Actes des Septièmes journées nationales du

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

Université de Bordeaux

Laboratoire LaBRI

10 au 12 juin 2015

Editeurs : Xavier Blanc
Laurence Duchien

Impression : service de reprographie, Université de Bordeaux

Table des matières

Préface

C'est avec grand plaisir que je vous accueille pour les Septièmes Journées Nationales du GDR Génie de la Programmation et du Logiciel (GPL) à l'Université de Bordeaux. Ces journées sont l'occasion de rassembler la communauté du GDR GPL. 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 septième 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 4ème édition de la conférence en IngénieErie Logicielle CIEL 2015, ainsi que 14ème édition d'AFADL 2015, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels.

Ces journées sont une vitrine où chaque groupe de travail ou action transverse donne un aperçu de ses recherches. Une trentaine de 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.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Lionel Briand (Université Luxembourg, SnT), de François Taiani (Université Rennes 1, IRISA) et de François Pelligrini, (Université Bordeaux, LaBRI). Une table ronde, animée par Mireille Blay-Fornarino et Jean-Michel Bruel, abordera le thème des méthodes agiles.

Le GDR GPL a à cœur de mettre à l'honneur les jeunes chercheurs. C'est pourquoi nous décernerons un prix de thèse du GDR pour la troisième année consécutive. Nous aurons le plaisir de remettre le premier prix de thèse GPL à Clément Quinton pour sa thèse intitulée *Une Approche Basée sur les Lignes de Produits Logiciels pour Sélectionner et Configurer un Environnement Informatique dans les Nuages*, ainsi qu'un accessit à Julien Henry pour sa thèse intitulée *Static Analysis by Abstract Interpretation and Decision Procedures*. Le jury chargé de sélectionner le lauréat a été présidé par Dominique Méry. Que ce dernier soit ici remercié ainsi que l'ensemble des membres du jury, pour l'excellent travail de sélection.

Ces journées ont aussi pour objectif de préparer l'avenir en favorisant l'intégration des jeunes chercheurs dans la communauté et leur future mobilité. Dans cet esprit, nous les avons encouragés à proposer un poster ou une démonstration de leurs travaux. Une dizaine ont répondu à cet appel. Un prix du meilleur poster sera égalemment remis par un jury présidé par Catherine Dubois.

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 ou d'actions transverses, les membres du comité de direction du GDR GPL et, tout particulièrement le comité d'organisation de ces journées nationales présidé par Xavier Blanc. Je remercie chaleureusement l'ensemble des collègues bordelais qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Finalement, c'est également ma dernière année à la direction du GDR GPL. Je tiens donc à remercier le bureau, le conseil scientifique, les responsables de groupe ainsi que l'ensemble des membres du GDR pour avoir permis à notre communauté de vivre, échanger et se réunir régulièrement autour des thèmes de la programmation et du logiciel. De nombreuses interactions ont eu lieu pendant ces quatre années et ont permis des montages de projet, des publications en commun dans de prestigieuses conférences et dans d'excellents journaux, des mobilités géographiques, l'émergence de nouvelles thématiques, la mise en avant de jeunes chercheurs et, finalement, l'émergence d'un club industriel qui sera présidé par Patrick Farail. Je me retire donc en vous remerciant pour votre participation à la vie de cette formidable structure CNRS qu'est le GDR et en laissant la place à Pierre-Etienne Moreau en lui souhaitant de prendre grand plaisir à animer notre communauté.

Laurence Duchien
Directrice du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2015 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.
Nous sommes en période de renouvellement des groupes et de leurs responsables. Certains groupes ont choisi leurs nouveaux responsables qui ont travaillé déjà pour ces journées. D'autres groupes ont continué à travailler avec les responsables du présent quadriennal. Les noms mentionnés ci-dessous correspondent à l'ensemble des responsables des groupes pour les deux quadriennaux, à l'exception des nouveaux groupes.

Laurence Duchien (présidente), CRIStAL, Inria, Université Lille 1

Yamine Ait Ameur, IRIT, ENSEEIHT
Nicolas Anquetil, CRIStAL, Inria, Université de Lille 1
Xavier Blanc, LaBRI, Université de Bordeaux, IUF
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Sandrine Blazy, IRISA, Inria, Université de Rennes 1
Florian Brandner, ENSTA
Yohan Boichut, LIFO, Université d'Orléans
Isabelle Borne, IRISA, Université de Bretagne Sud
Eric Cariou, LIUPPA, Université de Pau et des Pays de l'Adour
Frédérice Dadeau, FEMTO-ST, Université de Franche-Comté
Catherine Dubois, CEDRIC, ENSIIE
Lydie Du Bousquet, LIG, Université Joseph Fourier
Frédéric Gava, LACL, Université Paris-Est
Jean-Louis Giavitto (Président du jury des posters), IRCAM, CNRS
Laure Gonnord, LIP (ENS Lyon), Université Lyon1
Gaetan Hains, LACL, Université Paris-Est
Pierre-Cyrille Heam, FEMTO-ST, Université Franche-Comté
Aurélie Hurault, IRIT, ENSEEIHT
Akram Idani, LIG, ENSIMAG
Claude Jard, AtlanSTIC, LINA, Université de Nantes
Yves Ledru, LIG, Université Joseph Fourier
Pierre-Etienne Moreau, LORIA, INRIA
Sébastien Mosser, I3S, Université Nice-Sophia-Antipolis
Clémentine Nebut, LIRMM, Université Montpellier II
Pascal Poizat, LIP6, Université Paris-Nanterre
Marc Pouzet, Ecole Normale Supérieure, Université Pierre et Marie Curie, IUF

Fabrice Rastello, INRIA, ENS Lyon
Romain Rouvoy, CRIStAL, Inria, Université Lille 1
Olivier H. Roux, IRCCyN, Université de Nantes
Salah Sadou, IRISA, Université Bretagne-Sud
Christel Seguin, ONERA Centre de Toulouse
Chouki Tibermacine, LIRMM, Université Montpellier II
Sarah Tucci, CEA LIST
Christelle Urtado, Ecole des Mines, Alès
Virginie Wiels, ONERA, Centre de Toulouse

Comité scientifique du GDR GPL

Franck Barbier(LIUPPA, Pau)
Charles Consel (LABRI, Bordeaux)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Stéphane Ducasse (INRIA, Lille)
Jacky Estublier (LIG, Grenoble)
Nicolas Halbwachs (Verimag, Grenoble)
Marie-Claude Gaudel (LRI, Orsay)
Gatan Hains (LACL, Créteil)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christine Paulin (LRI, Orsay)

Comité d'organisation

Xavier Blanc, (Président), Université de Bordeaux
Jean-Rémi Falleri, Université de Bordeaux
Floréal Morandat, Université de Bordeaux

Conférenciers invités

Scalable Software Testing and Verification Through Heuristic Search and Optimization : Experiences and Lessons Learned

Auteur : Lionel Briand (Université Luxembourg, SnT)

Résumé :
Testing and verification problems in the software industry come in many different forms, due to significant differences across domains and contexts. But one common challenge is scalability, the capacity to test and verify increasingly large, complex systems. Another concern relates to practicality. Can the inputs required by a given technique be realistically provided by engineers ? This talk reports on 10 years of research tackling verification and testing as a search and optimization problem, often but not always relying on abstractions and models of the system under test. Our observation is that most of the problems we faced could be re-expressed so as to make use of appropriate search and optimization techniques to automate a specific testing or verification strategy. One significant advantage of such an approach is that it often leads to solutions that scale in large problem spaces and that are less demanding in terms of the level of detail and precision required in models and abstractions. Their drawback, as heuristics, is that they are not amenable to proof and need to be thoroughly evaluated by empirical means. However, in the real world of software development, proof is usually not an option, even for smaller and critical systems. In practice, testing and verification is a means to reduce risk as much as possible given available resources and time. Concrete examples of problems we have addressed and that I will cover in my talk include schedulability analysis, stress/load testing, CPU usage analysis, robustness testing, testing closed-loop dynamic controllers, and SQL Injection testing. Most of these projects have been performed in industrial contexts and solutions were validated on industrial software. There are, however, many other examples in the literature, a growing research trend that has given rise to a new field of study named search-based software testing.

Biographie :
Lionel C. Briand is professor and FNR PEARL chair in software verification and validation at the SnT centre for Security, Reliability, and Trust, University of Luxembourg. He also acts as vice- director of the centre. Lionel started his career as a software engineer in France (CS Communications & Systems) and has conducted applied research in collaboration with industry for more than 20 years. Until moving to Luxembourg in January 2012, he was heading the Certus center for software verification and validation at Simula Research Laboratory, where he was leading applied research projects in collaboration with industrial partners. Before that, he was on the faculty of the department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, where he was full professor and held the Canada Research Chair (Tier I) in Software Quality Engineering. He has also been the software quality engineering department head at the Fraunhofer Institute for Experimental Software Engineering, Germany, and worked as a research scientist for the Software Engineering Laboratory, a consortium of the NASA Goddard Space Flight Center, CSC, and the University of Maryland, USA. Lionel was elevated to the grade of IEEE Fellow for his work on the testing of object-oriented systems.He was recently granted the IEEE Computer Society Harlan Mills award and the IEEE Reliability Society engineer-of-the-year award for his work on model-based verification and testing. His research interests include : software testing and verification, model-driven software development, search-based software engineering, and empirical software engineering. Lionel has been on the program, steering, or organization committees of many international, IEEE and ACM conferences. He is the coeditor-in-chief of Empirical Software Engineering (Springer) and is a member of the editorial boards of Systems and Software Modeling (Springer) and Software Testing, Verification, and Reliability (Wiley). More details can be found on : `http://people.svv.lu/briand/`

Génie logiciel et outils de distribution pour les systèmes répartis de très grande taille

Auteur : François Taiani (Université Rennes 1, IRISA)

Résumé :
Les systèmes informatiques répartis de très grande taille sont aujourd'hui omniprésents. Il suffit de penser aux plates-formes du cloud, ou aux infrastructures intelligentes pour l'énergie et la distribution de l'eau (smart grids), pour ne citer que deux exemples. Contrairement aux systèmes informatiques répartis classiques, ces systèmes sont très grands, impliquant généralement des millions d'utilisateurs et des dizaines de milliers de noeuds ; ils sont hétérogènes, combinant souvent de grands centres de données, des appareils mobiles, et des capteurs, souvent gérés par différentes organisations ; et ils sont composites, intégrant une nombre croissant de technologies logiciels et de services distribués fournis par des tiers.

En dépit de leur succès phénoménal, ces systèmes sont aujourd'hui de plus en plus difficiles à développer en utilisant des techniques logicielles et des architectures distribuées traditionnelles. Dans cette présentation, je m'attacherai à présenter, sur la base de travaux récents, comment l'algorithmique distribuée, en particulier au travers des systèmes auto-stables ou encore des protocoles épidémiques, peut être une source d'inspiration pour fournir des outils à la conception et à la réalisation de ces systèmes, en particulier lorsqu'ils sont combinés à des approches de génie logiciel (composants, aspects, modèles à l'exécution) qui ont fait leurs preuves pour le développement de logiciels complexes.

Biographie :
François Taïani est professeur à l'Université de Rennes 1. Il effectue sa recherche à l'IRISA et au centre Inria de Rennes, au sein de l'équipe projet ASAP dont il a pris la responsabilité depuis avril 2015. Il est aussi directeur des études de l'ESIR (l'école Supérieure d'Ingénieurs de Rennes). Il est diplômé de l'école Centrale Paris et de l'université de Stuttgart, et a reçu son doctorant en informatique de l'Université de Toulouse III en 2004 pour son travail effectué au LAAS-CNRS sur les intergiciels tolérant les pannes. Après un séjour d'un an à AT&T Labs (New Jersey, Etats-Unis) en 2004, François a travaillé de 2005 à 2012 comme Lecturer (Maître de Conférences) à l'Université de Lancaster (Royaume-Uni) où il a co-animé le groupe de recherche "Next Generation Middleware" avec Gordon Blair et Geoff Coulson. Ses intérêts de recherche portent sur les canevas de programmation ouverts pour les systèmes informatiques décentralisés de très grande taille, avec un accent particulier sur les réseaux sociaux et les mécanismes épidémiques. url : `http://ftaiani.ouvaton.org/`

Aspects juridiques de la création logicielle

Auteur : François Pelligrini, (Université Bordeaux, LaBRI)

Résumé :
Le développement logiciel ne peut se réduire à la seule production de code. Afin d'assurer la réussite d'un projet, il est nécessaire de prendre en compte, dès ses phases préliminaires, certains éléments de nature économique et juridique. Ceux-ci sont essentiels pour déterminer les moyens les plus économiques et pérennes d'atteindre l'objectif visé. Ils influent en particulier sur la décision de réutiliser des briques existantes, dont les types de licences et les éventuels coûts de redéveloppement conditionnent les modèles économiques possibles et la viabilité du projet. Au cours de cet exposé, après avoir rappelé les principes du droit applicable aux logiciels, nous présenterons les principes et méthodes nécessaires à cette analyse. Nous conclurons pas quelques recommandations pratiques permettant d'éviter certains écueils juridiques ultérieurs.

Biographie :
François Pellegrini est professeur en informatique et vice-président délégué au numérique de l'Université de Bordeaux, et chercheur au LaBRI et à Inria Bordeaux Sud-Ouest. Il est le président d'Aquinetic, une association accompagnant vers l'entrepreneuriat les porteurs de projets à base de technologies libres. Il est également, depuis de nombreuses années, le vice-président de l'ABUL, Association bordelaise des utilisateurs de logiciels libres, au sein de laquelle il a co-fondé les Rencontres mondiales du logiciel libre. Co-auteur, avec Sébastien Canevet, d'un ouvrage sur le droit des logiciels, il est Commissaire à la Cnil depuis janvier 2014.

Sessions des groupes de travail

Session de l'action AFSEC

Approches Formelles des Systémes Embarqués Communicants

Auteur : Claire Pagetti (ONERA)

Titre :
Calcul d'ordonnancement hors-ligne sur plateformes multi/pluri-cœurs à base de méthodes formelles

Résumé :
L'implantation sûre d'applications de type contrôle-commande sur des processeurs multi-cœurs (quelques cœurs intégrés sur une même puce reliés par un bus partagé) ou pluri-cœurs (nombreux cœurs intégrés sur une même carte connectés par un réseau sur puce) requiert de maîtriser la prédictibilité des cibles. Notre approche consiste à calculer hors-ligne des séquenceurs partitionnés non pré-emptifs et à imposer un modèle d'exécution adéquat, c'est-à-dire un ensemble de règles de programmation à suivre par le concepteur de façon à éviter ou au moins réduire les comportements non prédictibles.

Cet exposé portera sur les techniques utilisées de façon à générer une configuration (ou un placement) définie par (1) la localité (ou le cœur) exécutant chaque tâche, (2) les zones mémoires hébergeant les sections de manière temporaire ou permanente, (3) les moments de démarrage des tâches. Le problème de placement consiste à allouer les tâches sur la plate-forme de sorte que toutes les contraintes fonctionnelles (précédences et échéances) et non fonctionnelles (capacités mémoire et processeur) soient satisfaites. Il s'agit d'une variation du problème NP-complet de sac à dos (bin packing problem). Deux approches seront proposées: la première repose sur du model checking et la deuxième sur de la programmation par contraintes.

SCADE Hybrid: une extension de SCADE avec des ODEs

Marc Pouzet

DI, École normale supérieure, 45 rue d'Ulm, 75230 Paris cedex 05.

Les langages de modélisation de systèmes hybrides permettent de décrire à partir d'un source unique, le logiciel de contrôle et un modèle de son environnement physique. Ce source de référence est utilisé pour la simulation, le test, la vérification, puis la génération de code embarqué. Les langages de modélisation de systèmes hybrides explicites ou *causaux* tels que Simulink/StateFlow[1] combinent des équations différentielles ordinaires (EDOs), des équations de suites, des automates hiérarchiques à la manière des Statecharts [12], ainsi que des traits impératifs. Dans ces langages, les modèles combinent des signaux évoluant sur une base de temps discrète et d'autres sur une base de temps continue. La vérification formelle des systèmes hybrides est étudiée en détail [8]. Le présent travail considère la question différente, mais non moins importante, de la génération de code séquentiel (typiquement C) pour obtenir des simulation efficaces, fiables et une implémentation embarquée temps réel.

La génération de code séquentiel à partir d'un langage synchrone tel que Lustre a été étudiée en détail [11]. Elle peut être décrite précisément par une suite de transformations source-à-source qui élimitent progressivement les constructions de haut niveau (e.g., automates hiérarchiques, conditions d'activation) vers un langage data-flow élémentaire [10]. Ce langage est simplifié à nouveau et traduit vers un langage intermédiare générique servant à représenter des fonctions de transition [5], et qui est ensuite traduit vers C. C'est l'approche suivie dans Scade Suite KCG code generator de Scade 6[2], utilisé pour développer de nombreuses applications critiques.

Les langages synchrones manipulent des signaux à temps discret seulement. Leur expressivité a été volontairement contrainte pour garantir le détermisme, une exécution en espace et temps bornés et une génération simple et traçable. L'exécution cyclique d'un langage synchrone est extrêmement simple et ne souffre pas des complications causées par la présence d'un solveur numérique d'équations différentielles. Dans des langages tels que Simulink/StateFlow, au contraire, l'interaction entre temps discret et temps continu, combinée à des constructions de programme peu sûres (effets de bord, boucles `while`) n'est cependant pas suffisamment contrainte ni spécifiée avec précision. C'est la cause de comportements non déterministes inattendus [9,4] et de bugs de compilation [1]. La description précise de la génération de code — c'est-à-dire la sémantique réellement implémentée — est indispensable dans une chaîne de développement où le code embarqué et la simulation de l'ensemble du système doivent offrir le maximum de confiance. On aimerait, en somme, avoir l'expressivité des langages synchrones combinée à des équations différentielles, sans ne rien sacrifier à la confiance qu'offrent leur compilateurs.

Dans des travaux récents, nous avions indroduit une approche nouvelle consistant à bâtir un modeleur hybride a la Simulink/StateFlow en réutilisant les principes et les techniques de compilation d'un langage synchrone. Ce travail s'est appuyé sur une sémantique synchrone étendue fondée sur l'analyse non standard [4], la définition d'un noyau Lustre étendu avec des EDOs [3] puis des automates hiérarchiques [2], et enfin, une analyse de causalité [1]. Ces résultats ont été mis en oeuvre dans le langage Zélus [7] [3]. Cependant, une validation en vraie grandeur dans un compilateur synchrone industriel restait à faire.

Dans cet exposé, je montrerai comment obtenir un compilateur d'un langage synchrone avec EDOs à partir d'une compilateur synchrone existant. Les principes présentés ont été validés dans deux compilateurs, le compilateur de Zélus et le Scade Suite KCG code generator (Release 6.4, 2014) développé à Esterel-Technologies/ANSYS. Dans ce dernier, il a été possible de réutiliser entièrement l'infrastructure existante: le typage statique, les analyses de causalité et

[1] http://mathworks.org/simulink

[2] http://www.esterel-technologies.com/products/scade-suite/

[3] zelus.di.ens.fr

d'initialisation, les langages intermédiaires et les différentes optimisations. Le traitement des nouvelles constructions a nécessité des ajouts mineurs. Nous avons été très surpris de découvrir que l'extension hybride de Scade 6 représentait en tout moins de 5% de lignes de code (OCaml) supplémentaires, ce qui est très peu. Ce résultat confirme l'intérêt de bâtir un modeleur hybride par dessus un langage synchrone existant. Il montre également la versatilité de l'architecture du compilateur KCG basé sur des réécritures successives. Enfin, l'extension proposée est conservatrice en ce sens que les fonctions synchrones existantes sont compilées exactement comme avant — le même code généré des fonctions synchrones est utilisé durant la simulation et l'exécution sur la plateforme cible. Il n'y pas de nécessité de ré-implémenter (en C ou en Scade) les contrôleurs discrets développés durant les phases amont de modélisation et utilisés durant la simulation.

Ce travail a été réalisé en collaboration avec Timothy Bourke (INRIA Paris-Rocquencourt), Jean-Louis Colaco, Bruno Pagano et Cédric Pasteur (Esterel-Technologies/ANSYS, Core Team). Il reprend des résultats présentés dans [6].

References

1. Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. A Type-based Analysis of Causality Loops in Hybrid Systems Modelers. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.
2. Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. A Hybrid Synchronous Language with Hierarchical Automata: Static Typing and Translation to Synchronous Code. In *ACM SIGPLAN/SIGBED Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011.
3. Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA, April 2011.
4. Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3):877–910, May 2012. Special issue in honor of Amir Pnueli.
5. Darek Biernacki, Jean-Louis Colaco, Grégoire Hamon, and Marc Pouzet. Clock-directed Modular Code Generation of Synchronous Data-flow Languages. In *ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.
6. Timothy Bourke, Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. A Synchronous-based Code Generator For Explicit Hybrid Systems Languages. In *International Conference on Compiler Construction (CC)*, LNCS, London, UK, April 11-18 2015.
7. Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.
8. Luca Carloni, Maria D. Di Benedetto, Alessandro Pinto, and Alberto Sangiovanni-Vincentelli. Modeling Techniques, Programming Languages, Design Toolsets and Interchange Formats for Hybrid Systems. Technical report, IST-2001-38314 WPHS, Columbus Project, March 19 2004.
9. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. *ACM Transactions on Embedded Computing Systems*, 2005. Special Issue on Embedded Software.
10. Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. In *ACM International Conference on Embedded Software (EMSOFT'05)*, Jersey city, New Jersey, USA, September 2005.
11. N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991.
12. D. Harel. StateCharts: a Visual Approach to Complex Systems. *Science of Computer Programming*, 8-3:231–275, 1987.

Processus de branchement des réseaux de Petri à reset arcs

Maurice Comlan[1], David Delfieu[1] and Médésu Sogbohossou[2]

[1] Institut de Recherche en Communications et Cybernétique de Nantes, France
{maurice.comlan, david.delfieu} @irccyn.ec-nantes.fr
[2] Université d'Abomey-Calavi, Bénin
medesu.sogbohossou@gmail.com

Résumé

Les réseaux de Petri sont un outil très répandu de modélisation et d'étude des systèmes concurrents autour desquels s'est constituée une large communauté scientifique. Il existe des extensions qui ont pour but d'étendre le formalisme de référence par les réseaux de Petri pour proposer soit des modèles plus compacts en termes de description, soit plus puissants en termes de pouvoir d'expression. Nous pouvons citer entre autre l'ajout des reset arcs qui ajoutent un pouvoir expressif utile en ce sens qu'ils permettent de remettre à zéro (notion de réinitialisation) le contenu d'une place sans impacter sur les règles de sensibilisation des transitions dans le réseau. Nous proposons pour les réseaux de Petri à reset arcs ayant un réseau normal sous-jacent borné, une approche pour construire le dépliage (méthode d'ordre partiel pour contourner le problème d'explosion combinatoire lié à la construction du graphe d'états) et d'un préfixe complet et fini du dépliage. Nous présenterons aussi un algorithme pour l'identification des processus de branchement issus d'un tel dépliage.

1 Introduction

Les systèmes sont de plus en plus très complexes (multitâches, réactifs, ...), s'exécutent sur des architectures mono ou multiprocesseurs, parallèles ou répartis. Les tâches peuvent être soumises à des contraintes de synchronisation, de concurrence, de communication, etc. De par leur complexité, de tels systèmes sont difficiles à appréhender et il est nécessaire de disposer d'outils et de modèle fiables pour, d'une part, les concevoir et les comprendre, et d'autre part, pour les vérifier en s'assurant qu'un certain nombre de contraintes sont bien respectées. Par exemple, on peut s'assurer de l'absence de blocage, garantir le non dépassement de capacité d'un buffer à capacité limitée, garantir l'exclusion mutuelle sur les ressources, prouver que les contraintes temporelles sont bien respectés et bien d'autres. La finalité est de garantir la fiabilité et la sureté du fonctionnement du système et pour cela, il faut, pendant toute la phase de conception, de disposer d'une approche permettant d'expliciter précisément le comportement du système et d'analyser ce comportement afin de s'assurer qu'il est bien conforme au cahier de charges du système [12].

