

where the side condition $\mathsf{no\text{-}ro\text{-}in}\, H'$ means that H' does not contain any read-only assertions. The intuition is as follows. With the traditional frame rule, the assertion H' that is "framed out" disappears within a certain scope and reappears when that scope is exited. With the read-only frame rule (above), instead of disappearing altogether, the assertion H' is turned into $\mathsf{RO}(H')$ within the scope, and reappears when that scope is exited. The soundness of this extension of Separation Logic has been entirely formalized in Coq.

References

1. Boyland, J.: Checking interference with fractional permissions. In: Static Analysis Symposium (SAS). Lecture Notes in Computer Science, vol. 2694, pp. 55–72. Springer (2003)
2. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Logic in Computer Science (LICS). pp. 55–74 (2002)

Session commune aux groupes de travail GLE et RIMEL

Génie Logiciel Empirique — Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Automatic Microbenchmark Generation to Prevent Dead Code Elimination and Constant Folding

Marcelino Rodriguez-Cancio, Benoit Combemale, Benoit Baudry

Abstract

Microbenchmarking evaluates, in isolation, the execution time of small code segments that play a critical role in large applications. The accuracy of a microbenchmark depends on two critical tasks: wrap the code segment into a payload that faithfully recreates the execution conditions of the large application; build a scaffold that runs the payload a large number of times to get a statistical estimate of the execution time. While recent frameworks such as the Java Microbenchmark Harness (JMH) address the scaffold challenge, developers have very limited support to build a correct payload.

This work focuses on the automatic generation of payloads, starting from a code segment selected in a large application. Our generative technique prevents two of the most common mistakes made in microbenchmarks: dead code elimination and constant folding. A microbenchmark is such a small program that can be "over-optimized" by the JIT and result in distorted time measures, if not designed carefully. Our technique automatically extracts the segment into a compilable payload and generates additional code to prevent the risks of "over-optimization". The whole approach is embedded in a tool called AutoJMH, which generates payloads for JMH scaffolds.

We validate the capabilities AutoJMH, showing that the tool is able to process a large percentage of segments in real programs. We also show that AutoJMH can match the quality of payloads handwritten by performance experts and outperform those written by professional Java developers without experience in microbenchmarking.

Diagnosys: Automatic Generation of a Debugging Interface to the Linux Kernel

Tegawendé F. Bissyandé, Laurent Réveillère
University of Bordeaux, France
{bissyande,reveillere}@labri.fr

Julia L. Lawall, Gilles Muller
INRIA/LIP6-Regal, France
{Julia.Lawall,Gilles.Muller}@lip6.fr

ABSTRACT

The Linux kernel does not export a stable, well-defined kernel interface, complicating the development of kernel-level services, such as device drivers and file systems. While there does exist a set of functions that are exported to external modules, these frequently change, and have implicit, ill-documented preconditions. However, no specific debugging support is provided.

We present *Diagnosys*, an approach to automatically constructing a debugging interface for the Linux kernel. First, a designated kernel maintainer uses Diagnosys to identify constraints on the use of the exported functions. Based on this information, service developers can then use Diagnosys to generate a debugging interface specialized to their code. When a service including this interface is tested, it records information about potential problems. This information is preserved following a kernel crash or hang. Our experiments show that the generated debugging interface provides useful log information and incurs a low performance penalty.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; D.4.5 [**Operating Systems**]: Reliability

General Terms

Design, Experimentation, Reliability

Keywords

Diagnosys, Debugging, Wrappers, Linux, Device drivers

1. INTRODUCTION

Debugging is difficult. And debugging an operating system kernel-level service, such as a device driver, file system, or network protocol, is even more difficult. When a crash occurs, the service developer is presented with a backtrace, containing the location of the instruction that caused the crash and the pending return pointers on the stack. This information may be unreliable or incomplete. Even when the backtrace information is present and correct, it does not capture context information such as the values of local variables and the effect of recent decisions that are often essential to identify the problem. Indeed, kernel service code contains many execution paths, taking conditions from the operating environment into account, and is difficult to test deterministically. Support is needed for providing more information at the time of the crash, without introducing a substantial performance penalty or imposing an additional burden on the developer.

As Linux is becoming more and more widely used, in platforms ranging from embedded systems to supercomputers, there is an increasing interest from third-party developers, having little expertise in Linux internals, in developing new Linux kernel services. Such services must integrate with the Linux kernel via the various kernel-level APIs. Developing code at this level is a challenging task. Indeed, the Linux kernel development process is based on the assumption that the source code of all kernel-level services is available within the publicly available kernel source tree, and thus kernel APIs are, for efficiency, only as robust as required by their internal client services. Furthermore, kernel developers can freely adjust the kernel APIs, as long as they are willing to update all of the affected service code. The kernel implementation is thus, by design, maximally efficient and evolvable, enabling it to rapidly meet new performance requirements, address security issues, and accommodate new functionalities. But these assumptions complicate the task of the developers of new services who require more safety and help in debugging. Advances in bug-finding tools [3, 18, 20], specialized testing techniques [17, 21], and code generation from specifications [30] have eased but not yet fully solved these difficulties. Current approaches put substantial demands on the developer, both to learn how to use the approach and to effectively integrate it into his development process.

We concretize the difficulty confronting a Linux service developer as the notion of a *safety hole*. We define a safety hole as a fragment of code that introduces the potential for a fault to occur in the interaction between a kernel-level service and the rest of the kernel. For example, code in the definition of a kernel API function that dereferences a parameter without testing its value represents a safety hole, because a service could invoke the function with `NULL` as the corresponding argument. Likewise, code in the definition of a kernel internal API function that returns `NULL` as the result represents a safety hole, because a calling service could

ASE '12, September 3–7, 2012, Essen, Germany

dereference this result without checking its value.

To address the problem of safety holes in Linux kernel internal API functions, we propose an approach, named *Diagnosys*, that automatically generates a debugging interface to the Linux kernel tailored for a particular kernel-level service under development, based on a prior static analysis of the Linux kernel source code. To limit the runtime overhead, the generated debugging interface is localized at the boundary of the interaction between the service and the OS kernel. This strategy focuses the feedback provided by the interface on the part of the code that the developer has written, and thus is expected to be the most familiar with. Because the interface is only visible to the service, it has no impact on the performance of code within the kernel, even code that uses functions that contain safety holes. When the service executes, the interface generates log messages whenever service code invokes a kernel API function containing a safety hole in a potentially risky way. Such a debugging interface requires no manual intervention from the service developer until there is a crash or hang, and is thus well-suited to intensive service development, when the developer is modifying the code frequently, and bugs are likewise frequent. Because the debugging interface is automatically generated, it can be regenerated for each new version of the Linux kernel, as the properties of the kernel APIs change.

Diagnosys is composed of two tools: SHAna (Safety Hole Analyzer), which statically analyzes the kernel source code to identify safety holes in the definitions of the kernel exported functions, and DIGen (Debugging Interface Generator), which uses the information about the identified safety holes to construct a debugging interface tailored to a given service. Diagnosys also includes a runtime system, provided as a kernel patch. SHAna is run by a Linux kernel maintainer once for each Linux version, to take into account the current definitions of the Linux kernel internal API functions. DIGen is run by a service developer as part of the service compilation process. During the execution of the resulting service, the debugging interface uses the runtime system to log information in a crash-resilient buffer about any unsafe uses of functions containing safety holes. On a kernel crash or hang, the service developer can subsequently consult the buffer to obtain the logged information.

The main contributions of this paper are as follows:

- We identify the interface of kernel exported functions as a sweet spot at which it is possible to interpose the generation of debugging information, in a way that improves debuggability but does not introduce an excessive runtime overhead.
- We identify safety holes as a significant problem in the interface between a service and the kernel. Indeed, of the 703 Linux 2.6 commits for which the changelog refers explicitly to a function exported in Linux 2.6.32, 38% corrected faults that are related to one of our identified safety holes.
- We propose an approach to allow a service developer to seamlessly generate, integrate, and exploit a kernel debugging interface specialized to the service code. This approach has a low learning curve, and in particular does not require any particular Linux kernel expertise.
- Using fault-injection experiments on 10 Linux kernel services, we demonstrate the improvement in debuggability provided by our approach. We find that in 90% of the cases in which a crash occurs, the log contains information relevant to the origin of the defect, and in 95% of these cases, a message relevant to the crash is the last piece of logged information. We also find that in 93% of the cases in which a crash or hang occurs, the log information reduces the number of files that have to be consulted to find the cause of the bug.
- We show that the generated debugging interface incurs only a minimal runtime overhead on service execution, allowing it to be used up through the phase of early deployment.

The rest of this paper is organized as follows. Section 2 illustrates problems in kernel development that have been related to safety holes and gives an overview of kinds of safety holes that we take into account. Section 3 discusses the challenges in kernel debugging, focusing on crashes and hangs derived from safety holes. Section 4 presents Diagnosys, including the process of collecting information about the occurrences of safety holes and their associated preconditions, and the process of generating a debugging interface. Section 5 evaluates our approach. Finally, Section 6 discusses related work, and Section 7 concludes.

2. SAFETY HOLES

To understand the challenges posed by safety holes, we first consider some typical examples in Linux kernel internal API functions and the problems that these examples have caused, as reflected by Linux patches. Then, we present a methodology for identifying kinds of safety holes, and use this methodology to enumerate the kinds of safety holes considered in the rest of the paper. Finally, we consider how to statically identify preconditions on these safety holes, to limit the generation of log messages in the debugging interface to cases that may actually cause a crash or hang.

2.1 Examples of safety holes

Because the Linux kernel does not define a precise internal API, we focus on the set of functions that are made available to dynamically loadable kernel modules using either `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL`. Dynamically loadable kernel modules provide a convenient means to develop new services, as they allow the service to be loaded into and removed from a running kernel for the testing of new service versions. We refer to kernel functions that are made available to such modules as *kernel exported functions.*

Fig. 1a shows an excerpt of the definition of the kernel exported function `skb_put`, which dereferences its first argument without first checking its value. Many kernel functions are written in this way, assuming that all arguments are valid. This code represents a safety hole, because the dereference is invalid if the corresponding argument is `NULL`. Such a fault occurred in Linux 2.6.18 in the file *drivers/net/forcedepth.c.* In the function `nv_loopback_test`, `skb_put` is called with its *skb* argument being the result of calling `dev_alloc_skb`, which can be `NULL`. The fix, as implemented by the patch shown in Fig. 1b, is to avoid calling `skb_put` in this case. `skb_put` remains unchanged.

Fig. 2a shows an excerpt of the definition of the kernel exported function `open_bdev_exclusive`, which returns a value constructed using the kernel function `ERR_PTR` when an

```
1  unsigned char *skb_put(struct sk_buff *skb, unsigned int len)
2  {   unsigned char *tmp = skb_tail_pointer(skb);
3      SKB_LINEAR_ASSERT(skb);
4      skb->tail += len; ...
5  }
```

a) Excerpt of the definition of `skb_put`

```
1  commit 46798c897e235e71e1e9c46a5e6e9adfffd8b85d
2      tx_skb = dev_alloc_skb(pkt_len);
3  +   if (!tx_skb) { ... goto out; }
4      pkt_data = skb_put(tx_skb, pkt_len);
```

b) Excerpt of the bug fix patch

Figure 1: Bug fix of the usage of `skb_put`

error is detected. Dereferencing such a value will crash the kernel. Thus, this return statement also represents a safety hole. In Linux 2.6.32, in the file *fs/btrfs/volumes.c*, the function `btrfs_init_new_device` called `open_bdev_exclusive` and compared the result to `NULL` before dereferencing the value. This test, however, does not prevent a kernel crash, because an `ERR_PTR` value is different from `NULL`. Fig. 2b shows a patch fixing the fault.

```
1  struct block_device *open_bdev_exclusive(
2          const char *path, fmode_t mode, void *holder)
3  {
4      ...
5      return ERR_PTR(error);
6  }
```

a) Excerpt of the definition of `open_bdev_exclusive`

```
1  commit 7f59203abeaf18bf3497b308891f95a4489810ad
2      bdev = open_bdev_exclusive(...);
3  -   if (!bdev) return -EIO;
4  +   if (IS_ERR(bdev)) return PTR_ERR(bdev);
```

b) Excerpt of the bug fix patch

Figure 2: Bug fix of error handling code

In the previous cases, the safety hole is apparent in the definition of a kernel exported function. A safety hole, however, may also be interprocedural, making the danger that it poses more difficult to spot. For example, as shown in Fig. 3(a,b), the kernel exported function `kmap`, defined in *arch/x86/mm/highmem_32.c*, passes its argument to the function `page_zone` via the macro `PageHighMem`, which in turn forwards the pointer, again without ensuring its validity, to the function `page_to_nid`. This function then dereferences it, unchecked. This safety hole resulted in a fault, which was fixed by the patch shown in Fig. 3c.

```
1  void *kmap(struct page *page)
2  { might_sleep();
3    if ( !PageHighMem(page))
4    ...
5  }
```

a) Excerpt of `kmap`

```
1  static inline int page_to_nid
2          (struct page *page) {
3    return ( page->flags >> ...)
4                  & NODES_MASK;
5  }
```

b) Excerpt of `page_to_nid`

```
1  commit 649f1ee6c705aab644035a7998d7b574193a598a
2      page = read_mapping_page(...);
3  +   if (IS_ERR(page)) { ... goto out; }
4      pptr = kmap (page);
```

c) Excerpt of the bug fix patch

Figure 3: Bug fix of a use of `kmap`

2.2 Taxonomy of safety holes

As illustrated in Section 2.1, some fragments of code executed by kernel exported functions, while themselves being correct, can provoke kernel crashes or hangs when the function is used incorrectly. We distinguish between *entry* safety holes, in which the crash or hang is provoked within the execution of the kernel exported function, due to an invalid argument provided by the service, and *exit* safety holes, in which the crash or hang is provoked within the subsequent execution of the service due to a possible effect of the kernel exported function that the service has not taken into account.

As a first source of kinds of safety holes, we consider the fault kinds identified by Chou *et al.* in their 2001 study of Linux code [5]. A fault is not in itself a safety hole, because the faulty code can be completely contained within a single function definition. Likewise, a safety hole is not in itself a fault, as illustrated by the above examples. Nonetheless, we observe that many fault kinds involve the conjunction of multiple disjoint code fragments. When some of these fragments are present in a kernel exported function and the remainder may be present in a service implementation, we say that the kernel exported function contains a safety hole. For example, a `NULL` pointer dereference fault typically involves an initialization of a variable to `NULL` followed by a dereference of this variable. Returning `NULL` from a kernel exported function can cause it to be dereferenced in service code, and receiving `NULL` as an argument in a kernel exported function can lead to a `NULL` pointer dereference in the kernel exported function code.

These observations suggest a methodology for translating fault kinds into kinds of safety holes. When the suffix of a sequence of code fragments associated with a fault kind is found in a kernel exported function and depends in some way on the calling context, *e.g.*, via arguments of that function, then that suffix represents an entry safety hole. Likewise, when a prefix of such a sequence is found in a kernel exported function and has some impact on the function's result, then that prefix represents an exit safety hole.

Table 1 summarizes the fault kinds identified by Chou *et al.*, as well as the entry and exit safety hole kinds that we have derived from these fault kinds according to the above methodology. For example, given the above analysis of the structure of a Null fault, the corresponding entry safety hole is a dereference of an unchecked pointer parameter, while the corresponding exit safety hole is a return of a `NULL` value.

2.3 Safety hole preconditions

From a collection of safety holes, our goal is to create a debugging interface that informs the service developer of possibly dangerous uses of kernel exported functions within his code. Nevertheless, merely invoking a kernel exported function that contains an entry or exit safety hole does not necessarily cause a fault. Instead, some properties of the argument or return values, such as the presence of a `NULL` value, must typically be satisfied. Thus, we require information not just about safety holes, but also about the preconditions that must be satisfied for a fault to occur [16]. We furthermore distinguish between safety holes that are *certain*, if satisfaction of the precondition is guaranteed to result in a crash or hang within the execution of the kernel exported function, or *possible*, if satisfaction of the precondition may cause a crash or hang on at least one possible execution path. All exit safety holes are *possible*, as the usage context of the function result is unknown.

3. KERNEL DEBUGGING

Category	Actions to avoid faults	safety hole	safety hole description	Analysis type
Block	To avoid deadlock, do not call blocking functions with interrupts disabled or a spinlock held	`entry` `exit`	EF calls a blocking function (function referencing GFP_KERNEL) EF returns after disabling interrupts or while holding a lock	interprocedural intra/interprocedural
Null	Check potentially NULL/ERR_PTR pointers returned from routines	`entry` `exit`	EF dereferences an argument without checking its validity EF returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Var	Do not allocate large stack variables ($> 1K$) on the fixed-size kernel stack	`entry` `exit`	EF allocates an array whose size depend on a parameter EF returns a large value	intraprocedural interprocedural
INull	Do not make inconsistent assumptions about whether a pointer is NULL/ERR_PTR	`entry` `exit`	EF dereferences an argument without checking its validity EF returns a NULL/ERR_PTR pointer	interprocedural interprocedural
Range	Always check bounds of array indices and loop bounds derived from user data	`entry` `exit`	EF uses an unchecked parameter to compute an array index EF returns a value obtained from user level	intraprocedural interprocedural
Lock	Released acquired locks; do not double-acquire locks	`entry` `exit`	EF acquires a lock derived from a parameter EF returns without releasing an acquired lock	interprocedural interprocedural
Intr	Restore disabled interrupts	`entry` `exit`	EF calls a blocking function EF returns with interrupts disabled	interprocedural intraprocedural
Free	Do not use freed memory	`entry` `exit`	EF dereferences a pointer-typed parameter value EF frees memory derived from a parameter	none interprocedural
Float	Do not use floating point in the kernel		*These fault kinds depends on local properties and are therefore*	none
Real	Do not leak memory by updating pointers with potentially NULL realloc return values		*not relevant to the interface between a service and the kernel exported functions*	none
Param	Do not dereference user pointers	`entry` `exit`	EF dereferences a pointer-typed parameter EF returns a pointer-typed value obtained from user level	none interprocedural
Size	Allocate enough memory to hold the type for which you are allocating	`entry` `exit`	EF allocates memory of a size depending on a parameter EF returns an integer value	intraprocedural none

Table 1: Categorization of common faults in Linux [5]. EF refers to the *exported function*.

Each of the examples presented in Section 2 could crash the kernel. When this occurs, the kernel generates an oops report, consisting of the reason for the crash, the values of some registers and a backtrace, listing the function calls pending on the stack. Using this information in debugging raises two issues: 1) the reliability of the provided information, and 2) the relevance of the provided information to the actual fault.

Reliability of kernel oops reports. Linux kernel backtraces suffer from the problem of *stale pointers*, *i.e.* addresses in functions that have actually already returned at the current point in the execution. To illustrate this problem, we consider a crash occurring in the function `btrfs_init_new_device` previously shown in Fig. 2. The crash occurred because the kernel exported function `open_bdev_exclusive` returns an `ERR_PTR` value in case of an error, while `btrfs_init_new_device` expects that the value will be `NULL`. This caused a subsequent invalid pointer dereference.

To replay the crash, we installed a version of the `btrfs` module from just before the application of the patch. To cause `open_bdev_exclusive` to fail we first create and mount a `btrfs` volume and then attempt to add to this volume a new device that we have not yet created. This operation is handled by the `btrfs_ioctl_add_dev` ioctl which calls `btrfs_init_new_device` with the device path as an argument. This path value is then passed to `open_bdev_exclusive` which fails to locate the device and returns an `ERR_PTR` value. Fig. 4 shows an extract of the resulting oops report. Line 1 shows that the crash is due to an attempt to access an invalid memory address. Line 5 shows that the faulty operation occurred in the function `btrfs_init_new_device` a priori during a call to `btrfs_ioctl_add_dev` (line 8). Source files and line numbers can be obtained by applying the standard debugger `gdb` to the compiled module and to the compiled kernel.

This backtrace contains possibly stale pointers, as indicated by the `?` symbol on lines 8 and 9. While `btrfs_ioctl_add_dev` really does call `btrfs_init_new_device`, this is not the case of `memdup_user`. Since it cannot be known a priori whether a function annotated with `?` is really stale, the service developer has to find and study the definitions of all of the functions at the top of the backtrace, until finding the reason for the crash, including the definitions of functions that may be completely unrelated to the problem. A goal of the kernel debugger kdb,[1] which was merged into the mainline in Linux 2.6.35, was to improve the quality of backtraces. Nevertheless, backtrace quality remains an issue.[2]

```
1  [  847.353202] BUG: unable to handle kernel paging request at ffffffee
2  [  847.353205] IP: [<fbc722d9>] btrfs_init_new_device+0xcf/0x5c5 [btrfs]
3  [  847.353229] *pdpt = 00000000007ee001 *pde = 00000000007ff067
4  [  847.353233] Oops: 0000 [#1] ...
5  [  847.353291] EIP is at btrfs_init_new_device+0xcf/0x5c5 [btrfs] ...
6  [  847.353298] Process btrfs-vol (pid: 3699, ...
7  [  847.353312] Call Trace:
8  [  847.353327] [<fbc7b84e>] ? btrfs_ioctl_add_dev+0x33/0x74 [btrfs]
9  [  847.353334] [<c01c52a8>] ? memdup_user+0x38/0x70 ...
10 [  847.353451] ---[ end trace 69edaf4b4d3762ce ]---
```

Figure 4: Oops report following a `btrfs` `ERR_PTR` pointer dereference crash.

Relevance of kernel oops reports. A kernel oops backtrace contains only the instruction causing the crash and the sequence of function calls considered to be on the stack. The actual reason for a crash, however, may occur in previously executed code that is not represented. For the fault shown in Fig. 2, the oops report mentions a dereference of the variable *bdev* in the function `btrfs_init_new_device`, but the real source of the problem is at the initialization of *bdev*, to the result of calling `open_bdev_exclusive`. This call has returned and thus no longer appears on the stack. Such situations make debugging more difficult as the developer must thoroughly consult kernel and service source code to localize important initialization code sites.

[1] https://kgdb.wiki.kernel.org/

[2] https://lkml.org/lkml/2012/2/10/129

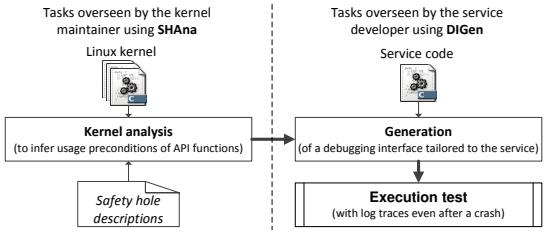


Figure 5: The steps in using Diagnosys

Kernel hangs. By default, the Linux kernel gives no feedback in the case of a kernel hang. It can, however, be configured to panic when it detects no progress over a certain period of time. When the hang is due to an infinite loop, the backtrace resulting from the panic can occur anywhere within this loop; the point of the panic may thus have no relation to the actual source of the problem.

4. DIAGNOSYS

The goal of Diagnosys is to improve the quality of the logs available when a crash or hang occurs and this crash or hang results from a safety hole in a kernel exported function. The use of Diagnosys involves three phases (Fig. 5): 1) identification of safety holes in kernel exported functions and inference of the associated preconditions, using the static analysis tool SHAna, 2) automatic generation of a debugging interface using DIGen based on the inferred preconditions, and 3) testing service code with the support of the debugging interface. The first phase is carried out only once by a kernel maintainer, for each new version of the mainline Linux kernel,[3] and the remaining phases are carried out by each service developer who would like to use Diagnosys.

4.1 Identifying safety holes and their preconditions

SHAna first searches the kernel code for occurrences of the kinds of safety holes listed in Table 1 in the implementations of exported functions, and then computes the preconditions that must be satisfied for these safety holes to cause a kernel crash or hang. The analysis focuses on unsafe operations that occur in code that is in or is reachable from an exported function. For each such occurrence, a backward analysis amounting to a simple version of Hoare logic [16] produces the weakest precondition to be satisfied on entry to the function, for entry safety holes, and on exit from the function, for exit safety holes, such that the safety hole may cause a crash. The result of SHAna is a list mapping each kernel exported function identified as containing safety holes to the associated preconditions.

The analysis starts from the definition of an exported function, recognized as one declared using `EXPORT_SYMBOL` or `EXPORT_SYMBOL_GPL`. Table 1 indicates for each category of safety hole whether intraprocedural, interprocedural or no analysis is used. In search scenarios that only require intraprocedural analysis, the analyzer scans the definition of the exported function to identify code fragments that represent safety holes. For example, in searching for Intr exit safety holes, SHAna only looks for interrupt disabling operations in the kernel exported function itself, because interrupt state flags should not be passed from one function to another [29]. In the case of interprocedural analysis, SHAna starts from the definition of an exported function and iteratively analyzes all called functions. For example, in searching for Null entry safety holes, SHAna searches through both the kernel exported function itself and all called functions that receive a parameter of the kernel exported function as an argument to find unchecked dereferences. SHAna furthermore includes unchecked dereferences of values that somehow depend on the value of an unchecked parameter.

[3]Each Linux distribution may add some specific patches to the Linux kernel. These are unlikely to affect the kernel API. Furthermore a service that should ultimately be integrated into the mainline kernel must be developed against the API supported by that kernel.

4.2 Generating and integrating a debugging interface

Based on the results of SHAna, DIGen generates a debugging interface in the form of a collection of wrapper functions that augment the definitions of kernel exported functions with the necessary checks and calls to logging primitives in order to detect and record violations of safety hole preconditions. Ideally, the kernel maintainer who runs SHAna would also generate a single debugging interface for the entire kernel that could be used by all service developers. The Linux kernel, however, is highly configurable, targeting a wide range of hardware platforms, and thus many kernel source files have incompatible header file dependencies. Therefore, it is not possible to compile a single debugging interface wrapping all of the kernel exported functions at the same time. Accordingly, we shift the interface generation process into the hands of the service developer, who generates an interface specific to his service. Because the functions invoked by a single service can necessarily be compiled together, this approach avoids all compilation difficulties, while producing a debugging interface that is sufficient for and individual service's needs. We now describe the generation of the debugging interface and how it is integrated into a service under development.