Pour une approche fiable, on a recours à des méthodes formelles qui permettent une modélisation mathématique du fonctionnement du système à partir de laquelle il est possible d'effectuer des preuves. Il s'agit en fait d'une représentation abstraite et approchée du système réel. Plusieurs modèles existent et chaque modèle possède ses caractéristiques propres, plus ou moins pertinentes dans une conception spécifique. En conséquence, le choix du modèle dépendra du système conçu et des propriétés à analyser. Parmi l'ensemble des modèles existants, les réseaux de Petri [13] et leurs extensions possèdent un intérêt fondamental en ayant fourni les premières approches de modélisation basées sur un support graphique facilitant l'expression et la compréhension des mécanismes de base pour des systèmes communicants et enfin, en n'étant pas lié à un langage particulier, ils assurent l'indépendance de la modélisation vis-à-vis des implémentations. Rapidement, les premiers réseaux de Petri se sont révélés être néanmoins un modèle trop limité pour les concepteurs d'applications informatiques ou plus généralement industrielles. Cette limitation est due principalement à trois facteurs : il est impossible de modéliser des comportements similaires au moyen d'une seule représentation condensée ; l'analyse du réseau n'est pas paramétrable ; les conditions et les conséquences de l'évolution d'un réseau, de nature purement quantitative sont inadéquates pour modéliser des conditions et des évolutions qualitatives. Plusieurs extensions ont donc été introduites soit plus compacts en terme

de description, c'est-à-dire des abréviations qui n'augmentent pas le pouvoir d'expression mais améliore juste la simplicité des modélisations ; soit plus puissant en termes de pouvoir d'expression, c'est-à-dire des extensions qui permettent de décrire des mécanismes et des fonctionnement qui ne pouvaient se faire avec le modèle de base [12]. Nous pouvons citer les réseaux de Petri à arcs inhibiteurs, permettant le test à zéro d'une place, les réseaux de Petri à reset arcs permettant la remise à zéro d'une place, les réseaux de Petri temporels pour prendre en compte les contraintes temporelles, etc.

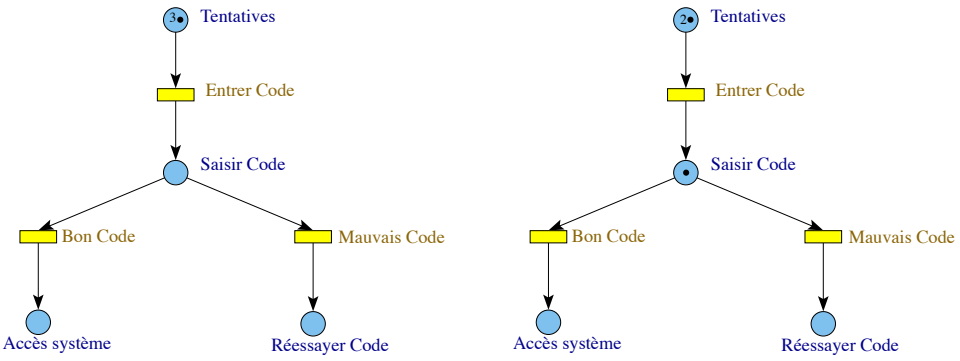
Pour conduire la vérification des systèmes, il est courant de construire le graphe d'états qui énumère de manière exhaustive les états possibles du système. Mais seulement, la construction de l'espace d'état (même fini) d'un système n'est toujours pas possible. L'une des raisons est l'explosion combinatoire des états due à la complexité du système, la forte concurrence dans le système, la présence d'un comportement infini dans le système, etc. Ce qui induit le problème de la limitation des ressources mémoires [1]. Les techniques dites à «ordre partiel» et plus particulièrement le dépliage [7] est une méthode d'ordre partiel largement utilisé. Il contourne le problème d'explosion combinatoire en éliminant la représentation du parallélisme par l'entrelacement des actions. Nous présentons dans ce papier une approche pour la construction du dépliage d'une extension particulière des réseaux de Petri : les réseaux de Petri à reset arcs.

Ce papier comporte six sections. La section 2 présente les réseaux de Petri en général et ceux à reset arcs en particulier, ce qui nous permet de définir dans la section 3 la notion de dépliage d'un réseau de Petri. Notre contribution commence dans la section 4 où nous présentons une approche pour construire le dépliage et le préfixe complet fini des réseaux de Petri avec des reset arcs et un algorithme pour identifier les processus de branchement d'un tel dépliage dans la section 5. La dernière section est consacrée à la conclusion et aux perspectives de ce travail.

2 Réseaux de Petri et les reset arcs

2.1 Réseaux de Petri

Un RdP [13] est un graphe biparti fait de deux types de sommets : les *places* et les *transitions*. Des arcs orientés relient des places à des transitions et *vice – versa* mais jamais deux sommets de même nature. Généralement, les places (ressources du système) sont représentées par des cercles qui sont constitués de jetons (instances des ressources), les transitions (événements du système) sont représentées par des rectangles ou des barres qui consomment et/ou produisent des jetons dans les places qui sont liées à la transition. Les places et les transitions sont reliés par des arcs orientés et valués (Place-Transition ou Transition-Place) qui indiquent le nombre de jetons à consommer et/ou à produire (par défaut le poids d'un arc est égal à 1). Le marquage d'un RdP est un vecteur à composantes entières positives ou nulles et dont la dimension est égale au nombre de places. La $n^{ième}$ composante de ce vecteur, représente le nombre de jetons dans la place n du RdP. La Figure 1(a) montre un exemple de réseau de Petri avec quatre places et trois transitions.



(a) Avant le tir de Entrer Code

(b) Après le tir de Entrer Code

FIGURE 1 – Exemple de réseau de Petri

Plus formellement,

Définition 1 (Réseau de Petri)**.** *Un réseau de Petri (normal) est un triplet $N =< P,T,W >$ avec :*

- *P un ensemble fini de places ;*
- *T un ensemble de transitions ;*
- *$P \cap T = \emptyset$ et $P \cup T \neq \emptyset$*
- *$W : P \times T \cup T \times P \to \mathbb{N}$ la fonction de valuation des arcs.*

Pour tout $(x,y) \in (P \times T)^2$, $W(x,y)$ désigne le poids de l'arc allant de x à y (s'il n'y a aucun arc, $W(x,y) = 0$). On définit aussi les fonctions *Pre* et *Post* respectivement les restrictions de W à $P \times T$ et à $T \times P$ ($Pre(p,t) = W(p,t)$ et $Post(p,t) = W(t,p)$). Pour tout $t \in T$, les pré-conditions de t, ${}^\bullet t = \{p \in P | W(p,t) > 0\}$ (reps. les post-conditions de t, $t^\bullet = \{p \in P | W(t,p) > 0\}$.

Un marquage M de N est l'application $M : P \to \mathbb{N}$ et le marquage initial est noté M_0 (le coupe $< N, M_0 >$ est appelé réseau marqué). Une transition $t \in T$ est sensibilisée par M, noté $M \xrightarrow{t}$, ssi $\forall p \in {}^\bullet t,\ M \geq W(p,t)$. Le tir de t conduit à un nouveau marquage M' ($M \xrightarrow{t} M'$) avec $\forall p \in P,\ M' = M - Pre(p,t) + Post(t,p)$. La Figure Figure 1(b) montre le réseau de Petri de la Figure 1(a) après le tir de la transition *Entrer Code*. Si le code entré est bon, que deviennent les jetons dans la place *Tentatives* ? On devait les retirer et les reset arcs nous permettent de réaliser le plus simplement cette action de retrait.

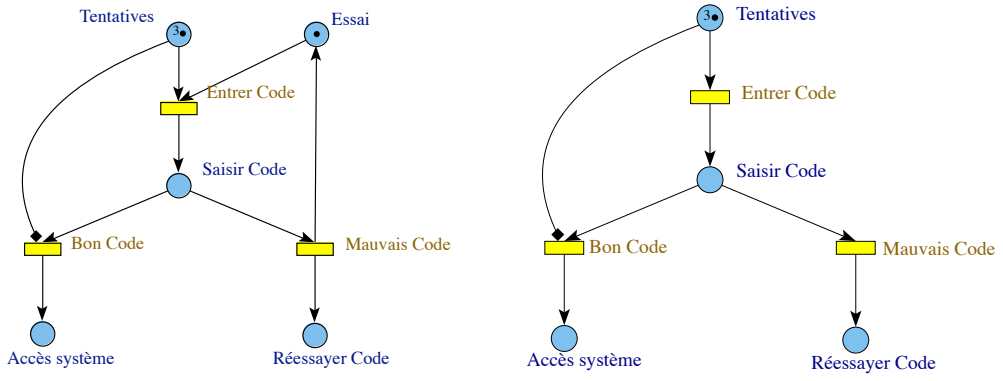
2.2 Réseaux de Petri avec des reset arcs

Les reset arcs constituent une des extensions des réseaux de Petri. Ils ne changent pas les règles de sensibilisation des transitions [5]. Par contre, si $M \xrightarrow{t} M'$ (i.e le tir de t fait passer le réseau de l'état M à l'état M') alors $\forall p \in P$ tel que $R(p,t) = 0$, $M'(p) = 0$. Mais si $W(t,p) > 0$ alors $M'(p) = W(t,p)$. De façon générale,

$$\forall p \in P,\ \ M' = (M - Pre(p,t)) \ . \ R(p,t) + Post(p,t)$$

avec . le produit matriciel de Hadamard (produit terme à terme).
La Figure 2(a) représente un réseau de Petri avec un reset arc (l'arc reliant *Tentatives* à *BonCode*). C'est une modélisation d'accès à un système avec trois essais. Si le code est correct, on a accès au système sans avoir à retaper le code, sinon, dans la limite des trois tentatives, on est invité à retaper le code. Nous avons :

- *P*={Tentatives, Essai,SaisirCode, Accès Système ,RéessayerCode}, ;
- *T*={EntrerCode,Bon Code, MauvaisCode} ;
- *R*={(Essais, Bon Code)} ;
- M_0=(3,1,0,0,0).



(a) Version complète (b) Version simplifiée

FIGURE 2 – Réseau de Petri à reset arc

Le tir de la transition *EnterCode* conduit à un marquage $M_1 = (2,0,1,0,0)$ et ensuite le tir de *BonCode* conduit au marquage $M_2 = (0,0,0,1,0)$. Pour la suite et pour des raisons de simplification, nous utiliserons la version simplifiée de ce système tel représenté par la Figure 2(b).

Définition 2 (Réseau de Petri avec des reset arcs)**.** *Un réseau de Petri avec des reset arcs est un tuple* $N_R =< P,T,W,R >$ *avec* $< P,T,W >$ *un réseau de Petri tel défini dans la Définition 1 et* $R : P \times T \to \{0,1\}$ *est l'ensemble des reset arcs (*$R(p,t) = 0$ *s'il y a un reset arc qui relie p à t, sinon* $R(p,t) = 1$*).*

Lemme 1. *Les reset arcs n'enlève que des comportements au réseau de Petri.*

Preuve. Comme dit plus haut, les resets arcs n'influent pas sur la sensibilisation des transitions du réseau. Ainsi à la limite, le réseau de Petri avec des reset arcs comportement les mêmes comportement que le réseau sans reset arcs sous-jacent. De plus le tir de la transition intervenant dans un reset arc annule le contenu de la place. Ce qui pourrait empêcher la sensibilisation de certaines transitions qui aurait pu être sensibilisées dans le réseau sans reset arcs. Nous pouvons donc admettre le lemme 1. □

3 Dépliage des réseaux de Petri

Le dépliage des réseaux de Petri fait partir des méthodes dites d'ordre partiel qui permettent d'optimiser la vérification des systèmes finis. L'objectif du dépliage est de représenter explicitement l'ordre partiel des événements du système étudié par un réseau de Petri particulier. Cette catégorie de réseau de Petri correspond aux réseaux d'occurrence [2] et utilise une fonction d'homomorphisme [6].

3.1 Homomorphisme de réseaux

Définition 3 (Homomorphisme de réseaux)**.** *Soient* (N_1,M_{o1}) *et* (N_2,M_{o2}) *deux réseaux marqués avec* $N_i = \{P_i,T_i,W_i\}$ *pour* $i = 1,2$*. Soit h une application* $h : P_1 \cup T_1 \to P_2 \cup T_2$ *telle que* $h(P_1) \subseteq h(P_2)$*,* $h(T_1) \subseteq h(T_2)$*. h est un homéomorphisme de réseaux de* (N_1,M_{o1}) *vers* (N_2,M_{o2}) *si :*

- $\forall t_i \in T_1 : Pré_2(h(t_i)) = h(Pré_1(t_i))$ *;*
- $\forall t_i \in T_1 : Post_2(h(t_i)) = h(Post_1(t_i))$ *;*
- $M_{o2} = M_{o1}$.

L'homomorphisme de réseaux préserve donc les noeuds. Les deux premières conditions de la définition assurent que l'environnement des transitions est préservé par l'application h et la troisième impose à h que les marquages initiaux des deux réseaux soient les mêmes.

3.2 Réseau d'occurrence

Le dépliage des réseaux de Petri est représenté par une structure particulière de réseau de Petri.

Définition 4 (Réseau d'occurrence)**.** *Un réseau d'occurrence est un réseau* $O =< B,E,F >$ *tel que :*

- $|^\bullet b| \leq 1, \forall b \in B$ *;*
- *O est acyclique ;*
- *O est fini par précédence, i.e* $\forall x \in B \cup E$*,* $\{ y \in B \cup E | y < x\}$ *est fini ;*
- *aucun élément n'est en conflit avec lui-même.*

Un réseau d'occurrence est un graphe sans circuit et chaque noeud est précédé par un nombre fini de noeuds. Trois types de relations, mutuellement exclusives, sont définis entre deux n$\frac{1}{2}$uds quelconques de O :

- la relation de causalité notée $\prec : \forall x,y \in B \cup E$ avec $x \neq y$, $x \prec y$ *ssi* le réseau contient au moins un chemin allant de x à y. Plus généralement, $x \preceq y$ *ssi* $x \prec y \vee x = y$;
- la relation de conflit notée $\perp : \forall x,y \in B \cup E$ avec $x \neq y$, $x \perp y$ *ssi* le réseau contient deux chemins $pt_1...x$ et $pt_2...y$ qui commence par la même place p et telle que $x \neq y$ $(x \perp y \Leftrightarrow y \perp x)$;
- et la relation de concurrence notée $\wr : \forall x,y \in B \cup E$ avec $x \neq y$, $x \wr y$ *ssi* $\neg((x \prec y) \vee (y \prec x) \vee (x \perp y))$ $(x \wr y \Leftrightarrow y \wr x)$.

Les comportements possibles d'un réseau d'occurrence sont capturés par la notion de configuration. La *configuration* C d'un réseau d'occurrence est un ensemble d'événements satisfaisant les deux conditions suivantes :

— $e \in C \Rightarrow \forall e' \preceq e,\ e' \in C$

— $\forall e, e' \in C,\ \neg(e \perp e')$

On désigne par $Conf(O)$ l'ensemble des configurations du réseau d'occurrence O et par $Cut(C)$ le marquage atteint par le franchissement des événements de C à partir du marquage initial : $Cut(C) = Min(N) + C^\bullet - {}^\bullet C$. Dans la pratique on considère les préfixes de dépliages et la notion de *configuration* est remplacée par celle de *configuration locale*. La configuration locale d'un événement e est notée $[e] : \forall e \in E, [e] = \{y \in B \cup E \mid y \prec e\}$.

3.3 Dépliage et préfixe du dépliage

Dans [6], les processus de branchement (ou processus arborescent) regroupent un ensemble de processus lié par une relation de causalité. Ils permettent de définir la notion de *dépliage* qui n'est rien d'autre que le plus grand processus de branchement (en général infini) qu'on peut construire pour un RDP. Le réseau résultant du dépliage est un réseau d'occurrence, un autre réseau de Petri où les places sont appelées *conditions* (étiquetés par les places correspondantes dans le réseau initial) et les transitions sont appelées *les évènements* (étiquetés par les transitions correspondantes dans le réseau initial). Pour un réseau marqué $< N, M_0 >$ et pour un réseau d'occurrence $O =< B, E, F >$, nous construisons le dépliage réseau marqué avec la procédure suivante.

1. Initialiser B avec les conditions initiales. Créer b_i (i allant de 1 à $|M_0|$) pour chacun des $|M_0|$ jetons et l'ajouter à B ;
2. Pour tout $t \in T$, s'il existe un $B' \subseteq B$ tel que $(B' =^\bullet t \wedge (\nexists e' \in E, {}^\bullet e' = B' \wedge (e') = t))$ alors :
   (a) Créer et ajouter l'événement e (lié à la transition t) à E ;
   (b) Créer et ajouter à B toutes les conditions liées aux post-conditions de t.
3. Répéter l'item 2 tant qu'il existe un événement e_n possible.

Le processus de construction peut donc s'exécuter indéfiniment si $< N, M_0 >$ permet un ou plusieurs comportements infinis. Alors même si le dépliage contourne le problème lié à la construction du graphe d'état [10], il n'en reste pas moins que c'est une structure potentiellement infinie alors qu'il est peu souhaitable d'avoir à manipuler de telle structure. Plus formellement,

Définition 5 (Dépliage)**.** *Le réseau marqué $< N, M_0 >$ (avec $N =< P, T, W >$) admet $Unf =< O, \lambda >$ comme dépliage ssi :*

- *$O =< B, E, F >$ est un réseau d'occurrence ;*
- *λ est une fonction d'étiquetage telle que $\lambda : B \cup E \to P \cup T$. Elle vérifie les propriétés suivantes :*
  — *$\lambda(B) \subseteq P$, $\lambda(E) \subseteq T$;*
  — *$\lambda(Min(O)) = m_0$;*
  — *Pour tout $e \in E$, la restriction de λ à ${}^\bullet e$ (resp. $e^\bullet$) est une bijection entre ${}^\bullet e$ (resp. $e^\bullet$) et ${}^\bullet\lambda(e)$ (resp. $\lambda(e)^\bullet$). Nous avons ${}^\bullet\lambda(e) = \lambda({}^\bullet e)$ et $\lambda(e)^\bullet = \lambda(e^\bullet)$, ce qui signifie que λ préserve l'environnement des transitions.*

La solution pour conduire des vérifications serait de ne considérer qu'un préfixe fini du dépliage du réseau qui contiendrait toutes les informations pertinentes. Plusieurs travaux [11, 8, 9, 7] ont permis de construire un tel préfixe fini complet des réseau de Petri en identifiant les événements (cut-offs) à partir desquels le dépliage n'est plus considéré. L'algorithme 1 permet de construire un tel préfixe. Il est prouvé [9] qu'un préfixe fini complet du dépliage existe toujours pour un réseau marqué même ceux présentant des comportements infinis. En pratique, la terminaison du calcul est basée sur la notion de *configuration locale* notée $[x]$. $[x] = \{y \in B \cup E \mid y \prec e\}$. La notion d'événement cut-off permet d'arrêter la dérivation de noeuds à partir d'un événement e donné du dépliage. Un événement cut-off est un événement déjà arriver dans la construction du dépliage et il est inutile de le représenter à nouveau.

Algorithme 1 : Algorithme du préfix fini et complet du dépliage

Entrées : Réseau de Petri marqué $< P,T,W,M_0 >$ avec $M_0 = \{p_1, p_2, ..., p_k\}$.
Sorties : Préfixe fini et complet du dépliage $Unf =< B,E,F >$ de $< N,M_0 >$ et les événements cut-offs co.
début
  $Unf := ((p_1,\emptyset),(p_2,\emptyset),...,(p_k,\emptyset))$;
  $pe := PE(Unf)$;
  $co := \emptyset$;
  tant que $pe \neq \emptyset$ **faire**
    Choisir un événement $e = (t,X)$ dans pe tel que [e] soit minimale;
    $pe := pe \backslash \{e\}$;
    si *e est cut-off* **alors**
      $co := co \cup \{e\}$;
    sinon
      Ajouter à Unf l'événement e et ses post-conditions de la forme (p,e);
      $pe := PE(Unf)$;
    fin
  fin
fin

4 Dépliage des réseaux de Petri avec des reset arcs

En prenant en compte le fait que la définition du dépliage des réseaux de Petri est bien formalisée dans la littérature, une première intuition serait de trouver un réseau de Petri normal, dont on construira le dépliage, équivalent au réseau de Petri à reset arcs. Mais à priori, trouver ce réseau équivalent est impossible. Si un tel réseau équivalent existait, on devait être capable, par exemple, de vérifier les mêmes propriétés au niveau des deux réseaux comme la *boundedness* ou l'accessibilité. Les deux propriétés sont indécidables pour un réseau de Petri à reset arcs, mais décidables pour un réseau de Petri normal [5]. Les réseaux de Petri à reset arcs sont plus puissants (réseau de haut niveau) et plus expressifs que les réseaux de Petri normaux. Pour notre part, pour pouvoir les déplier, il faudra alors raisonner à un haut niveau et nous le décrivons dans la Section 4.2. De ce point de vue, il est claire que la notion de réseau d'occurrence utilisée dans la Section 3.2 pour définir le dépliage des réseaux de Petri normaux ne nous permettra plus de définir celui avec des resets arcs. Nous introduisons dans ce papier la notion de R-réseau d'occurrence (R comme reset).

4.1 R-réseau d'occurrence

Un R-réseau d'occurrence est un réseau d'occurrence étendue aux reset arcs. C'est un réseau d'occurrence auquel on ajoute l'ensemble F_R formés de reset arcs. Plus particulièrement, un réseau d'occurrence est alors un R-réseau d'occurrence dont l'ensemble F_R est vide.

Définition 6 (R-réseau d'occurrence)**.** *Un R-réseau d'occurrence est un réseau $O_R =< O,F_R >$ tel que :*

- *$O =< B,E,F >$ est un réseau d'occurrence ;*
- *$F_R : B \times E \rightarrow \{0,1\}$ est un ensemble formé des reset arcs ($F_R(b,e) = 0$ s'il y a un reset arc qui relie b à e, sinon $F_R(b,e) = 1$).*

4.2 Dépliage des réseaux de Petri à reset arcs

La définition du dépliage d'un réseau de Petri avec des reset arcs est similaire à cette d'un réseau de Petri mais en remplaçant le réseau d'occurrence par le R-réseau d'occurrence. En nous basant sur le lemme 1, nous pouvons alors construire le dépliage d'un réseau de Petri à reset arcs de la manière suivante (Algorithme 2) :

Algorithme 2 : Algorithme du préfix fini et complet du dépliage d'un réseau de Petri avec reset arc

Entrées : Réseau de Petri avec reset arcs marqué $< P,T,W,R,M_0 >$ avec $M_0 = \{p_1, p_2, ..., p_k\}$.
début
1. Unf_R := Dépliage de $< N_R, M_0 >$ en ignorant les reset arcs;
2. $\forall (b,e) \in B \times E$, si $W(\lambda(b),\lambda(e)) = 0 \cap (b \notin [e] \cup e \notin [b])$ alors créer le reset arc (b,e) et l'ajouter à F_R i.e $F_R(b,e) = 0$;

fin

Plus précisément, le dépliage d'un réseau de Petri avec reset arcs ($N_R =< P,T,W,R >$) est un R-réseau d'occurrence ($O_R =< B,E,F,F_R >$) obtenu en dépliant le réseau de Petri avec reset arcs en ignorant les reset arcs (Algorithme 1). Puis pour toute condition b de B et pour tout événement e de E, si $R(\lambda(b),\lambda(e)) = 0$ alors $F_R(b,e) = 0$. Autrement dit, pour chaque place p et pour chaque transition t, relier par un reset arc, chaque copie de p et chaque copie de t dans le dépliage obtenu en ignorant les reset arcs. Mais si $e \in [b]$ ou $b \in [e]$, il n'est pas nécessaire de relier b et e par un reset arc (definition du dépliage).

Preuve. Le lemme 1 montre que les reset arcs n'enlèvent que des comportements au réseau de Petri. A l'item 1, on construit le dépliage du réseau de Petri sous-jacent. On a ainsi la garantie que tous les comportements du réseau de Petri à reset arcs y sont représentés. Avec l'item 2, on remet judicieusement les reset arcs pour enlever les comportement indésirables. On obtient bien avec l'algorithme2 le dépliage du réseau de Petri à reset arcs. Il en est de même pour le préfixe fini et complet du dépliage. On considère dans ce cas le préfixe fini et complet du réseau de Petri normal sous-jacent). □

La Figure 4.2 montre le dépliage du réseau de Petri à reset arcs de la Figure 2(b)

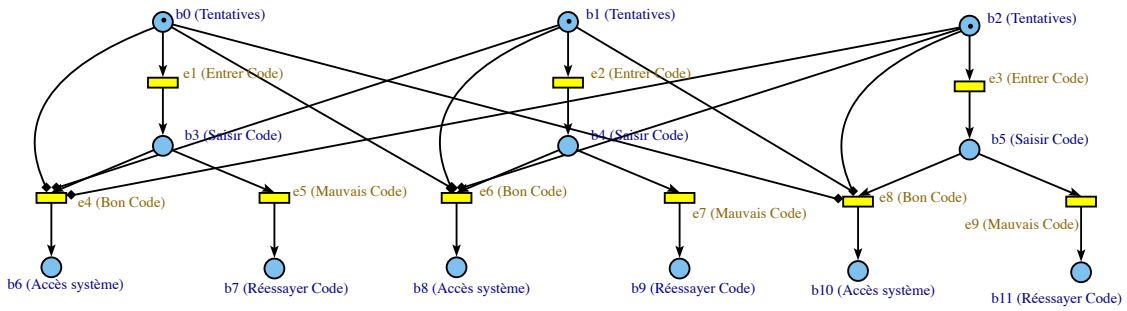


FIGURE 3 – Préfixe fini et complet du dépliage de réseau de la Figure 2(b)

4.3 Limites de l'approche

La première limite est liée au fait qu'on ne considère que les réseaux de Petri à reset arcs ayant des réseaux de Petri normaux sous-jacents bornés. Les reset arcs sont utilisés parfois pour borner les réseaux de Petri (Figure 4(a)) et *a priori* il n'y a pas de raison pour ne pas construire le dépliage mais en passant d'abord par le réseau normal sous-jacent, on ne peut plus construire ce le dépliage. La raison est que le réseau de Petri normal sous-jacent n'est pas borné.

La deuxième 'limite' est liée au fait qu'on ne peut pas garantir d'avoir le plus petit motif du dépliage. Prenons l'exemple du réseau de la Figure 4(b), en considérons le réseau de Petri normal sous-jacent, on commence la construction du dépliage par la représentation de trois conditions liées à $P1$ et donc la possibilité de produire trois événements liés $T1$. Mais seulement le comportement réel du réseau de Petri avec le reset arc, l'événement lié à

$T1$ ne peut qu'être réalisé qu'une seule fois. Le fait de remettre les reset arcs dans le dépliage obtenu permet de s'assurer de ces contraintes. Le motif obtenu n'est pas forcément pas le plus petit mais représente parfaitement le comportement du réseau. La deuxième limite n'en est vraiment pas une.

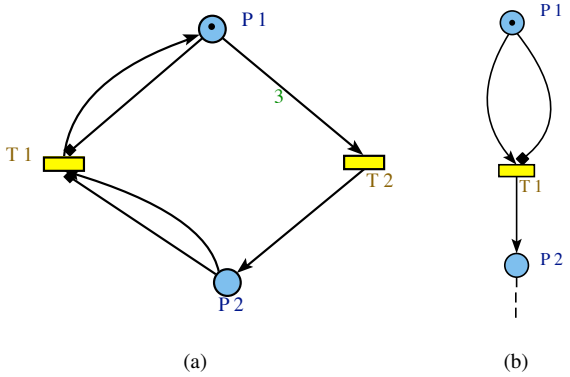


FIGURE 4 – Exemples de limites

5 Calcul des processus de branchement

De part sa structure arborescente, le dépliage permet de vite calculer les processus (appelés processus de branchement) d'un réseau de Petri. Pour le dépliage d'un réseau de Petri normal, trois relations existent entre ses événements (causalité, concurrence et conflit). Pour un réseau de Petri normal, un processus de branchement est un ensemble d'événements liés par deux relations (causalité et concurrence), pas de conflit. Mais le dépliage d'un réseau de Petri à rest arcs, on observe un autre comportement, que l'on matérialisera par la définition d'une nouvelle relation (relation de reset) qui pourrait empêcher deux événements, même n'étant pas en conflit, de ne pas appartenir à un même processus. C'est le cas pas exemple, dans l'exemple de la Figure 4.2 des événements $e4$ et $e2$. La présente de $e4$ pourrait inhiber l'événement $e2$.

Définition 7 (Relation de reset)**.** *e_1 et e_2 sont dans une relation de reset ($e_1 \top e_2$) ssi $\exists\, b \in {}^\bullet e_2$ tel que $(b, e_1) \in F_R$. De façon générale, $e_1 \top e_2$ ssi $\exists\, e \in E$ tel que $((e \prec e_2) \wedge (e_1 \top e))$. $\top$ n'est pas commutative ; Si $(e_2 \prec e_3)$ et $(e_1 \top e_2)$ alors $(e_1 \top e_3)$.*

Nous pouvons alors formaliser la notion de processus de branchement pour un réseau de Petri à reset arcs comme étant un ensemble d'événements en relations de causalité et de concurrence et dans lequel il ne subsiste aucun conflit et pour lequel deux événements ne sont en relation de reset. Soit C_i un ensemble formé par les conditions et les événements du dépliage et construit comme suit :

1. $Min(O) \subseteq C_i$
2. Ajouter un $e_i \in E$ à C_i ssi :
   — $e_i \notin C_i$;
   — $\exists b \in C_i$ *tel que* $e_i \in b^\bullet$;
   — $\forall e_j \in E, \neg(e_j \perp e_i) \vee \neg(e_j \top e_i)$.
3. Si e_i est ajouté à C_i alors $C_i = C_i \cup e_i^\bullet$
4. Répétez l'étape 2 autant de fois.

L'ensemble des événements $E_i \subseteq E$ de C_i définit un processus de branchement. Pour l'exemple de la Figure 4.2, nous pouvons identifier les processus suivants :
— $P_1 = \{e_1, e_4\}$
— $P_2 = \{e_1, e_5, e_4, e_2, e_6\}$

— $P_3 = \{e_1, e_5, e_4, e_2, e_7, e_3, e_8\}$
— $P_4 = \{e_1, e_5, e_4, e_2, e_7, e_3, e_9\}$

6 Conclusion et perspectives

Dans ce travail, nous avons présenté une approche pour construire le dépliage des réseaux de Petri à reset arcs et un algorithme pour construire son préfixe complet et fini. Cet algorithme permet d'étendre alors la notion de dépliage à une extension particulière des réseaux de Petri, les réseaux de Petri à reset arcs. L'intérêt de cette approche est de pouvoir construire le dépliage des réseaux de Petri avec des reset arcs sans avoir à trouver une quelconque équivalence avec un réseau de Petri.

Nos travaux visent à définir une algèbre des processus de branchement [3, 4], les processus issus du dépliage des réseaux de Petri. La perspective de ce travail est de pouvoir exprimer le dépliage des réseaux de Petri à reset arcs sous forme algébrique, d'en déduire les processus (voire les processus maximaux) et de trouver des équivalences de conflits.

Références

[1] Bernard Berthomieu and Francois Vernadat. State class constructions for branching analysis of time petri nets. In *In TACAS ?2003, volume 2619 of LNCS*, pages 442–457. Springer Verlag, 2003.

[2] Thomas Chatain and Claude Jard. Complete finite prefixes of symbolic unfoldings of safe time petri nets. In Susanna Donatelli and P.S. Thiagarajan, editors, *Petri Nets and Other Models of Concurrency - ICATPN 2006*, volume 4024 of *Lecture Notes in Computer Science*, pages 125–145. Springer Berlin Heidelberg, 2006.

[3] Delfieu D., M. Comlan, and Sogbohossou. Algebraic analysis of branching processes. In *Sixth International Conference on Advances in System Testing and Validation Lifecycle*, pages 21–27, 2014. Best paper award.

[4] Delfieu D. and M. Sogbohossou. An algebra for branching processes. In *(CoDIT, 2013 International Conference on*, pages 625–634, May 2013.

[5] Catherine Dufourd, Petr Jančar, and Philippe Schnoebelen. In Jirí Wiedermann, Peter van Emde Boas, and Mogens Nielsen, editors, *Proceedings of the 26th ICALP'99*, volume 1644 of *Lecture Notes in Computer Science*, pages 301–310, Prague, Czech Republic, July 1999. Springer.

[6] Joost Engelfriet. Branching processes of petri nets. *Acta Informatica*, 28(6) :575–591, 1991.

[7] Javier Esparza and Keijo Heljanko. Unfoldings - a partial-order approach to model checking. *EATCS Monographs in Theoretical Computer Science*, 2008.

[8] Javier Esparza, Stefan Römer, and Walter Vogler. *An Improvement of McMillan's Unfolding Algorithm*. Mit Press, 1996.

[9] Javier Esparza, Stefan Römer, and Walter Vogler. An improvement of mcmillan's unfolding algorithm. *Formal Methods in System Design*, 20(3) :285–310, 2002.

[10] Kenneth L McMillan. Using unfoldings to avoid the state explosion problem in the verification of asynchronous circuits. In *Computer Aided Verification*, pages 164–177. Springer, 1993.

[11] K.L. McMillan and D.K. Probst. A technique of state space search based on unfolding. *Formal Methods in System Design*, 6(1) :45–65, 1995.

[12] Diaz Michel. *Vérification et mise en oeuvre des réseaux de Petri*. Traité IC2. Hermès Science, Paris, 2003.

[13] Carl Adam Petri. Communication with automata. 1962.

Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures et Langages

Towards a continuum for COMPOnent-based development

Christophe Dony[1], Petr Spacek[2], Chouki Tibermacine[1], Frédéric Verdier[3], Antony Ferrand[3]

[1] LIRMM - CNRS and Montpellier University - France - dony,tibermacin@lirmm.fr

[2] Faculty of Information Technology - Czech Technical University in Prague - Czech Republic - spacepe2@fit.cvut.cz

[3] Montpellier-University - FDS - Master AIGLE

1 Context

Component-based software engineering studies the production of reusable components and their combination into connection architectures. It appears that component-orientation has been more studied from the design stage point of view, with modeling languages and ADLs [7,3], or from the framework and middleware points of view (with e.g. [8]) or from the deployment and packaging point of view (with e.g. OSGI bundles) than from the component implementation stage. As stated in [3] "most component models use standard programming languages ... for the implementation stage"; and most of today's solutions [5] for the implementation stage use object-oriented languages. Such a choice is natural because object gather a signigicant part of what makes component essence : composition, encapsulation, provisions, generic service invocation and have practical advantages related to the availability and maturity of object-oriented programming languages, tools and practices.

But this state of fact raises the issues that most of languages and technologies used to achieve component-based development are not uniform and are only partially component-based, i.e. that there is no conceptual continuum between component models and their implementations (or at least it is incomplete). Many design concepts (descriptors, requirements, ports or connections, architectures, components themselves) vanish (or are duplicated in another form) in the implementation. This makes tasks such as debugging, reverse-engineering, models transformations after code generation or runtime meta-modelling, more complex.

Component-based programming languages (CBPLs) [1] have been inspired by this need for a continuum between components models or architectures descriptions and their implementations. But primary CBPLs did not solve the uniformity issue; they did not eliminate the object/component redundancy in which component can be connected and objects can be passed as arguments, in which it is not clear to understand how to choose whether an entity should be represented by an object or by a component.

2 Compo, Everything is a Component

The primary `Scl` [4,6] project has been held to imagine such a continuum, founded on the "everything is a component" idea, making it possible to de-

fine, in a unified component-based context defined by a unique meta-model, and possibly (but not mandatorily) using the same language, as well components, their connection architectures and their implementations.

The following and current `Compo` project goes further by extending the continuum principle to encompass analysis, transformations and model-driven activities. The resulting language Compo [9,10] is a pure reflective modeling (architecture description) and programming component-based language. Compo's novelties lie in an integrated solution in which:

- Everything is a component. Compo's components subsume objects; they are objects that explicitly expose their requirements and internal (in the object's encapsulation meaning of the term) architecture. Elements of primitive date types are boxed as components when needed and can be used as such. Components can be passed as arguments via temporary connections.
- Compo's meta-model requires an inheritance system, which has been reintroduced at the component descriptor level [11], together with concrete solutions to the questions of knowing what are requirement and architecture specializations.
- Models, architectures and their implementations can be defined at the same conceptual level using the same language [5,12],
- "Models@runtime" is achieved via a component-based executable reflective architecture [10] allowing for any kind of static or runtime model or program verification or transformation. Architectural constraints can for example be checked either statically or at run-time [13].
- The entire meta-model (including component descriptors, ports and services) is self-described [10].

3 Prospective

A prototype version of Compo has been implemented in Pharo () by Petr Spacek, complemented by Frédéric Verdier and Antony Ferrand; it can be downloaded at . In the above quoted projects, the focus has been put more particularly on the questions of finding what could be a "pure component" language and of how to make it reflective. Meanwhile, we have been able to identify many interesting new issues among which we can note the following.

1. The architecture description language part of `Compo` is poor and should be enriched in various ways with particular references to recent ADLs, for example make it possible to represent how architectures can dynamically evolve during program execution (known as dynamic architectures) as in SOFA2.
2. The requirements and provisions description language part of `Compo` to describe required ports is also very limited, it should be enriched in various ways with particular references to various existing results in the context of requirement engineering, for example by reference to contract aware components [2].
3. Establishing a formal semantics of Compo is a work in progress that will include a description of the two protocols (passing by requirement or passing by provision) implemented for passing a component to an an operation.

Acknowledgments. The authors wish to thank David Delahaye, Roland Ducournau, and Marianne Huchard for many fruitful discussions.

References

1. J. Aldrich, C. Chambers, and D. Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 187–197, New York, NY, USA, 2002. ACM.
2. A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Making components contract aware. *IEEE Computer*, pages 38–45, 1999.
3. I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593 –615, sept.-oct. 2011.
4. L. Fabresse. *From decoupling to unanticipated assembly of components: design and implementation of the component-oriented language Scl.* PhD thesis, Montpellier II University, Montpellier, France, December 2007.
5. L. Fabresse, N. Bouraqadi, C. Dony, and M. Huchard. A language to bridge the gap between component-based design and implementation. *COMLAN : Journal on Computer Languages, Systems and Structures*, 38(1):29–43, Apr. 2012.
6. L. Fabresse, C. Dony, and M. Huchard. Foundations of a simple and unified component-oriented language. *Comput. Lang. Syst. Struct.*, 34(2-3):130–149, July 2008.
7. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, Jan. 2000.
8. L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A component-based middleware platform for reconfigurable service-oriented architectures. *Softw. Pract. Exper.*, 42(5):559–583, May 2012.
9. P. Spacek. *Design and Implementation of a Reflective Component-Oriented Programming and Modeling Language.* PhD thesis, Montpellier II University, Montpellier, France, December 2013.
10. P. Spacek, C. Dony, and C. Tibermacine. A component-based meta-level architecture and prototypical implementation of a reflective component-based programming and modeling language. In *Procs. of ACM CBSE'14 - The 17th Int. ACM SIGSOFT Conf. on Component Based Software Engineering*, pages 13–23, July 2014. CBSE'2014 Distinguished Paper Award.
11. P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. An inheritance system for structural & behavioral reuse in component-based software programming. In *Proceedings of the 11th GPCE*, pages 60–69. ACM, 2012.
12. P. Spacek, C. Dony, C. Tibermacine, and L. Fabresse. Wringing out objects for programming and modeling component-based systems. In *procs. of the 2nd Int. Workshop on Combined Object-Oriented Modeling and Programming Languages (COOMPL'13) - co-located with ECOOP*. ACM Digital Library, July 2013.
13. C. Tibermacine, S. Sadou, C. Dony, and L. Fabresse. Component-based specification of software architecture constraints. In *Procs. of ACM CBSE'11 - The 14th Int. ACM SIGSOFT Conf. on Component Based Software Engineering*, pages 31–40, June 2011. CBSE'2011 Distinguished Paper Award.

Reasoning at Runtime using time-distorted Contexts: A Models@run.time based Approach

Thomas Hartmann*, Francois Fouquet*, Gregory Nain*, Brice Morin‡, Jacques Klein* and Yves Le Traon*
*Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, first.last@uni.lu
‡SINTEF ICT Norway, Norway, first.last@sintef.no

***Abstract*—Intelligent systems continuously analyze their context to autonomously take corrective actions. Building a proper knowledge representation of the context is the key to take adequate actions. This requires numerous and complex data models, for example formalized as ontologies or meta-models. As these systems evolve in a dynamic context, reasoning processes typically need to analyze and compare the current context with its history. A common approach consists in a temporal discretization, which regularly samples the context (snapshots) at specific timestamps to keep track of the history. Reasoning processes would then need to mine a huge amount of data, extract a relevant view, and finally analyze it. This would require lots of computational power and be time-consuming, conflicting with the near real-time response time requirements of intelligent systems. This paper introduces a novel temporal modeling approach together with a time-relative navigation between context concepts to overcome this limitation. Similarly to time distortion theory, our approach enables building time-distorted views of a context, composed by elements coming from different times, which speeds up the reasoning. We demonstrate the efficiency of our approach with a smart grid load prediction reasoning engine.**

***Keywords*—Temporal data, Time-aware context modeling, Knowledge representation, Reactive systems, Intelligent systems**

I. INTRODUCTION

An intelligent system needs to analyze both its surrounding environment and its internal state, which together we refer to as the *context* of a system, in order to continuously adapt itself to varying conditions. Therefore, building an appropriate context representation, which reflects the current context of a system is of key importance. This task is not trivial [1] and different approaches and languages are currently used to build such context representations, *e.g.* ontologies [2] or DSLs [3]. Most approaches describe a context using a set of concepts (also called classes or elements), attributes (or properties), and the relations between them. Recently, in the domain of model-driven engineering, the paradigm of models@run.time [4], [5] has rapidly proved its suitability to safely represent, reason about and dynamically adapt a running software system. Nevertheless, context representations (or models), as abstractions of real system states and environments, are only able to reflect a snapshot of a real system at a specific timestamp. However, context data of intelligent systems rapidly change and evolve over time (at different paces for each element) and reasoning processes not only need to analyze the current snapshot of their contexts but also historical data.

The research leading to this publication is supported by the National Research Fund Luxembourg (grant 6816126) and Creos Luxembourg S.A. under the SnT-Creos partnership program.

Let us take a smart grid as an example. Due to changes in the production/consumption chain over time, or to the sporadic availability of natural resources (heavy rain or wind), the properties of the smart grid must be continuously monitored and adapted to regulate the electric load in order to positively impact costs and/or eco-friendliness. For instance, predicting the electric load for a particular region requires a good understanding of the past electricity production and consumption in this region, as well as other data coming from the current context (such as current and forecast weather). Since the electrical grid cannot maintain an overload for more than a few seconds or minutes [6], it is important that protection mechanisms work in this time range. This is what we call *near real-time*.

It is a common approach for such systems to regularly sample and store the context of a system at a very high rate in order to provide reasoning algorithms with historical data. Fig. 1 shows a context —represented as a graph (inspired by object graphs)— sampled at three different timestamps, t_i, t_{i+1}, and t_{i+2}. Each graph in the figure represents the context at a given point in time, where all context variables, independently from their actual values, belong to the same time. Therefore, each graph lies in a horizontal plane (in time).

This systematic, regular context sampling, however, yields to a vast amount of data and redundancy, which is very difficult to analyze and process efficiently. Moreover, it is usually not sufficient to consider and reason just with data from one timestamp, *e.g.* t_i or t_{i+1}. Instead, for many reasoning processes, *e.g.* to investigate a potential causality between two phenomena, it is necessary to simultaneously consider and correlate data from different timestamps (*i.e.* t_i and t_{i+1}). Reasoning processes therefore need to mine a huge amount of data, extract a relevant view (containing context elements from different snapshots), and analyze this view. This overall process would require some heavy resources and/or be time-consuming, conflicting with the near real-time response time requirements such systems usually need to meet.

Going back to the smart grid reasoning engine example: In order to predict the electric load for a region, a linear regression of the average electric load values of the meters in this region, over a certain period of time, has to be computed. Therefore, reasoning processes would need to mine all context snapshots in this time period, extract the relevant meters and electric load values, and then compute a value.

To address these issues, we propose to make context models aware of time *i.e.* to allow context elements (data) from different timestamps in the same model. We refer to such contexts as *time-distorted* contexts. Fig. 2 shows such a context

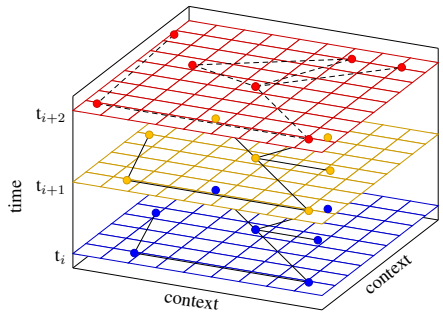


Fig. 1. Linear sampled context

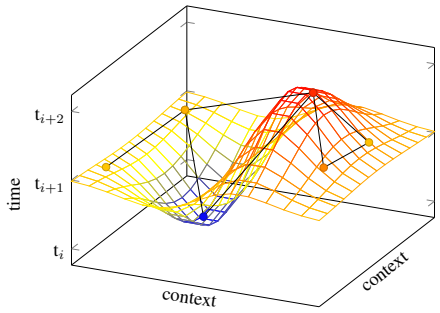


Fig. 2. Time-distorted context

representation, again represented as a graph. Here, the context variables —again independently from their actual values— belong to different timestamps. Such a context can no longer be represented as a graph lying entirely in one horizontal plane (in time). Instead, graphs representing time-distorted contexts lie in a curved plane. They can be considered as specialized *views*, dedicated for a specific reasoning task, composing navigable contexts to reach elements from different times. In contrast to the usage of the term view in database communities we do not aggregate data but offer a way to efficiently traverse specific time windows of a context.
Physics, and especially the study of laser [7], relies on a time distortion [8] property, specifying that the current time is different depending on the point of observation. Applied to our case, this means that context elements can have different values depending on the origin of the navigation context, *i.e.* depending on the timestamp of the inquiring actor. **We claim that time-distorted context representations can efficiently empower continuous reasoning processes and can outperform traditional full sampling approaches by far. The contribution of this paper is to consider temporal information as a first-class property crosscutting any context element, allowing to organize context representations as time-distorted views dedicated for reasoning processes, rather than a mere stack of snapshots. We argue that this approach enables many reasoning processes to react in near real-time (the range of milliseconds to seconds).**

The remainder of this paper is as follows. Section II introduces the background of this work. Section III describes the concepts of our approach and section IV the details on how we implement and integrate these into the open source modeling framework KMF. The provided API is presented in section V. We evaluate our general approach in section VI on a concrete smart grid reasoning engine for electric load prediction. After a discussion about the approach and related work in section VII the conclusion of the paper is presented in section VIII.

II. Background

Over time different languages, formalisms, and concepts to build and represent the context of intelligent systems have been developed and used [1], [9], [10] for different purposes. Entity-relationship models [11], as a general modeling concept for describing entities and the relationships between them, are widely used for building context representations. Ontologies, RDF [12], and OWL [13] are particularly used in the domain of the Semantic Web. These allow to describe facts in a *subject-predicate-object* manner and provide means to reason about these facts. Over the past few years, an emerging paradigm called *models@run.time* [4], [5] proposes to use models both at design and runtime in order to support intelligent systems. At design time, following the model-driven engineering (MDE) paradigm [14], models support the design and implementation of the system. The same (or similar) models are then embedded at runtime in order to support the reasoning processes of intelligent systems, as models offer a *simpler, safer and cheaper* [15] means to reason. Most of these approaches (RDF, OWL, models) have in common that they describe a context using a set of concepts (also called: classes, types, elements), attributes (or properties), and the relations between them. We refer to the representation of a context (set of described elements) as a *context model* or simply as *model* and to a single element (concept) as *model element* or simply as *element*. The concepts of our approach are, in principle, independent of a concrete context representation strategy. However, the implementation of our approach and the provided API are build on a models@run.time based context representation approach and are integrated into an open source modeling framework, called Kevoree Modeling Framework [16] (KMF [1]). KMF is the modeling pillar supporting the Kevoree models@run.time platform [17]. We decided to leverage a models@run.time based approach for several reasons: First of all, models provide a semantically rich way to define a context. Second, models can be used to formally define reasoning activities. Last but not least, the models@run.time paradigm has been proved to be suitable to represent this context during runtime [4], [5].

KMF is an alternative to EMF [18] and specifically designed to support the models@run.time paradigm in terms of memory usage and runtime performance. Two properties of KMF are particularly important for the implementation of our approach: First, in KMF, each model element can be accessed within the model by a path (from the root element of a model to a specific element following containment [19] references), which defines the semantic to efficiently navigate in the model. Our contribution extends the path of model elements with temporal data in order to provide seamless navigation, not only in the model but as well in time. Second, in KMF each model element can be serialized independently, using paths to represent elements of its relationships. We use this property in our implementation to incrementally store model element modifications.