Generating a debugging interface. For each kernel exported function that is used in the service and for which SHAna identified at least one safety hole, DIGen generates a wrapper function. The general structure of such a wrapper is shown in Figure 6. Based on the argument values, the wrapper first checks each entry safety-hole precondition (line 4) and then, if the precondition is not satisfied, logs a message indicating the violation. This message includes the safety hole category, which specifies the kind of safety hole and whether the violation is *certain* or *possible* (line 5), as defined in Section 2.3. The wrapper then calls the original function. If the original function has a return value, this value is stored in a local variable, `__ret`, and then the preconditions on any exit safety holes are checked based on this information (lines 9-10). Finally, the return value, if any, of the original function is returned as the result of the wrapper (line 12).

For performance reasons, Diagnosys does not log formatted strings in kernel memory, instead it logs integers representing unique information identifiers that are decoded and translated on-the-fly during log retrieval.

Compiling a debugging interface into a service. The generated debugging interface is implemented as a header

```
1  static inline ⟨rtype⟩ __debug_⟨kernel function⟩ (...) {
2    ⟨rtype⟩ __ret;
3    /* Check preconditions for entry safety holes */
4    if ⟨an entry safety−hole precondition is violated⟩
5      diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., arg number)⟩);
6    /* Invocation of the intended kernel function */
7    __ret = ⟨call to kernel function⟩;
8    /* Check preconditions for exit safety holes */
9    if ⟨an exit safety−hole precondition is violated⟩
10     diagnosys_log(⟨EF id⟩, ⟨SH cat⟩, ⟨info (e.g., err ret type)⟩);
11   /* Forward the return value */
12   return __ret;
13 }
14 #define ⟨kernel function⟩ __debug_⟨kernel function⟩
```

Figure 6: Wrapper structure for a non-void function

file to be included in the service code. Once compiled with the interface included, the service uses the wrapper functions instead of the corresponding kernel exported functions.

Diagnosys provides an automated script, `dmake`, that manages the generation of a debugging interface. This script (1) compiles the original service code, (2) identifies the kernel exported functions referenced by the resulting object files, (3) generates an interface dedicated to these functions, and (4) recompiles the service with the interface included.

4.3 Running service code with Diagnosys

To be able to use a Diagnosys-generated debugging interface, the service developer must use a version of the Linux kernel in which support for the Diagnosys runtime system has been installed. This support is expressed as a kernel patch, which we have implemented for Linux 2.6.32, that extends the kernel with a crash resilient logging system. The patch additionally configures the kernel to send all crashes and hangs (Linux soft and hard lockups) to the kernel panic function, which the patch extends to reboot into a crash kernel if Diagnosys is activated or to continue with a normal panic, otherwise. Finally, the Diagnosys runtime system includes a tool that can be run from user space to install a copy of the Diagnosys kernel as a crash kernel, initialize the reserved log buffer, and activate and deactivate logging.

Once the Diagnosys logging system has been activated, the service developer may test his code as usual. During service execution, if a wrapper function detects a safety hole for which the precondition is violated, the wrapper logs information about the safety hole in a reserved area of memory, annotated with a timestamp and including the memory address of the call site. The reserved area of memory is managed through a ring buffer that retains information about only the most recent violations.

On a kernel crash or hang, the Diagnosys runtime system uses a Kexec-based [24] mechanism to reboot into a new instance of the Diagnosys-enabled kernel. The Kexec-based mechanism performs the reboot without reinitializing any hardware, including the memory, thus ensuring that the accumulated Diagnosys log is still available. The service developer may then access the log messages through a pseudo character device. The messages are made available in the order in which they were generated. When a crash occurs, the Diagnosys runtime system also inserts the kernel stack trace into the Diagnosys log before rebooting.

4.4 Implementation

Table 2 gives the code sizes of the various parts of our prototype Diagnosys implementation. The implementation includes the SHAna analysis of Linux kernel, DIGen and `dmake` for generating and compiling wrappers for a given service, and the patch for the runtime system.

Diagnosys component	Tool		Code size (LOC)	Language
Kernel code analyzer	SHAna		2438 + 1331	SmPL[25] + OCaml
Wrapper generator	`dmake` + DIGen		49 + 1301	sh + OCaml
Logging system		user-space	115 + 355	sh + ansi C
		kernel-space	645	ansi C code patch

Table 2: Diagnosys prototype code size

5. EVALUATION

In designing Diagnosys, we have chosen to focus on the interface between the service code and the kernel. We first assess the number of safety holes in this interface and their past impact on kernel robustness, as evidenced by commits to the Linux kernel. We then assess the difficulty of debugging kernel faults derived from safety holes, by studying the feedback made available to the service developer on a crash or hang without Diagnosys, namely the kernel backtrace. Then, we assess the coverage of Diagnosys with respect to the possible crashes and hangs that are triggered by misuse of the interface between the service code and the kernel, and show that the Diagnosys log messages allow the service developer to find the cause of a crash or hang more rapidly than when relying on a kernel backtrace alone. Finally, we show that Diagnosys incurs a sufficiently low runtime overhead to be embedded in a service, up to the early deployment phase.

Our experiments use code from Linux 2.6.32, which was released in December 2009. This version is used in the current Long Term Support version of Ubuntu® (10.04), in Red Hat Enterprise Linux 6, in Oracle Linux, etc. Our performance experiments are carried out on a Dell 2.40 GHz Intel® Core™ 2 Duo with 3.9 GB of RAM. Unless otherwise indicated, the OS is running a Linux 2.6.32 kernel that has been modified to support the Diagnosys logging infrastructure. 1MB is reserved for the crash-resilient log buffer.

5.1 Prevalence and impact of safety holes

Diagnosys is only beneficial if SHAna identifies safety holes in functions that are used by a wide range of drivers and if these functions are likely to be used in an incorrect way. In this section, we assess the number of safety holes collected by SHAna and then study the impact these safety holes have had on the robustness of the Linux kernel itself.

In Linux 2.6.32, SHAna reports 22,940 safety holes in 7,505 exported functions. Table 3 summarizes for each kind of safety hole the number of functions that SHAna identifies as containing at least one occurrence of that kind of safety hole. In the largest category, INull/Null, about 94% of the reported functions perform unsafe dereferences directly, and 5% forward the parameter value to other functions that unsafely use them with no prior check. Around 1% perform unsafe dereferences on variables whose validity are indirectly correlated to that of pointer parameters. Defects due to safety holes in the latter two categories are more difficult for the service developer to identify.

Static analysis is necessarily approximate, as it does not have complete access to run-time values. This may lead to false positives, in which a safety hole is reported that in fact cannot lead to a crash. Such false positives can increase the logging time and clutter the log with irrelevant messages. Nevertheless, having studied the complete set of results for

Safety hole	Number of exported functions collected in the	
	entry sub-category	exit sub-category
Block	367	815
INull/Null	7,220	1,124
Var	5	11
Lock/Intr/LockIntr	815	23
Free	-	11
Size	8	-
Range	-	8

Table 3: Prevalence of safety holes in Linux 2.6.32

Linux 2.6.32, we have found only 414 false positives. 405 of these are due to the presence of multiple, configuration-specific, definitions of some functions. SHAna annotates safety holes derived from calls to such functions with the file in which the relevant function instance is defined, so that the service developer can remove those that are not relevant to his configuration.

Of the 147,403 call sites across the entire kernel source code where exported functions are used, half invoke a function containing an identified safety hole. This suggests that the kernel exported functions containing safety holes are likely to be useful to new services.

To assess the past impact of the identified safety holes over the course of the development of Linux, we have searched through 278,078 commits to Linux 2.6,[4] from 2.6.12 to 2.6.39.3, to identify those whose changelog mentions the name of at least one kernel function exported in Linux 2.6.32, ignoring changelogs in which the function name is used as a common word (*e.g.*, "sort", "panic", etc.). Linux commits by convention make only a single logical change, thus making the analysis precise.[5] 703 of these commits contain bugs described in kernel changelogs[6] that are related to the usage of exported functions. 267 of them, *i.e.*, 38% are related to the categories of safety holes that we consider in this paper.

5.2 Kernel debugging with Diagnosys

As discussed in Section 3, kernel debugging is made difficult by unreliable backtraces and by the questionable relevance of the information in crash reports. To assess the qualitative benefits of Diagnosys, we have replayed a crash and a hang reported in kernel commit logs.

Replaying a kernel crash. As an example of kernel crash, we again consider the *btrfs* example of Fig. 2. Study of the corresponding crash report in Fig. 4 showed that the source of the problem was not readily available in the backtrace. We have therefore replayed the same execution scenario when using Diagnosys. A typical Diagnosys log line contains the timestamp of the log, the source file and line number where the unsafe call was performed, the name of the exported function, the category of the safety_hole and eventually the name of a relevant argument or an unsafe return value. In the case of the replay of the *btrfs* crash, Fig. 7 shows the last line added to the Diagnosys log before the crash, which is the line that the developer is likely to consult first. This line shows that the function `open_bdev_exclusive` activated an Inull exit safety hole by returning an ERR_PTR. It also reports the runtime timestamp and the call site where the safety hole was violated. Combining this information with the information about the crash site in the oops report and the service source code shows that the problem is the inadequate error handling code after `open_bdev_exclusive`.

```
1 [4294934950]|@/var/diagnosys/tests/my_btrfs/volumes.c:1441|
2                 ↪open_bdev_exclusive|INULL(EXITED)|ERR PTR|
```

Figure 7: Diagnosys log line in the execution of `btrfs`

Replaying a kernel hang. Kernel hangs are notoriously hard to debug[7] as the panic, which occurs long after the actual fault, can produce a backtrace that is hard to correlate to the source of the problem. Diagnosys records information about previous potentially dangerous operations.

Just before the release of Linux 2.6.33, the `nouveau_drm` nVidia® graphics card driver contained a hang resulting from the use of the kernel exported function `ttm_bo_wait`. This function exhibits a Lock entry safety hole and a Lock exit safety hole, as it first unlocks and then relocks a lock received via its first argument. The `nouveau_drm` driver called this function without holding this lock, hanging the kernel.

In Fig. 8a, the last line of the Diagnosys log shows that `ttm_bo_wait` has been called without the expected lock held. Correlating this information with the source code suggests taking the lock before the call and releasing it after the call, as shown in the Linux patch in Fig. 8b.

```
1 [437126]|@/var/diagnosys/tests/nouveau/nouveau_gem.c:929|
2           ↪ttm_bo_wait|LOCK/ACQUIRE(POSSIBLE)|bo->lock|
```

a)Diagnosys log line in the execution of `nouveau_drm`.

```
1 commit f0fbe3eb5f65fe5948219f4ceac68f8a665b1fc6
2 + spin_lock(&nvbo->bo.lock);
3   ret = ttm_bo_wait(&nvbo->bo, false, false, no_wait);
4 + spin_unlock(&nvbo->bo.lock);
```

b) Bug fix related to the usage of `ttm_bo_wait`.

Figure 8: Fault involving a Lock safety hole in `nouveau_drm`

5.3 Quantifying the debugging benefit

To be useful, Diagnosys must cover a high percentage of the misuses of kernel exported functions. We first evaluate this by artificially creating and activating misuses of exported functions in kernel services and measuring how many are trapped by Diagnosys. Additionally, Diagnosys must be able to produce log messages that significantly ease the debugging process. We evaluate the debugging effort by measuring the number of files and functions that have to be studied to identify the cause of a crash, with and without Diagnosys. Our experiments involve a number of commonly used kinds of services: networking code, USB drivers, multimedia drivers, and file systems. Services of these kinds make up over a third of the Linux 2.6.32 source code. We have selected a range of services in Linux 2.6.32 that run on our test hardware. These are presented in Table 4.

Coverage of Diagnosys. To determine the coverage of Diagnosys, we first mutate existing services so as to artificially create bugs. Then, we inject faults at run-time to potentially cause the mutation to trigger a crash.

[4] git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[5] Documentation/SubmittingPatches in the Linux kernel source tree, http://www.kernel.org

[6] http://www.kernel.org/pub/linux/kernel/v2.6/

[7] http://www.linuxjournal.com/article/5749

Since the largest percentage of our identified safety holes are related to `NULL` and `ERR_PTR` dereferences, we focus on these safety holes. One prominent source of such values is as the result of a call to a function that has failed in performing some sort of allocation. Robust kernel code checks for these values and aborts the ongoing computation. Nevertheless, omission of these tests is common. For example, in Linux 2.6.32, even for the standard kernel memory allocation functions `kmalloc`, `kzalloc`, and `kcalloc`, over 8% of the calls that may fail do not test the result before dereferencing the returned value or passing it to another function.

Based on these observations, our experiments focus on missing `NULL` and `ERR_PTR` tests in the service code. Our mutations remove such tests from the service code, one by one, and use the `failslab` feature of the Linux fault injection infrastructure [6] within the initialization of the tested value to inject failures into the execution of any call to a basic memory allocation function that this initialization involves. Because the initialization can invoke basic memory allocation functions multiple time, a single mutation experiment may involve multiple injected faults.

A first possible result is that there is no observable effect. This occurs when the called function does not involve a memory allocation, when the failure of memory allocations does not lead to a `NULL` or `ERR_PTR` result, or when the safety hole is *possible* and is not encountered in the actual execution. A second possibility is that there is a crash, but there is no relevant information in the Diagnosys log. In this case, either the information has been overwritten in the ring buffer, SHAna has not detected the safety hole, or the call to a kernel exported function occurs in a header file that, for technical reasons, has to be included before the Diagnosys wrapper function definitions. The third possibility is that there is a crash and the information is logged, representing a success for Diagnosys.

We have evaluated the coverage of Diagnosys on the 10 services listed in Table 4. Removing the `NULL` and `ERR_PTR` tests one by one leads to 555 mutated services. For each mutated service, we have exercised the various execution paths of the affected module in order to execute the mutated code. The results are shown in Table 5. 56% of the mutations resulted in a kernel crash. After reboot, in 90% of the crashes, the log contained information relevant to the crash origin and in 86% of the crashes, a log was present and it was additionally in the last position. For one service, the latter only holds for 66% of the crashes, but this amounts to only one missing log, as this service has few mutation sites.

Ease of the debugging process. Provided with an oops report containing a backtrace and debugging tools that can translate stack entries into file names and line numbers, a developer typically starts from the point of the crash, visiting all files and caller functions until the origin of the crash is localized. When the crash occurs deep in the execution, the number of functions and files to visit can become large.

Category	Service module	Description	Used functions with safety holes
Networking	e1000e	Ethernet adapter	57
	iwlagn	Intel WiFi Next Gen AGN	57
	btusb	Bluetooth generic driver	26
USB drivers	usb-storage	Mass storage device driver	51
	ftdi_sio	USB to serial converter	31
Multimedia device drivers	uvcvideo	Webcam device driver	28
	snd-intel8x0	ALSA driver	35
File systems	isofs	ISO 9660 file system	26
	nfs	Network file system	198
	fuse	File system in userspace	86

Table 4: Tested services

Category	Kernel module	# of mutations	# of crashes with			Coverage
			no log	log is not last	log is last	
Networking	e1000e	57	0	0	20	100%
	iwlagn	18	1	0	8	88.9%
	btusb	9	1	0	7	87.5%
USB drivers	usb-storage	11	0	0	3	100%
	ftdi_sio	9	0	0	6	100%
Multimedia device drivers	snd-intel8x0	3	1	0	2	66.7%
	uvcvideo	34	3	3	17	73.9%
File systems	isofs	28	3	0	9	75.0%
	nfs	309	13	9	157	87.7%
	fuse	77	3	1	41	91.1%

Table 5: Results of the mutation experiments

We have considered 199 of the mutations performed in our coverage tests that lead to crashes, from btusb, nfs, and isofs. We also consider 31 mutations in nfs code that add statements for arbitrarily acquiring and releasing locks in services in order to provoke kernel hangs, focusing on locks that are passed between functions as they can trigger safety holes in core kernel code.

We have compared the 230 oops reports with the corresponding Diagnosys logs. In 92% of these crashes, the Diagnosys log contains information on the origin of the fault. We have found that for those cases, debugging with the oops report alone required consulting 1 to 14 functions, including on average one possibly stale pointer, in up to 4 different files distributed across kernel and service code. In 73% of the cases for which the Diagnosys log contains relevant information, we find that using Diagnosys reduces by at least 50% the number of files and functions to consult. In 19% of the cases for which the Diagnosys log contains relevant information, the crash occurred in the same file as the mutation, but the Diagnosys log made it possible to more readily pinpoint the fault by providing line numbers that are closer to the mutation site.

5.4 Overhead

Introducing wrappers on kernel-exported functions incurs a performance overhead on service execution. To assess the impact of this overhead, we execute various real-world kernel services with and without a debugging interface.

Network driver performance. Our first test involves a Gigabit Ethernet device that requires both low latency and high throughput to guarantee high performance. We evaluate the impact of a debugging interface by exercising the e1000e Linux device driver using the TCP_STREAM, UDP_STREAM and UDP_RR tests from the netperf benchmark.[8] For these experiments, the netperf utility was configured to report results accurate to 5% with 99% confidence. Table 6 summarizes the performance for the e1000e driver when it is run without and with a debugging interface. The debugging interface only reduces the throughput by 0.4% to 6.4%.

File system performance. Our second test involves the NFS file system, whose implementation uses around 200 exported functions exhibiting safety holes. The experiment consists of sequential block read and write phases based on

[8] http://www.netperf.org

Test		Without Diagnosys	With Diagnosys	Overhead
TCP_STREAM	Throughput	907.91 Mb/s	904.32 Mb/s	0.39%
UDP_STREAM	Throughput	951.00 Mb/s	947.73 Mb/s	0.34%
UDP_RR	Throughput	7371.69 Tx/s	6902.81 Tx/s	6.36%

Table 6: Performance of the e1000e driver

patterns generated by the IOzone file system benchmark[9] during which 8G of data are accessed. For this experiment, the client and server run on the same machine, connected using a loopback interface. Read and write operations are performed in the direct I/O mode with varying record sizes. With a debugging interface integrated into the `nfs` file system, we have recorded around 16 million calls to the interface wrapper functions when using a record size of 512 Kb. As shown in Table 7, the overhead varies between 3% and 11%, depending on the record size.

Record block size(Kb)	Without Diagnosys (Access rate - K/sec) read/write	With Diagnosys (Access rate - K/sec) read/write	Overhead read/write
128	45309/31672	42141/28072	6.99%/11.36%
256	49780/36577	48196/32900	3.18%/10.05%
512	49764/39957	45765/37981	8.03%/4.94%

Table 7: Performance of the NFS file system

6. RELATED WORK

In the last decade, studies have shown that kernel-level services, in particular device drivers, are responsible for the majority of OS crashes. Ganapathi *et al.* have found that 65% of all Windows XP crashes are due to device drivers [14]. Ten years ago, Chou *et al.* found that the fault rate in Linux drivers is 3–7 times higher than that of other parts of the kernel [5]. Palix *et al.* have shown that while this error rate is decreasing, Linux drivers still contain many defects [26]. They have also found that file systems have recently had a high fault rate, indeed even higher than that of drivers.

System robustness testing. Fault injection has been applied to the Linux kernel to evaluate the impact of various fault classes [1, 7]. Our work identifies the safety holes in kernel interfaces that explain their observations. Marinescu and Candea [21] focus on the returns of error codes from userspace library functions. These are analogous to our Null exit safety holes. Their approach, however, is not applicable to other types of safety holes.

Static bug finding. Model checking, theorem proving, and program analysis have been used to analyze OS code to find thousands of bugs [3, 9, 18, 27]. Nevertheless, these tools take time to run and the results require time and expertise to interpret. Thus, these tools are not well suited to the frequent modifications and tests that are typical of initial code development. A number of approaches have proposed to statically infer so-called *protocols*, describing expected sequences of function calls [9, 18, 19, 20, 28]. These approaches have focused on sequences of function calls that are expected to appear within a single function, rather than the specific interaction between a service and the rest of the kernel.

Some of our kinds of safety holes could be eliminated by the use of advanced type systems. For example, Bugrara and Aiken propose an analysis that differentiates between safe and unsafe userspace pointers in kernel code [4]. Their work, however, focuses on the kernel as a whole, and not on the interface between the kernel and a new service under development, thus potentially informing the service developer about faults in code with which he is not familiar.

Logging. Runtime logs are frequently insufficient for failure diagnosis especially in case of unexpected crashes [8]. *LogEnhancer* [32] enriches log messages with extra information, but does not create new messages. Diagnosys creates new log messages along the kernel-service boundary, where they can be most helpful to service developers.

Robust interfaces. LXFI [31] isolates kernel modules and includes the concept of *API integrity*, which allows developers to define the usage contract of kernel interfaces by annotating the source code. LXFI, however aims at limiting the security threat posed by the privileges granted to kernel modules, while Diagnosys focuses on various categories of common faults encountered in kernel code.

Healers automatically generates a robust interface to a user-level library without access to the source code [11]. It relies on fault injection to identify the set of assumptions that a library function makes about its arguments. Healers can obtain information about runtime values, such as array bounds, that may be difficult to detect using static analysis. However, Healers does not address safety hole kinds such as Lock that require calling-context information. Supporting Lock would require testing the state of all available locks, which would be expensive and are likely unknown.

Programming with contracts. A software *contract* represents the agreement between the developer of a component and its user on the component's functional behavior [13, 15, 22, 23]. Contracts include pre- and post-conditions, as well as invariants. A safety hole is essentially the dual of a contract, in that a contract describes properties that the context should have, while a safety hole describes properties that it should not have.

Contract inference is analogous to the execution of SHAna. Arnout and Meyer infer contracts based on exceptions found in .NET code [2]. Daikon infers invariants dynamically by running the program with multiple inputs and generalizing the observations [10]. Diagnosys targets situations that lead to unhandled exceptions, either in the kernel or the service code. Linux kernel execution is highly dependent on the particular architecture and devices involved, and thus a service developer would have to actively use Daikon in his own environment. SHAna allows the collection of safety holes to be centralized. Finally, only one of the invariants targeted by Daikon,[10] NonZero, may correspond to one of our safety hole kinds, namely INull. Daikon does not handle common safety hole kinds such as Free, or kernel-specific safety hole kinds such as Param, for user/pointer bugs.

The Extended Static Checker for Java (ESC/Java) [13] relies on programmer annotations to check method contracts. Annotation assistants such as Houdini [12] automate the inference of annotations. Houdini supports various exceptions involving arguments, such as NullPointerException and IndexOutOfBoundsException, but does not provide tests for the validity of allocated memory.

[9] http://www.iozone.org/

[10] http://groups.csail.mit.edu/pag/daikon/download/doc/daikon.html#Invariant-list

7. CONCLUSION

Defects in kernel-level services can cause the demise of the entire system, often leaving developers without any clue as to what went wrong. Debugging such problems is particularly challenging during the initial development of a service, when the code changes frequently and the developer is not necessarily aware of the usage preconditions of kernel interfaces. We have designed Diagnosys, a tool that detects safety holes in Linux kernel exported functions and supports the generation of a debugging interface, tailored for a particular service, according to this information. At runtime Diagnosys provides a crash-resilient logging system for recording information about risky uses of functions containing safety holes.

Using fault injection tests on 10 Linux kernel-level services, we have shown that our interface alerts the developer to the critical defects in his code. Using a driver for a Gigabit Ethernet device and a NFS file system, we have shown that the performance impact of our approach is within the limits of what is acceptable when testing a kernel-level service in the initial stages of development, and can even be used up to the phase of initial deployment.

8. REFERENCES

[1] A. Albinet, J. Arlat, and J.-C. Fabre. Characterization of the impact of faulty drivers on the robustness of the Linux kernel. In *DSN'04*, pages 867–876.

[2] K. Arnout and B. Meyer. Uncovering hidden contracts: The .net example. *Computer*, 36:48–55, 2003.

[3] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *EuroSys'06*, pages 73–85.

[4] S. Bugrara and A. Aiken. Verifying the safety of user pointer dereferences. In *IEEE Symposium on Security and Privacy*, pages 325–338, Oakland, CA, USA, 2008.

[5] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating systems errors. In *SOSP' 01*, pages 73–88, Banff, Canada.

[6] J. Corbet. Injecting faults into the kernel. `http://lwn.net/Articles/209257/`, November 2004.

[7] D. Cotroneo, R. Natella, and S. Russo. Assessment and improvement of hang detection in the Linux operating system. In *SRDS'09*, pages 288–294.

[8] Y. Ding, M. Haohui, X. Weiwei, T. Lin, Z. Yuanyuan, and P. Shankar. Sherlog: Error diagnosis by connecting clues from run-time logs. In *ASPLOS'10*, pages 143–154, Pittsburgh, PA, USA.

[9] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: a general approach to inferring errors in systems code. In *SOSP'01*, pages 57–72, Banff, Alberta, Canada.

[10] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69:35–45, December 2007.

[11] C. Fetzer and Z. Xiao. Healers: a toolkit for enhancing the robustness and security of existing applications. In *DSN'03*, pages 317–322, San Francisco, CA, USA.

[12] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In *FME'01*, pages 500–517, London, UK.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI'02*, pages 234–245.

[14] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *LISA'06*, pages 49–159, Washington, DC, USA.

[15] R. Hirschfeld, M. Perscheid, C. Schubert, and M. Appeltauer. Dynamic contract layers. In *SAC'10*, pages 2169–2175. ACM, 2010.

[16] M. Huth and M. Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2000.

[17] V. Kuznetsov, V. Chipounov, and G. Candea. Testing closed-source binary device drivers with DDT. In *USENIX Annual Technical Conference*, June 2010.

[18] J. L. Lawall, J. Brunel, N. Palix, R. R. Hansen, H. Stuart, and G. Muller. WYSIWIB: A declarative approach to finding API protocols and bugs in Linux code. In *DSN'09*, pages 43–52, Lisbon, Portugal.

[19] C. Le Goues and W. Weimer. Specification mining with few false positives. In *TACAS'09*, volume 5505 of *LNCS*, pages 292–306, York, UK.