[1] http://kevoree.org/kmf

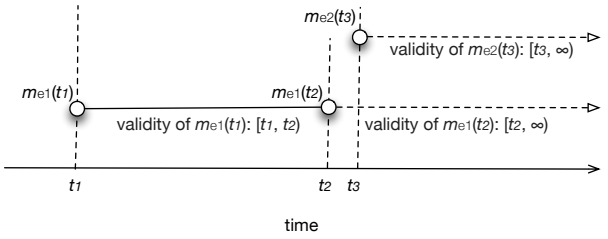


Fig. 3. Continous validity of model elements

III. ENABLING TIME-DISTORTED CONTEXTS

Our hypothesis is that temporal knowledge is part of a domain itself (*e.g.* electric load or wave propagation prediction, medical recommender systems, financial applications) and that defining and navigating temporal data directly within domain contexts is far more efficient and convenient than regularly sampling a context and independently querying each model element with the appropriate time. Therefore, we provide a concept to define and navigate into the time dimension of contexts. Most importantly, we enable a context representation to seamlessly combine elements from different points in time, forming a *time-distorted* context, which is especially useful for time related reasoning. We claim that our approach, which weaves time directly into the context model, as opposed to a full sampling and classic data mining approach, is compatible with near real-time requirements. In the following we present the concepts to enable time-distorted context models.

A. Temporal Validity for Model Elements

Instead of relying on context snapshots, we define a context as a continuous structure. Nevertheless, each context (model) element of this structure can evolve independently. We first define an implicit *validity* for model elements. Therefore, we associate a timestamp to each model element. They reflect a domain-dependent notion of the time at which a model element was captured and can be accessed on the element itself. In other words, a timestamp defines a *version* $v_{m_e}(t)$ of a model element m_e at a time t. If a model element now evolves, an additional version of the same element is created and associated to a new timestamp (the domain time at which the new version is captured). Timestamps can be compared and thus form a chronological sequence. Therefore, although timestamps are discrete values, they logically define intervals in which a model element can be considered as *valid* with regards to a timestamp. A model element is valid from the moment it is captured until a new version is captured. New versions are only created if necessary, *i.e.* if the model element changed. Fig. 3 shows two model elements, m_{e1} with two versions and m_{e2} with one version, and their corresponding validity periods. As represented in the figure, version $m_{e1}(t_1)$ is valid in interval $[t_1, t_2[$. Since there is no version of model element m_{e1}, which is captured later than t_2, the validity of version $m_{e1}(t_2)$ is the open interval $[t_2, +\infty[$. Accordingly, version $m_{e2}(t_3)$ is valid in $[t_3, +\infty[$. Since model elements now have a temporal validity, a relationship r from a model element m_{e1} to m_{e2} is no longer uniquely defined. Instead, the timestamps of model elements have to be taken into account for the resolution of relationships (described in III-C). The association of each model element with a timestamp and thus a validity, provides the foundation for a continuous context representation. Although model elements are still sampled at discrete timestamps, the definition of a continuous validity for each model element allows to represent a context as a continuous structure. This structure provides the foundation for our time-distorted context models.

B. Navigating through Time

Based on the idea that it is necessary for intelligent systems to consider not only the current context but also historical data to correlate or investigate potential causalities between two phenomena, we provide means to enable an efficient navigation into time. Therefore, we define three basic operations for model elements. These can be called on each model element: The *shift* operation is the main possibility to navigate a model element through time. It takes a timestamp as parameter, looks for the model version of itself, which is valid at the required timestamp, loads the corresponding element from storage (can be RAM) and returns the loaded version.
The *previous* operation is a shortcut to retrieve the direct predecessor (in terms of time) of the current model element.
The *next* method is similar to the *previous* operation but retrieves the direct successor of the current model element.
These operations allow us to shift model elements independently from each other through time. This makes it possible to create context models, combining model elements from different points in time. Now only the last step is missing to have time-distorted context models for efficient runtime reasoning processes: A concept to take the time of model elements into account when navigating between model elements.

C. Time-relative Navigation

Navigating temporal data is a complex task, since a relationship r from an element m_{e1} to an element m_{e2} is no longer uniquely defined. Instead —depending on the timestamps t_1 and t_2 of m_{e1} and m_{e2}, and depending on the set of versions of m_{e1} and m_{e2}— a relationship r from m_{e1} to m_{e2} can link different versions of m_{e2}. This means which version of m_{e2} is related to m_{e1} by r depends on the timestamp t of m_{e1}. Processing this time-relative navigation manually, *e.g.* to correlate or investigate potential causalities between two phenomena, is complicated and error-prone. We therefore provide a concept to automatically resolve relationships, taking the time aspect into account, while navigating the context model. This time related resolution is completely transparent and hidden behind methods to navigate in the context model. Hereby, a context time can be defined (the curve in fig. 2) and each model element is then resolved accordingly to this definition while traversing the model. For example, the context time can be defined as the current time of a model element minus one day. When navigating from model element m_{e1} at timestamp t_i to element m_{e2}, the version of m_{e2}, which is valid at timestamp $t_i - 1$ *day* is resolved. In case that at timestamp t_i, object m_{e2} does not exist, the *prior existing version of* m_{e2} is returned. Considering model elements in the context of a specific time interval creates a navigable time dimension for model elements. This time relative data resolution is one of the novel concepts of this contribution. Unlike in previous approaches (*e.g.* relationships in MOF [19] or predicates in RDF [12]), the navigation function is not constant but yields

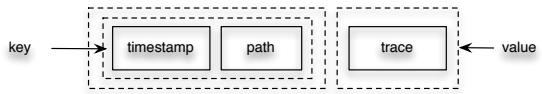


Fig. 4. Key/value structure for time-relative storage

different results depending on the navigation context (*i.e.* the current observation date). This distortion in terms of navigable relations finally enables what we call a time-distorted context.

IV. Integration into KMF

In this section we describe how we integrate our time-distorted modeling approach into the open source modeling framework KMF. We rely on two properties to integrate the time dimension as a crosscutting concern into model elements: *i*) each model element must be uniquely identifiable, and *ii*) it must be possible to get a serialized representation of all attributes and relationships of each model element, with no relativity to a time. To ensure the first property, KMF defines a *path* for each model element, starting from the root element of a model to the element following the containment relationships, as a unique identifier. Since the containment graph is actually a tree (each element, except the root, has to be contained exactly once), the path is a unique full qualified identifier. For the second property, KMF serializes models and model elements into *traces*. A trace defines a sequence of atomic actions to construct a model element, using the path concept (including relationship information). Each model element can be transformed into a trace and vice versa [20], [21]. As stated in III-A we inject a timestamp into all model elements. We do that by extending the KMF generator to automatically generate a timestamp attribute for all model elements. As a consequence, a model element, or more precisely a version of a model element, is now no longer simply defined by its path alone, but by a combination of its path and timestamp. Using the path together with a timestamp as key and the trace as value allows us to store and retrieve model elements within their time dimension in a simple *key/value* manner. The resolution (and storage) of a model element is therefore always relative to the context time. This is shown in fig. 4.

This context data organization allows us to use technologies eligible for big data to efficiently store and load context data. The data can be stored using different back ends, *e.g.* key/value stores, relational databases, or simply in memory (as a cache). In our implementation we use Google LevelDB[2] since it has proven to be suitable for handling big context data and, most importantly, it is very fast for our purpose (see VI). The storage implementation itself, however, is not part of our contribution. We intend to provide an efficient layer for runtime reasoning processes, on top of already existing storage technologies.

Since data captured in context models usually evolve at a very high pace (*e.g.* milliseconds or seconds), and our approach foresees to not only store the current version of model elements but also historical ones, context models can quickly become very large. In such cases, context models may no longer fit completely into memory, or at least it is no longer practical to do so. Therefore, based on our storage concept and the uniqueness of KMF paths in a model, we implement a *lazy loading*[3] mechanism to enable efficient loading of big context models. We use *proxies*[3], containing only path and timestamp, to reduce the overall memory usage. Attributes and referenced elements are only loaded when they are read or written. To enable this we extend KMF so that, while the context model is traversed, relationships are dynamically resolved. First, it must be determined which version of a related model element must be retrieved. This depends on the timestamp of the model element version (and the context time) from which the navigation starts (discussed in III-C). Second, the actual model element version must be loaded from storage. Our implementation allows to manage context models of arbitrary sizes efficiently and it hides the complexity of resolving —and navigating— temporal data behind an API.

V. API

Modeling approaches use meta-model definitions (*i.e.* concept descriptions) to generate domain specific APIs. The following section illustrates our API on a simplified smart grid meta-model definition. The model consists of a smart meter with an attribute for electric load and a relationship to reachable surrounding meters. In addition to a classical modeling API, our time-distorted context extension provides functions for creating, deleting, storing, loading, and shifting versions of model elements. Applications can use this API to create new model elements, specify their timestamps, store them, change their attributes and relationships, and store new versions (with different timestamps). In addition, the API can be used to specify the *context time* on which elements should be resolved while traversing the model. One can imagine the definition of the context time as the curve shown in fig. 2. Listing 1 shows Java code that uses a *Context ctx* (abstraction to manipulate model elements) to perform these actions.

```
// creating and manipulating model elements
m1 = ctx.createSmartMeter("m1","2014/3/1");
m1.setElectricLoad(125.6);
m1.addReachables(ctx.load("m2"));
m1_2 = m1.shift("2014/3/2");
m1_2.setElectricLoad(193.7);
// definition of the context time
ctx.timeDef("m1","2014/3/1");
ctx.timeDef("m2","2014/3/2");
r_m1 = ctx.load("m1");
assert(r_m1.getElectricLoad()==125.6);
r_m2 = r_m1.getReachables().get(0);
assert(r_m2.getTime()=="2014/3/2")
```

Listing 1. Usage of the time-distorted modeling API

The API provides a seamless way to create, manipulate, and navigate in time-distorted context representations.

VI. Evaluation

To quantify the advantages of our approach, we evaluate it on a smart grid reasoning engine to predict the electric load in a region based on current load and historical data. This problem is taken from our cooperation with Creos Luxembourg S.A. and led to the research behind this approach.

[2]https://code.google.com/p/leveldb/

[3]http://wiki.eclipse.org/CDO

TABLE I. BENCHMARK USING FULL SAMPLING

Scenario	Reasoning	Insert
Small Deep Prediction (SDP)	1075.6 ms	267 s
Small Wide Prediction (SWP)	1088.4 ms	
Large Deep Prediction (LDP)	180109.0 ms	
Large Wide Prediction (LWP)	181596.1 ms	

TABLE II. BENCHMARK USING TIME-DISTORTED CONTEXTS

Scenario	Reasoning	Insert
Small Deep Prediction (SDP)	1.8 ms	16 s
Small Wide Prediction (SWP)	0.8 ms	
Large Deep Prediction (LDP)	187.0 ms	
Large Wide Prediction (LWP)	157.6 ms	

Smart grids are infrastructures characterized by the introduction of reactive entities modernizing the electricity distribution grid. Smart meters, entities installed at customer sites to continuously measure consumption data and propagate it through network communication links, are one of the main building blocks of smart grids. Based on the electrical consumption smart meters can determine the electrical load. The load is regularly sampled, which leads to big context models. The idea for this reasoning engine is to predict if the load in a certain region will likely exceed or surpass a critical value. Our experimental validation focuses on two key indicators: **(1)** performance of the reasoning process and **(2)** insertion time.

Experimental results: We implemented the reasoning engine case study leveraging our approach on top of the KMF framework, presented in IV. It has been implemented twice, once with a classical systematic sampling strategy, and once using a time-distorted context model. The full sampling approach is implemented using the same data storage (Google LevelDB) as our time-distorted context representation approach. Our context model definition consists of one concept, a *smart meter*, one attribute for the *electrical load*, and a *reachable* relationship, which connects smart meters. The context model under study contains 100 smart meters with 10000 values history each, resulting in one million elements to store and analyze. This amount of data corresponds to roughly 140 days history. The experimental validation consists of an electrical load prediction for a specific point of the grid. We define two kinds of predictions both based on a linear regression: 1) *deep* uses a deep history (in time) of the meter, 2) *wide* uses history and in addition the electric load of surrounding meters. These two strategies are each executed with two different ranges: 1) *small* using ten hours of history (30 time units), 2) *large* uses a history of two months (4800 time units). Using the two strategies and two ranges for each, we create four different test series (SDP, SWP, LDP, LWP) that represent typical use cases for our case study. The experiments were carried out on a MacBook Pro i5 2.4 Ghz, 16 GB RAM. The results using full sampling are presented in table I, the results leveraging time-distorted contexts in table II. As shown in the tables, our time-distorted context strategy leads to a reduction of the reasoning time by factors of: **598** for *SDP*, **1361** for *SWP*, **963** for *LDP*, and **1152** for *LWP*, compared to the classic full sampling strategy. The insert time (for storing the context values) has also been significantly improved by a factor of **17**. The two reasoning strategies (full sampling and time-distorted models) predict the same electric load values for all tests. Our evaluation shows a reasoning time in the order of magnitude of minutes for regular sampling and milliseconds for time-distorted contexts, to analyze a particular section of the grid. Moreover, according to our smart grid case study, this electric load prediction has to be continuously performed on several hundred grid points. Thus, our solution reduces the computation time from hours to a few seconds, compatible with the near real-time requirements of smart grid overload capacity. The huge gains compared to full sampling can be explained by the fact that time-distorted contexts allow to directly navigate across the time dimension without costly mining the necessary data from different context models.

VII. DISCUSSION AND RELATED WORK

The lack of a temporal dimension in data modeling has been discussed in detail, especially in the area of databases. In an early work Clifford *et al.* [22] provide a formal semantic for historical databases. Rose and Segev [23] suggest to extend the entity-relationship data model into a temporal, object-oriented one, incorporating temporal structures and constraints in the data model itself rather than at the application level. They also propose a temporal query language for the model. Ariav [24] suggests a temporally-oriented data model as a restricted superset of the relational model. He adds a temporal aspect to the tabular notion of data and provides a framework and a SQL-like query language for storing and retrieving data, taking their temporal context into account. Works of Mahmood *et al.* [25] and Segev and Shoshani [26] take a similar direction and seek to extend the relational model with temporal aspects. In an earlier work Segev and Shoshani [27] examine semantics of temporal data and corresponding operators independently from a data model. In a more recent work, Shih *et al.* [28] describe how including time in a context model helps triggering and handling exceptions in system processes. Selig *et al.* [29] describe an interesting mathematical approach to examine time-dependent association between variables. In the Bigtable [30] implementation, Google incorporates time at its core by allowing each cell in a Bigtable to contain multiple versions of the same data, associated to different timestamps. All this work addresses mainly efficient storage and querying of time related data but largely ignores handling of time in the application domain itself. However, considering time in reasoning processes, like correlating causalities between phenomena, is complex and time-consuming, conflicting with the strict response time requirements intelligent systems usually face. The need to represent and reason about temporal knowledge has also been discussed in RDF [12], OWL [13] and the Semantic Web. Motik [31] suggests a logic-based approach for representing validity time in RDF and OWL. He also proposes to extend SPARQL to temporal RDF graphs and presents a query evaluation algorithm. We suggest to add a time dimension directly into the knowledge representation of a domain itself, *i.e.* into the core of context models. Therefore, we not only efficiently store and query historical data (what is done in other works before), but we propose a way to use time-distorted data sets specifically for intelligent reasoning. Also, we do not extend a specific data model (*e.g.* the relational data model) with temporal structures but use model-driven engineering techniques to integrate a time dimension as crosscutting property of any model element. We do not rely on a complex query language for retrieving temporal data. Instead, our approach aims at providing a natural, query-less

and seamless navigation into the time dimension of model elements, allowing a composition of different time-related values to build a dedicated context model for reasoning purposes (inspired by temporal logic [32]). Like version control systems, *e.g.* Git[4], our approach stores incremental changes (over time) rather than snapshots of a complete system.
Our approach is especially useful if model elements evolve at different paces. If all elements of a context evolve at the same pace the main advantage is the navigation concept as well as the lazy loading mechanism. In future work we want to improve and optimize our implementation. Especially the insertion of new model element versions between two existing versions has to be improved. In addition, we want to investigate how to distribute storage across multiple machines and how this affects the performance of our approach.

VIII. Conclusion

Considering time as a crosscutting concern of data modeling has been discussed since more than two decades. However, recent data modeling approaches mostly still rely on a discrete time representation, which can hardly consider model elements (*e.g.* context variables) coming from different points in time. In this paper, we presented a novel approach which considers time as a first-class property crosscutting any model element, allowing to organize context representations as time-distorted views dedicated for reasoning processes, rather than a mere stack of snapshots. By introducing a temporal validity independently for each model element we allowed model elements to evolve independently and at different paces, making the full sampling of a context model unnecessary. Instead of introducing a dedicated querying language we provided operations to move model elements independently through time, enabling the creation of context models, which combine model elements from different timestamps. Finally, we added a time-relative navigation, which makes an efficient navigation between model elements, coming from different timestamps, possible. This allows us to assemble a time-distorted context model for a specific reasoning purpose and seamlessly and efficiently navigate along this time distortion without manual and costly mining of the necessary data from different context models. Our approach has been implemented and integrated into the open source modeling framework KMF and evaluated on a smart grid reasoning engine for electric load prediction. We showed that our approach supports reasoning processes, outperforms a full context sampling by far, and is compatible with most of near real-time requirements.

References

[1] M. Perttunen, J. Riekki, and O. Lassila, "Context representation and reasoning in pervasive computing: a review," *Int. Journal of Multimedia and Ubiquitous Engineering*, pp. 1–28, 2009.

[2] C.-H. Liu, K.-L. Chang, J.-Y. Chen, and S.-C. Hung, "Ontology-based context representation and reasoning using owl and swrl," in *Communication Networks and Services Research Conf. (CNSR), 2010 8th Annu.*, 2010, pp. 215–220.

[3] K. Henricksen, J. Indulska, and A. Rakotonirainy, "Modeling context information in pervasive computing systems," in *Proc. 1st Int. Conf. Pervasive Computing*, ser. Pervasive '02, 2002, pp. 167–180.

[4] G. Blair, N. Bencomo, and R. France, "Models@ run.time," *Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[5] B. Morin, O. Barais, J. Jezequel, F. Fleurey, and A. Solberg, "Models@ run.time to support dynamic adaptation," *Computer*, vol. 42, 2009.

[4] http://git-scm.com/

[6] J. C. Cepeda, D. Ramirez, and D. Colome, "Probabilistic-based overload estimation for real-time smart grid vulnerability assessment," in *Transmission and Distribution: Latin America Conf. and Expo. (T D-LA), 2012 6th IEEE/PES*, 2012, pp. 1–8.

[7] H. W. Tom, G. Aumiller, and C. Brito-Cruz, "Time-resolved study of laser-induced disorder of si surfaces," *Physical review letters*, vol. 60, no. 14, p. 1438, 1988.

[8] P. Hubral, "Time migration-some ray theoretical aspects*," *Geophysical Prospecting*, 1977.

[9] M. Baldauf, S. Dustdar, and F. Rosenberg, "A survey on context-aware systems," *Int. J. Ad Hoc Ubiquitous Comput.*, vol. 2, no. 4, 2007.

[10] T. Strang and C. L. Popien, "A context modeling survey," in *UbiComp 1st Int. Workshop on Advanced Context Modelling, Reasoning and Management*, 2004, pp. 31–41.

[11] P. P. shan Chen, "The entity-relationship model: Toward a unified view of data," *ACM Trans. Database Syst.*, vol. 1, pp. 9–36, 1976.

[12] O. Lassila and R. R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," W3C, W3C Recommendation, 1999.

[13] W. W. W. C. W3C, *OWL 2 Web Ontology Language. Structural Specification and Functional-Style Syntax*, Std., 2009.

[14] S. Kent, "Model driven engineering," in *IFM*, 2002.

[15] J. Rothenberg, L. E. Widman, K. A. Loparo, and N. R. Nielsen, "The nature of modeling," in *Artificial Intelligence, Simulation and Modeling*, 1989, pp. 75–92.

[16] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J. Jézéquel, "An eclipse modelling framework alternative to meet the models@runtime requirements." in *MoDELS*, 2012.

[17] F. Fouquet, E. Daubert, N. Plouzeau, O. Barais, J. Bourcier, and J.-M. Jézéquel, "Dissemination of reconfiguration policies on mesh networks, DAIS 2012."

[18] F. Budinsky, D. Steinberg, and R. Ellersick, *Eclipse Modeling Framework : A Developer's Guide*, 2003.

[19] OMG, *OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1*, Object Management Group Std., Rev. 2.4.1, 2011.

[20] X. Blanc, I. Mounier, A. Mougenot, and T. Mens, "Detecting model inconsistency through operation-based model construction," in *Proc. 30th Int. Conf. Software Engineering*, 2008, pp. 511–520.

[21] J. Klein, J. Kienzle, B. Morin, and J.-M. Jézéquel, "Aspect model unweaving," in *In 12th International Conference on Model Driven Engineering Languages and Systems (MODELS 2009)*, L. 5795, Ed., Denver, Colorado, USA, 2009, pp. p 514–530.

[22] J. Clifford and D. S. Warren, "Formal semantics for time in databases." in *XP2 Workshop*, 1981.

[23] E. Rose and A. Segev, "Toodm - a temporal object-oriented data model with temporal constraints." in *ER*, T. J. Teorey, Ed., 1991.

[24] G. Ariav, "A temporally oriented data model," *ACM Trans. Database Syst.*, vol. 11, no. 4, pp. 499–527, 1986.

[25] N. Mahmood, A. Burney, and K. Ahsan, "A logical temporal relational data model," *CoRR*, 2010.

[26] A. Segev and A. Shoshani, "The representation of a temporal data model in the relational environment." in *SSDBM*, 1988.

[27] ——, "Logical modeling of temporal data," in *Proc. ACM SIGMOD Int. Conf. on Management of Data*, ser. SIGMOD '87, 1987.

[28] C.-H. Shih, N. Wakabayashi, S. Yamamura, and C.-Y. Chen, "A context model with a time-dependent multi-layer exception handling policy," *IJICIC*, vol. 7, no. 5A, pp. 2225–2234, 2011.

[29] J. P. Selig, K. J. Preacher, and T. D. Little, "Modeling time-dependent association in longitudinal data: A lag as moderator approach," *Multivariate Behavioral Research*, vol. 47, no. 5, pp. 697–716, 2012.

[30] F. Chang, J. Dean, S. Ghemawat, W.-C. Hsieh, D.-A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.-E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proc. 7th USENIX Symp. OSDI - Volume 7*, ser. OSDI '06, 2006, pp. 15–15.

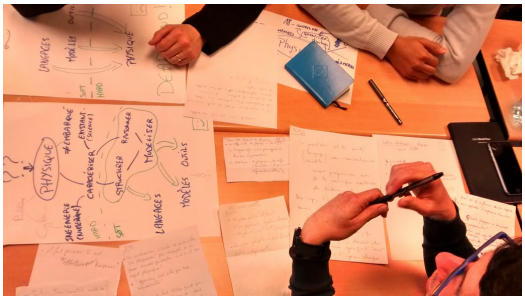
[31] B. Motik, "Representing and querying validity time in rdf and owl: A logic-based approach," in *Proc. 9th Int. Semantic Web Conf. The Semantic Web - Volume Part I*, ser. ISWC'10, 2010, pp. 550–565.

[32] A. Pnueli, "The temporal logic of programs," in *Foundations of Computer Science, 1977., 18th Annu. Symp.*, 1977, pp. 46–57.

Groupe de Travail GL/\CE : Génie Logiciel pour les systèmes Cyber-physiquEs

Sébastien Mosser Romain Rouvoy Pascal Poizat

Compte rendu de la journée du 28 avril 2015




La première journée du GT GL/\CE (*Génie Logiciel pour les systèmes Cyber-physiquEs*) s'est tenue le 28 avril, dans les locaux du LIP6 sur le campus de Jussieu. Elle a réuni 15 personnes, pour travailler ensemble sur les thématiques du groupe de travail et identifier les défis qu'il se propose d'attaquer sur le prochain quadriennal du GDR (2016 - 2020). Nous avons organisé cette première journée comme un atelier de création et d'innovation. Le but du jeu était d'amorcer une communauté, en identifiant ensemble les thématiques que le groupe allait aborder.

La première partie de la journée a consisté en une présentation "flash" de chaque membre du groupe (1 diapo), et de ses intérêts pour le Génie Logiciel et les *Systèmes Cyber-Physiques* (CPS). Un des premiers points abordés lors de ces présentations fut critique pour le reste de la journée : le GT considère t'il les CPS comme un cas d'étude (parmi d'autres) pour le Génie Logiciel, ou bien la vision CPS influence notre manière de concevoir l'ingénierie logicielle et la recherche en génie logiciel associée ?

Sur la base de ces prémisses de discussions, la suite de la journée se poursuit, avec un "World Café" qui durera jusqu'au repas. Les participants se coupent en 4 tables, échangent entre eux sur ce qui a émergé des présentations "flash", puis toutes les 10 minutes les participants changent de table pour continuer leurs discussions avec de nouveaux intervenants. Ce brassage d'échange permet l'émergence d'idées, de questions, de problématiques, qui sont assemblées au fur et à mesure sur une grande rangées de tables (5) dressée pour l'occasion.

Une fois les échanges finis, tout le groupe s'est réuni autour des notes émises pour classifier les idées et faire émerger les thèmes qui seront abordées après le repas. Quatre grands thèmes de recherche sont identifiés sur la base de ces notes : *(i)* gestion de l'incertitude dans les CPS, *(ii)* génie logiciel pour lier Cyber et Physique dans les CPS, *(iii) CPS for the masses* et finalement *(iv)* en quoi les CPS sont ils si différents des systèmes classiques ?

La pause repas permet a tout le monde de souffler un peu et aux équipes de se former autour des 4 grands thèmes identifiés. Après le repas, les équipes s'activent pour définir un poster représentant le défi. Ce poster est présenté au reste du groupe au bout de 30 minutes, et des retours sont faits aux auteurs sous la forme de phrases courtes écrites sur des petites notes. Un deuxième round permet d'améliorer les idées et les posters en tenant compte des retours fournis par l'assistance.

Pour plus d'informations sur la journée et le groupe de travail, vous pouvez consulter le site web du GDR: `http://gdr-gpl.cnrs.fr/node/171`

Session du groupe de travail Compilation

Language Support For Better Polyhedral Compilation Targeting Accelerators

Riyadh Baghdadi

INRIA and École Normale Supérieure de Paris
riyadh.baghdadi@inria.fr

Abstract

The use of special-purpose accelerators such as GPUs can be more appealing than the use of general-purpose processors due to their performance and energy efficiency. Software for such accelerators is currently written using low-level APIs, such as OpenCL and CUDA, which increases the cost of its development and maintenance.

A compelling alternative for developers is to work with higher-level programming languages, and to leverage compilation technology to automatically generate efficient low level code. For general-purpose languages in the C family, this approach is hindered by the difficulty of static analysis in the presence of pointer aliasing. The possibility of aliasing often forces a parallelizing compiler to assume that it is *not* safe to parallelize a region of source code, even though aliasing might not actually occur at runtime. Domain-specific languages (DSLs) can circumvent this problem: it is often clear how parallelism can be exploited given high-level knowledge about standard operations in a given domain, such as linear algebra, image processing or partial differential equations.

We present Pencil, a platform-neutral compute intermediate language intended as a target for DSL compilers and intended to be used by domain experts to target OpenCL-enabled devices. As a subset of GNU C99, Pencil relies on specific language constructs (many of them implemented as builtin functions and attributes) and on enforcing specific coding rules. Although the target platforms are highly parallel, we chose a sequential semantics for Pencil in order to simplify DSL compiler development and the work of a domain expert directly developing in Pencil, and more importantly, to avoid committing to any particular pattern(s) of parallelism. To enable parallelization and loop nest optimizations, we have carefully designed Pencil to incorporate a combination of analysis-friendly restrictions (on pointers and interprocedural data flow) and analysis-enhancing properties (assume predicates, side-effect summaries for functions). We demonstrate these features on a state-of-the-art polyhedral compilation flow, extended with a Pencil front-end and implementing advanced combinations of loop and data transfer optimizations. We illustrate this flow on irregular, data-dependent control and data flow, generating efficient OpenCL code on a variety of targets. Figure 1 shows an example of a Pencil code implementing a general 2D convolution (image processing).

We evaluate Pencil on a set of benchmarks and DSL compilers including:

2

```
__pencil_kill(conv);
__pencil_assume(ker_mat_rows <= 15);
__pencil_assume(ker_mat_cols <= 15);

for (int q = 0; q < rows; q++)
  for (int w = 0; w < cols; w++) {
    float prod = 0.;
    for (int e = 0; e < ker_mat_rows; e++)
      for (int r = 0; r < ker_mat_cols; r++) {
        row = clamp(q+e-ker_mat_rows/2, 0, rows-1);
        col = clamp(w+r-ker_mat_cols/2, 0, cols-1);
        prod += src[row][col] * kern_mat[e][r];
      }
    conv[q][w] = prod;
  }
```

Fig. 1: General 2D convolution

- VOBLA, a linear algebra DSL;
- two signal processing radar applications generated from the SPEAR-DE streaming DSL and modeling environment;
- an image processing benchmark written in PENCIL and covering computationally intensive parts of a computer vision stack used by *RealEyes*, a leader in the automatic recognition of facial emotions and eye tracking;
- a selection of applications extracted from the SHOC and the Rodinia benchmarks re-written in PENCIL.

We automatically generate OpenCL code from PENCIL and target different architectures including an AMD Radeon HD 5670 GPU, an Nvidia GTX470 GPU and an ARM Mali-T604 GPU. The performance gains compared to the implementation efforts for these applications and benchmarks are very encouraging. For example, for the RealEyes image processing benchmark, we were able to match and sometimes outperform the OpenCV image processing library.

Benchmark	Nvidia GTX	ARM Mali	AMD Radeon
colour conversion	1.03	1.01	1.27
2D convolution	1.00	-	0.82
resize	0.98	1.01	1.36
affine warping	1.01	1.27	2.66
gaussian smoothing	0.94	1.36	0.81
dilate	0.63	0.22	0.79
basic histogram	0.41	0.60	0.47

Table 1: Speedups of the OpenCL code generated by `PPCG` over OpenCV

Table 1 shows the speedups of the OpenCL code generated from PENCIL compared to the OpenCV library on Nvidia GTX, on ARM Mali, and on AMD Radeon GPUs.

For the VOBLA linear algebra DSL, we were able to generate code that has close performance to the cuBlas and clMath BLAS linear algebra libraries. Table 2 shows the speedups of the code that we generate automatically from PENCIL over the highly optimized cuBlas and clMath BLAS libraries.

Benchmark	Nvidia GTX	AMD Radeon
gemver	1.17	2.14
2mm	0.91	0.62
3mm	0.87	0.65
gemm	1.10	0.69
atax	0.87	3.79
gesummv	1.03	1.83

Table 2: Speedups obtained with `PPCG` over highly optimized BLAS libraries

The contributions of this paper include

– the presentation of Pencil, a platform-neutral compute intermediate language for DSL compilers and domain experts;
– the extension of a polyhedral compilation framework leveraging the features of Pencil to handle applications that do not fit in the classical restrictions of the polyhedral model, including some forms of dynamic, data-dependent control flow and array accesses;
– the evaluation of Pencil on several application domains and on multiple GPUs.

Refinement of Worst-Case Execution Time Bounds by Graph Pruning

Florian Brandner[1] and Alexander Jordan[2]

[1] U2IS, ENSTA ParisTech
florian.brandner@ensta-paristech.fr
[2] DTU Compute, Technical University of Denmark
alejo@dtu.dk

Extended Abstract The overhead of analyzing safety-critical real-time programs is rapidly growing due to the increasing software size and complexity [6]. For WCET analysis this effect is amplified since in order to arrive at a safe WCET bound, hardware states need to be considered as well, adding to the number of potential states that have to be analyzed. Even when only small portions of a complex software program are relevant for the final WCET estimation, the analysis has to account for *all* of the program's code to derive a safe bound. This inevitably reduces the precision of the WCET analysis, as irrelevant code parts interfere with the analysis of relevant code parts and lead to an overestimation of the WCET bound compared to the actual worst-case behavior.

In this work, we propose an iterative approach that allows us to eliminate *irrelevant code* from the analysis problem, while still delivering provably safe WCET bounds. The approach is based on observations from our previous work on the Criticality metric [2, 3]. This metric characterizes the code parts of a real-time program with regard to their relevance to the obtained WCET bound. A Criticality value close to 0 indicates code that is not relevant for the WCET, while code close to 1 has to be considered highly relevant. Using this information, a code optimization tool, such as a compiler is able to apply optimizations in a directed manner. We found that in standard benchmarks, widely used in the real-time community, considerable portions of the programs' code have low Criticality values. This gave rise to the question: why analyze this *irrelevant* code?

Iterative graph pruning performs WCET analysis in two main steps. First, the Criticality metric is computed – potentially using less expensive approximation techniques [7]. The program's code fragments (typically basic blocks) are then grouped into sets according to their Criticality values. In the second phase of the algorithm, sub-graphs of the program's original control-flow graph (CFG) are constructed by iteratively unifying the most critical k sets, where k is the number of iterations performed so far. These sub-graphs can be analyzed using any standard WCET analysis tool, which will provide us a tentative WCET bound. This bound is typically lower than the WCET bound computed on the program's complete CFG. However, since only a fraction of the original program was considered, the bound might be unsafe at a given iteration k of the algorithm. In order to prove that the tentative bound is safe, we need to reason about the length of the execution paths considered so far, i.e., the paths through the current sub-graph, and the length of the execution path that have not yet been

considered. The tentative WCET bound naturally is safe w. r. t. all execution paths in the sub-graph. What is missing is a bound on the execution paths not yet considered. Fortunately, conservative bounds of these paths are available through the criticality metric pre-computed before the iterative processing. The algorithm thus can easily verify whether the newly computed tentative bound is safe or not. If the bound is safe, the algorithm terminates and the tentative bound becomes the final WCET bound. Otherwise, the iterative processing continues.

Related Work Sandberg et al. [10] propose to use program slicing to improve the analysis of flow facts (infeasible paths) in real-time programs. Similarly, Lokuciejewski et al. [9] use program slicing to improve the precision of loop-bounds analysis. These approaches are orthogonal to our work. In particular, they do not improve the precision of subsequent analyses, e.g., the analysis of cache states, while our pruning is potentially profitable to all analyses.

Bang and Kim [1] (later Zwirchmayer et al. [8]) also iteratively refine the WCET. A regular WCET analysis is performed and the resulting execution path is tested for feasibility. If the path is infeasible, additional flow facts exclude that path during longest path search on a new analysis run. The technique cannot improve the precision of analyses other than the longest path search.

Acknowledgments This abstract is based on a report [5] and an article [4]. The work was partially funded under the European Union's 7th Framework Programme under grant agreement no. 288008 (T-CREST) and the Austrian Science Fund under contract P21842.

References

1. Bang, H.J., Kim, T.H., Cha, S.D.: An iterative refinement framework for tighter worst-case execution time calculation. In: Proc. of the Symposium on Object and Component-Oriented Real-Time Distributed Computing. pp. 365–372 (2007)
2. Brandner, F., Hepp, S., Jordan, A.: Static profiling of the worst-case in real-time programs. In: Proc. of the Conf. on Real-Time and Net. Sys. pp. 101–110 (2012)
3. Brandner, F., Hepp, S., Jordan, A.: Criticality: Static profiling for real-time programs. Real-Time Systems pp. 1–34 (2013)
4. Brandner, F., Jordan, A.: Refinement of worst-case execution time bounds by graph pruning. Computer Languages, Systems & Structures 40(3-4), 155–170 (2014)
5. Brandner, F., Jordan, A.: Subgraph-Based Refinement of Worst-Case Execution Time Bounds. TR 00978015, ENSTA ParisTech (2014)
6. Dvorak (editor), D.L.: NASA study on flight software complexity. Technical excellence initiative, NASA Office of Chief Engineer (2009)
7. Jordan, A.: Evaluating and estimating the wcet criticality metric. In: Proc. of the Workshop on Optimizations for DSP and Embedded Systems. pp. 11–18 (2014)
8. Knoop, J., Kovács, L., Zwirchmayr, J.: WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds. In: Proc. of the Conf. on Real-Time and Net. Sys. (2013)
9. Lokuciejewski, P., Cordes, D., Falk, H., Marwedel, P.: A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In: Proc. of the Int. Symp. on Code Generation and Optimization. pp. 136–146 (2009)
10. Sandberg, C., Ermedahl, A., Gustafsson, J., Lisper, B.: Faster WCET flow analysis by program slicing. In: Proc. of the Conf. on Language, Compilers, and Tool Support for Embedded Systems. pp. 103–112 (2006)

Session de l'action IDM

Ingénierie dirigée par les modèles

Adaptation d'exécution de modèles

Olivier Le Goaer, Eric Cariou, Franck Barbier, and Samson Pierre

Université de Pau / LIUPPA
BP 1155, F-64013 PAU CEDEX, France
{prenom.nom}@univ-pau.fr

1 Introduction

Un des buts fondateurs de l'ingénierie des modèles (IDM) est de considérer les modèles comme les éléments principaux et productifs pour le développement d'applications. On distingue généralement deux approches productives des modèles : la génération de code à partir d'un modèle ou l'interprétation directe du contenu d'un modèle. Selon cette dernière approche, plus récente, on dispose d'un moteur d'exécution implémentant une sémantique opérationnelle capable d'interpréter un modèle. De tels modèles sont écrits dans des langages de modélisation (DSML pour *Domain- Specific Modeling Language*) intégrant la possibilité d'interpréter leur contenu. On parle alors de i-DSML pour *interpreted*-DSML. Dans cet article, nous étudions les i-DSML adaptables, c'est-à-dire des modèles qui sont exécutables et également adaptables pendant leur exécution. Cette adaptation a pour objectif de modifier le contenu du modèle en fonction par exemple d'un changement dans le contexte d'exécution.

Dans la section 2 nous caractérisons les éléments constitutifs d'un i-DSML adaptable. Puis dans la section 3, nous présentons l'intérêt de spécialiser les i-DSML adaptables pour définir des sémantiques d'adaptation qui font sens.

2 Caractérisation des i-DSML adaptables

Dans [2], nous avons caractérisé les éléments constitutifs d'un i-DSML adaptable. À la base, un i-DSML est un méta-modèle qui définit des éléments jouant un rôle statique et d'autres jouant un rôle dynamique. Par exemple, pour une machine à états, les états et les transitions forment le contenu statique et pendant l'exécution, il faudra connaître les états actifs courants qui eux forment la partie dynamique. Une sémantique d'exécution est associée à ce méta-modèle et est implémentée par un moteur interprétant le modèle. Pour une machine à états, la sémantique d'exécution précisera comment suivre les transitions quand un événement est généré.

Un i-DSML adaptable est une extension d'un i-DSML. Dans le méta-modèle est rajoutée une partie d'adaptation qui définit des éléments voués à l'adaptation. Par exemple, pour un processus modélisé qui doit se réaliser en un temps donné, une adaptation possible consiste à supprimer des activités non obligatoires pour gagner du temps. C'est au concepteur du processus de préciser ces activités optionnelles et, à cet effet, le méta-modèle doit permettre de les marquer comme telles. Ensuite, une sémantique d'adaptation est également ajoutée ; elle précisera comment adapter le modèle pendant son exécution. Nous considérons une sémantique d'adaptation comme étant une sémantique d'exécution spécialisée : elle traite des situations anormales ou extra-ordinaires tandis que l'autre s'occupe de l'exécution nominale. En pratique, la sémantique d'exécution ne modifie que les éléments dynamiques pour gérer un pas normal d'exécution tandis que la sémantique d'adaptation peut agir sur tout le contenu du modèle (voire sur les sémantiques d'exécution et d'adaptation elles-même). Par exemple, pour notre processus, des activités, donc des éléments statiques, sont supprimés.

Par conséquent, le moteur d'exécution est étendu pour implémenter en plus la sémantique d'adaptation. Concrètement, en plus des opérations d'exécution, on trouvera des opérations dédiées à l'adaptation et de deux natures : (i) des opérations de vérification pour déterminer si une adaptation est requise et (ii) des opérations réalisant l'adaptation.

3 Spécialisation des i-DSML pour favoriser l'adaptation

Dans [1], nous avons décrit un ensemble de critères qui permettent de caractériser une adaptation :

Fonctionnelle *vs* non-fonctionnelle (qualité de service) : l'adaptation peut modifier ou ajouter des fonctionnalités ou bien concerne la qualité de service de celles existantes.

Prévue à l'avance *vs* non-anticipée : les adaptations sont « pré-câblées » dans le système dès le départ ou bien on peut gérer des cas non prévus à l'avance.

Prédictible *vs* non-déterministe : l'adaptation amène le système dans un état bien défini et stable ou bien laisse place à une certaine incertitude.

Auto-adaptation *vs* par un tiers : l'adaptation est réalisée par le système lui-même ou bien par une entité extérieure qui agit sur le système.

Générique *vs* métier : l'adaptation est agnostique ou bien est dédiée à un contenu métier particulier.

Le challenge scientifique majeur est clairement de pouvoir faire de l'auto-adaptation fonctionnelle, non anticipée, prédictible et générique. Dans [2], nous montrons un exemple de cela pour une adaptation d'exécution de machines à états : en cas d'un événement inconnu (non anticipée), la machine à états se modifie (auto-adaptation) pour rajouter des états et transitions pour gérer ce nouvel événement (fonctionnel) et cela sans avoir besoin de savoir quel système est modélisé par la machine à états (générique). Bien sur, cela n'est pas magique et ne fonctionne qu'avec des hypothèses fortes sur la forme des machines à états considérées. Notamment, nous imposons qu'un événement donné mène par principe toujours au même état. Cela permet de rajouter les transitions associées à cet événement. Le corollaire est que cette adaptation générique ne marche pas pour toute machine à états. Néanmoins, par rapport à une adaptation purement métier, elle sera davantage réutilisable.

Partant de ces constats, afin de favoriser les adaptations génériques, nous avons défini le concept de « famille » [3]. Pour un i-DSML adaptable, on pourra avoir besoin de spécialiser et contraindre le méta-modèle pour faire émerger des adaptations qui n'ont de sens que sous certaines hypothèses. Concrètement, on pourra définir des opérations de vérification et de réalisation d'adaptation dans un contexte précis. Une famille organise tout cela en associant ces opérations avec un méta-modèle spécialisé donné. On pourra ensuite également spécialiser des familles entre-elles, créant ainsi des hiérarchies de classes d'adaptation pour un i-DSML donné.

4 Perspectives

Un moteur exécute des opérations d'exécution et d'adaptation. Pour l'heure, c'est au développeur du moteur d'orchestrer « en dur » dans son code ces différentes opérations. Nous travaillons à externaliser de telles orchestrations dans un modèle propre. Ce modèle sera interprété à son tour par un moteur, c'est donc également un i-DSML. Ce moteur d'orchestration pourra être couplé à n'importe quel moteur d'exécution pour prendre en charge l'adaptation d'un i-DSML. En considérant ce i-DSML d'orchestration comme lui-même adaptable, on pourra modifier ses modèles et donc les sémantiques d'adaptation qui s'y trouvent réifiées. On fera alors de l'adaptation d'adaptation d'exécution de modèles, dit autrement, de la méta-adaptation, en réutilisant nos principes des i-DSML adaptables sur des i-DSML adaptables.

Références

1. Franck Barbier, Eric Cariou, Olivier Le Goaer, and Samson Pierre. Software adaptation : classification and case study with State Chart XML. *IEEE Software*, (à paraître), 2015.
2. Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre. Characterization of Adaptable Interpreted-DSML. In *ECMFA 2013*, volume 7949 of *LNCS*, pages 37–53. Springer, 2013.
3. Samson Pierre, Eric Cariou, Olivier Le Goaer, and Franck Barbier. A Family-based Framework for i-DSML Adaptation. In *ECMFA 2014*, volume 8569 of *LNCS*, pages 164–179. Springer, 2014.

Génération Automatique de Modèles par CSP

Adel Ferdjoukh, Anne-Elisabeth Baert, Eric Bourreau, Annie Chateau, Remi Coletta and Clémentine Nebut

Lirmm, CNRS et Université de Montpellier, France
lastName@lirmm.fr

Résumé

En Ingénierie Dirigée par les Modèles, concevoir des méta-modèles et écrire des transformations de modèles sont deux tâches importantes et récurrentes. Un processus de génération de données de test -donc de modèles- est nécessaire pour tester les transformations de modèles et pour valider les méta-modèles. La génération de modèles doit être automatique et permettre de générer des modèles respectant les contraintes OCL d'un méta-modèle. Dans ce document, nous proposons une approche automatique pour la génération de modèles respectant les contraintes OCL et basée sur les Problèmes de Satisfaction de Contraintes (CSP).

Keywords : Problèmes de Satisfaction de Contraintes (CSP), Génération de Modèles, Contraintes OCL.

1 Introduction & État de l'art

Pour obtenir des modèles pouvant être utilisés pour tester des transformations de modèles et aider à la conception des méta-modèles et puis leurs validation, un processus de génération de modèles doit respecter les caractéristiques suivantes :

1. `automatisation` : processus le plus automatisé possible.
2. `validité` : modèles conformes au méta-modèle et respectant ses contraintes OCL.
3. `passage à l'échelle` : modèles générés et méta-modèles de grande taille.
4. `vraisemblance` : modèles générés réalistes, proches des modèles réels.
5. `diversité` : génération d'ensembles de modèles différents et éloignés les uns des autres.

Approche	**Caractéristiques**				
	1	**2**	**3**	**4**	**5**
Arbres Aléatoires [5]	plutôt oui	non	oui	non	plutôt oui
Grammaires de Graphes [2]	non	non	oui	non	non
contraintes Alloy [6]	plutôt non	plutôt oui	non	non	non
contraintes SMT [7]	oui	oui	plutôt non	non	?
contraintes CSP [1]	oui	oui	plutôt non	possible	non

Table 1 – Comparaison entre les différentes approches de génération selon les précédentes caractéristiques.

La table 1 présente une comparaison succincte des différentes approches de génération de modèles existantes selon les caractéristiques précédentes. Plusieurs remarques peuvent être faites sur cette comparaison :

1

— Les approches basées sur les Arbres aléatoires et les Grammaires de Graphes génèrent des modèles qui ne respectent pas les contraintes OCL.
— L'approche par contraintes Alloy souffre d'une automatisation insuffisante du processus de génération, notamment pour la prise en compte des contraintes OCL.
— La génération à l'aide de contraintes SMT ne génère que des petits modèles (1 à 2 instances par classe).
— L'approche par CSP est prometteuse. Néanmoins, un temps de génération conséquent est induit par une modélisation en CSP, inefficace, d'un méta-modèle et des ses contraintes OCL.

Nous avons choisit de proposer une nouvelle approche de génération par CSP. Un aperçu sur les CSP est donné à la référence [3]. La figure 1 explicite les différentes étapes d'un processus de génération de modèles basé sur les CSP. Ce processus prend en entrée un méta-modèle et un ensemble de ses contraintes OCL. Ces derniers sont traduits en un CSP qui sera résolu par un solveur. Il produit des modèles conformes à ce méta-modèle et respectant ses contraintes OCL.

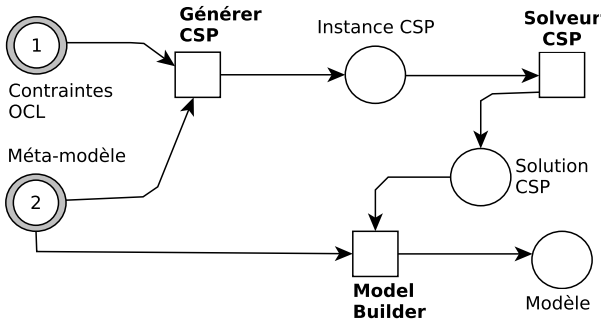


Figure 1 – Processus de génération de modèles par CSP.

2 Traduction d'un Méta-modèle et de ses Contraintes OCL en CSP

La modélisation en CSP d'un méta-modèle et des contraintes OCL influence grandement l'efficacité d'un processus de génération de modèles. L'objectif est d'obtenir un CSP le plus compact possible et de faire en sorte que sa résolution soit facile et rapide. Pour cela, le nombre de variables et contraintes de notre modélisation est optimisé par rapport à l'approche en CSP de *Cabot et al.* [1]. Les détails de notre modélisation d'un méta-modèle en CSP sont donnés à la référence [4]. Concernant les contraintes OCL, nous proposons une modélisation spécifique à chaque construction du langage OCL. Ceci évitera la création de structures lourdes pour des contraintes OCL simples comme par exemple celles comportant des expressions sur des attributs de classes. La modélisation en OCL que nous proposons est publiée dans [3].