[20] Z. Li and Y. Zhou. PR-miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE-13*, pages 306–315, Lisbon, Portugal, 2005.

[21] P. Marinescu and G. Candea. Efficient testing of recovery code using fault injection. *ACM Transactions on Computer Systems (TOCS)*, 29(3), Nov. 2011.

[22] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1st edition, 1988.

[23] C. Mills. *Using Design by Contract in C*. OnLamp.com, O'Reilly, 1st edition, October 2004.

[24] H. Nellitheertha. Reboot Linux faster using kexec. `http://www.ibm.com/developerworks/linux/library/l-kexec/index.html`, 2004.

[25] Y. Padioleau, J. L. Lawall, R. R. Hansen, and G. Muller. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys'08*, pages 247–260, Glasgow, Scotland.

[26] N. Palix, S. Saha, G. Thomas, C. Calvès, J. L. Lawall, and G. Muller. Faults in Linux: Ten years later. In *ASPLOS'11*, pages 305–318.

[27] H. Post and W. Küchlin. Integrated static analysis for Linux device driver verification. In *IFM'07*, pages 518–537, Oxford, UK.

[28] M. K. Ramanathan, A. Grama, and S. Jagannathan. Path-sensitive inference of function precedence protocols. In *ICSE'07*, pages 240–250.

[29] A. Rubini and J. Corbet. *Linux Device Drivers*, page 109. O'Reilly Media, second edition, 2001.

[30] L. Ryzhyk, P. Chubb, I. Kuz, and G. Heiser. Dingo: Taming device drivers. In *EuroSys'09*, pages 275–288.

[31] M. Yandong, C. Haogang, Z. Dong, W. Xi, Z. Nickolai, and K. M. Frans. Software fault isolation with API integrity and multi-principal modules. In *SOSP'11*.

[32] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *ASPLOS'11*, pages 3–14.

A formal approach for managing component-based architecture evolution

Abderrahman Mokni[1], Christelle Urtado[1], Sylvain Vauttier[1], Marianne Huchard[2], and Huaxi Yulin Zhang[3]

[1] LGI2P, Ecole Nationale Supérieure des Mines Alès, Nîmes, France, Abderrahman.Mokni@mines-ales.fr, Christelle.Urtado@mines-ales.fr, Sylvain.Vauttier@mines-ales.fr
[2] LIRMM, CNRS and Université de Montpellier, Montpellier, France, Marianne.Huchard@lirmm.fr
[3] Laboratoire MIS, Université de Picardie Jules Verne, Amiens, France, yulin.zhang@u-picardie.fr

Component-based software development (Cbsd) promotes a reuse-based approach to defining, implementing and composing loosely coupled independent software components into whole software systems [6]. While component reuse is crucial to shorten large-scale software systems development time, handling evolution in such processes is a significant issue [1]. Indeed, software systems have to evolve to extend their functionalities, correct bugs, improve performance and quality, or adapt to their environment. While unavoidable, software changes may engender several inconsistencies and system dysfunction if not analyzed and handled carefully. In turn, an ill-mastered evolution engenders software degradation, the loss of its evolvability and then its phase-out [3].

A famous problem of software evolution is software architecture erosion [5, 2]. It arises when modifications of the software implementation violate the design principles captured by its architecture. To increase confidence in reuse-centered, component-based software systems, all architecture descriptions must remain consistent and coherent with each other after every change.

While a lot of work has been dedicated to architectural modeling and evolution, there still is a lack of means and techniques to tackle architectural inconsistencies, and erosion in particular. Indeed, most existing approaches to architecture evolution hardly support the whole life-cycle of component-based software and only enable evolution of early stage models by propagating change impact to runtime models while evolution of runtime models are not fully dealt with, thus increasing the risks of architecture erosion.

This paper [4] proposes an approach and its implementation to automatically manage component-based architecture evolution at multiple abstraction levels in a manner that preserves architecture consistency and coherence all along the software lifecycle. The approach is based on the Dedal [7, 8] architectural model that explicitly models architectures at three abstraction levels, each corresponding to one of the three major steps of Cbsd – specification, implementation and deployment, thus granting a full evolution management process. Given a change request at any abstraction level, it transforms Dedal models into B formal models to analyze the requested change and generates an evolution plan

that guarantees the consistency of architecture descriptions and the coherence between them. The proposed approach is centered on a formal evolution management model that includes the generated B models, the architecture properties to preserve and a set of evolution rules. It is implemented as an Eclipse-based tool that generates B models from diagrammatic Dedal models and uses our specific solver to resolve architecture evolution. The overall approach is illustrated with a Home Automation Software case-study.

References

1. Hongyu Pei Breivold, Ivica Crnkovic, and Magnus Larsson. A systematic review of software architecture evolution research. *Information and Software Technology*, 54(1):16 – 40, 2012.
2. Lakshitha de Silva and Dharini Balasubramaniam. Controlling Software Architecture Erosion: A Survey. *Journal of Systems and Software*, 85(1):132–151, January 2012.
3. T. Mens and S. Demeyer. *Software Evolution*. Springer, 2008.
4. Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and H. Y. Zhang. A formal approach for managing component-based architecture evolution. *Science of Computer Programming, special issue of the 11th international symposium on Formal Aspects of Component Software*, (127):24–49, Octobre 2016.
5. Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
6. Hans Van Vliet. *Software Engineering: Principles and Practice*. Wiley Publishing, 3rd edition, 2008.
7. Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In *Proceedings of the 4th European Conference of Software Architecture*, volume 6285 of *Lecture Notes in Computer Science*, pages 295–310, Copenhagen, Denmark, August 2010. Springer.
8. Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component-based software development. In *Proceedings of the 11th of the International Conference on Generative Programming: Concepts and Experiences*, pages 70–79, Dresden, Germany, September 2012. ACM.

Session aux groupes de travail MFDL, IE et à AFADL

Méthodes Formelles dans le Développement Logiciel — Ingénierie des Exigences — Approches Formelles dans l'Assistance au Développement de Logiciels

Du cahier des charges à sa spécification

Imen Sayar et Jeanine Souquières

LORIA – CNRS UMR 7503 – Université de Lorraine
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex

Firstname.Lastname@loria.fr

Résumé. Le cahier des charges est un document de référence tout au long du développement d'un système. Il s'agit tout d'abord de le comprendre et de le structurer. Les liens entre ce document et la spécification en Event-B définie en termes de ses différents raffinements nous amène à utiliser des outils de vérification et de validation. Nous présentons quelques leçons sur le développement de la machine d'hémodialyse, avec l'introduction de patrons.

Mots clés. Cahier des charges, développement, spécification, raffinement, patron, validation, vérification, Event-B, outils.

1 Introduction

La compréhension des besoins est indispensable pour démarrer un processus formel de développement de logiciels. La qualité du cahier des charges, appelé *CdC* dans la suite du papier, affecte celle du logiciel obtenu. Ce document est souvent peu utilisable et difficile à utiliser par les parties prenantes tout au long du processus. Les rapports du Standish Group dont le premier date de 1994 [1], montrent qu'une mauvaise qualité des exigences entraîne des difficultés dans le développement et mène à des échecs et à des coûts importants en temps et en argent. Un soin accordé au *CdC* aide à réduire l'écart entre ce document et sa spécification formelle. Il améliore la qualité du logiciel final.

Notre approche commence par une étape de re-structuration des besoins. Celle-ci a pour objectif l'obtention d'un document lisible. Nous travaillons sur le *CdC* du client et le ré-écrivons sous forme de phrases courtes étiquetées en utilisant les recommandations d'Abrial [13] et l'outil ProR [8]. Différentes sortes de diagrammes sont disponibles pour aider à la compréhension des besoins [11]. La notion de patron avec ses paramètres est introduite au niveau du développement. Elle permet de décrire un sous-problème identifié et sa solution en réutilisant des connaissances acquises par l'expérience [7].

L'activité de vérification a pour objectif d'assurer la correction du modèle formel développé. La validation sert à montrer que le modèle satisfait les besoins du client. Dans notre approche, ces activités aident à améliorer la qualité du *CdC* et sa spécification formelle en Event-B [4]. Nous prenons en compte la validation en tant que processus rigoureux préparé dès le traitement des besoins et tout au long du développement de la spécification formelle [12]. Nous introduisons des termes formels dans le *CdC* structuré avec ProR, plug-in de la plateforme Rodin [1], [5].

La section 2 décrit rapidement notre approche et les outils utilisés. Un exemple de patron de développement est présenté dans la section 3. La section 4 présente quelques leçons retirées de plusieurs études de cas [2] avec utilisation du patron présenté dans la section précédente. Nous les illustrons par un système d'hémodialyse [10]. La section 5 conclut et décrit la suite de ce travail.

2 Notre approche

Un système informatique, voir figure 1, est composé de :

1. Rapport CHAOS du Standish Group (https ://www.standishgroup.com)
2. `http://dedale.loria.fr`

- son *CdC*, décrit à l'aide de l'outil ProR,
- sa spécification formelle, décrite en Event-B et
- les liens entre ce *CdC* et sa spécification formelle.

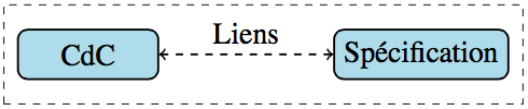


FIGURE 1 – Etat du système

L'ensemble *CdC*/*Spécification* évolue en permanence. Il est mis à jour grâce aux liens entre le *CdC* et la *Spécification* qui sont affinés. Cette mise à jour est réalisée grâce aux différents outils disponibles.

2.1 *CdC*

Le cahier des charges, informel au démarrage du développement, est ré-écrit sous forme d'une suite de phrases courtes et étiquetées. Une phrase peut contenir des termes formels intégrés dans le texte informel. Ces termes proviennent de la spécification formelle associée au *CdC*. Ses opérations sont celles des types utilisés, suites, ensembles et produit cartésien noté CP.

CdC is Sequence(*Sentence*)

Sentence is CP [ID: *TEXT*,
[Order: *INTEGER*,]
Description : *Set*(*Term*),
[*Sequence*(*Sentence*),]
[*Set*(*Term_ Type*)]]

Term is Informal_ Term | *Formal_ Term*

Term_ Type is Fact | *Functionality* | *Behavior* | *Obligation*

La figure 2 présente deux exemples d'exigences du cahier des charges de l'étude de cas de la machine d'hémodialyse. L'exigence *R-8-1-2* est un fils de *R-8-1*.

ID	*Description*
R-8-1	The software shall control the pressure at the AP transducer during initiation
R-8-1-2	The initiation is a working phase in the BEP component

FIGURE 2 – Exemple d'exigences avec ProR

2.2 Spécification

La spécification est définie en Event-B en utilisant le raffinement. La correspondance entre le *CdC* et sa spécification est définie par :

- *ID*. Il s'agit d'un commentaire.
- *Order*. L'ordre entre les événements dans la spécification est décrit par leur garde.
- *Informal_ Term*. Il s'agit d'un commentaire.
- *Formal_ Term*. Un terme formel introduit dans le *CdC* correspond à un élément d'Event-B ; il s'agit du nom d'une variable, d'une constante, d'un ensemble ou d'un événement.
- *Term_ Type*. Le type d'un terme est :
  - *Fact* pour des constantes, des ensembles et des variables,
  - *Functionality* pour des événements,

- *Obligation* pour des axiomes, invariants, théorèmes et gardes,
- *Behavior* pour l'animation et la simulation de machines.

2.3 Liens entre le *CdC* et sa spécification

Pour mémoriser le cahier des charges, nous prenons en compte ses différentes mises à jour et explicitons sa place dans le processus de développement. Pour cela, un ensemble de liens entre besoins et spécifications a été identifié et réalisé avec ProR. Ces liens sont automatiquement mis à jour et disponibles tout au long du développement. Initialement, nous n'avons pas de spécification formelle et il n'y a pas de terme formel dans le *CdC*. Au cours du développement, des termes formels sont intégrés dans le *CdC*.

Les informations pour la validation sont extraites du *CdC* en tenant compte du type de chaque terme de chaque phrase dans le modèle formel associé. Un terme formel introduit dans la spécification est introduit dans le *CdC*. Nous avons défini :

- un lien entre un terme formel et sa définition informelle dans le *CdC*. Il s'agit d'un élément du glossaire ;
- un lien entre ce terme formel, dénoté entre crochets dans le *CdC*, et sa définition dans la spécification Event-B. Ce lien est géré par ProR via une adresse.

2.4 Outils

Gestion des exigences. L'outil ProR, *plug-in* de Rodin, permet d'exprimer de manière hiérarchique des exigences [8]. Il commence par l'élicitation des exigences initiales et les hypothèses du domaine. Il ne propose pas de notation particulière mais un classement des artefacts, voir figure 2.

Les liens entre exigences et éléments du modèle Event-B en cours de construction sont créés manuellement et peuvent être annotés. Pour gérer la traçabilité entre exigences et modèles formels, ProR permet :

- de définir manuellement des liens depuis la spécification Event-B vers les exigences texte. Initialement, ces exigences sont informelles,
- d'insérer des éléments formels dans les exigences, ces éléments étant issus de la spécification Event-B.

La vérification. Nous cherchons à formaliser les propriétés du système. Ces propriétés sont abstraites et générales au début du développement puis elles se précisent. Tout au long du développement de la spécification formelle [2], nous prouvons sa correction via des preuves formelles. Les obligations de preuve, OPs, sont générées automatiquement ou semi-automatiquement par Rodin.

La validation. Cette activité est utilisée tout au long du processus de développement avec l'outil ProB [9] sous Rodin. Des informations nécessaires à la validation de la future spécification sont extraites de chaque exigence. Elles sont mises à jour par des termes formels au fur et à mesure du développement. La validation s'effectue par la preuve de propriétés. Il s'agit de raisonner en termes de propriétés auxquelles le futur système doit obéir.

3 Un exemple de patron de développement

La lecture du texte de référence du *CdC* [3] nous invite à étudier la notion de patrons pour réutiliser une solution dans ce développement. La présentation des besoins de la machine de l'hémodialyse utilise une forme particulière. Elle nous a conduit à définir plusieurs patrons [7] ou modèles génériques adaptés à notre approche dans laquelle nous manipulons des triplets *<CdC, Liens, Spec>*. Chaque patron générique est mémorisé ainsi que ses paramètres. La défintion du patron utilise le raffinement en Event-B, les preuves correspondantes et précise les éléments restant à décrire.

Exemple. Regardons le besoin *R-8* :

> *R-8 During initiation, if the software detects that the pressure at the AP Arterial Pressure transducer falls below the lower pressure limit, then the software shall stop the BP Blood Pumping and execute an alarm signal.*

Ce besoin a la même forme que la plupart des exigences de ce cahier des charges. Nous définissons un patron pour faciliter le développement de ce dernier.

3.1 Forme générique de ce besoin

R-i During actual phase, if the software detects an error on p [for more/less than n seconds], then the software shall stop the BP Blood Pumping and execute an alarm signal.

Ce patron a un paramètre p qui désigne l'élément physique de la machine d'hémodialyse.

3.2 Présentation du patron

Le besoin *R-i* paramétré par p a la même forme qu'un besoin *R-j* décrit précédemment et paramétré par *prev_p*. Le patron de développement associé à *R-i* crée sa spécification *R-i_Mch*, voir figure 3. Cette construction s'effectue en fonction du besoin *R-j* et de sa spécification *R-j_Mch* associée. Les éléments dénotés par "..." sont à définir par le développeur.

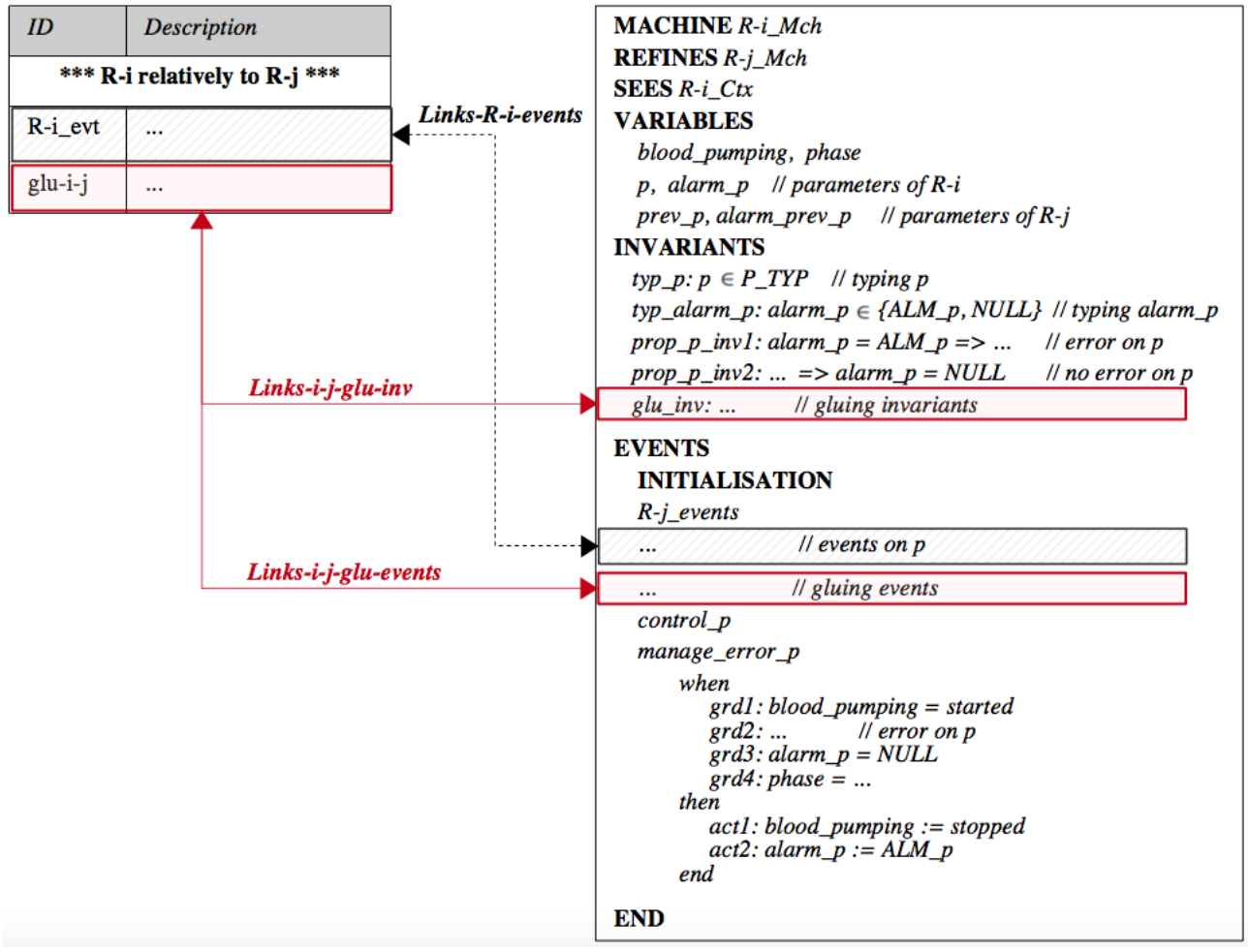


FIGURE 3 – Patron pour décrire *R-i* relativement à *R-j*

3.2.1 Réutilisation de la spécification R-j_Mch existante

La machine R-i_Mch est un raffinement de R-j_Mch. Les éléments de collage en Event-B doivent être définis.

CdC'. De nouveaux besoins *glu-i-j* sont ajoutés, voir la partie gauche de la figure 3. Ces besoins prennent en compte les différents cas en combinant les différentes valeurs de p et $prev\_p$. Ils contiennent des termes formels venant de la spécification.

Spec'. Des invariants de collage, comme *glu_ inv*, et des événements de collage sont introduits. Ils sont définis dans la partie droite de la figure 3.

Liens'. Les besoins *glu-i-j* sont liés aux invariants et événements de collage via les liens nommés respectivement *Links-i-j-glu-inv* et *Links-i-j-glu-events*.

3.2.2 Evénements liés au paramètre *p*

Ces événements correspondent à l'évolution du paramètre du patron. Ils doivent être décrits par le développeur. Les descriptions formelle et informelle de ces événements sont reliées via les liens *Links-R-i-events*, voir figure 3.

4 Leçons retirées des différentes études de cas

L'utilisation des outils aide à améliorer le processus de développement, que ce soit avec ProR ou avec les outils de vérification et de validation. Le *CdC* et ses liens sont automatiquement mis à jour lors de l'évolution de la spécification. Notre propos est illustré par le développement d'un système d'hémodialyse [10], système critique, hybride et interactif contrôlant sa machine. Nous abordons les points suivants :

– rôle des alarmes,
– forme particulière des besoins présentant des anomalies,
– rôle de l'invariant de collage de la spécification relativement au triplet $<$ *CdC, Liens, Spec* $>$.

4.1 Compréhension et analyse des besoins

La prise en compte des propriétés de sécurité, de vivacité et de terminaison sont examinées lors de la compréhension du *CdC*. Certaines ambiguïtés et imprécisions sont détectées très tôt dans le développement.

Dans l'étude de cas de la machine d'hémodialyse :

- les propriétés de sécurité sont clairement exprimées dans son *CdC*,
- la notion d'alarme dans ce système est à clarifier :
  . Existe-t'il une alarme commune à toutes ses composantes ?
  . Existe-t'il une alarme spécifique à chaque composante ?
  Peut-on alors déceler plusieurs alarmes différentes à un instant donné ?

4.2 Evolution du développement à l'aide d'un patron

Nous développons le besoin *R-8* sachant que le besoin *R-6* de même forme a été développé.

R-6 During initiation, if the software detects that the pressure at the VP transducer falls below the lower limit, then the software shall stop the BP and execute an alarm signal.

La spécification formelle de *R-6* est :

MACHINE *R−6_Mch*
SEES *R−6_Ctx*
VARIABLES
blood_pumping, *phase*
vp, *alarm_vp*
INVARIANTS
typ_vp: $vp \in Z$
prop_vp_inv1: $alarm\_vp = ALM\_vp \Rightarrow vp < lower\_press\_limit$
EVENTS
INITIALISATION
increase_vp_initiation
decrease_vp_initiation
control_vp
manage_deficit_vp
END

Nous utilisons le patron présenté dans la section 3 pour décrire le besoin *R-8* en termes de la spécification *R-6_mch*. Le paramètre formel *p* de *R-8* est *ap* et le paramètre *prev_p* de *R-6* est *vp*. Les éléments suivants doivent être complétés.

4.2.1 Réutilisation de la spécification R-6_Mch existante

CdC'. Les besoins concernant le collage entre *R-8* et *R-6* sont décrits par :

ID	*Description*
glu-8-6-1	If [ap] and [vp] fall below [lower_press_limit], then the software shall [manage_def icit_ap_def icit_vp]
glu-8-6-2	If [ap] falls below [lower_press_limit] and [vp] is normal, then the software shall [manage_def icit_ap]
glu-8-6-3	If [ap] and [vp] are normal, then the software does nothing
glu-8-6-4	If [ap] is normal and and [vp] falls below [lower_press_limit] , then the software shall [manage_def icit_vp]

Spec'. L'invariant de collage *glu_1* exprime la présence d'anomalies conjointes pour prendre en compte les besoins *R-8* et *R-6*. Il est défini par :

glu_1: $alarm\_ap = ALM\_ap \wedge alarm\_vp = ALM\_vp \Rightarrow ap < lower\_press\_limit \wedge vp < lower\_press\_limit$

L'événement de collage *manage_deficit_ap_deficit_vp* exprime le traitement simultané d'anomalies pour *R-8* et *R-6*. Il est défini comme suit :

EVENT manage_deficit_ap_deficit_vp
WHEN
grd1: $blood\_pumping = started$
grd2: $phase = initiation$
grd3: $alarm\_ap = NULL$
grd4: $alarm\_vp = NULL$
grd5: $ap < lower\_press\_limit$
grd6: $vp < lower\_press\_limit$
THEN
act1: $blood\_pumping := stopped$
act2: $alarm\_ap := ALM\_ap$
act3: $alarm\_vp := ALM\_vp$
END

4.2.2 Evénements liés au paramètre *ap*

Deux événements *decrease_ap_initiation* et *increase_ap_initiation* font évoluer le paramètre *ap*. L'action de décroître *ap* est définie comme suit :

EVENT decrease_ap_initiation
WHEN
grd1: $blood\_pumping = started$
grd2: $phase = initiation$
grd3: $alarm\_ap = NULL$
grd4: $ap \geq lower\_press\_limit$
THEN
act1: $ap := ap - 10$
END

4.3 Amélioration du *CdC* à partir de la spécification

Le développement de la spécification formelle permet de détecter des lacunes dans le *CdC* avec les outils de vérification et de validation. Ces lacunes concernent des oublis, des imprécisions et des incohérences. Elles n'ont pas été détectées dans l'étape de compréhension et analyse des besoins.

Exemple. Regardons le besoin *R-9*.

R-9 While connecting the patient, if the software detects that the pressure at the VP transducer exceeds + 450 mmHg for more than 3 seconds, then the software shall stop the BP and execute an alarm signal.

ID	*Description*
init1	The [blood_pumping] is [stopped]
init2	There is no pressure at the [ap]
init3	There is no pressure at the [vp]
init4	All the alarms are disabled

MACHINE *R−9_Mch*
REFINES *R−8_Mch*
VARIABLES
ap, *phase*, *alarm_ap*, *blood_pumping*, *vp*, *alarm_vp*
EVENTS
INITIALISATION
init_act1: $blood\_pumping := stopped$
init_act2:...
...
END

Figure 4 – Etat initial

Son modèle Event-B est défini par un raffinement de la *Machine R-8_Mch*. Nous avons détecté les anomalies suivantes :

- *Invariant de collage.* Dans le document initial, chaque besoin est isolé. Il est décrit séparément des autres besoins et ne tient pas compte de possibles interactions entre eux.
- *Etat initial.* L'activité de validation nécessite un état initial. Cet état n'a pas été mentionné explicitement dans le *CdC*. Nous avons proposé l'ajout de phrases décrivant cet état, voir figure 4.
- *Omission.* L'alarme peut être commune à toutes les composantes du système ou bien définie pour chaque composant. Les preuves aident à détecter les cas prévus dans la spécification. Par exemple, le choix d'une alarme générale signifie qu'il n'y a pas de changement de valeur de sa variable abstraite.

5 Conclusion et perspectives

Dans ce papier, nous avons abordé la gestion des besoins de la machine d'hémodialyse, ceux-ci sont décrits dans une forme identique [10]. Le document décrit les cas anormaux et présente la gestion des alarmes. L'utilisation et la ré-utilisation d'une spécification existante aborde le rôle de l'invariant de collage dans la spécification. Le patron génère automatiquement une partie de la spécification en cours de construction. La notion de collage est présentée via le *CdC*. L'effort de vérification est réduit. Les patrons utilisés dans cette

étude de cas peuvent être utilisés et adaptés à d'autres études de cas, comme celle concernant le système hybride de contrôle du train d'atterrissage d'un avion [6]. En effet, la présentation de leurs besoins est identique.

La validation en tant que processus rigoureux démarre avec la structuration des besoins, avant que la spécification associée ne soit introduite, jusqu'à l'animation des modèles Event-B. Ces modèles sont validés relativement aux besoins du client, en vue de détecter des problèmes [12].

Les outils disponibles dans la plateforme Rodin ont un rôle important tout au long du processus de développement. Nous utilisons l'outil ProR pour la ré-écriture des besoins, pour leur hiérarchisation et pour la mise à jour du *CdC* et des liens avec sa spécification. Nous utilisons les outils de vérification et de validation pour assurer la correction de la spécification avec les générateurs d'obligations de preuve, les prouveurs et l'animateur/model-checker ProB.

Pour la suite de notre travail, il est important de décrire rigoureusement les paramètres intervenant dans la prise en compte d'un besoin par rapport à un système existant. Ces paramètres servent à définir des patrons afin de réutiliser des modèles existants et corrects. L'étude de l'apport de l'étape d'analyse sert à la détection de lacunes dans le système existant.

Dans notre approche, nous étudions des systèmes hybrides. Ce sont des systèmes discrets et temporels qui fonctionnent dans un environnement continu. De tels systèmes combinent des aspects matériels et logiciels. Il serait important de prendre en compte les contraintes de l'environnement et de ses hypothèses relativement aux nouveaux besoins.

Références

[1] Rodin platform, http ://wiki.event-b.org.

[2] J-R. Abrial. B : passé, présent, futur. *Technique et Science Informatiques*, 22(1) :89–118, 2003.

[3] J.-R. Abrial. Faultless Systems : Yes We Can ! *IEEE Computer*, 42(9) :30–36, 2009.

[4] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering.* Cambridge University Press, 2010.

[5] J-R. Abrial, M. J. Butler, S. Hallerstede, T. Son Hoang, F. Mehta, and L. Voisin. Rodin : an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6) :447–466, 2010.

[6] F. Boniol and V. Wiels. Landing Gear System case Study. In *ABZ Conference, Communications in Computer and Information Science, Springer*, volume 433, pages 1–18, 2014.

[7] T. S. Hoang, A. Fürst, and J.-R. Abrial. Event-B Patterns and their Tool Support. *Software and System Modeling*, 12(2) :229–244, 2013.

[8] M. Jastram. ProR, an Open Source Platform for Requirements Engineering based RIF. *SEISCONF*, 2010.

[9] M. Leuschel and M. J. Butler. ProB : A Model Checker for B. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *International Symposium of Formal Methods Europe, Pisa, Italy, Proceedings*, volume 2805 of *LNCS*, pages 855–874. Springer, 2003.

[10] A. Mashkoor. The Hemodialysis Machine Case Study. In *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 5th International Conference, ABZ 2016, Linz, Austria, Proceedings*, pages 329–343. Springer International Publishing, 2016.

[11] C. Ponsard, R. Darimont, and A. Michot. Combining Models, Diagrams and Tables for Efficient Requirements Engineering : Lessons Learned from the Industry. In *Actes du XXXIIIème Congrès INFORSID, Biarritz, France, May 26-29*, pages 235–250, 2015.

[12] I. Sayar and J. Souquières. La Validation dans le Processus de Développement. In *Actes du XXXIVème Congrès INFORSID, Grenoble, France, May 31 - June 3*, pages 67–82, 2016.

[13] W. Su, J.-R. Abrial, R. Huang, and H. Zhu. From Requirements to Development : Methodology and Example. In *13th International Conference on Formal Engineering Methods, Durham, UK*, pages 437–455, 2011.

Sur l'assignation de buts comportementaux à des coalitions d'agents

Christophe Chareton
LORIA CNRS
54506 Vandœuvre-lès-Nancy

Julien Brunel
ONERA/DTIS
31055 Toulouse cedex 2

David Chemouil
ONERA/DTIS
31055 Toulouse cedex 2

Résumé

Dans cet article, nous présentons un cadre de modélisation formelle pour l'ingénierie du besoin qui prenne simultanément en compte les buts comportementaux et les agents. Pour ce faire, nous introduisons un langage noyau, appelé KHI, ainsi que sa sémantique dans une logique de stratégies appelée USL. Dans KHI, les agents sont décrits par leurs capacités et les buts sont définis par des formules de logique temporelle linéaire. Une « assignation » associe alors chacun des buts à un ensemble (une coalition) d'agents, qui sont responsables de sa satisfaction. Nous présentons et discutons ensuite différents critères de correction pour cette relation d'assignation. Ceux-ci permettent d'évaluer la « pertinence » d'une assignation de buts à des coalitions. Ils diffèrent selon les interactions qu'ils permettent entre les coalitions d'agents. Nous proposons alors une procédure décidable de vérification pour la satisfaction des critères de correction pour l'assignation. Elle consiste à réduire la satisfaction des critères à des instances du problème de *model-checking* pour des formules d'USL dans une structure dérivée des capacités des agents.

1 Contexte

Si, en toute rigueur, la discipline de la modélisation du besoin ne se restreint pas à elles seules [17, 14], les approches dites par buts [18] ou par agents [2, 9] ont le vent en poupe dans la communauté idoine (cf. les citations précédentes mais aussi [12, 15]).

En KAOS [18], la question première est de déterminer les besoins dont il faut tenir compte pour rendre compte d'un *système* au sein d'un *environnement,* le tout formant un *système global* à mettre au point. Celui-ci doit répondre à des *buts* et est constitué *d'agents* (entités actives).

Un *but* est défini comme un *énoncé prescriptif* sous la responsabilité d'agents du système global. Les buts peuvent être de toutes sortes (on retrouve les traditionnelles taxonomies autour des buts *non-fonctionnels* [11]). Mais on distingue en particulier les buts *comportementaux* qui caractérisent des traces et peuvent donc faire l'objet d'une formalisation dans une logique temporelle telle que LTL.

Bien que partageant superficiellement de nombreuses notions avec KAOS, TROPOS se concentre avant tout sur la notion *d'acteur,* défini comme un agent *intentionnel*. Un tel agent est muni de buts qu'il souhaite voir remplis mais dont la satisfaction, partielle comme complète, n'est pas nécessairement de sa responsabilité. Celle-ci peut être déléguée à d'autres acteurs. TROPOS [2] pousse ainsi à l'explicitation des liens de dépendance et de collaboration entre acteurs. Ceci s'explique en particulier par le fait que les systèmes visés par la méthode sont susceptibles de comprendre des acteurs « humains » ou institutionnels.

TROPOS a aussi fait l'objet d'une proposition formelle visant à étudier dans quelle mesure des acteurs peuvent contribuer à satisfaire des buts pour d'autres acteurs. L'approche en question [9, 10] introduit à cette fin les notions, dites « sociales », de *rôle, d'engagement (commitment)* et de *protocole.* Le rôle représente le comportement *attendu* des acteurs. Une *assignation* de rôles à des acteurs est alors évaluée au moyen d'un critère de correction. Celui-ci revient essentiellement à vérifier que les capacités d'un acteur entraînent les conséquents des engagements où le rôle assigné apparaît comme débiteur.

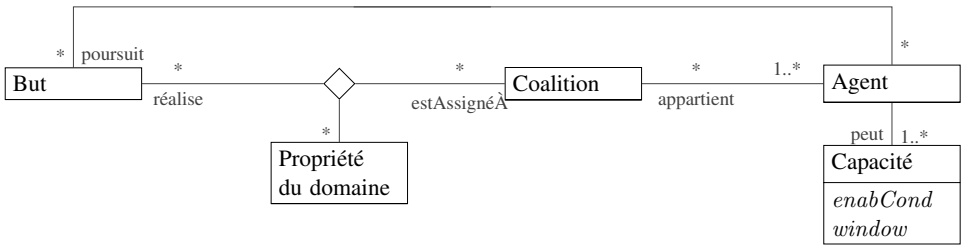


FIGURE 1 – Métamodèle du langage KHI.

Si la dimension « sociale » de cette proposition est plus forte qu'en KAOS, un certain nombre de faiblesses demeurent. Premièrement, on ne peut pas assigner de rôle à un ensemble d'acteurs qui collaboreraient pour l'assurer. Comme en KAOS, la question du partage de variables d'état entre plusieurs acteurs n'est pas traitée ; et les capacités des acteurs demeurent indépendantes du contexte. Surtout, la restriction du formalisme à la logique propositionnelle constitue une lacune importante : l'emploi d'un formalisme temporel, à l'image de ce qui est fait en KAOS, permettrait de véritablement caractériser des comportements attendus des acteurs.

Ces considérations nous ont mené à l'étude formelle ce que nous avons appelé le *problème de l'assignation.* Informellement, étant donnée une assignation de buts à des coalitions (ensembles) d'agents, la question est de déterminer dans quelle mesure ces dernières sont en mesure de satisfaire les premiers, y compris en tenant compte des interactions avec les autres coalitions. Cette question nous a amenés à définir un langage de modélisation appelé KHI et à élaborer une logique temporelle multi-agent, USL, fournissant un domaine sémantique permettant de formaliser et d'évaluer le problème de l'assignation selon différents critères.

2 KHI

Le cadre de modélisation KHI (figure 1) comprend un ensemble restreint de concepts destinés à permettre la mise en œuvre d'une modélisation du besoin *à la fois* par buts et par agents. Il peut en un sens être vu comme une union enrichie (des aspects principaux) de KAOS et des variations autour de TROPOS. Une première version de KHI a été introduite dans [4, 5] ; nous présentons ici une version simplifiée d'une seconde version, plus aboutie et décrite en détail dans [3].

Comme en TROPOS, les buts sont poursuivis par des agents, possiblement mais pas nécessairement aptes à contribuer à leur satisfaction. Les agents disposent de capacités plus fines que dans les propositions précédentes : une capacité ne peut s'exercer que dans certaines conditions *($enabCond$)* et consiste à pouvoir agir sur une variable du système mais seulement dans une certaine fenêtre *($window$)*, c'est-à-dire dans un ensemble fini de valeurs possible. Originalité supplémentaire : les agents sont regroupés en coalitions (ensembles) qui se voient assigner les buts à remplir (autrement dit, l'assignation n'est pas restreinte à des agents isolés).

Par ailleurs, la formalisation des buts suit le même principe qu'en KAOS : les buts sont décrits dans la logique LTL. Ces formules de LTL apparaîtront comme sous-formules de formules de la logique USL sur laquelle s'appuie la formalisation complète de KHI. Or cette logique repose sur des modèles admettant des exécutions finies. La sémantique utilisée pour LTL pour le critère de correction du raffinement des buts par des sous-buts et de l'opérationnalisation s'appuie donc sur des traces possiblement finies (et non-vides), de manière par ailleurs standard (techniquement comme la logique LTL^n d'[13]).

L'existence d'exécutions finies résulte du choix de modélisation de l'assignation entre buts et coalitions d'agents : celle-ci est définie comme une application de l'ensemble des buts dans les coalitions non vides d'agents (donc de type But $\to \mathscr{P}_{>0}(\text{Agent})$). On remarque qu'il est tout à fait possible pour un agent de se retrouver dans plusieurs coalitions. Dès lors, la participation de cet agent à ces différentes coalitions peut induire des spécifications contradictoires pour son comportement. Cet agent pourra alors

être engagé envers la réalisation d'actions *incompatibles* entre elles, depuis un même état. La présence d'exécutions finies dans notre formalisme provient donc du fait que nous avons souhaité favoriser la description fine des capacités des agents à composer leurs comportements pour la satisfaction de différents buts. Par ailleurs, il faut noter que notre formalisme permet ensuite de spécifier des modèles où les exécutions sont forcément infinies.

3 Problème de l'assignation

L'évaluation d'une assignation ne se fait pas de manière binaire. Un ensemble de critères permet de la caractériser plus finement ; en particulier en déterminant si des coalitions peuvent interagir de sorte à voir assurée la satisfaction de leurs buts :

Correction locale Ce critère consiste à s'assurer que chaque but est assigné à une coalition capable d'assurer sa satisfaction, quoi que fassent les autres agents.

Correction globale Le critère de correction locale est insuffisant dès lors qu'un agent appartient à plusieurs coalitions, car rien ne dit qu'il est apte à participer à toutes ces coalitions (en vue de réaliser leurs buts respectifs) *à la fois* (les comportements demandés pourraient être contradictoires). Le critère de *correction globale* demande donc s'il y a *un* comportement (par agent) qui permette à chaque coalition de satisfaire les buts qui lui sont assignés. Cet unique comportement permet de s'assurer de la cohérence des choix des agents.

Collaboration Le critère précédent peut être trop fort au sens où il stipule que chaque coalition doit pouvoir satisfaire ses buts assignés, quoi que fassent les autres agents. Or on peut souhaiter que certaines coalitions *collaborent* avec d'autres pour assurer la satisfaction de leurs buts.

Contribution Les trois critères précédents reposent sur l'hypothèse qu'il est possible de contrôler (et spécifier) tous les agents dans le système. Or il se pourrait que ce ne soit pas le cas. Le critère de *contribution* pose alors la question de savoir si une coalition indépendante, en satisfaisant ses propres buts assignés, est en mesure de produire des « effets de bord » qui contribuent à la satisfaction des buts assignés à d'autres coalitions.

Ces critères sont tous traduits en problèmes de *model-checking* dans la logique USL. On aboutit donc à des problèmes de la forme $\mathfrak{K} \models c$, où c est une formule d'USL traduisant un critère et où $\mathfrak{K}$ est une CGS *(Concurrent Game structure)*, soit un système de transitions particulier.

Intuitivement, les états de la CGS $\mathfrak{K}$ sont déterminés par les valuations possibles des variables décrivant le système (et compatibles avec les propriétés du domaine). En ce qui concerne la fonction de transition, les états suivants d'un état donné dépendent de celui-ci ainsi que de l'intersection des choix des agents, choix eux-mêmes déterminés par leurs capacités (pour cette raison, la fenêtre d'action d'un agent doit être exprimée comme un ensemble fini de valeurs possibles pour une variable donnée).

Pour la formule c, il s'agit à chaque fois d'exprimer que des coalitions d'agents sont en mesure d'assurer la satisfaction de propriétés temporelles, en tenant compte du fait que les autres coalitions agissent elles-aussi et qu'un agent peut appartenir à plusieurs coalitions. Les coalitions pouvant poursuivre des objectifs variés, il faut pouvoir spécifier qu'un agent peut enrichir de *différentes* manières son comportement selon les objectifs auxquels il concourt.

4 Logique des stratégies actualisables

Ces motifs mènent naturellement vers les *logiques temporelles multi-agents*. Toutefois, aucune proposition ne présentant les caractéristiques nécessaires [1] à l'élaboration des critères, nous avons défini notre propre logique : USL. Dans les logiques telles qu'ATL [1] ou SL [8, 16], chaque agent est muni d'une *stratégie* qui, en fonction du déroulement du jeu jusqu'alors, indique quelle action il choisit, ce

1. En particulier, donc, la possibilité pour un agent de suivre diverses « stratégies » et de les raffiner de diverses façons.

qui concourt à établir une décision et donc à déterminer l'état successeur. Eu égard à KHI, cette construction contribue à matérialiser la notion selon laquelle un agent a le moyen (lire : une stratégie) d'assurer une propriété. Toutefois, ici, en conformité avec les critères d'assignation, il est nécessaire de pouvoir composer des stratégies de plusieurs manières différentes. Dans USL, un agent auquel est assigné une stratégie ς peut se voir à nouveau assigner une autre stratégie ς'. Il composera alors son comportement de manière à satisfaire à la fois ς et ς'. On dit qu'il *raffine* sa stratégie. Ceci nous a menés, pour USL, à l'utilisation de stratégies *non-déterministes :*

Définition 4.1 (Multistratégie)**.** *Une* multistratégie *est une application qui, à tout déroulement*[2] *d'un jeu, associe un ensemble non-vide d'actions.*

Nous sommes maintenant en mesure de définir une syntaxe pour USL, avec les contraintes suivantes : *(a)* pouvoir raisonner sur les multistratégies, ce qui mène à l'introduction de modalités *ad hoc* ; *(b)* pouvoir se référer à des multistratégies données, d'où une notion de *variable* associée (et donc de *quantificateur*) ; *(c)* pouvoir composer, pour un même agent, plusieurs multistratégies différentes dans des sous-formules distinctes, et ce sans *révoquer* les multistratégies déjà associées, ce qui nécessite des opérateurs liant ou révoquant explicitement une multistratégie à un agent.

Définition 4.2 (Formules d'USL)**.** *Soient Ag un ensemble d'agents, At un ensemble de propositions, et X un ensemble de variables de multistratégies. Alors, l'ensemble des* pseudoformules d'USL *sur (Ag, At, X) est engendré par la grammaire suivante :*

— *Pseudoformules d'états :* $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \langle\langle x \rangle\rangle \varphi \mid (A \rhd x)\psi \mid (A \not\rhd x)\psi$

— *Pseudoformules de chemins :* $\psi ::= \varphi \mid \neg\psi \mid \psi \wedge \psi \mid \mathsf{X}\ \psi \mid \psi\ \mathsf{U}\ \psi$

où $p \in At$, $x \in X$ et $A \subseteq Ag$.

Une formule (bien formée) *d'*USL *est une pseudoformule dans laquelle chaque variable de multistratégie introduite par un quantificateur est fraîche par rapport à la portée dans laquelle elle apparaît.*

Naturellement, lors de la détermination de la possible satisfaction d'une formule dans une CGS, la présence de quantificateurs et d'opérateurs de liaison et révocation doit être prise en compte dans un *contexte* (d'évaluation). Un tel contexte κ est un couple comprenant : *(a)* une *assignation* α, qui a pour objet de mémoriser la multistratégie effective associée à chaque variable liée rencontrée jusqu'alors ; *(b)* et un *engagement* γ, c'est-à-dire l'inventaire, pour chaque agent, des (variables de) multistratégies selon lesquelles il joue. Ce couple doit par ailleurs être cohérent : toute variable de multistratégie listée dans l'engagement doit disposer d'une instanciation, donnée par l'assignation.

Nous renvoyons le lecteur à [6] pour une exposition plus fine, qui fait appel à un certain nombre de détails techniques, mais nous donnons ici une intuition des points saillants de la sémantique :

— L'opérateur $\langle\langle x \rangle\rangle$ est un quantificateur existentiel sur les multistratégies : une formule $\langle\langle x \rangle\rangle \varphi$ est vraie dans un état s, dans un contexte κ, ssi il existe une multistratégie ς t. q. la formule φ est vraie dans le contexte κ enrichi du fait que x est instanciée par ς. On peut aussi définir un quantificateur universel de la sorte : $[\![x]\!]\varphi \coloneqq \neg\langle\langle x \rangle\rangle\neg\varphi$.

— Une formule $(A \rhd x)\psi$ est vraie dans un état s, dans le contexte κ, ssi la formule ψ est vraie dans toute exécution *issue* de s et $\kappa[A \oplus x]$, où $\kappa[A \oplus x]$ est le contexte κ enrichi du fait que les agents dans A sont maintenant liés à la multistratégie instanciant x dans κ (en plus des multistratégies auxquelles ils étaient éventuellement déjà liés).

Les *issues* $out(\kappa, s)$ d'un état dans un certain contexte constituent les exécutions possibles à partir de cet état dans la CGS si chaque agent ne joue que des actions autorisées par toutes les multistratégies auxquelles il est lié dans ce contexte.

Il faut remarquer que parmi ces issues, certaines exécutions peuvent être finies, ce en raison de la possibilité pour un agent de jouer en même temps selon des multistratégies contradictoires (c'est-à-dire renvoyant des ensembles disjoints d'actions). Ceci implique l'usage d'une sémantique idoine pour les opérateurs temporels dans les formules de chemin.

2. C'est-à-dire un préfixe non-vide d'exécution dans la CGS.

— $(A \not\rhd x)$ délie les agents dans A de la multistratégie instanciant x dans le contexte courant.

Theorème 4.1 ([6]). *Pour finir, on remarque :* (a) USL *est strictement plus expressive que* SL *[8, 16] ;* (b) *comme pour cette dernière, la satisfaisabilité est indécidable ;* (c) *en revanche, le* model-checking *sur une* CGS finie [3] *est décidable, quoique en temps non-élémentaire, ce qui est aussi le cas de* SL.

5 Évaluation d'une assignation

Les caractéristiques uniques d'USL (en particulier le raffinement de multistratégies) permettent de formaliser les critères de correction présentés en § 3. On ne présente pas ici la procédure (complexe) de construction d'une CGS *finie* à partir d'un modèle KHI, cf. [3, 7]. Cette finitude dépend en particulier du type de formules atomiques et de la forme des fenêtres que KHI permet de spécifier. Elle rend décidable la vérification des critères ci-dessous.

Soit un modèle KHI et soit G un ensemble de buts. On note $\mathscr{A}$ l'application qui à tout but associe la coalition qui lui est assignée. Étant donnée une coalition $A = \{a_1, \ldots, a_n\}$ et $\overrightarrow{x}$ un vecteur de variables de multistratégies ; on note $(A \rhd \overrightarrow{x})$ pour $(a_1 \rhd x^{a_1}) \ldots (a_n \rhd x^{a_n})$.

Correction locale L'assignation est localement correcte ssi pour tout but $g \in G$, il existe un vecteur de multistratégies t. q., en les jouant, les agents concernés peuvent assurer la satisfaction de g :

$$\mathrm{LC}_{\mathscr{A}}[G] \coloneqq \bigwedge_{g \in G} \left[\langle\!\langle \overrightarrow{x_g} \rangle\!\rangle (\mathscr{A}_g \rhd \overrightarrow{x_g}) g\right]$$

Correction globale Ici, on considère un unique vecteur de multistratégies, ce qui impose aux agents de jouer en cohérence ($\overrightarrow{x_g}$ représente la partie de $\overrightarrow{x}$ qui concerne les agents dans $\mathscr{A}_g$) :

$$\mathrm{GC}_{\mathscr{A}}[G] \coloneqq \langle\!\langle \overrightarrow{x} \rangle\!\rangle \Big[\bigwedge_{g \in G} (\mathscr{A}_g \rhd \overrightarrow{x_g}) g \Big]$$

Collaboration Pour qu'une coalition associée à un but h collabore à la satisfaction de l'ensemble de buts G, il faut un vecteur de multistratégies $\overrightarrow{x_h}$ t. q. si les agents dans $\mathscr{A}_h$ les jouent, alors ces multistratégies assurent *à la fois* que le but h est satisfait et que l'évolution du modèle est contrainte de sorte à ce que l'assignation devienne globalement correcte pour G. Soit donc h un but t. q. $h \notin G$. :

$$\mathrm{Coll}_{\mathscr{A}}[h, G] \coloneqq \langle\!\langle \overrightarrow{x_h} \rangle\!\rangle (\mathscr{A}_h \rhd \overrightarrow{x_h})(h \wedge \mathrm{GC}_{\mathscr{A}_G}[G])$$

Contribution La contribution est définie comme une variante universellement quantifiée de la collaboration ; les agents concernés doivent pouvoir assurer h et tout vecteur de multistratégies qui permet aux dits agents d'assurer h doit aussi contraindre l'évolution du système de sorte à ce que l'assignation devienne globalement correcte pour G :

$$\mathrm{Contr}_{\mathscr{A}}[h, G] \coloneqq \left[\langle\!\langle y_h \rangle\!\rangle (\mathscr{A}_h \rhd \overrightarrow{y_h}) h\right] \wedge \left[[\![\overrightarrow{x_h}]\!] \Big\{ (\mathscr{A}_h \rhd \overrightarrow{x_h}) h \rightarrow \big((\mathscr{A}_h \rhd \overrightarrow{x_h})\, \mathrm{GC}_{\mathscr{A}_G}[G] \big) \Big\} \right]$$

6 Perspectives

KHI constitue à notre connaissance la première proposition de cadre de modélisation *formelle* pour l'ingénierie du besoin qui prenne simultanément en compte les buts comportementaux et les agents.

Ceci étant, le cadre doit clairement être amélioré. Tout d'abord, la sémantique de KHI ne fait pas appel à toute l'expressivité d'USL. Une question importante en termes d'applicabilité de nos propositions serait de déterminer s'il est possible de caractériser un fragment d'USL suffisant pour la traduction de KHI et pour lequel le *model-checking* disposerait d'une complexité « raisonnable » en pratique.

3. C'est-à-dire t. q. les ensembles d'états et d'actions sont finis.

De son côté, USL constitue aussi une proposition originale. Elle bénéficie par ailleurs de propriétés métathéoriques comparables aux logiques similaires et dispose d'un fort pouvoir expressif dont il s'agirait de creuser les conséquences en profondeur.

Hors de l'ingénierie du besoin proprement dite, USL pourrait bien avoir des applications intéressantes. Ainsi de l'analyse de « systèmes de systèmes », ensembles dans lesquels, en particulier, les sous-systèmes apparaissent comme des entités autonomes dont les capacités sont connues et qui disposent de leurs objectifs propres en sus de ceux du système global. Une autre application possible réside dans la sécurité : en effet, USL permet en principe de raisonner sur des agents dont les objectifs sont malveillants.