Nos principales contributions sont une nouvelle approche de génération de modèles par CSP plus efficace par rapport à la seule approche de génération pour laquelle un outil fonctionnel était disponible [1]. De plus, nous avons mené des expérimentations sur le passage à l'échelle de

notre approche, au cours desquelles nous générons des modèles de grande tailles (jusqu'à 1000 éléments) conformes à de grands méta-modèles (jusqu'à 50 classes). Enfin, nous proposons une modélisation des contraintes OCL en CSP qui est spécifique à chaque construction du langage OCL et qui utilise au mieux les concepts des CSP (optimisation du nombre de variables et contraintes, réduction des domaines et utilisation de contraintes globales). Un outil totalement automatisé et implémentant notre approche est disponible en téléchargement ou en utilisation en ligne à l'adresse : `http://www.lirmm.fr/~ferdjoukh/english/research.html`.

Dans le but d'améliorer notre processus de génération de modèles et couvrir toutes les caractéristiques d'un modèle généré automatiquement, nous travaillons sur la vraisemblance et la diversité des modèles générés. Ainsi, nous projetons d'obtenir les modèles les plus proches possibles des modèles réels mais aussi de générer un ensemble de solutions différentes les unes de autres. Nous menons également des expérimentations avec des utilisateurs pour les aider durant la conception des méta-modèles et leur validation.

3 Conclusion & Perspectives

Dans ce document, nous abordons de manière concise l'approche de génération automatique de modèles que nous proposons. Nous donnons, d'abord, les caractéristiques d'un processus de génération que nous pensons être indispensables pour l'obtention de modèles réellement utilisables. Ensuite, nous comparons les approches de génération existantes selon ces caractéristiques. Ainsi, les points forts et les points faibles de chaque approche sont donnés. Enfin, nous donnons, brièvement, les contributions apportées par notre approche par rapport aux approches existantes.

Références

[1] Cabot, J., Clarisó, R., and Riera, D. Verification of UML/OCL Class Diagrams using Constraint Programming. In *ICSTW, IEEE International Conference on Software Testing Verification and Validation Workshop* (2008), pp. 73–80.

[2] Ehrig, K., Küster, J., and Taentzer, G. Generating Instance Models from Meta models. *Software and Systems Modeling* (2009), 479–500.

[3] Ferdjoukh, A., Baert, A.-E., Bourreau, E., Chateau, A., Coletta, R., and Nebut, C. Instantiation of Meta-models Constrained with OCL : a CSP Approach. In *MODELSWARD, International Conference on Model-Driven Engineering and Software Development* (2015), pp. 213–222.

[4] Ferdjoukh, A., Baert, A.-E., Chateau, A., Coletta, R., and Nebut, C. A CSP Approach for Metamodel Instantiation. In *ICTAI, IEEE International Conference on Tools with Artificial Intelligence* (2013), pp. 1044–1051.

[5] Mougenot, A., Darrasse, A., Blanc, X., and Soria, M. Uniform Random Generation of Huge Metamodel Instances. In *ECMDA, European Conference on Model-Driven Architecture Foundations and Applications* (2009), pp. 130–145.

[6] Sen, S., Baudry, B., and Mottu, J.-M. Automatic Model Generation Strategies for Model Transformation Testing. In *ICMT, International Conference on Model Transformation* (2009), pp. 148–164.

[7] Wu, H., Monahan, R., and Power, J. F. Exploiting Attributed Type Graphs to Generate Metamodel Instances Using an SMT Solver. In *TASE, International Symposium on Theoretical Aspects of Software Engineering* (2013), pp. 175–182.

Processus flexible de configuration pour lignes de produits logiciels complexes

Auteur : Simon Urli (Université de Nice Sophia-Antipolis, I3S)

Résumé :

La nécessité de produire des logiciels de qualité en adéquation avec les besoins spécifiques du marché a conduit à l'émergence de nouvelles approches de développements telles que les *Lignes de Produits Logiciels* (LPL). Cependant pour répondre aux exigences croissantes des nouveaux systèmes informatiques, il convient aujourd'hui d'envisager la production de ces systèmes comme des compositions d'un grand nombre de systèmes interconnectés que l'on nomme aujourd'hui des *systèmes-de-systèmes*. En terme de lignes de produits, il s'agit de supporter la modularité et la très grande variabilité de ces systèmes, aussi bien du point de vue de la définition des sous-systèmes, que du point de vue de leur composition tout en garantissant la viabilité des systèmes construits.

Pour supporter la construction et l'utilisation de lignes de produits logiciels complexes, nous proposons une nouvelle approche basée sur *(i)* la définition du modèle du domaine de la ligne, *(ii)* la formalisation de la variabilité des éléments du domaine par des feature models (FM) et *(iii)* l'expression des dépendances entre ces différents FM. Pour maîtriser la complexité de telles lignes nous avons complété cette approche de modélisation par d'une part, des algorithmes visant à assurer la cohérence des lignes ainsi modélisées et d'autre part, la conception d'un processus de configuration des produits logiciels complexes garantissant la cohérence des produits sans imposer d'ordre dans les choix utilisateurs et en autorisant l'annulation des choix.

Cette thèse présente une formalisation de ces travaux démontrant ainsi les propriétés attendues de ces LPL comme la maîtrise de la complexité de la ligne par des algorithmes incrémentiels exploitant la topologie du modèle du domaine, la définition formelle et la preuve de la flexibilité du processus de configuration ou les notions de cohérence du processus lui-même. Sur cette base bien fondée, nous proposons une implémentation possible intégrant des éléments additionnels pour supporter le développement de telles lignes tels qu'une interface graphique de configuration générique qui nous a servi de support aux expérimentations. Nous validons nos travaux sur une LPL dédiée à un système-de-systèmes de portée industrielle pour la production de systèmes de diffusion d'informations. Cette ligne est aujourd'hui le support d'une start-up qui vend des écrans dynamiques de diffusion d'information

Session du groupe de travail LaMHA

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

Programmation BSP chez Huawei

Auteur : Gaétan Hains (Huawei FRC, Boulogne-Billancourt)

Résumé :

Dans cet exposé nous allons présenter le modèle isochrone de la programmation parallèle, connu sous le nom de « bulk-synchronous parallelism » ou BSP. Ce modèle a été popularisé par Valiant et McColl depuis les années 1990 et depuis adapté à de très nombreuses applications et à tous les types de plateformes matérielles. L ?isosynchronisme a de très nombreux avantages pour le logiciels parallèle : déterminisme, accélérations prévisibles et portables, programmation parallèle déclarative simplifiée et démonstration de programmes parallèles simplifiée. Nous présentons un tour d ?horizon de la R&D en informatique BSP, ainsi que des travaux français du domaine. L ?équipe d ?algorithmique parallèle du French R&D Center Huawei développe des algorithmes et outils de programmation parallèle BSP pour les applications temps-réel comme l ?optimisation des réseaux virtualisés et l ?apprentissage machine.

Vérification de composants distribués : le modèle pNets et son utilisation en pratique

Auteur : Ludovic Henrio et Eric Madelaine (Université de Nice Sophia-Antipolis, I3S, Inria)

Résumé :

Dans cet exposé nous présenterons les pNets, un formalisme de spécification pour systèmes distribués, et les études aussi bien théoriques que pratiques qui portent sur ce formalisme. Nous utilisons les pNets pour modéliser le comportement des applications faites de composants distribués et pour vérifier la correction de ce comportement (généralement par model-checking). Dans cet exposé nous présenterons aussi bien la plateforme de spécification de composants distribués que nous développons que nos derniers résultats théoriques fournissant une théorie de la bisimulation spécifique pour ce modèle.

Modèles fonctionnels de MapReduce en Coq

Auteur : Frederic Loulergue (Inria & LIFO, Université d'Orléans) et Kiminori Matsuzaki (Kochi University of Technology)

Résumé :

MapReduce, d'abord proposé par Google, est un modèle de programmation adapté au traitement de très grands volumes de données. Hadoop, une implantation Java libre de MapReduce, est maintenant utilisée dans de nombreuses applications. Avoir des modèles fonctionnels et formels de MapReduce permet de mieux appréhender les calculs effectués, de prouver la correction de programmes MapReduce, et même de les optimiser. Nous présentons cinq modèles fonctionnels qui capturent la sémantique de Hadoop MapReduce, formalisés en Coq. Nous les utilisons pour développer plusieurs algorithmes MapReduce pour le calcul des préfixes en parallèle (scan).

Session du groupe de travail LTP

Langages, Types et Preuves

Un analyseur statique pour C formellement vérifié

Jacques-Henri Jourdan[1], Vincent Laporte[2,3], Sandrine Blazy[2,3], Xavier Leroy[1] et David Pichardie[2,4]

[1] Inria Paris-Rocquencourt
[2] IRISA
[3] U. Rennes 1
[4] ENS Rennes

1 Introduction

Les outils de vérification sont de plus en plus utilisés dans l'industrie du logiciel critique. Ils utilisent des outils variés, allant de l'analyse statique au model checking, en passant par la vérification déductive de programmes. Les garanties qu'ils apportent peuvent être une simple sécurité de la mémoire ou aller jusqu'à la correction fonctionnelle complète.

Les outils d'analyse statique peuvent être utilisés de deux manières différentes : ils peuvent être de puissants outils de recherche de bugs, ou alors des outils de vérification de programme, qui établissent des propriétés de sûretés ou de sécurités. Dans le cas de la recherche de bugs, la précision de l'analyse est essentielle, mais aucune garantie formelle de correction n'est nécessaire. À l'inverse, pour les outils de vérification formelle, la correction de l'analyse est primordiale. Si l'analyseur ne lève pas d'alarme, il est essentiel que le programme ne contienne pas d'erreur du type recherché.

Afin d'atteindre ce but de sûreté, nous proposons une nouvelle approche : utiliser les technologies de preuves de programmes fournies par l'assistant de preuve Coq afin de donner une preuve formelle de la correction d'un analyseur statique utilisant l'interprétation abstraite.

Notre analyseur, Verasco, est décrit plus en détail dans une publication récente [1]. Il permet de garantir l'absence de comportement indéfini dans un code écrit en C99 sans allocation dynamique ni récursion. Il réutilise le front-end du compilateur formellement vérifié CompCert [2], ce qui, en utilisant les théorèmes de CompCert, lui permet d'apporter des garantie formelles sur le code assembleur généré. L'analyseur contient de nombreux domaines abstraits : des abstractions pour la mémoire utilisant des ensemble de points-to, des intervalles entiers et flottants, des égalités symboliques, et un domaine polyédrique.

2 Une architecture modulaire

Comme on peut le voir dans la figure 1, Verasco est construit en utilisant une architecture très modulaire, grâce au mécanisme des classes de types de Coq. Le premier composant est l'interprète abstrait, qui analyse le flot de contrôle du programme. Il utilise les techniques bien connues en interprétation abstraite de widening et de narrowing afin de calculer les invariants de boucles, notamment. Sa preuve repose sur l'utilisation d'une logique de Hoare spécialisée au langage intermédiaire C#minor de CompCert. Sa complexité réside dans le fait qu'il prend en compte toutes les constructions de de contrôle disponibles dans ce langage (boucles, tests, gotos, appels de fonctions). Ce composant est paramétré par un domaine abstrait d'états, dont le rôle est d'abstraire les valeurs de variables, la structure du tas et la pile d'appel. Il est lui-même paramétré par un domaine numérique, composé de plusieurs modules : un foncteur transforme un domaine numérique sur les entiers mathématiques non bornés et les flottants en un domaine d'arithmétique machine, et une combinaison de domaines permet d'obtenir de précises approximations d'environnements numériques. Cette combinaison comprend deux domaines non relationnels (les congruences et les intervalles), qui utilisent une couche d'adaptation générique pour avoir une interface relationnelle ;

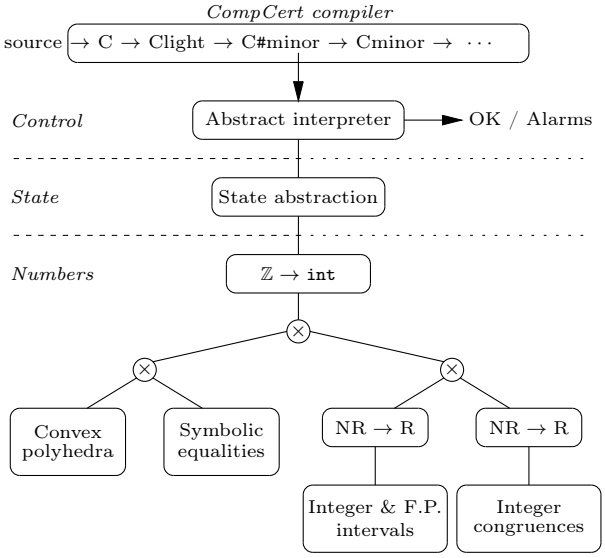


Fig. 1. Modular architecture of the Verasco static analyzer

un domaine d'égalités symboliques qui permet notamment de reconstruire les expressions booléennes ; et un domaine polyédrique. Un système de canaux permet à ces domaines numériques de partager de l'information, tout en gardant une interface compositionnelle.

Chacun de ces modules a une interface qui décrit les opérations mise à la disposition des autres composants, ainsi que les spécifications de ces opérations. Typiquement, un domaine abstrait fournit les opérations classiques de treillis (union, top, comparaison, widening), ainsi que les opérations spécifiques, comme `assign`, qui implémente l'affectation. Elles sont spécifiées en utilisant une fonction de concrétisation γ, qui associe à une valeur abstraite un ensemble de valeurs concrète, comme souvent en interprétation abstraite. Il est notable que dans ces spécifications, ainsi que dans la formalisation dans son ensemble, une large partie de la théorie de l'interprétation abstraite n'est pas formalisée. Ainsi, les résultats d'optimalité de l'analyse n'étant pas essentiels pour nous, nous ne faisons jamais référence à la fonction de concrétisation α. De même, nous évitons une fastidieuse preuve de terminaison de l'analyse, reposant sur des propriétés de l'opérateur de widening en utilisant la technique du *fuel*.

3 Résultats

Bien qu'encore à l'état de prototype, Verasco est déjà capable d'analyser et de prouver l'absence de comportement indéfini pour des programmes de plusieurs centaines de lignes, et dont les arguments nécessaires pour prouver la sûreté peuvent être complexes. Nous analysons, par exemple, deux programmes du banc d'essai de CompCert, contenant des calculs numériques complexes sur les nombres flottants (`nbody.c` and `almabench.c`) ; ainsi que `smult.c`, un programme pris de la bibliothèque cryptographique NaCl, effectuant des calculs de multiplication dans une courbe elliptique. L'analyse de ces programme prend au plus quelques dizaines de secondes.

Références

1. Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D. : A formally-verified C static analyzer. In : POPL. pp. 247–259. ACM (2015)
2. Leroy, X. : Formal verification of a realistic compiler. Comm. ACM 52(7), 107–115 (2009)

Une formalisation en Coq du modèle relationel de données ⋆

Véronique Benzaken[1], Évelyne Contejean[2], and Stefania Dumbrava[1]

[1] Université Paris Sud, LRI, France
[2] CNRS, LRI, Université Paris Sud, France

1 Contexte et motivation

De nombreux domaines scientifiques et socio-économiques (sciences fondamentales, économie numérique, informatique décisionnelle, environnement, sécurité, industrie...) produisent désormais de très grands volumes de données, et cette explosion est en passe de produire une révolution dite « sociétale ». Il est donc de la plus haute importance d'avoir des garanties fortes sur la fiabilité des systèmes et des applications qui traitent ces données massives. Curieusement ce domaine a été peu exploré par la recherche, tant académique qu'industrielle à l'exception de [4,5]. Une piste pour obtenir ces garanties consiste à appliquer des techniques déjà bien connues dans le domaine de la vérification et de la preuve de programmes. Le recours à des assistants à la preuve tels que Coq [6] pour formaliser les systèmes et certifier formellement les algorithmes est une voie très prometteuse. Parmi les systèmes de gestion de données, les systèmes relationnels occupent une place prépondérente et sont incontournables. C'est pourquoi nous avons choisi comme premier objet d'étude le modèle relationnel qui constitue leur fondement théorique et partant, permet d'exprimer leur sémantique et de justifier des optimisations, des choix de conception etc. Le modèle relationel a différentes finalités qui sont liées, il permet :

1. de *représenter* l'information au moyen de *relations* ;
2. d'*extraire* cette information grâce à des *langages de requêtes*, fondés soit sur l'algèbre relationelle, soit sur la logique du premier ordre ;
3. d'*optimiser* l'exécution de ces requêtes au moyen d'*équivalences algébriques* ;
4. d'*affiner* l'information ainsi représentée en la restreignant par des *contraintes d'intégrité*.

Deux versions, équivalentes, du modèle relationel coexistent : la version par positions et la version par noms. Dans le cadre par positions, les attributs spécifiques à une relation sont ignorés, et seule l'arité (c'est-à-dire le nombre de ces attributs) est utilisée par les langages de requêtes. Dans le cas nommé, au contraire, les attributs font partie intégrante de la base et sont utilisés tant par les langages de requêtes que pour définir les contraintes d'intégrité. En pratique, les systèmes tels qu'Oracle, DB2, PostgreSQL ou Microsoft Access reposent, principalement, sur la version *nommée* du modèle. Ce choix est motivé par plusieurs raisons. Les noms portent davantage de sens que des numéros, ce qui est appréciable à des fins de modélisation. Ensuite les optimiseurs tirent parti de structures de données auxiliaires à des fins d'optimisation physique. De telles structures, en particulier les index, sont définies en utilisant explicitement les noms des attributs.

Notre objectif à long terme est de *prouver* que les systèmes existants se conforment à leur spécification et de *vérifier* que les programmes utilisant intensivement des requêtes à des bases de données sont corrects. Ainsi, la première étape incontournable de ce travail est de formaliser le modèle relationnel.

2 Contributions

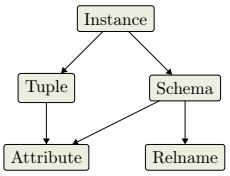
Nos contributions couvrent les quatre aspects du modèle relationnel décrits ci dessus. Dans ce court résumé, nous nous bornerons à présenter nos choix de modélisation essentiellement pour ce qui concerne le premier d'entre eux, puisque tout le reste en découle, et à ne survoler que très brièvement l'algèbre, l'optimisation et les contraintes. Le lecteur intéressé trouvera la version complète dans [3] et le code Coq associé dans [2].

⋆. Ce travail a été partiellement financé par le projet ANR Typex n^o 11BS0200702.

2.1 Représentation des données

L'information est représentée par des tables (relations) dont les lignes (ou n-uplets ou tuples) partagent une structure uniforme et une même signification, une ligne pour une entité. Chaque élément d'une ligne est une valeur associée à un attribut. Suivant la ligne fixée dans [1], nous supposerons donné un ensemble *attribute* (de noms) d'attributs, muni d'un ordre total $\leq_{\mathbf{att}}$. Si ces attributs doivent avoir des domaines (entiers, chaînes de caractères, *etc*) distincts, cela est précisé par une fonction *dom* d'*attribute* vers *domain*. En outre, nous nous donnerons un ensemble de valeurs, elles aussi typées. Dans le contexte des bases de données, le domaine est l'analogue parfait des types dans les langages de programmation, c'est pourquoi notre modélisation utilise type_of_attribute plutôt que de *dom*.

Notre vision des n-uplets est abstraite, et ne fait aucune supposition liée à une quelconque implantation concrète. Au contraire, nous supposerons qu'un n-uplet n'est connu que par l'ensemble de ses attributs (support), et une fonction qui associe une valeur à chaque attribut (dot), et que ces informations sont suffisantes pour caractériser l'équivalence entre n-uplets. Un *schéma de relation* est modélisé par un nom et un ensemble fini d'attributs appelé sa *sorte* (basesort). Les noms des relations et les sortes associées sont regroupés dans le *schéma de la base*, et les ensembles de n-uplets associés constituent l'*instance* du schéma. Les dépendances entre ces notions sont décrites par la figure ci-dessous.




```
(** Basic ingredients,
     attributes, domains and values *)
attribute : Type;
type : Type;
value : Type;
(* Typing attributes and values. *)
type_of_attribute : attribute → type;
type_of_value : value → type;
(* A module for handling finite sets
   of attributes *)
A : Fset.Rcd attribute;
(** Tuples *)
tuple : Type;
support : tuple → set A;
dot : tuple → attribute → value;
```

```
mk_tuple : set A → (attribute → value) → tuple;
support_mk_tuple_ok :
  ∀ V f, support (mk_tuple V f) =set= V;
dot_mk_tuple_ok :
  ∀ a V f, a ∈ V → dot (mk_tuple V f) a = f a;
tuple_compare : tuple → tuple → comparison;
tuple_compare_ok : ∀ t1 t2 : tuple,
 (tuple_compare t1 t2 = Eq) ⟷
    (support t1 =set= support t2 ⋀
    ∀ a, a ∈ (support t1) → dot t1 a = dot t2 a)
```

Le postulat d'uniformité est formalisé (well_sorted_instance) par le fait que tous les n-uplets d'une même relation ont le même support, la sorte de la relation.

```
Definition well_sorted_instance (I : relname → set T) := ∀ (r : relname) (t : tuple), t ∈ (I r) → support t =set= basesort r.
```

Le bon typage des n-uplets est formalisé (well_typed_tuple) par le fait le type de la valeur de chaque attribut varie dans le domaine correspondant à cet attribut.

```
Definition well_typed_tuple (t : tuple) := ∀ a, a ∈ (support t) → type_of_value (dot t a) = type_of_attribute a.
```

La propriété well_sorted_instance s'est avérée cruciale dans la preuve des différents théorèmes "bases de données". En revanche et de façon surprenante la propriété well_typed_tuple s'est avérée inutile dans les développements ultérieurs de la formalisation.

2.2 Requêtes : algèbre et calcul

L'algèbre relationnelle est construite à partir d'un ensemble d'opérateurs (algèbriques) ayant pour opérandes des relations. Les opérateurs classiques décrits dans [1] sont la sélection par rapport à une fonction booléenne (σ), la projection sur un ensemble w d'attributs (π), la jointure naturelle ($\bowtie$), le renommage des attributs (ρ), l'union, l'intersection et la différence ensemblistes.

$$q := r \mid \sigma_f(q) \mid \pi_w(q) \mid \rho_g(q) \mid q \bowtie q \mid q \cup q \mid q \cap q \mid q \setminus q$$

Dans le cas des requêtes conjonctives, le langage est quelque peu différent : plutôt que de s'appuyer sur les opérateurs algébriques, les requêtes sont exprimées au moyen de formules logiques

$$\{(a_1, \ldots, a_n) \mid \exists b_1, \ldots \exists b_m, P_1 \wedge \ldots \wedge P_k\}$$

où les a_i, b_i dénotent des variables qui seront interprétées par des valeurs, et où les P_i sont soit des égalités entre variables soit des prédicats qui correspondent à l'appartenance à une relation de base. Les requêtes conjonctives jouent un rôle central pour l'optimisation, puisqu'elles admettent une forme équivalente minimale (c'est-à-dire optimale).

```
Inductive query : Type :=
  | Query_Basename : relname → query
  | Query_Sigma : formula → query → query
  | Query_Pi : setA → query → query
  | Query_NaturalJoin : query → query → query
  | Query_Rename : renaming → query → query
  | Query_Union : query → query → query
  | Query_Inter : query → query → query
  | Query_Diff : query → query → query
with ...
```

```
Inductive tvar : Type :=
    Tvar : nat → tvar
  | Tval : value → tvar.
Inductive trow : Type :=
    Trow : relname → (attribute → tvar) → trow.
Definition tableau := (* set of trows *) (Fset.set (Ftrow T DBS)).
Inductive summary : Type :=
    Summary : set A → (attribute → tvar) → summary.
Definition tableau_query := (tableau * summary)
```

La sémantique des requêtes est formalisée en Coq par une fonction qui capture le comportement classique des opérateurs de l'algèbre relationnelle, comme en témoignent les lemmes d'adéquation prouvés pour chaque opérateur. Ci-dessous, nous en donnons trois.

```
Lemma mem_Basename :
  ∀ r t, t ∈_I (Query_Basename r) ⟷ t ∈ (I r).

Lemma mem_Union : ∀ q1 q2, sort q1 =S= sort q2 →
   ∀ t, t ∈_I (Query_Union q1 q2) ⟷ (t ∈_I q1 ⋁ t ∈_I q2).
```

```
Lemma mem_NaturalJoin : well_sorted_instance I →
  ∀ q1 q2 t, t ∈_I Query_NaturalJoin q1 q2 ⟷
     ∃t1, ∃t2, (t1 ∈_I q1 ⋀ t2 ∈_I q2 ⋀ ...
           (* join semantics *) ... )
```

Nous avons modélisé l'algèbre relationnelle ainsi que les requêtes conjonctives, et donné une sémantique formelle pour ces deux langages. Nos lemmes d'adéquation illustrent la nature hétérogène de l'algèbre relationnelle : pour les opérateurs purement ensemblistes, il est nécessaire que *les sortes des opérandes coïncident*, tandis que pour les autres, les hypothèses portent sur le fait que *l'instance est bien sortée*. Enfin nous avons formalisé un algorithme de traduction des requêtes de l'algèbre relationnelle vers des requêtes conjonctives, et certifié qu'il préserve les sémantiques. Pour ce faire, nous avons dû clarifier certains points passés sous silence dans les livres.