Pour finir, USL permet de raisonner sur le comportement (les multistratégies) possible des agents. Une question supplémentaire serait celle du comportement *effectif* du système. Cette question appelle certainement plusieurs directions de recherche autour de la *synthèse* de contrôleurs, de normes, de stratégies...

Références

[1] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5) :672–713, 2002.

[2] J. Castro, M. Kolp, and J. Mylopoulos. A requirements-driven development methodology. In *Proceedings of the 13th International Conference on Advanced Information Systems Engineering*, CAiSE '01, pages 108–123, London, UK, UK, 2001. Springer-Verlag.

[3] C. Chareton. *Modélisation formelle d'exigences et logiques temporelles multi-agents*. PhD thesis, Institut supérieur de l'aéronautique et de l'espace (ISAE), Université de Toulouse, 2014.

[4] C. Chareton, J. Brunel, and D. Chemouil. A Formal Treatment of Agents, Goals and Operations Using Alternating-Time Temporal Logic. In *Formal Methods, Foundations and Applications (SBMF)*, 2011.

[5] C. Chareton, J. Brunel, and D. Chemouil. Vers une sémantique des jeux pour un langage d'ingénierie des exigences par buts et agents. In *Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL)*, 2012.

[6] C. Chareton, J. Brunel, and D. Chemouil. A Logic with Revocable and Refinable Strategies. *Information and Computation*, 242 :157–182, 2015.

[7] C. Chareton, J. Brunel, and D. Chemouil. Evaluating the assignment of behavioral goals to coalitions of agents. In *Proc. 18th Brazilian Symposium, SBMF 2015, Belo Horizonte, Brazil, 2015*. Springer, 2015.

[8] K. Chatterjee, T. A. Henzinger, and N. Piterman. Strategy logic. *Inf. Comput.*, 208(6) :677–693, 2010.

[9] A. K. Chopra, F. Dalpiaz, P. Giorgini, and J. Mylopoulos. Modeling and reasoning about service-oriented applications via goals and commitments. In *Proceedings of the 22nd International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 6051 of *LNCS*, pages 113–128. Springer, 2010.

[10] A. K. Chopra and M. P. Singh. Multiagent commitment alignment. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 937–944. IFAAMAS, 2009.

[11] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[12] E. Dubois, P. Du Bois, and M. Petit. ALBERT : An agent-oriented language for building and eliciting requirements for real-time systems. In *HICSS (4)*, pages 713–722, 1994.

[13] C. Eisner, D. Fisman, J. Havlicek, Y. Lustig, A. McIsaac, and D. V. Campenhout. Reasoning with temporal logic on truncated paths. In *Proceedings of the 15th International Conference on Computer Aided Verification (CAV)*, pages 27–39, 2003.

[14] M. Jackson. *Problem Frames*. Springer, 2001.

[15] T. P. Kelly and R. A. Weaver. The goal structuring notation - a safety argument notation. In *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[16] F. Mogavero, A. Murano, and M. Y. Vardi. Reasoning about strategies. In *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2010*.

[17] B. Nuseibeh and S. M. Easterbrook. Requirements engineering : a roadmap. In *22nd International Conference on Software Engineering, Future of Software Engineering Track*, 2000.

[18] A. van Lamsweerde. *Requirements Engineering — From System Goals to UML Models to Software Specifications*. Wiley, 2009.

Intégration des (multi-)exigences tout au long du développement des systèmes complexes

Florian Galinier, Jean-Michel Bruel, Sophie Ebersold, Bertrand Meyer*

IRIT, Université de Toulouse, Toulouse, France
**Également Software Engineering Lab, Innopolis University, Russie et Politecnico di Milano, Italie*

1 Introduction

La création de systèmes complexes implique de nombreux acteurs, provenant de domaines différents. En raison de cette hétérogénéité, la spécification de ces systèmes est réalisée à l'aide d'outils différents, comme des documents textuels, des bases de données d'exigences, des diagrammes SysML (*Systems Modeling Language*)... Un des défis de l'IS (Ingénierie Système) est d'établir la cohérence et les liens entre ces différents artefacts dans l'objectif d'assurer la qualité du produit final.

En effet, il existe à ce jour un manque de cohérence entre les différentes vues de ces systèmes qui rend plus difficile l'analyse des exigences et la détection des conflits. Une approche multi-vues, avec un unique langage ou avec une abstraction commune des artefacts de spécification devrait ainsi permettre de détecter de telles incohérences en amont et faciliter les communications des parties prenantes. C'est une dimension que nous appellerons *horizontale* des multi-exigences.

Dans [1], Bertrand Meyer propose d'entremêler dans un langage de programmation la spécification et l'implémentation, afin de réduire l'écart entre les exigences et la réalisation concrète du système, dans une optique sans rupture. L'application de son concept de *multirequirements* à l'IS devrait ainsi permettre de faire le lien entre les différents niveaux d'abstraction de représentation du système et d'en faciliter la mesure d'impact du changement. C'est une dimension que nous appellerons *verticale* des multi-exigences. De plus, l'utilisation d'un langage commun pour la spécification et la conception devrait permettre la réintroduction dans le système des exigences déduites de l'implémentation.

L'objectif du travail à réaliser durant cette thèse est de définir des méthodes et outils pour permettre cette intégration sans rupture des exigences dans les deux dimensions vues précédemment, et ainsi contribuer à un problème important en ingénierie des exigences (IE) : le problème de la traçabilité entre les exigences et les artefacts développés pour y répondre, qui rend si difficile la mesure de l'impact des changements.

2 Expression des exigences dans différentes vues

L'INCOSE (*International Council on Systems Engineering*) a mis en avant le besoin de concilier les points de vues des différentes parties prenantes [2]. Notre démarche s'inscrit dans cet objectif. En effet, plutôt que d'imposer l'utilisation d'un unique langage pour exprimer les exigences, l'utilisation d'une interface commune permettrait de lier différents formalismes tout en permettant aux ingénieurs de continuer à utiliser leurs outils.

Outils en langue naturelle Il existe aujourd'hui de nombreux outils de gestion des exigences [3]. Parmi les plus connus, *IBM Rational DOORS* ou encore *Reqtify* de Dassault Systems fournissent des outils pour gérer les exigences. Ces solutions laissent les utilisateurs représenter de diverses façons les exigences et permettent de faire le lien entre elles. Ces outils permettent d'établir une traçabilité aussi bien entre les exigences et leurs réalisations, qu'entre les exigences elles-mêmes mais ils ne proposent pas de réelle sémantique pour ces liens.

Les approches GORE (*Goal-Oriented Requirements Engineering*) telles que KAOS [4] permettent également d'exprimer les exigences en langue naturelle ainsi que les relations existantes entre ces exigences. La sémantique sur les liens existants (raffinement d'une exigence, exigence composée d'autres exigences, etc.) est ici définie. Par contre, les exigences sont exprimées en langage naturel, sans syntaxe imposée, ce qui rend la déduction de liens difficile, ces derniers devant être indiqués par l'utilisateur.

Approche dirigée par les modèles L'initiative GEMOC [5] a pour objectif de définir une interface commune pour différents DSML (*Domain Specific Modeling Language*) utilisés pour exprimer les besoins spécifiques des différents acteurs impliqués dans un projet. L'utilisation des modèles comme artefacts de base est ainsi proposée, afin de combler l'écart entre différents DSML, de façon similaire à l'utilisation des artefacts comme passerelle entre spécification et implémentation proposée en ingénierie des modèles. De plus, l'acceptation de cette approche pourrait être facilitée par la croissance de l'intérêt pour l'approche MBSE (*Model-Based System Engineering*) par les industriels en IS, qui peut être utilisée afin d'exprimer les exigences comme des éléments de modèles, de la même manière que les autres artefacts de modélisation. Cette approche devrait ainsi permettre de créer des liens entre les exigences et les autres artefacts, que ce soit d'autres exigences ou des éléments de spécification fournis par les différentes parties prenantes. Ainsi, il serait possible de combiner des éléments de différents domaines dans une vue holistique, prenant en compte les liens entre les artefacts spécifiques à un domaine mais également entre ces artefacts et les exigences. Dans [6] par exemple, les auteurs proposent d'appliquer cette fédération de modèles aux exigences, considérant que chaque espace technologique est une technique d'expression des exigences. Cela permet par exemple de lier des exigences exprimées en langue

naturelle dans un document type Word avec des exigences et leurs liens exprimés grâce à l'approche KAOS.

3 Formalisation des exigences

Le standard ISO/IEC/IEEE 29148:2011 [7] définit un certain nombre de qualités nécessaires pour l'expression des exigences (la traçabilité, la vérifiabilité, la consistance et l'absence d'ambiguïtés, ...). L'utilisation d'un langage dédié aux exigences, plus formel que la langue naturelle habituellement utilisée, est un moyen possible d'assurer ces différentes qualités.

Expression des exigences L'utilisation d'un langage dédié a été étudiée à différentes occasions. Le profil SysML [8] propose ainsi un diagramme spécifique à l'expression des exigences (à la fois fonctionnelles et non-fonctionnelles), ainsi que les liens existant entre exigences ou entre les exigences et les autres éléments de modèles (blocs, cas d'utilisation). Les exigences en elles-mêmes y sont cependant représentées sous une forme textuelle (voir Fig. 1).

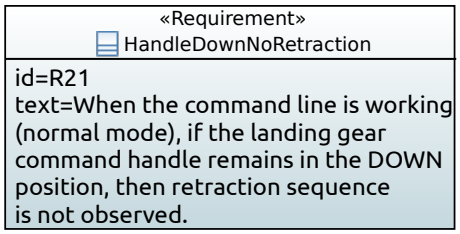


Figure 1: Représentation de l'exigence R21 de [9] en SysML

Cependant dans les approches proposées jusqu'à présent, les exigences sont toujours exprimées en langage naturel et sont, par conséquent ambiguës. Des travaux ont été proposés pour surmonter ce problème (e.g., [10], [11]). Si ces approches permettent de faire le pont entre une formalisation des exigences et une expression des exigences compréhensible par des non-informaticiens, l'utilisation de deux langages rend nécessaire de maintenir la cohérence entre les exigences ainsi exprimées, et la répercussion des changements n'est pas aussi immédiate que dans le cas de l'utilisation d'un formalisme unique.

Langages spécifiques d'expression des exigences Afin d'éviter cet écart entre langue naturelle et approches plus formelles, certains travaux mettent en avant des langages dédiés à l'expression des exigences (e.g., [12], [13]).

Cependant, à chaque fois, la syntaxe proposée n'est pas suffisamment simple ou proche des utilisateurs pour être adoptée facilement. De plus, ils s'adressent à des domaines particuliers et sont par conséquents très spécifiques.

S1: The synchronization process SHALL be initiated when Alice enters the room and at 30 minute intervals thereafter.

S2: The synchronization process SHALL distribute data to all connected devices in such a way that all devices are using the same data at all times.

Figure 2: Exemple d'exigences en RELAX extrait de [12].

Expression formelle des exigences De nombreux travaux proposent d'exprimer les exigences avec des méthodes formelles (e.g., [14], [15], [16]). Il est à noter que ces approches ne sont pas du tout destinées à des non-spécialistes et ne peuvent ainsi pas être utilisées comme interface de discussion entre acteurs de différents domaines.

Dans [1], Bertrand Meyer propose une approche pour exprimer les exigences directement dans le code source Eiffel. Il applique ainsi le *Single Model Principle* proposé dans [17]. L'exigence proposée en Fig. 1 peut ainsi être exprimée sous forme de préconditions et postconditions d'une opération exécutant la boucle d'exécution du système (donnée List. 1).

```
r21
    -- (R21) When the command line is working (normal
    -- mode), if the landing gear command handle
    -- remains in the DOWN position, then retraction
    -- sequence is not observed.
    require
      handle_status = is_handle_down
    do
      main
    ensure
      gear_status /= is_gear_retracting
end
```

Listing 1: Exemple de représentation en Eiffel de l'exigence R21 présentée Fig. 1.

L'expression des exigences à l'aide des contrats permet ainsi de fournir une interface plus accessible que les méthodes beaucoup plus formelles, tout en bénéficiant des outils formels d'Eiffel – comme AutoProof [18] un vérificateur pour Eiffel – afin de détecter les éventuelles incohérences du système. Il propose également de lier au sein d'un même formalisme les exigences, exprimées en langue naturelle, avec leur spécification (au travers de diagramme) et leur implémentation. Cette approche permet ainsi de faciliter la traçabilité.

4 Travaux futurs

L'intégration des exigences tout au long du développement d'un système complexe est un moyen de réduire les coûts engendrés par les erreurs dues à une mauvaise analyse des exigences. Les méthodes présentées ont toutes pour objectif l'ajout d'un certain formalisme afin d'améliorer le traitement des exigences. L'utilisation de la langue naturelle comme principal outil d'expression des exigences nous amène cependant à nous poser un certain nombre de questions :

- Comment réussir à exprimer les exigences de façon à ce qu'elles restent compréhensibles par un non-spécialiste tout en étant analysables de façon automatique ?
- Comment faire le lien entre les exigences exprimées par différentes parties prenantes ?
- Quelle sémantique donner aux liens existant entre les exigences ? Entre les exigences et le système ?
- Comment utiliser la formalisation des exigences pour prouver des propriétés des exigences ?

Si les travaux cités jusqu'à présent permettent de donner des pistes de réponses, il n'en existe, à notre connaissance, aucun qui permette de répondre à toutes ces questions à la fois. Un des objectifs principaux de notre travail est ainsi de définir une formalisation des exigences, via un langage qui permettra de faire le lien entre les exigences de différents domaines. Ce formalisme devra également être utilisé afin de faire le lien entre les exigences et les autres artefacts de spécification. Nous souhaitons également fournir des outils afin d'assister les ingénieurs des exigences dans le contrôle de la qualité et de la validité du système (à l'aide de traces, de couverture, ...).

Un autre objectif important de notre projet est la simplicité d'utilisation et d'appropriation de cette interface. En effet, afin de permettre à des ingénieurs non-informaticiens d'utiliser ces outils, nous souhaitons qu'ils soient aussi proches que possible de leurs outils habituels.

Afin d'expérimenter notre approche, nous utilisons l'étude de cas du système de train d'atterrissage proposée dans [9], qui définit un système et ses exigences de façon détaillée. Cet exemple, proposé durant la conférence ABZ2014 [19], est accompagné d'un ensemble d'articles proposant des formalisations des exigences identifiées, avec lesquelles nous pourrons comparer notre approche. Par la suite, nous prévoyons de valider notre approche sur un exemple industriel réel.

References

[1] B. Meyer. Multirequirements. *Modelling and Quality in Requirements Engineering (Martin Glinz Festscrhift)*, 2013.

[2] INCOSE. *SE Vision 2025*. 2014. http://www.incose.org/docs/default-source/aboutse/se-vision-2025.pdf.

[3] J. M. Carrillo de Gea, J. Nicolás, J. L. F. Alemán, A. Toval, C. Ebert, and A. Vizcaíno. Requirements Engineering Tools. *IEEE Software*, 28(4):86–91, July 2011.

[4] A. van Lamsweerde. Goal-oriented requirements engineering: a guided tour. In *Proceedings Fifth IEEE International Symposium on Requirements Engineering*, pages 249–262, 2001.

[5] B. Combemale, J. Deantoni, B. Baudry, R. B. France, J.-M. Jézéquel, and J. Gray. Globalizing Modeling Languages. *Computer*, pages 10–13, June 2014.

[6] Fahad R. Golra, Antoine Beugnard, Fabien Dagnat, Sylvain Guerin, and Christophe Guychard. Continuous Requirements Engineering using Model Federation. *RE:Next! Track at 24th IEEE International Requirements Engineering Conference 2016*, 2016.

[7] ISO/IEC/IEEE International Standard 29148:2011. *ISO/IEC/IEEE 29148:2011(E)*, pages 1–94, December 2011.

[8] Object Management Group (OMG). OMG Systems Modeling Language (OMG SysML^{TM}), V1.0, 2007. OMG Document Number: formal/2007-09-01 Standard document URL: http://www.omg.org/spec/SysML/1.0/PDF.

[9] F. Boniol and V. Wiels. The Landing Gear System Case Study. In Frédéric Boniol, Virginie Wiels, Yamine Ait Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study*, number 433 in Communications in Computer and Information Science, pages 1–18. Springer International Publishing, June 2014.

[10] W. Scott and S. C. Cook. *A Context-free Requirements Grammar to Facilitate Automatic Assessment*. PhD thesis, UniSA, 2004.

[11] R. Hähnle, K. Johannisson, and A. Ranta. An Authoring Tool for Informal and Formal Requirements Specifications. In Ralf-Detlef Kutsche and Herbert Weber, editors, *Fundamental Approaches to Software Engineering*, number 2306 in Lecture Notes in Computer Science, pages 233–248. Springer Berlin Heidelberg, April 2002.

[12] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *2009 17th IEEE International Requirements Engineering Conference*, pages 79–88, August 2009.

[13] T. Nguyen. Verification of Behavioural Requirements for Complex Systems with FORM-L, a MODELICA Extension. In *26th International Conference on Software & Systems Engineering and their Applications*, EDF R&D, 6 quai Watier, 78110 Chatou, FRANCE, 2015.

[14] F.-L. Li, J. Horkoff, A. Borgida, G. Guizzardi, L. Liu, and J. Mylopoulos. From Stakeholder Requirements to Formal Specifications Through Refinement. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, Lecture Notes in Computer Science, pages 164–180. Springer International Publishing, March 2015.

[15] A. Mammar and R. Laleau. On the Use of Domain and System Knowledge Modeling in Goal-Based Event-B Specifications. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, number 9952 in Lecture Notes in Computer Science, pages 325–339. Springer International Publishing, October 2016.

[16] A. Matoussi, F. Gervais, and R. Laleau. An Event-B formalization of KAOS goal refinement patterns. Technical Report Tech. Rep. TRLACL-2010-1, LACL, University of Paris-Est, 2010.

[17] R. Paige and J. Ostroff. The Single Model Principle. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering*, RE '01, pages 292–, Washington, DC, USA, 2001. IEEE Computer Society.

[18] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, number 9035 in Lecture Notes in Computer Science, pages 566–580. Springer Berlin Heidelberg, April 2015.

[19] F. Boniol, V. Wiels, Y. Ait Ameur, K.-D. Schewe, S. D. Junqueira Barbosa, P. Chen, A. Cuzzocrea, X. Du, J. Filipe, O. Kara, I. Kotenko, K. M. Sivalingam, D. Slezak, T. Washio, and X. Yang, editors. *ABZ 2014: The Landing Gear Case Study*, volume 433 of *Communications in Computer and Information Science*. Springer International Publishing, Cham, 2014.

MODELISATION D'EXIGENCES POUR LA SYNTHESE D'ARCHITECTURE AVIONIQUE : APPLICATION A LA SURETE DE FONCTIONNEMENT

L. ZIMMER, M. LAFAYE

Dassault Aviation – Direction Générale Technique
78 quai Marcel Dassault - 92552 Saint-Cloud - France
laurent.zimmer@dassault-aviation.com
michael.lafaye@dassault-aviation.com

P.A. YVARS

Institut Supérieur de Mécanique de Paris (SupMéca)
QUARTZ
3 rue Fernand Hainaut – 93407 Saint-Ouen - France
pierre-alain.yvars@supmeca.fr

RESUME : Les plateformes informatiques embarquées doivent supporter l'exécution d'un nombre croissant de fonctions systèmes et respecter scrupuleusement de multiples contraintes fonctionnelles et non fonctionnelles. Leur architecture est donc de plus en plus complexe et par conséquent la définition préliminaire d'architectures admissibles basées sur la compétence et l'expérience d'un petit nombre de concepteurs experts est une activité de plus en plus difficile. A l'avenir, ils devront s'aider d'outils d'assistance à la génération d'architectures correctes par construction. Dans ce contexte, nous nous sommes intéressés à la modélisation formelle des exigences et à la résolution informatique de ces modèles formels pour trouver des architectures admissibles. Nous avons étudié une problématique de déploiement assisté de fonctions systèmes sur une plateforme d'avionique modulaire embarquée avec des exigences de sûreté de fonctionnement portant sur les fonctions. Nous avons développé une approche à base de modèle qui utilise le langage DEPS (DEsign Problem Specification), un langage dédié à la modélisation et à la résolution des problèmes de conception. Les résultats obtenus montrent qu'il est possible de modéliser des exigences complexes de sûreté de fonctionnement au niveau requis par l'architecte système et que leur prise en compte pendant la résolution génère des solutions correctes vis-à-vis de celles-ci.

MOTS-CLES : *modélisation, synthèse d'architecture, spécification formelle, déploiement, exigences, sûreté de fonctionnement.*

1 INTRODUCTION

Les systèmes avioniques sont de plus en en plus complexes. D'après [1], dans le domaine militaire nous sommes passés de 15 sous-systèmes et moins de 40% des fonctions systèmes à dominante logicielle pour le F16 à 135 sous-systèmes dont 90% des fonctions à dominante logicielle pour le F35. L'évolution dans le domaine civil est moins extrême mais elle suit la même tendance. En parallèle le nombre et la complexité des spécifications techniques ou réglementaires ont augmenté tout comme la complexité de l'organisation industrielle. Cet accroissement global de la complexité engendre une explosion des coûts et des délais de développement qui amène les industriels à revoir leurs méthodes et leurs outils de conception. Selon des études financées par la DARPA [1, 2, 3], le nouveau processus de conception à imaginer doit reposer sur le développement de 4 éléments clefs :

1. des outils de conception à base d'abstraction ;
2. des métriques de complexité des systèmes ;
3. des méthodes avancées de synthèse d'architecture ;
4. une gestion robuste des incertitudes.

Le travail présenté concerne exclusivement le premier [2] et le troisième point [3]. Il s'agit pour le premier point de développer ou d'utiliser un langage de description formelle (FDL) capable de représenter à la fois un système complexe ainsi que les spécifications ou exigences qui portent sur celui-ci. Le FDL doit notamment permettre de représenter le système à des niveaux d'abstraction compatibles des différentes étapes de la conception et notamment les étapes de conception préliminaire. Les langages envisagés sont AADL, UML ou SysML.

Il s'agit pour le deuxième point de disposer très tôt dans la conception de moyens « avancés de synthèse d'architecture » qui permettraient d'explorer automatiquement l'espace de conception pour rechercher des architectures admissibles c'est-à-dire compatibles avec les différentes exigences système.
L'idée étant que la complexité des systèmes à concevoir mettra à terme la tâche d'énumération et d'évaluation des architectures candidates hors de portée des experts s'ils ne bénéficient pas d'une assistance informatique.

Pour développer un outillage de conception capable de synthétiser des architectures admissibles des chaînes de traitement logicielles ont été proposées dans des travaux précurseurs [4]. Elles comportent un générateur de contraintes dédié qui prend en entrée des données système et dont la sortie est exploitée par un outil de résolution. Dans cette approche l'essentiel du développement porte sur le générateur ; l'outil de résolution est pris sur étagère et il n'y a pas de langage de modélisation en entrée. D'autres travaux s'appuient sur l'utilisation ou l'enrichissement d'un langage de modélisation de système proposant différentes descriptions (exigences, structure, comportement etc.) et l'exploite à l'aide d'outils, d'algorithmes de vérification ou de résolution [3, 5] pour énumérer ou évaluer des architectures candidates.

Plus récemment des travaux portent sur le développement d'un langage de modélisation de problème de conception en vue de leur résolution [6]. Il s'agit donc d'enrichir les langages de résolution pour aborder les problèmes de synthèse. Dans ce cadre, les travaux que nous présentons montrent l'intérêt d'utiliser le langage DEPS pour d'une part modéliser un système et des exigences qui lui sont associées et d'autre part résoudre pour trouver une solution de synthèse.

L'étude de cas, objet de cette évaluation, est une problématique de déploiement de fonctions avion sur une architecture informatique embarquée de type avionique modulaire intégrée. Nous nous sommes limités dans ce papier à la fois en types d'éléments d'architecture (les calculateurs) et en types d'exigences (la sûreté de fonctionnement) mais nous avons veillé attentivement au caractère généralisable de l'approche.

Le papier s'organise comme suit : nous positionnons d'abord nos travaux dans le contexte de l'avionique modulaire, de la sûreté de fonctionnement et de la répartition des rôles entre acteurs puis nous présentons le langage DEPS et l'outillage associé que nous utilisons. Ensuite nous décrirons successivement l'étude de cas et le problème posé. Puis nous discuterons de la modélisation qui a été faite en DEPS et des résultats que nous avons obtenus. Enfin nous présenterons des perspectives d'évolution de ce travail tant du point de vue de la généralisation de l'étude de cas que des évolutions du formalisme DEPS.

2 PROBLEMATIQUE

2.1 Cadre général

Le concept d'avionique modulaire intégrée (IMA) [7, 8], défini dans les années 1990, s'est aujourd'hui imposé comme l'une des références pour le développement de plateformes avioniques civiles. Permis par l'émergence de technologies augmentant la puissance des calculateurs, ce concept vise à regrouper plusieurs fonctions logicielles non vitales, auparavant exécutées par des calculateurs dédiés, sur un même calculateur.

Un des apports de l'IMA est de découpler le développement logiciel du matériel sous-jacent, i.e. assurer une modularité de ces logiciels qui peuvent ainsi être déployés indifféremment sur les calculateurs en fonction de leurs besoins en ressources, pourvu que ces derniers soient « compatibles » IMA.

Cette compatibilité se traduit par un ensemble d'interfaces applicatives (API) générique à tous les calculateurs et une allocation statique des ressources temporelles et d'I/O.
En IMA le concept de partition traduit cette allocation des ressources (cf. Figure 1). Un logiciel, réalisant tout ou partie d'une fonction avion, peut ainsi être projeté sur une ou plusieurs partitions suivant le besoin en ressources et les exigences associées.

En contrepartie, cette capacité de mutualisation des logiciels associée à l'accroissement de leur nombre a entrainé une augmentation de la complexité du problème d'intégration. Il s'agit de trouver un schéma d'allocation des calculateurs aux logiciels répondant aux besoins en termes de ressources tout en respectant les contraintes de latence d'exécution des chaines fonctionnelles traversantes (i.e. traitées via l'exécution de plusieurs logiciels) ou encore de sûreté de fonctionnement. Du fait de ces contraintes multiples et du nombre important de déploiements, trouver manuellement une allocation satisfaisante devient très difficile.

L'intégrateur système va utiliser sa connaissance des logiciels et des ressources calculateurs et réseau pour proposer plusieurs allocations puis évaluer à l'aide d'outils la conformité de ces allocations aux exigences jusqu'à trouver un déploiement satisfaisant. Il s'agit d'une démarche d'analyse a posteriori.

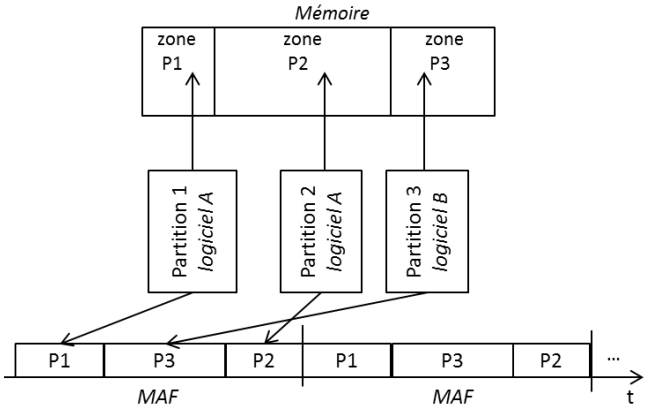


Figure 1 : principe d'allocation statique des ressources via le concept de partition

Partant de ce constat, nous avons développé une approche de déploiement des logiciels qui prend en compte a priori et non a posteriori les exigences qui portent sur les fonctions avion.

Afin d'illustrer notre démarche, nous nous placerons par la suite dans le cas d'un problème de déploiement de logiciels sur un ensemble de calculateurs IMA devant respecter uniquement des exigences de sûreté de fonctionnement. Ces exigences sont parmi les plus importantes, tout avion civil étant pensé dans une approche visant à réduire au maximum les risques d'accident et donc potentiellement de pertes humaines.

2.2 Les exigences de sûreté de fonctionnement

2.2.1 Production des exigences

La sûreté de fonctionnement vise à établir des niveaux admissibles de fiabilité, disponibilité et maintenabilité des systèmes essentiels de l'avion afin de garantir la sécurité de l'appareil en vol comme au sol. Les exigences permettant d'atteindre ces niveaux résultent de l'analyse des cas de pannes et défaillances, pouvant amener par exemple à une perte de fonctionnalité ou une corruption de données.

En terme d'architecture plateforme, ces analyses conduisent à un ensemble de « patterns ». Un « pattern » est une proposition de découpage d'une fonction avion en composants logiciels avec des exigences associées de ségrégation ou de dissimilarité. Par exemple, la robustesse à un cas de panne d'une fonction conduit à un « pattern » de type redondance du logiciel exécutant cette fonction et son déploiement sur deux calculateurs distincts. On parlera de ségrégation des ressources utilisées. Ces exigences ne sont pas uniquement applicables au niveau logiciel, mais peuvent porter par exemple sur l'ensemble de la chaine de traitement d'une fonction avion donnée, en exigeant que cette fonction soit réalisée via deux chaines ségréguées. Dans ce cas, l'ensemble des logiciels réalisant la première chaine devront impérativement être déployés sur des calculateurs distincts des logiciels réalisant la seconde chaine. Une illustration de « patterns » et d'exigences associées sera donnée dans la partie cas d'étude.

2.2.2 Utilisation à l'intégration

Afin de prendre en compte au plus tôt ces exigences et patterns pour l'identification de schémas d'intégration satisfaisants, il est nécessaire de pouvoir formaliser ces éléments de sûreté de fonctionnement à partir des sorties « manuscrites » livrées après analyse amont. Pour que cette formalisation ne soit pas seulement exploitable par des spécialistes du domaine de la sûreté de fonctionnement mais également par l'intégrateur plateforme, il est nécessaire que cette formalisation se situe au bon niveau entre pouvoir d'expression et abstraction. Notre objectif est ensuite d'utiliser ces éléments afin de déterminer au plus tôt et de manière informatisée, les possibilités de déploiements candidats (cf.
Figure 2).

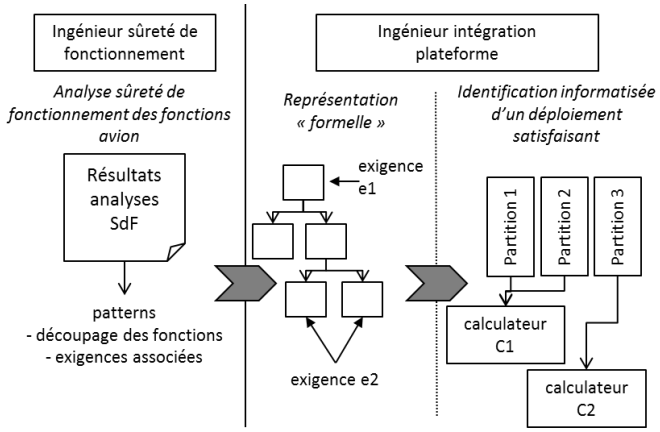


Figure 2 : principe de prise en compte au plus tôt des exigences de Sûreté de Fonctionnement

Nous sommes donc confrontés à un problème de représentation « formelle » des éléments d'entrée, ainsi qu'à un problème d'identification de solutions compatibles des exigences. Nous nous positionnons ainsi dans une démarche de modélisation et de synthèse, par opposition aux démarches actuelles centrée sur l'analyse.
Nous avons fait le choix d'utiliser un même formalisme pour traiter conjointement la question de la représentation des éléments système et celle de la recherche de solutions compatibles de l'espace des exigences. Il s'agit du langage DEPS.

3 LE LANGAGE DEPS

3.1 Paradigme

Le langage DEPS (Design Problem Specification) est ce qu'on appelle communément un langage dédié ou DSL (Domain Specific Language). Le domaine d'application visé est la spécification et la résolution de problèmes de l'ingénieur, en particulier ceux que l'on rencontre en conception de produits ou de systèmes.
DEPS est un langage dédié externe (par opposition à interne ou embarqué). Le code source est donc indépendant de tout langage généraliste hôte.

Le langage DEPS peut être vu comme une combinaison entre un langage de modélisation logiciel ou système et un langage de programmation mathématique. Aux premiers ont été empruntés les traits de structuration et d'abstraction qui permettent de représenter les éléments et le système étudié. Aux seconds ont été empruntés les concepts mathématiques nécessaires à la résolution des problèmes de l'ingénieur : inconnues, équations et inéquations.

Cette combinaison permet à la fois de représenter les problèmes de conception et poser puis résoudre ou optimiser les systèmes d'équations et d'inéquations qui les régissent [6].

3.2 Caractéristiques essentielles

3.2.1 Le Modèle

Le trait fondamental du langage est le Modèle. Tout Modèle est défini à l'aide du mot clé *Model* suivi de son nom et de sa liste d'arguments (éventuellement vide). Il comporte dans l'ordre : une zone de déclaration-définition des constantes du modèle, une zone de déclaration des variables, une zone de déclaration-création des éléments et une zone de définition des propriétés. La définition d'un Modèle DEPS se termine par le mot clé *End.*

Les propriétés d'un Modèle sont les équations et les inéquations qui portent sur les constantes et les variables de ce Modèle. Un Modèle contient donc tous les ingrédients nécessaires à la pose des contraintes qui régissent une instance de ce Modèle.

Comme dans un langage objet on dispose de l'héritage et de la composition : un Modèle peut être étendu et hériter des propriétés d'un autre Modèle et il peut être composé d'éléments, instances d'autres Modèles.

Des éléments peuvent être passés en argument d'un Modèle créant un lien d'agrégation avec celui-ci.
Des constantes peuvent aussi être passées en argument d'un Modèle ce qui permet de créer des Modèles paramétrés.

Modéliser un problème revient donc à spécifier des Modèles.

3.2.2 La Quantité

En DEPS, les constantes comme les variables sont associées à des types de grandeurs physiques ou technologiques qu'on appelle quantités (*Quantity*). Elles sont nécessaires en Ingénierie de Systèmes.

Une quantité possède :
- Un type de quantité de base (*QuantityKind*). Par exemple, réel (*Real*), entier (*Integer*), longueur (*Length*) ;
- Une borne min (resp. max) qui représente la valeur minimale (resp. maximale) pouvant être prise par toute constante ou variable ayant pour type la quantité définie ;
- Une dimension qui représente la dimension au sens de l'analyse dimensionnelle de la quantité. Par exemple [L] pour une longueur ou [U] pour une grandeur sans dimension ;
- Une unité de la quantité. Par exemple le mètre m pour une longueur.

4 L'OUTILLAGE EXISTANT

4.1 L'environnement de développement

L'environnement de modélisation et de résolution intégré associé au langage DEPS comprend :

- Des fonctions d'édition de modèle,
- Des fonctions de gestion de projet basées sur un mécanisme de packages ;
- Un compilateur,
- Un solveur.

Une approche par intégration plutôt qu'une approche par transformation de modèles a été privilégiée, En effet, dans le cas de la résolution d'un problème de synthèse de système sous-défini, il va être nécessaire en cas de résultat de calcul non satisfaisant de réaliser une mise au point des modèles dans le langage DEPS. En optant délibérément pour une approche par intégration, nous favorisons ce processus de mise au point.

4.2 La résolution

Les méthodes de calcul que nous utilisons sont tirées des travaux sur la résolution des CSP.

Un CSP (Constraint Satisfaction Problem) est défini par un triplet (*X, D, C*) tel que [9] :
- *X* est un ensemble fini de variables dites variables contraintes.
- *D* est un ensemble fini de domaines de ces variables.
- *C* est un ensemble fini de contraintes sur les variables de l'ensemble *X*.

Les domaines des variables peuvent être discrets (CSP) ou continus (NCSP pour Numerical CSP). On appelle contrainte, n'importe quel type de relation mathématique qui porte sur les valeurs d'un ensemble de variables : égalité, différence, inégalité logique et/ou algébrique (linéaire ou non linéaire).

Résoudre un CSP revient ainsi à instancier chacune des variables de X tout en satisfaisant l'ensemble C des contraintes du problème. Partant des domaines de valeurs de chacune des variables du problème, l'algorithme de résolution alterne contraction des domaines et choix d'un sous domaine pour une variable du problème jusqu'à aboutir à une solution ou bien un échec. Ce dernier est alors traité par un retour en arrière sur les points de choix précédents.

L'étape de contraction est réalisée à l'aide d'algorithmes dédiés qui mettent à profit les contraintes explicitées du problème pour réduire les domaines de chaque variable.
Dans le cas d'un problème sur-contraint, un échec peut apparaitre dès la première propagation ou bien au final après avoir exploré les parties restantes de l'arbre de recherche. Dans ce cas, la méthode garantit qu'il n'y a pas de solution au problème posé.

Le solveur implémente une méthode de propagation de type HC4 révisé [10] sur des équations et inéquations portant sur quatre types de domaines : les intervalles ouverts de réels, les intervalles d'entiers, les ensembles énumérés de valeurs flottantes et les ensembles énumérés de valeurs entières signées. Les contractions sont réalisées directement sur les domaines typés sans repasser dans les intervalles de réels. L'algorithme de recherche de solution est une méthode de branch and prune. Les stratégies round-robin et first-fail sont disponibles.

Dans le cas d'un problème sur-contraint, un échec peut apparaitre dès la première propagation ou bien au final après avoir exploré les parties restantes de l'arbre de recherche. Dans ce cas, l'échec s'interprète comme la preuve qu'il n'y a pas de solution au problème posé et non pas comme une défaillance de l'algorithme de résolution.

L'architecture orientée-objet du solveur a été pensée de manière à pouvoir être étendue à d'autres méthodes de propagation et/ou de résolution (box-consistance, méthodes locales, …).

5 APPLICATION AU CAS D'ETUDE

5.1 Description du problème

Notre cas d'étude (cf.

Figure **3**) consiste en un problème de déploiement de sept fonctions avion (notamment des applications de freinage, de remontée de pannes, de communication, etc.) sur une plateforme composée de quatre calculateurs, à déterminer en fonction des exigences de sûreté de fonctionnement associées.

Pour rappel, nous nous plaçons dans le rôle de l'intégrateur plateforme et supposons l'analyse de sûreté de fonctionnement des sept fonctions avion terminée.

Nous avons donc en entrée pour chaque fonction un document décrivant le découpage en composants logiciels puis la projection en termes de partitions, avec des exigences de ségrégation matérielle (projection des composants ségrégués sur des calculateurs distincts) ou inversement de co-localisation pour assurer la cohérence de données à traiter séquentiellement dans un temps maitrisé court. Nous allons détailler ci-après la décomposition du cas d'étude (cf.
Figure 3) sur deux fonctions avion.

La première fonction consiste en une fonction de gestion du système d'atterrissage. Cette fonction dite SAT est initialement projetée sur un seul composant logiciel.

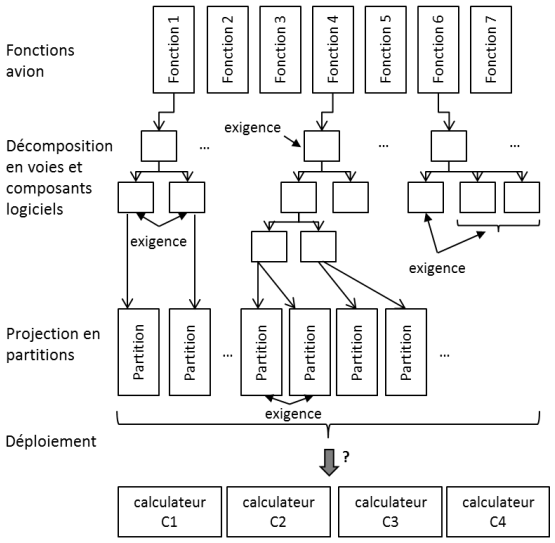


Figure 3 : description générale du cas d'étude

L'analyse de sûreté de fonctionnement nous impose deux exigences :
- Exigence 1 : un second composant logiciel dissimilaire (appelée CS pour composant « safety ») doit être prévu, afin de prendre le relais en cas de perte du composant principal (appelé CP). Ces composants doivent être ségrégués matériellement afin qu'une panne matérielle affectant CP n'affecte pas CS ;
- Exigence 2 : cette chaine de traitement (ou voie) CP + CS doit être redondée et ségréguée matériellement afin d'être robuste à la perte de la première chaine.

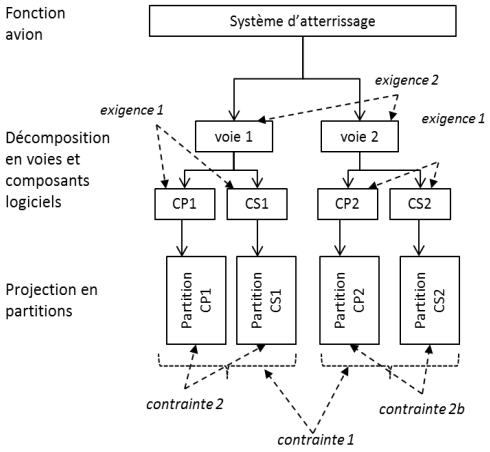


Figure 4 : décomposition de la fonction SAT

D'un point de vue traitement, le fournisseur de la fonction nous indique que chaque composant peut être projeté sur une seule partition. Nous nous retrouvons donc avec un schéma de décomposition résultant sur quatre partitions, avec les contraintes suivantes (cf. Figure 4) :
- Contrainte 1 : les partitions {CS1, CP1} doivent être matériellement ségréguées des partitions {CS2, CP2} (cf. Exigence 1);
- Contrainte 2 : les partitions CS1 et CP1 doivent être matériellement ségréguées (cf. Exigence 2) ;
- Contrainte 2bis : les partitions CS2 et CP2 doivent être matériellement ségréguées (cf. Exigence 2).

La seconde fonction que nous détaillons consiste en une fonction de gestion des remontées de pannes (SRP).
Cette fonction est initialement projetée sur un seul composant logiciel CGP (composant de gestion pannes).

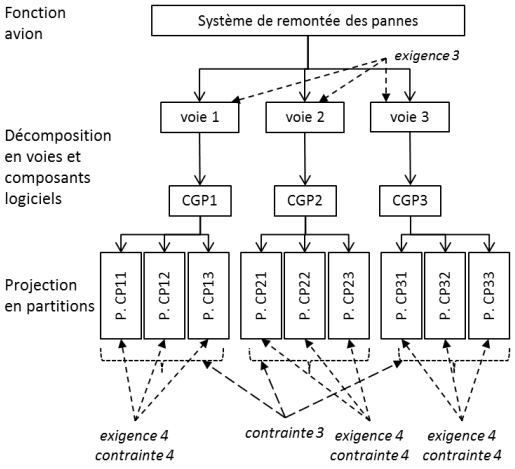


Figure 5 : décomposition de la fonction SRP

L'analyse de sûreté de fonctionnement nous impose deux exigences :
- Exigence 3 : afin d'offrir une disponibilité suffisante, le composant doit être tripliqué, chaque occurrence devant être matériellement ségréguée des autres ;

- Exigence 4 : chaque composant sera découpé suivant trois traitements projetés sur trois partitions différentes qui doivent, pour des raisons de temps, de séquencement et de rapidité des traitements, être co-localisées sur un même calculateur. Nous nous retrouvons avec un schéma de décomposition amenant à 9 partitions avec les contraintes suivantes (cf.

Figure 5):
- Contrainte 3 : les triplets de partitions {PCGP11, PCGP12, PCGP13}, {PCGP21, PCGP22, PCGP23} et {PCGP31, PCGP32, PCGP33} doivent être matériellement ségréguées;
- Contrainte 4 : toutes les partitions d'un même triplet doivent être projetées sur un même calculateur.

5.2 Formalisation en DEPS

5.2.1 Les éléments du système

Nous avons modélisé les éléments constitutifs d'une fonction avion aux différents niveaux d'abstraction nécessaires à l'expression des exigences de sûreté de fonctionnement.

```
Package Partition ;
Uses Universal;
Quantity CpuIndex
 Kind : Integer ;
 Min  : 1 ;
 Max : 4 ;
 Unit : u ;
End

Model Partition()
Constants
Variables
icpu : CpuIndex;
Elements
Properties
End
```

Figure 6 : Model DEPS d'une partition

Ce sont les modèles de : fonction avion, chaîne de traitement, composant logiciel, et partition (cf. Figure 3).
- Une fonction avion est composée d'une ou plusieurs voies. Les voies sont des éléments du modèle de la fonction avion et sont donc des instances de chaîne de traitement.
- Une chaîne de traitement est composée d'un ou plusieurs composants logiciels ainsi que d'autres éléments (capteurs, actionneurs, réseau ...) hors du champ d'étude de cet article.
- Un composant logiciel est découpé en une ou plusieurs partitions.
- Une partition doit être projetée sur une ressource de calcul pour pouvoir s'exécuter (cf. Figure 6).

5.2.2 Les exigences

Le modèle qui suit montre comment on modélise au niveau des composants logiciels et des partitions l'exigence de redondance et de ségrégation de la chaine de traitement du système d'atterrissage (cf. Figure 7).
Les exigences sont propres à chaque fonction avion et elles portent à différents niveaux de décomposition de la fonction considérée.

```
Model NonColoc (P1, P2)
Constants
Variables
Elements
 P1 : Partition ;
 P2 : Partition ;
Properties
 P1.icpu - P2.icpu <> 1;
End

Model Segregation(C1, C2)
Constants
Variables
Elements
 C1 : Compl ; (* composant
                logiciel 1 partition*)
 C2 : Comp1 ;
 NCL : NonColoc(C1.P, C2.P) ;
Properties
End

Model ChaineTraitementSA ()
Constants
Variables
Elements
 CP : Comp1() ;  (* composant principal *)
 CS : Comp1() ;  (* composant safety    *)
 SEG : Segregation (CP, CS) ;
Properties
End
```

Figure 7 : Modèle d'une chaîne de traitement du système d'atterrissage

En procédant ainsi on a pu modéliser l'exigence d'indépendance matérielle entre les deux voies dupliquées de la chaine de traitement du système d'atterrissage tout comme l'exigence de co-localisation sur un même calculateur des partitions des composant logiciels du système de remontée de pannes.

5.3 Résultats

Après modélisation des sept fonctions avions et de leurs exigences respectives on a obtenu les solutions de déploiement des partitions de ces fonctions sur les calculateurs. Les méthodes de résolution de DEPS retrouvent bien dans toutes les solutions que, conformément aux exigences, la fonction SAT a besoin de quatre calculateurs pour son déploiement tandis que la fonction SRP en a besoin de trois (cf. Figure 8).

.

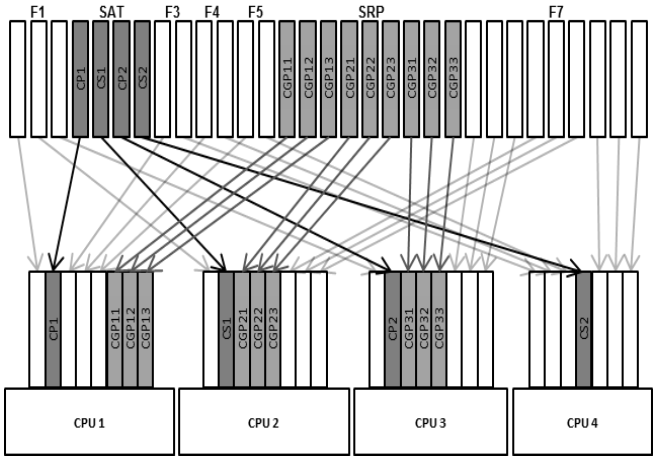


Figure 8 : Un déploiement des sept fonctions avion

6 CONCLUSIONS ET PERSPECTIVES

Dans cet article nous avons montré qu'il est possible de capturer des exigences de sûreté de fonctionnement difficiles à formaliser directement dans le formalisme (V, D, C) des CSP.

Nous avons utilisé le langage DEPS qui nous a permis de modéliser les bonnes abstractions pour d'une part décrire un modèle sous-défini d'architecture IMA et d'autre part des modèles d'exigences qui portent sur les bonnes abstractions.

Le problème de déploiement posé a été résolu en appliquant des méthodes de satisfaction de contraintes aux propriétés du problème.

Les traits de structuration offerts par le langage facilitent la réutilisation des modèles « métier » développés.

Les travaux en cours portent sur l'expression d'autres exigences d'architecture et sur leur prise en compte conjointe pour trouver des solutions de déploiement admissibles sur des problèmes à l'échelle.

REFERENCES

[1] Becz, S., Pinto, A., Zeidner, L. E., Banaszuk, A., Khire, R., and Reeve, H. M.,Design System for Managing Complexity in Aerospace Systems," 13th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference, Texas, 2010

[2] Pinto A., Becz S., Reeve H.M., Correct-by-construction design of aircraft electric power systems, in AIAA Aviation Technology, Integration, and Operations Conf., 2010.

[3] Zeidner L, Reeve H, Khire R, Becz S., Architectural Enumeration and Evaluation for Identification of Low-Complexity Systems, MATIO10, Texas, 2010.

[4] Bieber P., J.P Bodeveix J.P., Castel C., Doose D., Filali M., Minot F., Pralet C., Constraint-based Design of Avionics Platform - Preliminary Design Exploration ERTS2008 Toulouse January 2008.

[5] Albarello, N., Welcomme J.B. and Reyterou C., A formal design synthesis and optimization for systems architectures, MOSIM'12, Bordeaux, France, 2012.

[6] Yvars P.A., Zimmer L., DEPS un langage pour la spécification de problèmes de conception de systèmes, MOSIM'14, Nancy, 2014.

[7] ARINC Specifications 653-1 Avionics Application Software Standard Interface, SAE ITC, 2017

[8] DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations. RTCA, 2017.

[9] E. Tsang, Foundations of Constraint Satisfaction. London and San Diego: Academic Press, 1993.

[10] Benhamou F., Goualard F., Granvilliers L., Puget J.F., Revising Hull and Box consistency,16th International Conference on Logic Programming, 1993.

Prix de thèse et Accessits du GDR Génie de la Programmation et du Logiciel

Prix de thèse du GDR GPL
Verasco : a Formally Verified C Static Analyzer

Auteur : Jacques-Henri Jourdan (Inria Paris-Rocquencourt, GALLIUM)

Résumé :
Afin de développer des logiciels plus sûrs pour des applications critiques, certains analyseurs statiques tentent d'établir, avec une certitude mathématique, l'absence de certains types de bugs dans un programme donné. Une limite possible à cette approche est l'éventualité d'un bug affectant la correction de l'analyseur lui-même, éliminant ainsi les garanties qu'il est censé apporter.

Dans cette thèse, nous proposons d'établir des garanties formelles sur l'analyseur lui-même : nous présentons la conception, l'implantation et la preuve de sûreté en Coq de Verasco, un analyseur statique formellement vérifié utilisant l'interprétation abstraite pour le langage ISO C99 avec l'arithmétique flottante IEEE754 (à l'exception de la récursion et de l'allocation dynamique de mémoire). Verasco a pour but d'établir l'absence d'erreur à l'exécution des programmes donnés. Il est conçu selon une architecture modulaire et extensible contenant plusieurs domaines abstraits et des interfaces bien spécifiées. Nous détaillons le fonctionnement de l'itérateur abstrait de Verasco, son traitement des entiers bornés de la machine, son domaine abstrait d'intervalles, son domaine abstrait symbolique et son domaine abstrait d'octogones. Verasco a donné lieu au développement de nouvelles techniques pour implémenter des structures de données avec partage dans Coq.