2.3 Optimisation logique

Pour ce qui est de l'optimisation logique dans les deux formalismes, nous avons démontré les principaux théorèmes classiques du domaine : les équivalences algébriques, le théorème de l'homomorphisme. Ceci nous a permis de dériver de nouvelles équivalences algébriques, non répertoriées dans les livres de référence, telles que :

$$\pi_w(R_1 \bowtie R_2) = \pi_{w_1}(R_1) \ \bowtie \pi_{w_2}(R_2) \text{ si } \begin{cases} w = \ (w_1 \cup w_2) \\ w_1 \subseteq sort(q_1) \\ w_2 \subseteq sort(q_2) \\ sort(q_1) \cap sort(q_2) \subseteq (w_1 \cap w_2) \end{cases}$$

qui s'exprime formellement par le théorème suivant :

```
Lemma Pi_NaturalJoin_distr : well_sorted_instance I →
    ∀ w w1 w2 q1 q2, w =^set= (w1 ∪ w2) → w1 ⊆ (sort q1) → w2 ⊆ (sort q2) → (sort q1 ∩ sort q2) ⊆ (w1 ∩ w2) →
    Query_Pi w (Query_NaturalJoin q1 q2) =^I= Query_NaturalJoin (Query_Pi w1 q1) (Query_Pi w2 q2).
```

Enfin nous avons formalisé la minimisation des requêtes conjonctives, en avons prouvé la correction et extrait le premier algorithme de minimisation certifié.

2.4 Contraintes d'intégrité

Les contraintes d'intégrité constituent un élément essentiel du modèle relationnel. Les plus simples sont les dépendances fonctionnelles qui permettent de définir des clefs, les dépendances multivaluées et les dépendances d'inclusion permettant quant à elles d'exprimer la notion de clef étrangère. Toutes tombent dans la classe des dépendances généralisées. Nous avons modélisé ces contraintes et nous nous sommes attelées à formaliser le problème de *l'implication*, central dans le domaine. Plus précisément, il s'agit, à partir d'un ensemble donné de contraintes, d'en déduire de nouvelles, qui sont des conséquences logiques.

Nous avons modélisé les règles du système d'inférence d'Armstrong (pour les dépendances fonctionnelles) par le type inductif `dtree`, donné la sémantique de ces dépendances (`fd_sem`), ainsi qu'une preuve formelle directe de correction et de complétude.

```
Inductive fd : Type := FD : set A → set A → fd.
Notation "v '↪' w" := (FD v w).

Inductive dtree (F : set F) : fd → Type :=
 | D_Ax : ∀ X Y, (X ↪Y) ∈ F → dtree F (X ↪Y)
 | D_Refl : ∀ X Y, Y ⊆ X → dtree F (X ↪Y)
 | D_Aug : ∀ X Y Z XZ YZ,
     XZ =set (X ∪ Z) → YZ =set (Y ∪ Z) →
     dtree F (X ↪Y) → dtree F (XZ ↪YZ)
 | D_Trans : ∀ X Y Y' Z, Y =set Y' →
     dtree F (X ↪Y) → dtree F (Y' ↪Z) → dtree F (X ↪Z).

Definition fd_sem (s : set T) (d : fd) := match d with | v ↪w ⇒
    ∀ t1 t2, t1∈ s → t2∈ s → (∀ x, x ∈ v → dot T t1 x = dot T t2 x) → ∀ y, y ∈ w → dot T t1 y = dot T t2 y end.
Theorem Armstrong_soundness : ∀ F d (t : dtree F d) (s : set T), (∀ f, f ∈ F → fd_sem s f) → fd_sem s d.

Theorem Armstrong_completeness : ∀ U F X Y, X ⊆ U → Y ⊆ U → (∀ (s : set T), (∀ t, t ∈ s → support T t =set U)
  → (∀ f, f ∈ F → fd_sem s f) → fd_sem s (X ↪Y)) → (dtree F (X ↪Y)).
```

Pour traiter le cas général, nous avons formalisé la procédure dite du “chase”, et montré sa correction. Ce faisant, nous avons mis en lumière des imprécisions dans les descriptions livresques [1,7] et exhibé un contre-exemple dû au fait que les ouvrages de référence négligent les problèmes de capture de variable ou confondent renommage et unification.

3 Conclusion et perspectives

File name	size (loc)	contents
Data.v	262	attributes tuples relations
RelationalAlgebra.v	2,106	relational algebra definition and semantics
AlgebraOptimisation.v	1,565	algebra structural equivalences
Tableaux.v	1,958	tableau formalization, homomorphism, minimization
Translation.v	3,738	translation from algebra to tableaux
Matching.v	1,356	matching for tableau homomorphism and chase
Unify.v	4,439	unification for translation
Chase.v	4,187	chase procedure
Armstrong	1,649	Armstrong’s inference system
Tree.v	460	auxiliary definitions for orders

Notre formalisation atteint un haut degré d’abstraction et de modularité. C’est, à notre connaissance, la mécanisation la plus réaliste de la théorie classique des bases de données à ce jour. Une des contributions importantes de ce travail est d’avoir pu par ce processus de formalisation apporter de nouveaux éclairages aux deux communautés, Coq et bases de données. Nous avons ainsi tenté de souligner les subtilités qui étaient dissimulées ou même complétement absentes dans les ouvrages de référence. Enfin nous tenons à insister sur le fait que cette étape de formalisation n’est pas un pur exercice de style en Coq, mais bien une phase nécessaire pour la vérification d’un système de gestion de base de données pleinement fonctionnel (ou réaliste), et qu’il n’y a aucun moyen de la contourner.

Références

1. Abiteboul, S., Hull, R., Vianu, V. : Foundations of Databases. Addison-Wesley (1995)
2. Benzaken, V., Contejean, É., Dumbrava, S. : A Relational Library. `http://datacert.lri.fr/esop/html/Datacert.AdditionalMaterial.html` (2013)
3. Benzaken, V., Contejean, É., Dumbrava, S. : A Coq formalization of the relational data model. In : Shao, Z. (ed.) 23rd European Symposium on Programming, ESOP 2014, Grenoble, France, April 5-13, 2014, Proceedings. LNCS, vol. 8410, pp. 189–208. Springer (2014)
4. Gonzalia, C. : Towards a formalisation of relational database theory in constructive type theory. In : Berghammer, R., Möller, B., Struth, G. (eds.) RelMiCS. LNCS, vol. 3051, pp. 137–148. Springer (2003)
5. Malecha, G., Morrisett, G., Shinnar, A., Wisnesky, R. : Toward a verified relational database management system. In : ACM Int. Conf. POPL (2010)
6. The Coq Development Team : The Coq Proof Assistant Reference Manual (2010), `http://coq.inria.fr`, `http://coq.inria.fr`
7. Ullman, J.D. : Principles of Database Systems, 2nd Edition. Computer Science Press (1982)

Mezzo : sûreté du typage et absence de *data races*

Thibaut Balabonski[1,2], François Pottier[1], and Jonathan Protzenko[1,3]

[1] Inria
[2] Université Paris-Sud
[3] Microsoft Research

Résumé Le langage de programmation Mezzo est caractérisé par un système de types riche contrôlant l'*aliasing* et l'accès à la mémoire. Nous présentons ici une formalisation modulaire du noyau du système de types de Mezzo, sous la forme d'un λ-calcul concurrent étendu par des références et des verrous. Nous prouvons que les programmes bien typés s'exécutent sans erreur et sans *data races*.

Transformer une logique de programmes en un système de types

Les langages de programmation fortement typés éliminent certaines erreurs de programmation en s'assurant que chaque opération est appliquée à des arguments convenables. Pour obtenir de meilleures garanties, il faut en général utiliser en sus des techniques d'analyse statique ou de vérification. La logique de séparation concurrente [3] par exemple permet de prouver l'absence d'interférences entre des processus concurrents utilisant des ressources partagées.

Le langage Mezzo [5, 4, 2] est soumis à un système de types qui va plus loin que les systèmes de types habituels, en incorporant quelques idées de la logique de séparation. Le système de types de Mezzo intègre des raisonnements concernant l'*aliasing* ou la possession de blocs de mémoire. Ceci d'une part augmente l'expressivité du langage, en ce que cela permet par exemple l'initialisation graduelle d'une structure, et d'autre part élimine plus d'erreurs de programmation, comme les *data races*. Mezzo est inspiré de ML et en possède certaines caractéristiques clés : variables locales immuables, données potentiellement mutables allouées sur le tas, et fonctions de première classe. Nous présentons ici la théorie de son système de types, étendue à un cadre de concurrence à mémoire partagée.

Programmer avec des permissions

Si nous créons une référence en Mezzo, par exemple avec une instruction `val r = newref 0`, cela crée au niveau du typeur une *permission* que l'on note `r @ ref int`, et qui a deux significations conjointes :

- `r @ ref int` décrit l'agencement de la mémoire, en affirmant que la variable `r` dénote l'adresse d'une cellule mémoire qui contient une valeur entière. En ce sens, cette permission est similaire à la notation de type habituelle `r : ref int`.
- `r @ ref int` offre un *accès exclusif* en lecture et en écriture à cette cellule mémoire. Ainsi le type `ref` désigne des références à possesseur unique, et la permission `r @ ref int` est la clé unique qu'il faut posséder pour être autorisé à déréférencer `r`. Notons que cette clé n'existe qu'au moment de la vérification des types.

Une différence fondamentale entre la permission `r @ ref int` et l'habituelle hypothèse de typage `r : ref int` est que, tandis que l'hypothèse de typage est supposée valide dans toute la portée de `r`, la permission peut être passée d'une fonction appelante à une fonction appelée, ou être renvoyée par une fonction appelée à sa fonction appelante, ou être passée d'un processus à un autre. Cette permission n'existe qu'en un seul exemplaire. Si elle est passée à un nouveau processus, il n'est plus possible de lire ou écrire dans la cellule `r`, celle-ci fût-elle visible par ailleurs. Il s'agit d'un transfert de la possession de cette variable.

Bien que la permission `r @ ref int` ne puisse être copiée, certaines permissions sont duplicables. Par exemple `x @ int` est une permission duplicable, qui peut être partagée entre plusieurs

processus, et qui se comporte donc comme l'hypothèse de type habituelle `x : int`. Cette duplication est autorisée en Mezzo pour les types de données immuables.

Autrement dit, le principe général de Mezzo est d'autoriser deux modes pour ses données : chaque donnée est soit accessible à tous en lecture seule, soit accessible à un unique possesseur en lecture et en écriture.

Utiliser des verrous pour transférer des permissions

Bien que Mezzo interdise la duplication d'une permission `r @ ref int`, il peut être légitime que plusieurs processus partagent une même référence `r` et y accèdent à tour de rôle, pourvu que suffisamment de mécanismes de synchronisation soient présents pour éviter toute interférence au niveau de cette référence. Par exemple, on peut s'assurer que les accès à la référence partagée `r` ne se font que sous la protection d'un verrou.

Le système de types de Mezzo permet de vérifier que ces mécanismes de verrouillage sont bien utilisés et imposent effectivement une bonne synchronisation des accès aux différentes cellules mémoire. Pour ceci, chaque verrou est associé à une permission, qui est gagnée ou perdue par tout processus qui prendrait ou relâcherait ce verrou.

On peut par exemple créer un verrou `l` de type `lock (r @ ref int)` qui protège la permission `r @ ref int`. Point important : la permission `l @ lock (r @ ref int)` est duplicable, et le verrou `l` peut ainsi être partagé, ce qui permet à plusieurs processus d'entrer en compétition pour la prise de ce verrou. Un processus qui arrive à prendre le contrôle du verrou `l` gagne la permission exclusive `r @ ref int` et peut momentanément accéder à la référence `r` en lecture comme en écriture. En revanche, au moment de relâcher le verrou `l`, ce processus doit toujours être en possession de cette permission `r @ ref int`, et doit la « rendre » au verrou, c'est-à-dire l'abandonner. Ainsi, quand un deuxième processus aura réussi à prendre à nouveau le verrou et obtenir l'accès à `r`, nous pouvons être certain que le premier processus n'aura plus cet accès.

Formalisation

Nous prouvons que le système de types de Mezzo est correct, en ce qu'il vérifie les propriétés usuelles de préservation du typage et de progression. De plus, nous prouvons que les programmes Mezzo bien typés n'ont pas de *data race*, c'est-à-dire que deux processus ne peuvent jamais simultanément accéder au même bloc de mémoire quand l'un des deux accès est en écriture.

Cette preuve formalisée en Coq [1] cherche à être modulaire : l'ensemble de la structure est d'abord mise en place pour un λ-calcul concurrent avec sous-typage, puis les références et les verrous sont ajoutés comme deux extensions indépendantes (mais destinées à être utilisées ensemble) interférant peu avec la base déjà présente. Cette formalisation fait notamment apparaître un ensemble de ressources, qui est partagé entre les différents processus et les verrous, et qui passe de l'un à l'autre au gré des synchronisations. Notez pour finir que ces mouvements de ressources sont purement logiques : ils ne se traduisent par aucune manipulation de la mémoire à l'exécution.

Références

[1] Balabonski, T., Pottier, F. : A Coq formalization of *Mezzo*, take 2 (Jul 2014), `http://gallium.inria.fr/~fpottier/mezzo/mezzo-coq.tar.gz`

[2] Balabonski, T., Pottier, F., Protzenko, J. : Type soundness and race freedom for Mezzo. In : Proceedings of the 12th International Symposium on Functional and Logic Programming (FLOPS 2014). Lecture Notes in Computer Science, vol. 8475, pp. 253–269. Springer (2014)

[3] O'Hearn, P.W. : Resources, concurrency and local reasoning. Theoretical Computer Science 375(1–3), 271–307 (2007)

[4] Pottier, F., Protzenko, J. : Programming with permissions in Mezzo. In : International Conference on Functional Programming (ICFP). pp. 173–184 (2013)

[5] Pottier, F., Protzenko, J. : *Mezzo*(Jul 2014), `http://protz.github.io/mezzo/`

Session du groupe de travail MFDL

Méthodes Formelles dans le Développement Logiciel

Validation du standard RBAC ANSI 2012 avec B

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

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

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

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

Références

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

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

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

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

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

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

Construction de programmes parallèles en Coq avec des homomorphismes de listes

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

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

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

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

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

1 Une introduction à la théorie des listes

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Paralléliser avec des homomorphismes dans SyDPaCC

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

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

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

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

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

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

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

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

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

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

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

En définissant la classe RightInverse par :

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

L'impression du terme par_ms donne :

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

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

Références

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Verifying Reachability-Logic Properties on Rewriting-Logic Specifications

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

[1]INRIA Lille-Nord Europe, France

[2]Faculty of Computer Science UAIC, Romania

[3]CRIStAL, University of Lille, France

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

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

References

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

Session du groupe de travail MTV^2

Méthodes de test pour la validation et la vérification

Instrumentation de programmes C annotés pour la génération de tests*

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

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

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

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

Nos contributions sont :

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

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

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

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

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

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

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

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

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

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

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

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

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

Résumé

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

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

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

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

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

Remerciement

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

Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Analyse de Différences entre ASTs avec GumTree

Jean-Rémy Falleri[1], Floréal Morandat[1], Xavier Blanc[1], Matias Martinez[2], Martin Monperrus[2]

[1] Univ. Bordeaux, LaBRI, UMR 5800
F-33400, Talence, France {falleri,fmoranda,xblanc}@labri.fr
[2] INRIA and University of Lille, France {matias.martinez,martin.monperrus}@inria.fr

1 Introduction

L'observation de l'évolution logicielle passe par le calcul de différences entre fichiers de code source. Les différences sont usuellement représentées par une séquence d'actions d'édition. Les approches existantes calculent ces actions sur une représentation du fichier source comme étant une séquence de lignes de texte, les actions étant *ajouter*, *supprimer* ou *remplacer* une ligne de texte. Malheureusement, cette façon de procéder ne permet pas de calculer des différences au niveau syntaxique du code source considéré. En outre, ce cadre théorique ne permet pas de considérer le déplacement de code, qui est une action d'édition fréquemment utilisée par les développeurs. Pour calculer des différences au niveau syntaxique, il est nécessaire de travailler sur une représentation arborescente du code source : l'arbre de syntaxe. Or, calculer des différences sur ces structures est un problème avec une complexité au pire cas de $O(n^3)$ sans prendre en compte les déplacements. Si les déplacements sont pris en compte, le problème devient NP difficile [1]. Le calcul de telle différences est donc un problème difficile pour lequel il faut développer des heuristiques efficaces.

Notre nouvelle approche GumTree résout ces problèmes. Nous introduisons un algorithme, basé sur des heuristiques, qui calcule des différences sur une représentation arborescente du code source qui prennent en compte les déplacements. Le complexité au pire cas de notre algorithme est $O(n^2)$. Notre objectif est de calculer une séquence d'action qui est courte, et proche de l'intention originelle du développeur. Notre algorithme est implémenté dans un outil disponible gratuitement et validé intensivement, de manière empirique.

2 Approche

Voici le fonctionnement global de notre approche, qui prend en entrée deux fichiers de code source :

1. Les fichiers sont parsés et les arbres de syntaxe (ASTs) sont construits,
2. Notre algorithme calcule des mappings entre les noeuds des deux ASTs,
3. Nous réutilisons un algorithme existant [2] pour convertir ces mappings en une séquence d'actions.

Notre principale contribution se situe donc au niveau de l'étape 2, durant laquelle les mappings entre les noeuds des deux ASTs sont calculés. Nous introduisons à cet effet un algorithme qui s'inspire de la manière dont procède manuellement les développeurs pour analyser des différences entre deux fichiers de code source. Premièrement, ils cherchent les grosses parties communes entre les deux fichiers. Ensuite ils se servent de ces parties communes pour déduire quels sont les mappings entre les éléments "conteneur" du code source (comme les classes ou les functions). Enfin, quand ils ont trouvé deux conteneurs qui se correspondent, ils analysent leurs différences précises.

Notre algorithme procède en deux phases :

1. Une phase top-down et greedy qui permet de trouver des sous-arbres isomorphes de hauteur décroissante. Des mappings sont établis entre les noeuds de ces sous-arbres isomorphes. Ils sont appelés *mappings ancres*.

2. Une phase bottom-up où les noeuds internes sont mappés quand un grand nombre de leurs descendants ont été mappés ensembles au cours de la phase précédente. Ces mappings sont appelés *mappings conteneurs*. Enfin quand un mapping conteneur est découvert, nous appliquons un algorithme optimal qui ne prend pas en compte les déplacements pour trouver des mappings additionnels parmi les noeuds qui n'ont pas encore été mappés dans les descendants des conteneurs. Ces mappings sont appelés *mappings de dernière chance*.

La figure 1 montre un exemple d'application de notre algorithme sur du code Java. Une description détaillée de l'algorithme est disponible dans l'article originel [3].

Nous avons validé notre algorithme sur des données réelles provenant de programmes open-sources. Les différences produites par GumTree ont notamment été jugées plus souvent faciles à comprendre que celles de Unix diff. Les différentes validations auxquelles nous avons procédé sont disponibles dans l'article originel [3].

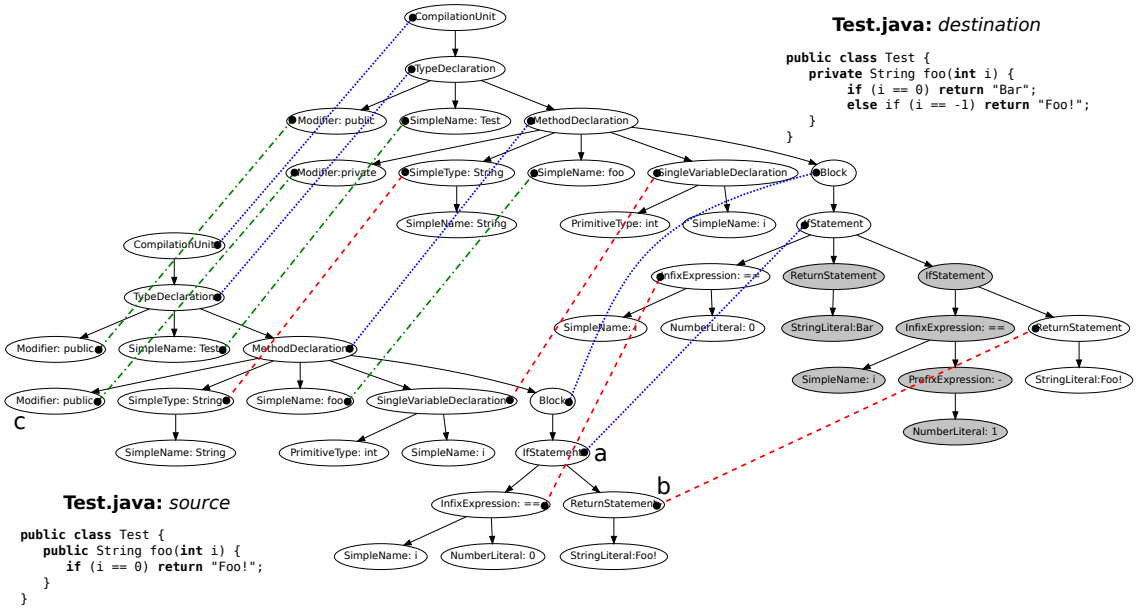


Fig. 1. Deux fichiers Java d'exemple, avec leurs ASTs et les mappings entre les noeuds des ASTs. Les mappings découverts lors de la phase top-down sont dessinés avec les pointillés longs (les descendants de ces noeuds sont aussi mappés, mais ils sont omis de la figure pour plus de clarté). Les mappings découverts lors de la phase bottom-up sont dessinés avec les pointillés courts (pour les mappings de conteneurs) ou des pointillés courts et longs (pour les mappings de dernière chance). Les noeuds non mappés sont dessinés en gris.

References

1. Bille, P.: A survey on tree edit distance and related problems. Theor. Comput. Sci. 337(1-3), 217–239 (2005)
2. Chawathe, S.S., Rajaraman, A., Garcia-Molina, H., Widom, J.: Change detection in hierarchically structured information. In: Proceedings of the 1996 International Conference on Management of Data. pp. 493–504. ACM Press (1996)
3. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014. pp. 313–324 (2014), http://doi.acm.org/10.1145/2642937.2642982

Easing Software Component Repository Evolution*

Jérôme Vouillon[1], Mehdi Dogguy[2], and Roberto Di Cosmo[3]

[1] CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité,
jerome.vouillon@pps.univ-paris-diderot.fr
[2] EDF S.A., Debian Release Team, Debian Project, mehdi@debian.org
[3] Univ Paris Diderot, Sorbonne Paris Cité, PPS, UMR 7126 CNRS, INRIA, roberto@dicosmo.org

Abstract. Modern software systems are built by composing components drawn from large *repositories*, whose size and complexity increase at a fast pace. Maintaining and evolving these software collections is a complex task, and a strict qualification process needs to be enforced to ensure that the modifications accepted into the reference repository do not disrupt its useability. We studied in depth the Debian software repository, one of the largest and most complex existing ones, which uses several separate repositories to stage the acceptance of new components, and we developed `comigrate`, an extremely efficient tool that is able to identify the largest sets of components that can migrate to the reference repository without violating its quality constraints. This tool outperforms significantly existing tools, and provides detailed information that is crucial to understand the reasons why some components cannot migrate. Extensive validation on the Debian distribution has been performed. The core architecture of the tool is quite general, and can be easily adapted to other software repositories.