Biographie :
Jacques-Henri Jourdan est chercheur postdoctoral à l'institut Max Planck pour les systèmes logiciels à Sarrebruck (Allemagne). Il travaille sur la formalisation du système de types de Rust, à l'aide de la logique de séparation concurrent Iris en Coq. Il a effectué sa thèse avec Xavier Leroy au sein de l'équipe Gallium d'Inria Paris-Rocquencourt. Il a aussi travaillé au CEA à Saclay avec Éric Goubault et à Microsoft Research à Redmond avec Francesco Logozzo.

Accessit
La sécurité des protocoles d'authentification sur le Web

Auteur : Antoine Delignat-Lavaud (Microsoft Research Cambridge)

Résumé :
Est-il possible de démontrer un théorème prouvant que l'accès aux données confidentielles d'un utilisateur d'un service Web (tel que GMail) est impossible sans connaître son mot de passe, en supposant certaines hypothèses sur ce qu'un attaquant est incapable de faire (par exemple, casser des primitives cryptographiques ou accéder directement aux bases de données de Google), sans toutefois le restreindre au point d'exclure des attaques possibles en pratique ?

Il existe plusieurs facteurs spécifiques aux protocoles du Web qui rendent impossible une application directe des méthodes et outils existants issus du domaine de l'analyse des protocoles cryptographiques. Tout d'abord, les capacités d'un attaquant sur le Web vont largement au-delà de la simple manipulation des messages échangés entre le client et le serveur sur le réseau. Par exemple, il est possible (et même fréquent en pratique) que l'utilisateur ait dans son navigateur un onglet contenant un site malhonnête en même temps qu'il consulte sa messagerie ou sa banque (par exemple, via une bannière publicitaire) ; ce qui permet à un attaquant d'exécuter du code malicieux pouvant provoquer l'envoi de requêtes arbitraires vers les sites honnêtes. Les navigateurs incluent des politiques d'isolation complexes entre les pages visant à limiter les interactions entre sites - mais comment être sûr que ces mécanismes sont bien utilisés et garantissent la protection des données alors même qu'un nombre croissants de modules tiers (Google, Facebook, Twitter...) est maintenant intégré à la plupart des services ?

La procédure pour se connecter à GMail implique un empilement complexe de protocoles : tout d'abord, un canal chiffré, et dont le serveur est authentifié, est établi avec le protocole TLS ; puis, une session HTTP est créée en utilisant un cookie ; enfin, le navigateur exécute le code JavaScript retourné par le client, qui se charge de demander son mot de passe à l'utilisateur. Même en supposant que la conception de ce système soit sûre, il suffit d'une erreur minime de programmation (par exemple, une simple instruction `goto` mal placée) pour que la sécurité de l'ensemble de l'édifice s'effondre.

L'objet de cette thèse est de bâtir un ensemble d'outils et de librairies permettant de programmer et d'analyser formellement de manière compositionnelle la sécurité d'applications Web confrontées à un modèle plausible des capacités actuelles d'un attaquant sur le Web. Dans cette optique, nous étudions la conception des divers protocoles utilisés à chaque niveau de l'infrastructure du Web (TLS, X.509, HTTP, HTML, JavaScript) et évaluons leurs compositions respectives. Nous nous intéressons aussi aux implémentations existantes et en créons de nouvelles que nous prouvons correctes afin de servir de référence lors de comparaisons.

Nos travaux mettent au jour un grand nombre de vulnérabilités aussi bien dans les protocoles que dans leurs implémentations, ainsi que dans les navigateurs, serveurs, et sites internet ; plusieurs de ces failles ont été reconnues d'importance critiques. Enfin, ces découvertes ont eu une influence sur les versions actuelles et futures du protocole TLS.

Biographie :
Antoine Delignat-Lavaud est chercheur au laboratoire Microsoft Research de Cambridge au Royaume-Uni depuis octobre 2016. Il travaille sur le projet Everest visant à vérifier formellement les librairies utilisées par les navigateurs, applications et serveurs pour les communications sur le Web, en utilisant le système de type dépendant $F^\star$ développé par Microsoft Research en partenariat avec l'Inria. Il est assisté dans cette tâche par plusieurs de ses anciens camarades de l'équipe Gallium de l'Inria spécialisés dans la programmation fonctionnelle et les preuves de programmes (Jonathan Protzenko, Tahina

Ramananandro). Il a préparé sa thèse sous la direction de Karthikeyan Bhargavan dans l'équipe Prosecco de l'Inria Paris après avoir été élève à l'ENS de Cachan.

Accessit
Towards Improving the Quality of Mobile Apps by Leveraging Crowd-sourced Feedback

Auteur : María Gómez Lacruz (Inria Lille - Université de Lille)

Résumé :
Le développement des applications mobiles (apps) est en pleine explosion, en grande partie en raison de l'utilisation généralisée des appareils mobiles. Les magasins d'applications (e.g., Google Play) sont les canaux de distribution courants pour les apps. La concurrence excessive sur les marchés d'applications actuels oblige les fournisseurs d'applications à diffuser des applications de haute qualité. En fait, des études antérieures ont démontré que les utilisateurs d'applications sont intolérants à des problèmes de qualité (e.g., des arrêts inopinés, des applications qui ne répondent pas). Les utilisateurs qui rencontrent des problèmes désinstallent fréquemment les applications et se dirigent vers des applications concurrentes. Par conséquent, détecter et prévenir rapidement des problèmes dans les applications est crucial pour rester compétitif sur le marché.

Même si les développeurs utilisent des émulateurs et testent les applications avant le déploiement, de nombreux bugs peuvent encore apparaître dans la nature. Développer des applications qui fonctionnent sans erreur à travers le temps reste donc une préoccupation majeure pour les développeurs. Le grand défi qui demeure est que l'environnement reste hors du contrôle des développeurs d'applications. Plus précisément, l'écosystème mobile est confronté à une rapide évolution des plateformes mobiles, une forte fragmentation des équipements, et une grande diversité des contextes d'exécution (les réseaux, les localisations, les capteurs).

Cette thèse présente donc une nouvelle génération de magasins d'applications mobiles —App Store 2.0— qui exploite des données collectées sur les applications, les appareils et les utilisateurs a fin d'augmenter la qualité globale des applications mobiles publiées en ligne. Les magasins d'applications ont déjà accès à différents types de communautés (de smartphones, d'utilisateurs, et d'applications). Nous affirrmons que cette nouvelle génération de magasins d'applications peut exploiter l'intelligence collective pour obtenir des indications pratiques à partir des données retournées par les utilisateurs. Ces indications concrètes aident les développeurs d'applications à traiter les erreurs et les menaces potentielles qui a ectent leurs applications avant la publication ou même lorsque les applications sont dans les mains des utilisateurs finaux.

Nous avons conçu un prototype du magasin d'applications envisagé pour les applications Android. Nous validons la solution proposée avec des bugs et des applications réels. Nos résultats ont prouvé l'applicabilité et la faisabilité de notre approche. Ces nouveaux magasins d'applications ont le potentiel de réduire l'effort humain et de gagner du temps précieux pour les développeurs, ce qui est un facteur déterminant pour le succès des applications mobiles sur les marchés d'applications actuels.

Biographie :
María Gómez Lacruz est actuellement chercheuse post-doctorante à l'Université de Saarland (Allemagne), dans le centre CISPA. Elle travaille avec Prof. Andreas Zeller, sur l'amméllioration de la securité des utilisateurs d'applications mobiles. María a obtenu le Doctorat en Informatique (avec les labels Européen et International) par l'Université de Lille, sous la direction de Prof. Romain Rouvoy et Prof. Lionel Seinturier au sein de l'équipe Spirals du centre Inria Lille.

Accessit
Exploration des variantes d'artefacts logiciels pour une analyse et une migration vers des lignes de produits

Auteur : Jabier Martinez (Université du Luxembourg, Luxembourg – Sorbonne Université UPMC Paris 6)

Résumé :
Les lignes de produits logiciels (LdPs) permettent la dérivation d'une famille de produits basés sur une gestion de la variabilité. Les LdPs utilisent des configurations de caractéristiques afin de satisfaire les besoins de chaque client et, de même, permettre une réutilisation systématique en utilisant des assets réutilisables. L'approche capitalisant sur des variantes des produits existants est appelé une approche extractive pour l'adoption de LdPs. L'identification des caractéristiques est nécessaire pour analyser la variabilité d'un ensemble de variantes. Il est également nécessaire de localiser les éléments associés à ces caractéristiques. Les contraintes entre ces caractéristiques doivent être identifiées afin de garantir la sélection de configurations valides. Par ailleurs, il faut construire les assets réutilisables et synthétiser un modèle de caractéristiques.

Cette thèse présente BUT4Reuse (Bottom-Up Technologies for Reuse), un framework unifié, générique et extensible pour l'adoption extractive de LdPs. Une attention particulière est accordée à des scénarios de développement dirigée par les modèles. Nous nous concentrons aussi sur l'analyse des techniques en proposant un benchmark pour la localisation de caractéristiques et une technique d'identification de familles de variantes. Nous présentons des paradigmes de visualisation pour accompagner les experts du domaine dans le nommage de caractéristiques et aider à la découverte de contraintes. Finalement, nous étudions l'exploitation des variantes pour l'analyse de la LdP après sa création. Nous présentons une approche pour trouver des variantes pertinentes guidée par des évaluations des utilisateurs finaux.

Biographie :
Jabier Martinez a participé à plusieurs implémentations de lignes de produits logiciels et services de conseil dans l'industrie. Il a obtenu un doctorat de l'Université du Luxembourg en cotutelle avec l'Université Pierre et Marie Curie et a mené ses recherches au sein de l'équipe Serval SnT et Lip6 (Yves le Traon, Mikal Ziane, Tewfik Ziadi, Jacques Klein). Sa thèse porte sur l'analyse de configurations et variantes d'artefacts logiciels pour obtenir des niveaux de réutilisation plus élevés. Il est post-doctorant à la Sorbonne Université depuis décembre 2016.

Démonstrations et Posters

An Expressive DSL for Parametric Monitoring

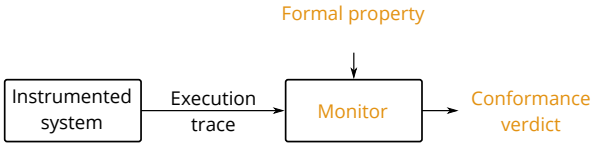
Yoann Blein, Lydie du Bousquet, Roland Groz and Yves Ledru
Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, F-38000 Grenoble France

The MODMED Project

- Industry of Medical Systems
- Bring formal monitoring
- Partnership with local companies
- Case study on a computer assisted surgery system

System monitoring to

- Check the correctness and the robustness of the system
- Validate the assumptions made on the environment
- Better understand the usage of the system
- Produce evidences for certification



A DSL to facilitate property formalization

Language objectives

- **Intuitive** and yet formal
- Expressive enough to fit **practical needs**
- Allow easy exploitation of **data** carried by events

Methodology to design the DSL

1. Gather properties that the system should verify
2. Extract a representative subset of those properties
3. Design a language allowing to express this subset concisely
4. Gather feedbacks from partners
5. Validate the language with field practitioners

Language features

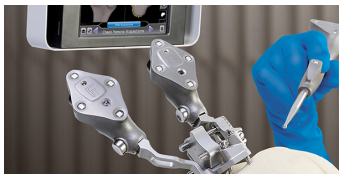
- Strong focus on **temporal propreties**
- Based on known **specification patterns**
- Most of the constructions can be **composed**
- **Declarative** style
- **Constraints on parameters** of events

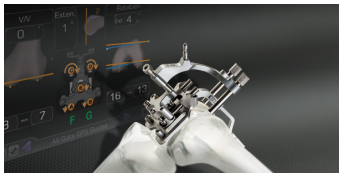
Case study: ExactechGPS Total Knee Arthroplasty

Surgery workflow

System initialization


Sensors calibration


Anatomic points acquisition


Cut planning

Abstract view of an execution trace

```
EnterState {
  state: "LocalizerConnection" }
CameraConnected {}
ExitState {
  state: "LocalizerConnection" }
EnterState {
  state: "TrackersConnection" }
SearchTrackers {
  types: ["P", "F", "T", "G"] }
TrackerDetected { ty: "P" }
TrackerDetected { ty: "F" }
TrackerDetected { ty: "G" }
TrackerDetected { ty: "T" }
ExitState {
  state: "TrackersConnection" }
...
Temperature { value: 49.75 }
ActionNext {}
EnterState {
  state: "AcquiAnkleMalleolus" }
AcquisitionBegin {}
AcquisitionCancel {}
ActionNext {}
AcquisitionBegin {}
AcquisitionSuccess {}
...
EnterState {
  state: "AcquiTibiaCenter" }
...
Temperature { value: 54.50 }
ExitState {
  state: "CutPlanning" }
EnterState {
  state: "CutNavigation" }
...
EnterState {
  state: "KneeControl" }
```

Natural and formalized properties

"The temperature is never lower than 45 degrees after the camera is connected."

```
after first CameraConnected,
absence_of Temperature t
  where t.value < 45.0
```

"All the searched trackers are detected before leaving the state `TrackersConnection`*"*

```
between SearchTrackers st
and ExitState e
  where e.state=="TrackersConnection",
forall ty in st.types,
occurrence_of TrackerDetected td
  where td.ty == ty
```

"Cancelling an acquisition prevents it from succeeding"

```
since AcquisitionCancel
until AcquisitionBegin,
absence_of AcquisitionSuccess
```

"The system does not allow navigating a cut before terminating its planification"

```
ExitState exit
  where exit.state == "CutPlanning"
precedes EnterState enter
  where enter.state == "CutNavigation"
```

VASCO - Génie des Logiciels et des Systèmes d'Information

This work is funded by the ANR project MODMED (ANR-15-CE25-0010).

Version Propagation in Three-Level Component-Based Architectures

1. Context and objectives

- Versioning models / architectures
  - Representing the whole life-cycle of an application and version its representations → Co-evolution
  - Versioning models / architectures
- Dedal
  - 3 abstraction levels (Figure 1): **Specification** (Roles)/ **Configuration** (Component classes)/ **Assembly** (Component instances) → Keeping track of the whole life-cycle
  - Changes may occur at any of the 3 architecture levels
- Problematics: Management of co-evolution and versioning of architecture models

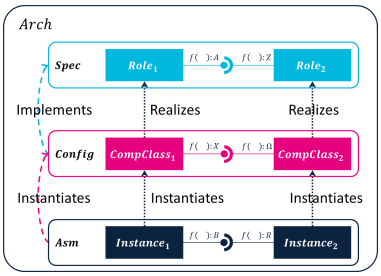


Figure 1. Base case: Dedal three-level architecture

Architectural rules

I. Intra-level consistency

1. Unique name
2. Connected interfaces are compatible
3. The architecture realizes its functional objectives and the architecture definition is composed of a connected graph

II. Inter-level coherence

1. All component roles from the specification are realized by component classes in the configuration
2. Each connected provided interface in the configuration is included in the specification.
3. Every component class from the configuration is instantiated at least once by a component instance in the assembly
4. Each connected provider in the assembly is an instance of a provided interface from the configuration.

2. Rules for predicting version propagation

Hypothesis on types (Figure 1): $B \preccurlyeq X \preccurlyeq A \preccurlyeq Z \preccurlyeq \Omega \preccurlyeq R$

Provided functionality

Specification	Configuration	Assembly
$Y \hookrightarrow A$	$Y \hookrightarrow X$	$Y \hookrightarrow B$
Non-propagation		
$X \preccurlyeq Y \preccurlyeq Z$	$B \preccurlyeq Y \preccurlyeq A$	$Y \preccurlyeq X$
Propagation		
Inter-level / **Intra-level**	**Inter-level** / **Intra-level**	**Inter-level** / **Intra-level**
$(Y \parallel X) \vee (Y \prec X)$ / $(Y \parallel Z) \vee (Y \succ Z)$	$(\neg(Y \preccurlyeq A \Rightarrow \uparrow)) \vee (\neg(Y \succcurlyeq B \Rightarrow \downarrow))$ / $\neg(Y \preccurlyeq \Omega)$	$\neg(Y \preccurlyeq X)$ / $\neg(Y \preccurlyeq R)$
$(Y \parallel X) \wedge (Y \parallel Z)$	$\left[\left(\neg(Y \preccurlyeq A)\right) \vee \left(\neg(Y \succcurlyeq B)\right)\right] \wedge [\neg(Y \preccurlyeq \Omega)]$	$\neg(Y \preccurlyeq X)$

Required functionality

Specification	Configuration	Assembly
$Y \hookrightarrow Z$	$Y \hookrightarrow \Omega$	$Y \hookrightarrow R$
Non-propagation		
$A \preccurlyeq Y \preccurlyeq \Omega$	$Z \preccurlyeq Y \preccurlyeq R$	$Y \succcurlyeq \Omega$
Propagation		
Inter-level	**Inter-level** / **Intra-level**	**Inter-level** / **Intra-level**
$\neg(Y \preccurlyeq \Omega)$	$(\neg(Y \succcurlyeq Z \Rightarrow \uparrow)) \vee (\neg(Y \preccurlyeq R \Rightarrow \downarrow))$ / $\neg(Y \succcurlyeq X)$	$\neg(Y \succcurlyeq \Omega)$ / $\neg(Y \succcurlyeq B)$
$(Y \parallel \Omega) \wedge (Y \parallel A)$	$\left[\left(\neg(Y \succcurlyeq Z)\right) \vee \left(\neg(Y \preccurlyeq R)\right)\right] \wedge [\neg(Y \succcurlyeq X)]$	$\left(\neg(Y \succcurlyeq \Omega)\right) \wedge \left(\neg(Y \succcurlyeq B)\right)$

Notations

$T_1 \prec T_2$: T_1 is a **subtype** of T_2

$T_1 \preccurlyeq T_2$: T_1 is a **subtype** of T_2 **or equal to** T_2.

$T_1 \succ T_2$: T_1 is a **supertype** of T_2.

$T_1 \succcurlyeq T_2$: T_1 is a **supertype** of T_2 **or equal to** T_2.

$T_1 \parallel T_2$: T_1 **is not comparable to** T_2.

$\neg(T_1 \preccurlyeq T_2) \Leftrightarrow ((T_1 \succ T_2) \vee (T_1 \parallel T_2))$: T_1 **is either a supertype of** T_2 **or not comparable to** T_2.

$\neg(T_1 \succcurlyeq T_2) \Leftrightarrow ((T_1 \prec T_2) \vee (T_1 \parallel T_2))$: T_1 **is either a subtype of** T_2 **or not comparable to** T_2.

$T_2 \hookrightarrow T_1$: T_2 **replaces** T_1.

3. Example of version propagation

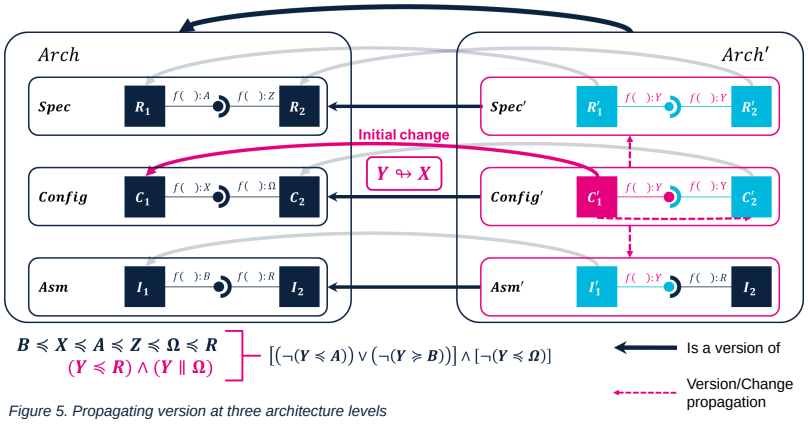


Figure 5. Propagating version at three architecture levels

4. Generalization

1 to n replacement

- Cases of 1 *to* n replacement:
  - A role may be realized by n component classes
  - Many roles may be realized by one component class
  - A component class may be instantiated by n component instances

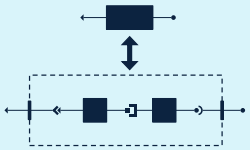


Figure 3. Connected components seen as a single composite component

- multiple connected components can be considered as a single composite component

Multiple connections

- separately study each connection

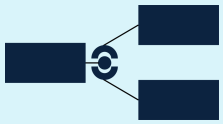


Figure 4. Multiple connections with an interface

5. Conclusion and future work

- Substitutability-based principles for predicting version propagation in three-level component-based architectures
  - Identification of component substitution scenarios
  - component **substitution is not a fine-grained enough criterion** → parameter types into signatures
- Future work
  - Formalization and automation of version propagation

Publications

[SATToSE 2017, Madrid, Spain] A. Le Borgne, D. Delahaye, M. Huchard, C. Urtado, and S. Vauttier, "*Preliminary study on predicting version propagation in three-level component-based architectures*", to appear in *Proceedings of the 7th Seminar on Advanced Techniques & Tools for Software Evolution*, 2017.

[SEKE 2017, Pittsburgh, USA] A. Le Borgne, D. Delahaye, M. Huchard, C. Urtado, and S. Vauttier, "*Substitutability-Based Version Propagation to Manage the Evolution of Three-Level Component-Based Architectures*", to appear in *Proceedings of the 29th International Conference on Software Engineering & Knowledge Engineering*, 2017.

Authors

Alexandre Le Borgne *
David Delahaye+
Marianne Huchard +
Christelle Urtado *
Sylvain Vauttier *

* IMT – Mines Alès
{ Alexandre.Le-Borgne, Christelle.Urtado, Sylvain.Vauttier } @mines-ales.fr

+ LIRMM / CNRS & Montpellier University, France
{ David.Delahaye, Marianne.Huchard } @lirmm.fr

Juin 2017 – GDR GPL, Montpellier

Androfleet: Testing WiFi P2P Mobile Apps in the Large

Lakhdar Meftah[1], Maria Gomez[2], Romain Rouvoy[1], and Isabelle Chrisment[3]

[1] Inria / Univ. Lille 1, France — first.last@inria.fr
[2] Saarland University, Germany — maria.gomez@uni-saarland.de
[3] Inria / Telecom Nancy, France — isabelle.chrisment@inria.fr

Abstract. WiFi Direct has become a popular communication technology among mobile apps, e.g., in multiplayer games, messaging, and social apps. However, testing WiFi P2P-based app remains a challenge. This poster introduces Androfleet, an acceptance testing framework to automate the testing of WiFi P2P apps at scale. Beyond the capability of testing p2p interactions under various conditions, Androfleet supports the deployment and the emulation of a fleet of mobile devices as part of an alpha testing phase in order to assess the robustness of a WiFi P2P app once deployed in the field.

1 Motivation

WiFi *peer-to-peer* (WiFi P2P, also known as WiFi Direct)[4] allows mobile applications (apps) to connect to each other via WiFi without an intermediate access point, as long as they comply with the WiFi Alliance's WiFi Direct certification. This communication mode has been commonly adopted by mobile apps to support single hop interactions and interact with one or more devices simultaneously (*e.g.*, file sharing, messaging, ad hoc routing).

Current practices to test WiFi P2P mobile apps present the following challenges:

Challenge #1: Lack of emulators. To the best of our knowledge, none of current Android emulators (e.g., GenyMotion, Android Emulator) support WiFi P2P communications. The only way to test a WiFi P2P app is to use real physical devices and test the app in them simultaneously. However, this approach presents a series of drawbacks: 1. *Scalability.* This approach does not scale to hundreds of mobile devices. 2. *Decentralization of tests.* Each app can be tested in one device individually. However, the behavior of the p2p system cannot be assessed as a whole.

Challenge #2: Lack of testing support for inter-app communications. There is a lack of testing framework, in both industry and academia, devoted to test p2p mobile apps.

To overcome the identified challenges, we present Androfleet, a large scale testing framework for WiFi P2P Android apps.

2 Testing P2p Mobile Apps

The goal of Androfleet is to automate WiFi P2P User Acceptance Testing. Androfleet provides 3 main features:

1. A DSL to describe WiFi P2P testing scenarios, such as peer discovery and peer interactions.
2. A large-scale platform which emulates multiple devices, and supports parallel testing on multiple devices as well as the communications among them.
3. A *Peer Discovery* node, that will provide the list of nearby devices for each emulator.

Figure 1 shows an overview of the Androfleet framework. Androfleet contains three main parts: the *Peer Discovery* node, the emulators nodes and the calabash-calabash features files. Androfleet uses Docker containers for each node and each the device node contains an Android emulator. The communication between nodes is done via weave network [5] and the communications

[4] http://www.wi-fi.org/discover-wi-fi/wi-fi-direct
[5] https://www.weave.works/oss/net/

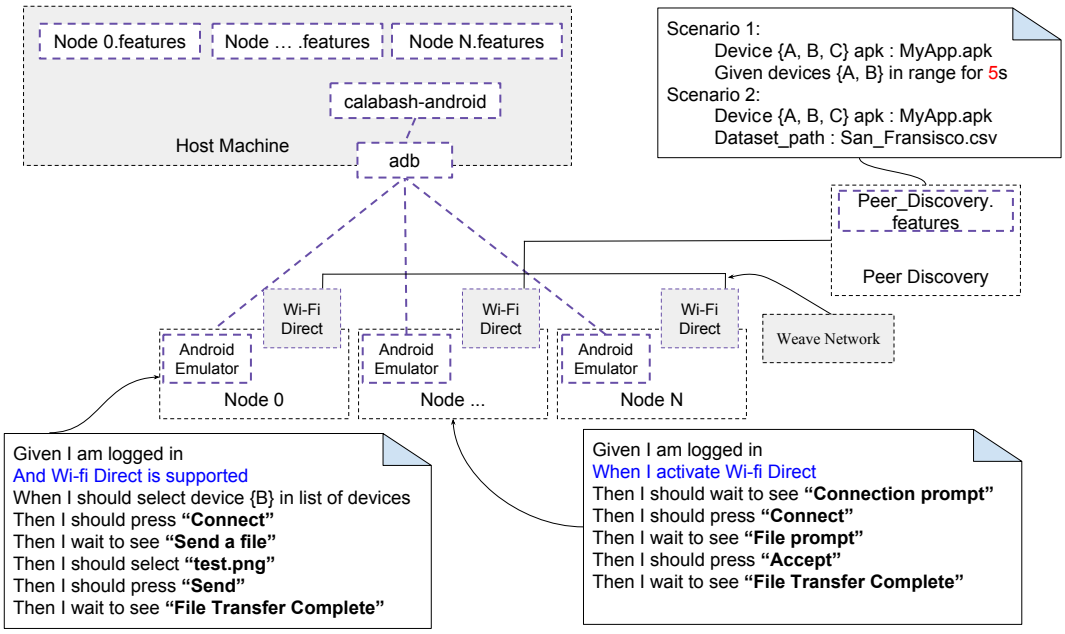


Fig. 1. Example of P2P scenario in Androfleet DSL

with the *Peer Discovery* node is done using Akka Scala to provide large scale deployment. The android emulators are then connected to the developer machine via ADB (Android Debug Bridge). The *Peer Discovery* node is responsible for emulating WiFi Direct behavior for all the emulators, it processes the *Peer_Discovery.feature* file to inform the Android emulators with nearby devices, the *Peer Discovery* node can also use a GPS mobility dataset to calculate the distance between devices and inform the devices that are in range.

2.1 Describing test scenarios

To describe testing scenarios, ANDROFLEET provides a DSL to developers. Previous research have extended Calabash framework [6] (A User Acceptance Testing framework) to add some context-based testing [2, 1]. We have also extended Calabash to support WiFi P2P specific vocabulary.

Figure 1 shows an example of a testing scenario created with the ANDROFLEET DSL. In this scenario, we are testing a WiFi P2P File Sharing App, and the file is sent from *Node 0* to *Node 1*.

2.2 Running the scenarios

To run the scenarios, the developer has to provide: 1. *testing scenarios* for each single node; 2. *device profiles* (e.g. OS version, hardware specification) to test the app; and 3. *Peer Discovery behavior* Then, the different scenarios are executed in parallel using Calabash-Android.

References

1. T. Griebe and V. Gruhn. A model-based approach to test automation for context-aware mobile applications. *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14*, pages 420–427, 2014.
2. M. Hesenius, T. Griebe, S. Gries, and V. Gruhn. Automating UI tests for mobile applications with formal gesture descriptions. *MobileHCI 2014 - Proceedings of the 16th ACM International Conference on Human-Computer Interaction with Mobile Devices and Services*, pages 213–222, 2014.

[6] http://calaba.sh/

Bottom-Up Technologies for Reuse

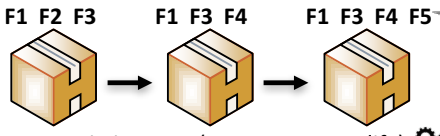
Automated Extractive Adoption of Software Product Lines

*UPMC SORBONNE UNIVERSITÉS

Jabier Martinez*, Tewfik Ziadi*
Tegawendé F. Bissyandé#, Jacques Klein# and Yves le Traon#

uni.lu UNIVERSITÉ DU LUXEMBOURG

Clone-and-own



- Opportunistic reuse (copy-paste-modify) to create variants
- Adding or removing features to respond quickly to different customers' needs

Maintenance issues

Extractive SPL Adoption

Software Product Lines

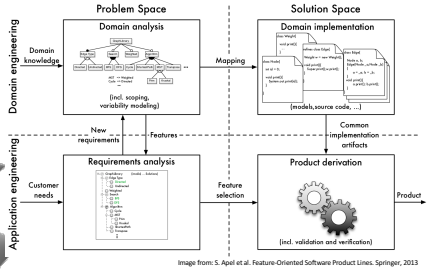
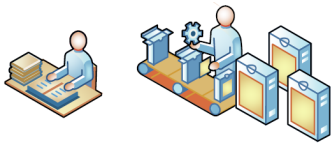
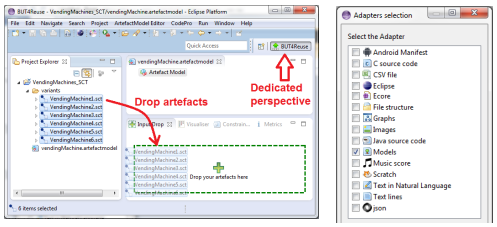


Image from: S. Apel et al. Feature-Oriented Software Product Lines. Springer, 2013

- Systematic reuse to create variants
- Satisfying the needs of a domain

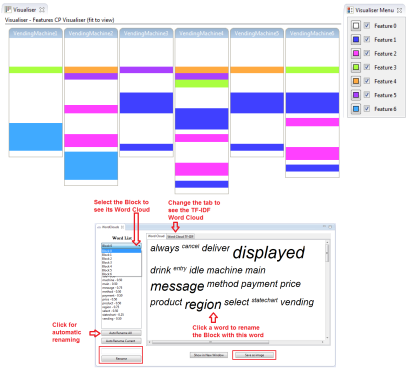


BUT4Reuse framework



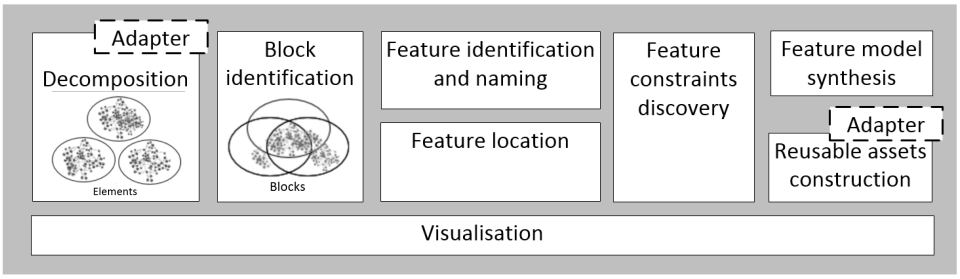
Generic and Extensible

- 15 adapters (Java, C++, Models...)
- Integrate your own adapters and techniques
- Benchmarks to compare techniques



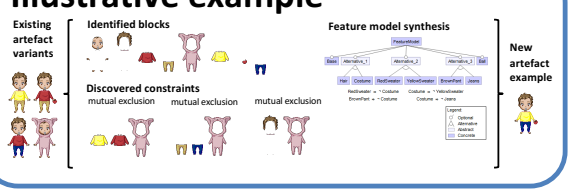
Artefact variants

Extractive SPL adoption



Software product line

Illustrative example



http://but4reuse.github.io

BUILT ON eclipse GitHub @BUT4Reuse

but4reuse@gmail.com

Du cahier des charges à sa spécification : patrons de développement

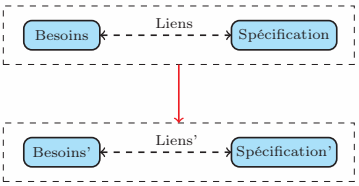
Imen Sayar imen.sayar@loria.fr

UNIVERSITÉ DE LORRAINE

Loria Laboratoire lorrain de recherche en informatique et ses applications

Contexte

- Développement formel en Event-B de systèmes critiques hybrides
  - raffinement
  - vérification et validation
- Définition des liens besoins/spécification
- Evolution et traçabilité des besoins



Problématique

- Problèmes
  - Complexité élevée des systèmes
  - Coût élevé du développement formel
    - temps
    - effort de preuve
    - effort de validation
- Données
  - Besoins décrits sous une forme répétitive
- Objectifs
  - Développer le <Besoins', Liens', Spécification'>
  - Diminuer le coût de développement

Pré-requis

- Patrons de conception

  ⇒ phase de programmation
- Patrons de spécification formelle

  ⇒ phase de spécification (Event-B)

Conclusion

- Le patron de développement
  - un sous-modèle prédéfini en Event-B : figure 1
  - ce qui reste à définir
    - CdC : besoins pour le collage
    - Spécification
      - ➙ invariants et événements de collage
      - ➙ événements pour changement d'état du paramètre p du patron
    - Liens : établis automatiquement
- Perspectives
  - Application du patron sur d'autres études de cas
  - Description de patrons à plusieurs paramètres
  - Construction de patrons de développement pour des systèmes hybrides à partir des patrons de conception de GoF

Patron de développement

- Patron pour <Besoins, Liens, Spécification>

  ⇒ phase de développement
- Le patron appliqué au besoin R-i
  - a un paramètre p
  - utilise <R-j, Liens, Spécification> existant
    - paramètre de R-j : prev_p
    - spécification de R-j : R-j_Mch
  - précise ce qui reste à définir
    - collage
    - gestion de l'évolution du paramètre p

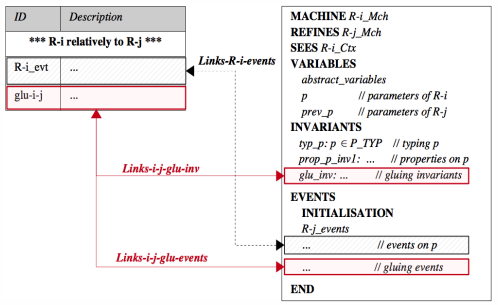


Figure 1 – Patron

Application à la machine d'hémodialyse

R-8. During initiation, if the software detects that the pressure at the AP Arterial Pressure transducer falls below the lower pressure limit, then the software shall stop the BP Blood Pumping and execute an alarm signal.

- Données
  - <R-6, Liens, Spécification> existe
  - Le besoin R-8 a la même forme que R-6
    - Le paramètre de R-6 est vp
    - Le paramètre de R-8 est ap
- Objectif : développer <R-8, Liens, Spécification> à partir de R-6
- Choix de développement : patron de la figure 1
- Résultat
  - Partie générée automatiquement
  - Partie à définir
    - CdC : des besoins relatifs au collage, figure 3
    - Spécification
      - ➙ des événements relatifs à l'avancement de ap, figure 4
      - ➙ des invariants de collage, figure 5
      - ➙ des événements de collage
    - Liens mémorisés entre R-8 et sa spécification

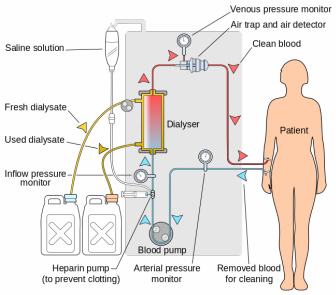


Figure 2 – Machine d'hémodialyse

ID	*Description*
glu-8-6-1	If [ap] and [vp] fall below [lower_press_limit], then the software shall [manage_deficit_ap_deficit_vp]
glu-8-6-2	If [ap] falls below [lower_press_limit] and [vp] is normal, then the software shall [manage_deficit_ap]
glu-8-6-3	If [ap] and [vp] are normal, then the software does nothing
glu-8-6-4	If [ap] is normal and [vp] falls below [lower_press_limit], then the software shall [manage_deficit_vp]

Figure 3 – CdC : collage entre R-8 et R-6

Event decrease_ap_initiation
  when
    grd1: $blood\_pumping = started$
    grd2: $phase = initiation$
    grd3: $alarm\_ap = NULL$
    grd4: $ap \geq lower\_press\_limit$
  then
    act1: $ap := ap - 10$
end

Figure 4 – Un événement lié à *ap*

$$glu\_1: alarm\_ap = ALM\_ap \wedge alarm\_vp = ALM\_vp \Rightarrow ap < lower\_press\_limit \wedge vp < lower\_press\_limit$$

Figure 5 – Un invariant de collage

Références

Références

[1] Erich Gamma and Richard Helm and Ralph Johnson and John Vlissides, *Design Patterns : Catalogue de modèles de conception réutilisables*, International Thomson Publishing France, Paris, 1996.

[2] Jean-Raymond Abrial and Thai Son Hoang, *Using Design Patterns in Formal Methods : An Event-B Approach*, in *ICTAC*, in 5th International Colloquium, Istanbul, Turkey, 2008.

[3] Imen Sayar and Jeanine Souquières, *Du cahier des charges à sa spécification*, in *AFADL* 16èmes journées Approches Formelles dans l'Assistance au Développement de Logiciels, Montpellier, France, 2017.

[4] Atif Mashkoor, *The Hemodialysis Machine Case Study*, in *Abstract State Machines, Alloy, B, TLA, VDM, and Z* - 5th International Conference, ABZ, Linz, Austria, 2016.

Spoon

Open source library to analyze, rewrite, transform, transpile Java source code.

http://github.com/INRIA/spoon

Spoon

Transfos

Spoon takes in input source code and builds a well-formed AST. It scales from one single Java class to ultra-large applications.

With templates and processors, Spoon can apply complex transformations on source code. It then pretty-prints the transformed code on disk. This is fully integrated in Maven and Gradle.

AST

SpoonLabs: usecases

Metrics

Code metrics: use Spoon to quickly compute customized code metrics on your projects

Architecture checking: use Spoon to automatically verify architectural rules on your projects

Testing

Transpiling

Code transpiling: use Spoon to easily transform Java code to other formats

Code instrumentation: use Spoon to insert probes in your applications for coverage, logging, hotpatching, etc

Instrumenting

Statistics

665 Pull Requests
Number of pull requests merged from the community.

303 Stars
Keep track of repositories that users find interesting.

66 Forks
Experiments changes without affecting the original project.

30 Contributors
Commiting to the master branch of the repository

1500+ Commits
Previous commits that were made to the project

Statistics taken from the GitHub project
January 23, 2017

Technologies

Java Git GitHub Gradle Maven Jenkins Travis

Inria INVENTEURS DU MONDE NUMÉRIQUE

SPIRALS

Université de Lille 1 SCIENCES ET TECHNOLOGIES

OW2

OntoEventB : Un outil pour la modélisation des ontologies dans B Événementiel

Linda Mohand-Oussaïd and Idir Aït-Sadoune

LRI, CentraleSupelec, Université Paris Saclay
Plateau du Moulon, Gif-sur-Yvette, France
{linda.mohandoussaid, idir.aitsadoune}@centralesupelec.fr

1 Introduction

Cet article présente un plug-in intégré à la plateforme Rodin [2] implémentant deux approches [3][4] de formalisation des ontologies décrites par des langages de description des ontologiques (OWL, Plib, ...) en utilisant la théorie des ensembles et la logique du premier ordre supportées par B Événementiel [1]. L'intérêt de cette formalisation est d'enrichir le processus de spécification et de vérification utilisant la méthode B Événementiel, en intégrant des modèles de données et de connaissances décrits dans des ontologies. L'utilisation des ontologies dans un développement B Événementiel va servir à annoter et/ou à typer les données manipulées par le système, ce qui permet de vérifier, en plus des propriétés caractéristiques du système, des propriétés liées à la cohérence des données manipulées et induites par les contraintes du domaine exprimées dans les ontologies [3].

Cet article est organisé de la manière suivante : la section 2 présente l'architecture interne du plug-in, et la section 3 est consacrée à la description de la procédure d'installation et d'utilisation du plug-in. Enfin, nous concluons l'article avec des perspectives d'évolution relatives à ce plug-in.

2 L'architecture interne de OntoEventB

Le plug-in *OntoEventB* est développé selon une architecture interne articulée autour de trois composants (voir Figure 1) :

- **Le composant Modèles d'Entrée.** Ce composant est dédié au traitement des fichiers contenant les descriptions des ontologiques. Ce composant parcourt le contenu des fichiers en entrée, extrait les concepts ontologiques de bases (classes, propriétés, relations, ...) et les envoie en entrée du composant Modèle Pivot.
- **Le composant Modèle Pivot.** Ce composant contient un modèle intermédiaire qui regroupe les concepts communs et spécifiques utilisés par les langages de description ontologiques comme OWL, Plib ou le diagramme de classe d'UML (classes, propriétés, relations entre classes, relations entre propriétés). Le composant Modèle Pivot récupère l'ensemble des concepts reçus depuis le composant Modèles d'Entrée, les fait correspondre aux concepts de son modèle pivot et les insère dans ce modèle. Il prépare ces concepts au traitement (formalisation B Événementiel) qui est effectué par le composant Modèles de Sortie.

— **Le composant Modèles de Sortie.** Ce composant implémente les deux approches de formalisation des ontologies, Shallow [3] et Deep [3][4]. L'approche Shallow modélise les concepts définis dans une ontologie en s'appuyant sur la sémantique de B Événementiel (les classes et les propriétés de l'ontologie sont formalisées par des ensembles et des relations dans un contexte B Événementiel) tandis que l'approche Deep utilise une modélisation en profondeur qui consiste, dans un premier temps, à formaliser les concepts ontologiques génériques dans un contexte B Événementiel générique (les notions de classes, de propriétés, d'héritage, ..., sont définies dans un contexte B Événementiel), et à modéliser les concepts définis dans une ontologie dans un second contexte B Événementiel (qui étend le contexte générique) par un ensemble d'instances reliées aux concepts génériques du contexte générique. Le composant Modèles de Sortie parcourt le modèle pivot et génère un projet Rodin contenant des contextes B Événementiel selon l'approche choisie par l'utilisateur.

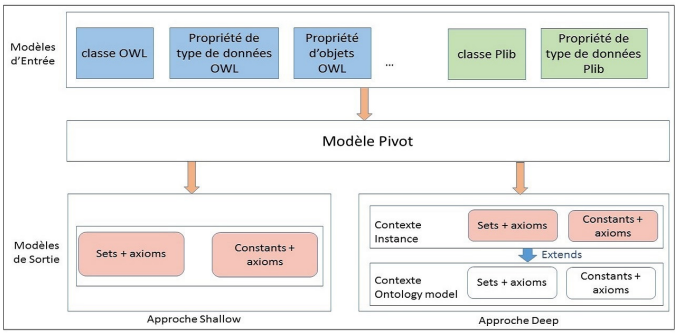


FIGURE 1. L'architecture interne de OntoEventB.

L'utilisation de cette architecture permet d'étendre le plug-in *OntoEventB* en intégrant de nouveaux langages de description des ontologies en entrée sans redéfinir les règles de formalisation en B Événementiel entre les composants Modèle Pivot et Modèles de Sortie. Une fois la correspondance entre les concepts du nouveau langage et les concepts du Modèle Pivot établie, la formalisation en B Événementiel est directement effectuée sans modifier les règles de formalisation définies par les approches Shallow et Deep.

3 Installation et utilisation de OntoEventB

Dans le soucis d'intégrer l'outil *OntoEventB* à la plateforme Rodin, nous avons fait le choix de le développer sous la forme d'un plug-in Eclipse. Ainsi, pour l'utiliser, il

faut d'abord l'intégrer à la plateforme Rodin en passant par l'assistant d'installation des nouveaux logiciels de la plateforme, et en utilisant l'adresse du update site du plug-in [1].

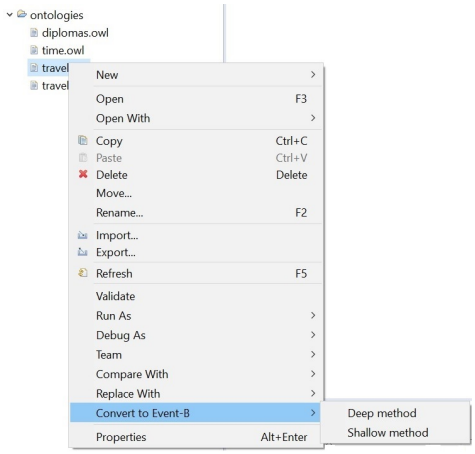


FIGURE 2. Le sous-menu OntoEventB.

Une fois l'installation du plug-in effectuée, un sous-menu portant l'étiquette *Convert to Event-B* devient accessible par clic droit sur un fichier possédant l'extension *.owl* [2] dans l'explorateur de projet comme indiqué dans la figure 2. Ce menu propose deux fonctions : la fonction Deep Method et la fonction Shallow Method. Nous illustrons le fonctionnement de OntoEventB sur une ontologie OWL relative au domaine des voyages. Un extrait de cette ontologie est présenté ci-dessous. Il contient deux classes *Accommodation* et *Destination* et une propriété d'objet *hasAccommodation*.

travel.owl

```
<owl:Class rdf:ID="Accommodation"></owl:Class>
<owl:Class rdf:ID="Destination"></owl:Class>
<owl:ObjectProperty rdf:ID="hasAccommodation">
        <rdfs:range rdf:resource="#Accommodation"/>
        <rdfs:domain rdf:resource="#Destination"/>
</owl:ObjectProperty>
```

1. **La fonction Shallow method.** L'invocation de cette fonction sur un fichier OWL conduit à la création d'un nouveau projet B Événementiel composé d'un unique contexte contenant la formalisation de l'ontologie reçue en entrée en utilisant des ensembles, des constantes et des axiomes, résultat de l'application de

1. `http://downloads.sourceforge.net/project/ontoeventb/update-site`
2. La version actuelle du plug-in supporte uniquement les ontologies OWL en entrée.

l'approche Shallow. La Figure 3 présente le contexte *Travel* généré relativement à l'extrait présenté ci-dessus. Les classes *Accommodation* et *Destination* sont formalisées par les ensembles *Accommodation* et *Destination* sous-ensembles de *Thing* (ensemble abstrait correspondant à la classe racine Thing de OWL). La relation d'héritage est formalisée par l'opérateur $\subseteq$. La propriété *hasAccommodation* est formalisée par une relation possédant comme domaine l'ensemble *Accommodation* et comme co-domaine l'ensemble *Destination*.

CONTEXT Travel
SETS
Thing
CONSTANTS
Accommodation Destination hasAccommodation
AXIOMS
axm1 : $Accommodation \subseteq Thing$
axm2 : $Destination \subseteq Thing$
axm3 : $hasAccommodation \in Destination \leftrightarrow Accommodation$
END

FIGURE 3. La fonction OntoEventB Shallow

2. **La fonction Deep method.** L'invocation de cette fonction sur un fichier OWL conduit à la création d'un nouveau projet B Événementiel contenant deux contextes : un contexte générique *ontologyModel* définissant les concepts ontologiques génériques, généré pour toutes les ontologies et un contexte instance *Travel* étendant le contexte *ontologyModel* et décrivant l'ontologie en appliquant l'approche Deep. La Figure 4 présente les contextes générés relativement à l'extrait présenté ci-dessus. Dans le contexte *ontologyModel*, sont définis les ensembles de classes *CLASS* et de propriétés *PROPERTY*. Les relations *HAS_PROPERTY* et *IS_A* sont définies pour modéliser respectivement la relation entre les classes et les propriétés qui les caractérisent et la relation de subsomption entre classes. Dans le contexte *Travel*, les classes *Accommodation* et *Destination* sont formalisées par des instances de l'ensemble *CLASS*, elle sont reliées à l'ensemble Thing par des instances de *IS_A*. La propriété *hasAccommodation* est formalisée par une instance de l'ensemble *HAS_PROPERTY*.

4 Conclusion

Cet article présente l'implémentation d'une approche de formalisation des ontologies dans une méthode formelle comme B Événementiel à travers le plug-in *OntoEventB*. Cet outil permet de générer automatiquement des contextes B Événementiel formalisant des ontologies de domaine à intégrer dans un processus de développement formel d'un système utilisant B Événementiel. Ces contextes ainsi générés permettent de spécifier et de vérifier des contraintes spécifiques au domaine. D'autre part, l'utilisation d'un formalisme tel que B Événementiel offre également la possibilité de valider

CONTEXT ontologyModel
SETS
CLASS PROPERTY
CONSTANTS
HAS_PROPERTY IS_A
AXIOMS
axm1 : $HAS\_PROPERTY = CLASS \leftrightarrow PROPERTY$
axm2 : $IS\_A = \{IsA \mid IsA \in CLASS \leftrightarrow CLASS \land (\forall x, y \cdot (x \in CLASS \land y \in CLASS \land x \mapsto y \in IsA \Leftrightarrow ...)\}$
END

CONTEXT Travel
EXTENDS OntologyModel
CONSTANTS
Thing Accommodation Destination hasAccommodation isA hasProperties
AXIOMS
axm1 : $partition(CLASS, \{Thing\}, \{Destination\}, \{Accommodation\})$
axm2 : $partition(PROPERTY, \{hasAccommodation\})$
axm3 : $hasProperties = \{Destination \mapsto hasAccommodation\}$
axm4 : $isA = \{Destination \mapsto Thing, Accommodation \mapsto Thing\}$
axm5 : $isA \in IS\_A$
axm6 : $hasProperties \in HAS\_PROPERTIES$
END

FIGURE 4. La fonction OntoEventB Deep

les règles d'inférence utilisés par les raisonneurs ontologiques, qui peuvent être formalisées par des théorèmes à prouver.

La version actuelle du plug-in se base sur une architecture qui facilite l'extension du plug-in à d'autres langages de description des ontologies en entrée. En effet, le Modèle Pivot intègre l'ensemble des constructeurs utilisés par des langages comme OWL, Plib ou UML. La formalisation de l'ensemble des éléments composant le modèle Pivot en B Événementiel a été effectuée. Comme l'outil ne supporter que des ontologies OWL en entrée, l'intégration de nouvelles ontologies en entrée (comme les ontologies Plib par exemple) se fera au niveau du composant Modèles d'Entrée qui doit définir la correspondance des éléments du nouveau langage avec les éléments du modèle Pivot.

Remerciement. Ce travail est supporté par le projet ANR IMPEX.

Références

1. J-R Abrial. *Modeling in Event-B : system and software engineering*, Cambridge University Press, 2010.
2. J-R Abrial, M J. Butler, S. Hallerstede, T. Son Hoang, F. Mehta et L. Voisin. *Rodin : an open toolset for modelling and reasoning in Event-B*, STTT, vol 12(6), 2010.
3. IMPEX Consortium. Formal models for ontologies, June 2016.
4. Kahina Hacid, Yamine Aït Ameur. *Strengthening MDE and Formal Design Models by References to Domain Ontologies. A Model Annotation Based Approach*. ISoLA (1) 2016 : 340-357.

Ce document contient les actes des neuvièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées au LIRMM – Université de Montpellier du 13 au 16 juin 2017.

Les contributions présentées dans ce document ont été sélectionnées par les différents groupes de travail du GDR. Il s'agit de résumés, de nouvelles versions, de posters et de démonstrations qui correspondent à des travaux qui ont déjà été validés par les comités de programmes d'autres conférences et revues et dont les droits appartiennent exclusivement à leurs auteurs.