1 Introduction

Component-based software architectures, maintained in a distributed fashion and evolving at a very quick pace have been popularised by the wide adoption of free and open source software (FOSS). These components are usually made available via a *repository*, which are storage locations from which they can be retrieved. A large variety of repositories are available, ranging from specialised ones for components written in a given programming language, like CPAN, Hackage or PyPI, application-specific ones, like the Eclipse Plugin collection [13], and more general repositories like Maven Central [6] or most GNU/Linux distributions. All these component repositories share the common concern of organising the process of integrating changes: new components are regularly added (Debian grew by more than 8000 packages since the last stable release two years ago), outdated versions are being replaced by more recent ones, and superseded or abandoned components get dropped.

To maintain the quality of a repository, it is necessary to set up a formal process that allows to add, update and remove components in a safe place, where they will be tested and qualified, before moving them to the official repository. For large repositories, this process can only be enacted with the help of automated tools. Existing tools are often unable to cope with all the quality requirements that, depending on the needs of the user community of a repository, may vary from basic unit testing to extensive bug tracking to sophisticated integration tests striving to ensure that components can be combined with each other, a property known as *co-installability* [22]. We have studied in depth the process used to evolve the Debian distribution, which has been in place for more than a decade, managing hundreds of thousands of components, called *packages*, for multiple architectures; since it is open to public inspection, we had access to its formal requirements and we collaborated closely with the Debian Release Team, of which the second author is a member, on solving the problems faced by repository maintainers due to the limitations of the current tools.

The Debian evolution process is organised around three repositories (see Figure 1): *stable*, which contains the latest official release and does not evolve anymore (apart for security and

* This work was partially performed at IRILL, center for Free Software Research and Innovation in Paris, France, http://www.irill.org

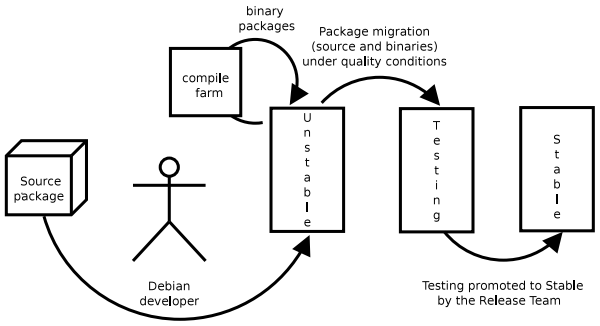


Fig. 1. The Debian process (simplified)

critical updates); *testing*, a constantly evolving repository to which packages are added under stringent qualification conditions, and that will eventually be released as the new *stable*; and *unstable*, a repository in which additions, removals and modifications are allowed under very liberal conditions. A stringent set of requirements, which are formally defined, must be satisfied by packages coming from *unstable* to be accepted in *testing* (also known as *package migration*), and the repository maintainers have responsibility for enforcing them with the help of ad hoc tools. Unfortunately, maintainers are currently all too often confronted with large sets of packages that are stuck in *unstable*, due to complex package interdependencies, with no useful clue to unblock them. A single package can prevent the migration of hundreds of others, and without effective tools to find the culprit, sometimes migration takes months to complete, with a huge amount of manual intervention. In other occasions, the current tool allows into *testing* packages that disrupt co-installability w.r.t. the previous state of the repository, with dire consequences for the users. Indeed, in Debian many applications are split into distinct package that need to be installed together in order to obtain their full functionality, and when this becomes impossible because of an unfortunate modification to the repository, the user need to file bugs, and the maintainers end up spending a significant amount of energy to restore their co-installability.
This article presents `comigrate`, a powerful tool able to efficiently compute maximal sets of packages that can migrate from *unstable* to *testing*: it significantly advances the state of the art by ensuring that no new co-installability issues arise and providing highly valuable explanations for those packages that cannot migrate, helping maintainers find the fix; it is so fast that it can be used *interactively* to narrow down repository issues; it has been validated on various complex migration problems, in collaboration with the Debian Release Team.

The `comigrate` tool is designed using a general architecture, similar to the one of a Sat Modulo Theory solver [19]: it uses a very efficient Boolean solver to quickly identify a large set of packages candidate for migration, starting from general migration criteria that can be encoded as Boolean clauses, and then uses `coinst-upgrade` [23] to look for co-installability issues; when some issues are found, new clauses are added to prevent migration of problematic packages. The process is iterated until a complete solution is found. This general architecture can be reused for building similar tools for other component-based repositories.

The article is structured as follows: in Section 2 we briefly recall basic notions from package based distributions, and give an overview of the Debian integration process with its requirements; Section 3 presents in details an example of how `comigrate` is used to manage a real complex migration; in Section 5, the full architecture of the tool and its algorithm are described; extensions to the algorithm are presented in Section 6; the different ways to use the tools are described in Section 4; Section 7 contains an evaluation of the tool on the Debian integration process; related works are discussed in Section 8 and Section 9 concludes.

2 Package migration in Debian

2.1 Packages and Repositories

Debian software components are called *packages* and come in two flavours: *binary packages* contain the files to be installed on the end user machine, and *source packages* contain all of the necessary files to build these binary packages.

A typical example of the metadata attached to a package is shown in Figure 2, where we can see that the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol '`,`'), disjunctions (symbol '`|`') and version constraints (the symbols '`>=`', '`<=`', '`>>`' and '`<<`' stand for the usual $\geq, \leq, >$ and $<$ operators); it is now well known that checking whether a component is installable is an NP-complete problem [2,4], though real-world instances are tractable [14,21,7,20].

In Figure 2, we find the binary package `ocaml-base` version `3.12.1-4`, which is built from the source package `ocaml` version `3.12.1-4`. Each binary package holds a pointer to its source in the `Source` entry, which may have the same version (in which case only its name is present), or not.

There is a different namespace for source packages and, regarding binary packages, for each architecture, so there can be both a source package and a collection of binary packages named `ocaml`, one for each architecture. In a given namespace, though, there cannot be two packages with the same name and version, and with the notable exception of *unstable*, there can only be one version of a package in any given namespace.

```
1  Package: ocaml-base
2  Source: ocaml
3  Version: 3.12.1-4
4  Architecture: amd64
5  Provides: ocaml-base-3.12.1
6  Depends: ocaml-base-nox, libc6 (>= 2.7), libx11-6, tcl8.5 (>= 8.5.0), tk8.5 (>= 8.5.0),
7    ocaml-base-nox-3.12.1
8
9  Package: ocaml
10 Version: 3.12.1-4
11 Build-Depends: debhelper (>= 8), pkg-config, autotools-dev, binutils-dev, tcl8.5-dev,
12   tk8.5-dev, libncurses5-dev, libgdbm-dev, bzip2, dh-ocaml (>= 1.0.0~)
```

Fig. 2. Inter-package relationships of the `ocaml` source component and `ocaml-base`, one of the binary packages generated from it.

The Debian distribution is huge: we gathered the following statistics on February 28, 2013. The number of binary packages is per architecture; an instance of each of these packages is typically built for the thirteen supported architectures. All these instances have to be considered individually during the migration process.

	source packages	binary packages
unstable	18 768	38 903
testing	17 635	36 404
stable	14 968	29 326

A *healthy installation* is a set of packages in which all dependencies are satisfied and with no conflict. If all is well, the set of packages currently installed on your machine under Debian is a healthy installation. A package is *installable* if there exists at least one healthy installation that contains it. A minimal requirement for a software repository is that all packages are installable. A set of packages are *co-installable* if there exists at least one healthy installation that contains them all. It is normal to have conflicts between some packages, such as for instance between two mail transport agents. Hence, one should not expect all subsets of a software repository to be co-installable. However, when switching to a new version of a software repository, an end user expects

to remain able to use all still supported packages. His installation may need to be modified by removing some packages which are no longer supported and by adding new packages, but having to remove any still supported package should remain exceptional. Hence, any set of packages present in both the old and new version of the software repository and which used to be co-installable should normally remain so.

2.2 Package Integration and Migration

The integration process of new packages, and new versions of existing packages, in the Debian distribution involves two repositories, *testing* and *unstable*, and is very complex. We only provide here a general overview and refer the interested reader to [10] for more details.

When a new version of a source package S is available, it is introduced in *unstable*, and then the corresponding binary packages are gradually built and added to *unstable* as well. When a binary package is rebuilt, it replaces the previous version, and when all binary packages are rebuilt, the old version of S is dropped. Binary packages that are no longer built from the new version of S are dropped as well. Building binary packages can be a long process, because of compilation errors and broken dependencies, so it is possible to find in *unstable* several versions of the same source package, and a mixture of binary packages coming from these different versions of the same source.

After a quarantine period, which is useful to detect critical bugs and ranges from zero to ten days according to the priority of a package, the following actions can be performed:

- *replace* a package in *testing* by a package of the same name and a newer version available in *unstable*;
- *remove* a package that no longer exists in *unstable*;
- *add* a new package from *unstable* in *testing*.

Following the Debian Release Team, we call *migration* of a package any of these operations[4].

2.3 Constraints

Package migrations must satisfy many different quality constraints, of different nature. Some of these constraints can be checked by looking only locally at each set of packages built from a source package:

bug reduction a package with new release critical bugs cannot migrate;
binary with source binary and source packages migrate together;
no downgrades packages that migrate have strictly greater version number.

Others require a global inspection of the repositories to be assessed:

installability a new package accepted in *testing* should be installable;
no breakage other packages in *testing* should not become non-installable;
co-installability sets of packages that were compatible before migration should remain compatible afterwards.

Often, because of these quality constraints, many packages have to migrate simultaneously: about 490 source packages for the lastest migration of Haskell packages, for instance. When such a large set of packages is stuck, it can be very difficult to manually find out which ones have issues. We need tools to find that out.

[4] The only anti-intuitive case being the second one, where the migration leads to suppressing a package from *testing*.

2.4 The Current Migration Tool

The tool currently used in Debian to help the Release Team migrate packages from *unstable* to *testing* is a program named `britney`, which heavily relies on heuristics: it first tries to migrate individual source packages together with the associated binary packages, and then looks for clusters of packages that need to migrate together. Since `britney` is unable to find all possible migrations by itself, it provides a complex *hint* mechanism used by the team to help `britney` find sets of packages that can migrate together (the `hint` and `easy` hints) or, when nothing else goes, to force a migration even when this breaks other packages (the `force-hint` hint).

The `britney` tool only checks for installability of individual packages, and not for co-installability of sets of packages, which can have a significant impact on the quality of the repository as we will see in Section 7. It can also be fairly slow and can use up considerable computing resources, but the main complaint from the repository managers is that it provides little help when a migration does not go through.

In the next section we provide a real example of how our replacement tool, `comigrate`, significantly improves over the current state of the art.

3 Finding Migration Issues

On June 13, 2012, we investigated the `ghc` compiler and other associated Haskell packages: the `britney` tool was unable to migrate the `ghc` source package right away, and this prevented the migration of hundreds of related binary pacakges, a most unsatisfactory situation.
The only information at the disposal of the maintainers was the cryptic output from `britney` that looked like:

```
Trying easy from autohinter: ghc/7.4.1-3 ...
leading: ghc,haskell-explicit-exception,haskell-hxt,...
start: 67+0: i-9:a-2:...
orig: 67+0: i-9:a-2:...
easy: 735+0: i-137:a-74:...
    * i386: haskell-platform, haskell-platform-prof, ...
    * amd64: libghc-attoparsec-text-dev, ...
    ...
FAILED
```

with each line containing hundreds of package names. Apart from the very clear last line that stated that the migration failed, there was no hint of what was going wrong.

```
> comigrate -c britney2.conf --migrate ghc
Package ghc cannot migrate:
 Package ghc: binary package ghc-haddock/i386 cannot migrate.
 Package ghc-haddock/i386: needs binary package libghc-happstack-state-doc
  (would break package libghc-happstack-state-doc):
  - libghc-happstack-state-doc (testing) depends on
    haddock-interface-16 {ghc-haddock (testing)}
 Package libghc-happstack-state-doc/i386: a dependency would not be satisfied
  (would break package libghc6-happstack-state-doc):
  - libghc6-happstack-state-doc depends on
    libghc-happstack-state-doc {libghc-happstack-state-doc (testing)}
```

Fig. 3. Output of `comigrate` explaining why package `ghc` cannot migrate.

Our `comigrate` tool is designed to provide valuable help when a migration cannot be performed: running it on exactly the same data, we got the output given in Figure 3, that conveys concisely

a wealth of information, using abbreviations and conventions that we detail below, and contains a precious starting hint to explain the situation.

Line 1 says that we are attempting to migrate the source package `ghc`, using the configuration file `britney2.conf`, and line 2 shows that it cannot migrate right away. The rest of the output explains why: line 3 tells us that the source package `ghc` cannot migrate because one of the binary packages built from it, `ghc-haddock`, cannot migrate, at least on architecture `i386`.

In turn, line 4 indicates that package `ghc-haddock` cannot migrate unless the binary package `libghc-happstack-state-doc` migrates (the architecture `i386` is omitted, as it is the same as for `ghc-haddock`). The excerpt `(would break package libghc-happstack-state-doc)` right after, in line 5, means that not migrating the two packages simultaneously would make package `libghc-happstack-state-doc` non installable.

Indeed, the version of package `libghc-happstack-state-doc` in *testing* depends on package `haddock-interface-16`[5], and we find in line 6, inside the curly braces, all the packages that can satisfy this dependency. As the binary package `ghc-haddock` from *testing* is the only one appearing in the braces, we know that this dependency is not satisfied by the version of `ghc-haddock` in *unstable*, and migrating `ghc-haddock` alone would render `libghc-happstack-state-doc` uninstallable.

But `libghc-happstack-state-doc` cannot migrate: on line 7 and 8 we discover that its migration would break `libghc6-happstack-state-doc`, and looking at the explanation of this fact on line 9, we see that the dependency `libghc-happstack-state-doc` is only satisfied by the version present in *testing*. This tells us that the migration of `libghc-happstack-state-doc` is actually a removal, and that removing it from *testing* breaks an existing package.

Hence, following these few lines of `comigrate` output we learned that migrating package `ghc` is not possible, as it would break `libghc6-happstack-state-doc`.

So we focus on `libghc6-happstack-state-doc` and looking at it we find out that it is in fact a bit special: it contains no file and is just there to ease upgrades when a package is renamed[6]. Following the history of modifications to this package, we discover that at some point in time all the `libghc6-*` packages were renamed into `libghc-*` packages and a single source package `haskell-dummy` was introduced in the distribution to build all the corresponding dummy packages whose role was just to make sure that if somebody needs a `libghc6-*` package, he will actually pull in the corresponding `libghc-*` one. But over time, some `libghc-*` stopped being supported and were removed from *unstable*, while the corresponding `libghc6-*` package were still generated by `haskell-dummy`. It thus seems worthwile to try removing this source package: for simplicity, we focus on the `i386` architecture.

```
> comigrate -c britney2.conf --arches i386 \
      --migrate ghc --remove haskell-dummy
Successful:
age-days 7 haskell-bindings-libzip/0.10-2
# source package xmonad/0.10-4: fix bugs #663470
# source package haskell-cryptocipher/0.3.3-1: fix
#   bugs #674811
age-days 9 haskell-platform/2012.2.0.0
easy [...]
```

This time, the tool has been able to find a way to perform the migration: it is enough to fix two bugs, and to wait for two packages to become old enough.

Making the migration go through on all architectures requires some extra effort, because some binary packages were not built successfully everywhere and have to be removed as well. At the end, we get a list of packages to remove and a large `britney` hint (489 source packages) which

[5] This is actually a *virtual package*, a named functionalities that can be provided by more than one package, and on which other packages may depend.

[6] This is known as a *transitional dummy package*, see `http://wiki.debian.org/Renaming_a_Package`

make it possible to migrate the `ghc` package. We posted this information on the Debian-release mailing list[7] and it helped successfully migrate the whole set of packages involved.

This concrete example shows how `comigrate` can be used to progressively (depending on the complexity of the migration problem) understand the actions needed to make packages go through.

4 Main applications

The `comigrate` tool we have seen at work in Section 3 is highly flexible and very fast (see Section 7 for real world benchmarks). This allows it to be be used for package migration in several ways.

4.1 Automatic Package Migration

By default, `comigrate` computes the largest set of packages that can migrate without breaking any other package or violating any of the constraints defined in the Debian migration process. In this modality, `comigrate` can output the list of packages that should be in *testing* after migration. This allows it to be used as a drop-in replacement for the `britney` tool for automatic package migration. As an option, `comigrate` can spend extra effort, as explained in Section 6.1, to split the result in clusters of packages that can migrate *independently of each other*. This makes it easier for humans to understand which package can migrate, which packages need to migrate together, and allows maintainers to finely control the process if needed.

Clustered results can also be fed directly into `britney` in the form of `easy` hints, easing the adoption of `comigrate` that can then fruitfully coexist with `britney` for a period.

Sometimes, there are good reasons for some packages to become incompatible after a migration, thus violating the preservation of co-installability. To this end, one can use the `--break` directive, followed by a list of one or more binary packages (for instance, `gnuplot-x11,gnuplot-nox`), to specify that these packages, and *just these packages*, are allowed to become non co-installable. An underscore can be used as a wildcard: package `libjpeg62-dev` was at some point superseded by the incompatible package `libjpeg8-dev`, and one can write `--break libjpeg62-dev,_` to specify that it is allowed to become in conflict with any other package. This option provides a means of controlling which set of packages become non co-installable which is way more precise than other approaches, like the `force-hint` hint of `britney`.

It can also be useful to temporarily remove a package that prevents the migration of other more important packages. In fact, this is a common way to guide package migration. To this end, one can use the `--remove` directive, which corresponds to the homonymous hint of `britney`. Followed by a source package name, it makes the tool behave as if this source package and all its associated binary packages had been removed from *unstable*: `comigrate` will perform the removal of these packages together with the expected migration if this preserves co-installability. An updated version of the removed packages can be put back by a subsequent run of the tool, but in this case, for `comigrate`, these will be new packages, so it will only guarantee that they are installable.

4.2 Explaining Migration Failures

For the packages that cannot migrate, it is very important to provide concise and informative explanations on the reasons that block them. When used with the `--excuses` directive, `comigrate` takes the set of constraints generated by a migration attempt, and outputs an HTML report presenting them in a user-friendly way, giving for each package the precise reasons that prevent its migration. In particular, a graph like the ones shown in Figure 5, in SVG format, is generated for each co-installability issue detected. As we have seen in the real-world example of Section 3, these explanations are precious for finding a way to unblock a migration.

It is important to detect co-installability issues as soon as possible, so that issues regarding a newly introduced package can be immediately brought to the attention of the maintainers, and

[7] `http://lists.debian.org/debian-release/2012/06/msg00317.html`

not only after the quarantine period of ten days. Hence, the migration algorithm is run initially while omitting age and bug constraints, so as to collect as much constraints due to co-installability issues as possible.

4.3 Focusing on a Given Package

As described in Section 3, one can use the `--migrate` directive to focus on the migration of a particular source package. In this modality, `comigrate` drops progressively the less important constraints (age, number of new bugs, out of date packages in *unstable*) that prevents this migration, until either the migration succeeds, or a hard reason to refuse the migration is reached. In case of success, the hints overriding these constraints and the list of packages that need to migrate together are printed. Otherwise, the constraints preventing the migration are printed.

By adding the `--break` and `--remove` directives described above, the user can interact with `comigrate` to get a clear view of all migration issues related to this particular package and find the best possible course of action in order to allow a given source package to migrate.

In conclusion, `comigrate` is a sophisticated and flexible tool that allows a full range of modes of operation, from automatic package migration, to fine grained, interactive analysis of the reasons why a package cannot migrate, and progressive relaxation of the migration constraints when needed.
Now it's time to look at the internals of `comigrate` and describe how it works.

5 Core Architecture of the tool

As explained in Section 2.3, the constraints imposed on migration can be broadly separated in two classes. The first one contains constraints that can be easily expressed using Boolean clauses (disjunctive Boolean formulas). Examples of such clauses are "a binary package cannot migrate without its source", or "this binary package can only migrate if these two other binary packages migrate as well." The second class contains constraints that cannot be easily encoded using Boolean clauses, and generally need a global analysis of the repository to be checked: ensuring that a package that migrates to *testing* does not break existing packages requires checking installability for all the packages in the candidate new version of *testing*; ensuring that co-installability is preserved by the migrations needs a special and sophisticated algorithm to be checked efficiently, which is implemented in a separate tool `coinst-upgrade` described in details in [22].

This is a quite interesting situation, that can be handled by using an architecture inspired by the architecture of SMT solvers, and summarised in Figure 4. To find a migration solution for

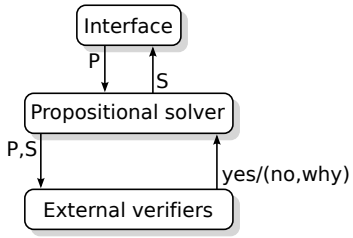


Fig. 4. Solver architecture used in `comigrate`

a particular repository configuration P, `comigrate` uses a simple and fast Boolean solver on the constraints that are easily encoded as Boolean clauses and comes up with the largest possible candidate solution S satisfying them all; then, a series of external verifiers are used to check

whether other constraints, not easily encoded as Boolean clauses, are satisfied. If this is not the case, these verifiers return an *explanation*, which is used to *learn* some extra Boolean clauses that approximate soundly the constraints, and are added to the original problem, on which the boolean solver is called again, producing a smaller solution. The process is iterated until a valid solution is found. In our case, there is always at least one solution: performing no migration. Each learnt clause forces at least one additional package not to migrate. Thus, the process eventually terminates.

We now detail each component of this core architecture.

5.1 The Boolean Solver

The encoding of the migration problem uses one Boolean variable per package, with the intended meaning that if the variable is true, the package cannot migrate, and if the variable is false, it can migrate. We have thus to deal with hundreds of thousands of variables (one per package), and the encoding of a typical Debian migration problem is quite big (1 097 490 clauses in the migration on December 18, 2012), so the choice of the Boolean solver has to be done carefully.

We observed that the clauses in the encoding that express constraints like "if some set of packages cannot migrate, then a given package cannot migrate" or assertions like "this package cannot migrate" are actually Horn clauses (Boolean clauses with at most one positive literal). For instance, the encoding of the fact that the binary package `ocaml-base` in Example 2 needs to migrate together with its source `ocaml` is the pair of clauses:

$$\neg\texttt{ocaml} \vee \texttt{ocaml-base}, \qquad \texttt{ocaml} \vee \neg\texttt{ocaml-base}.$$

Indeed, with the exception of the constraints that come from the external verifiers, all clauses in the encoding are Horn clauses, and for this reason, in the current version of the tool, we use a Horn clause solver, that has the following important advantages:

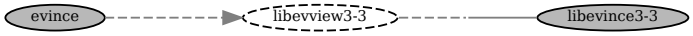
- simplicity: we just need to implement resolution (if the hypotheses of a rule hold, then so does its conclusion); no backtracking is needed;
- speed: it is well known that satisfiability of Horn clauses can be checked in linear time [12];
- optimality: if a set of Horn clauses is satisfiable, then there exists a minimal solution setting to true the least possible number of variables, and which can also be found in linear time [12]; this means that the Horn solver will propose the largest possible migration compatible with the constraints;
- flexibility: it is possible to remove some clauses and update the solver state incrementally to find a new minimal solution;
- easy explanation of why a variable is set: we can explain why a package cannot migrate by printing the tree of Horn clauses which justify it.

In the rare cases when the external verifiers return a Boolean clauses that is not a Horn clause, we approximate it using a stronger Horn clause and when it happens we may not find the optimal solution for the migration. As we will see in Section 7.1, this happens so rarely that it is not an issue in practice, and we have not felt the need to switch to a more sophisticated solver.

In any case, we do not envision any difficulty in replacing this solver with a full blown PMAX-SAT solver or pseudo-Boolean solver, able to handle all kind of Boolean clauses, with the addition of an optimisation function to maximise the number of packages that migrate. The performance should remain good: the fact that we are able to find optimal solutions with the Horn solver means that little backtracking will be needed. But its a significant engineering effort to integrate such a solver with our tool, in particular to still be able to provide readable explanations.

5.2 The External Verifiers

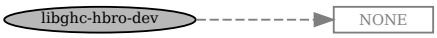
Our Boolean solver specialised for Horn clauses finds a solution that maximises the number of migrating packages given the constraints it knows about. This solution corresponds to a candidate



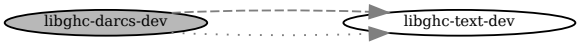
(a) Package `evince` cannot migrate as it depends on a new library which conflicts with the old `libevince3-3` library; this latter library should be removed.

(b) Package `mysql-common` cannot migrate as it conflicts with a previous version of `mysql-server-core`; this old version should be removed.

(c) Package `smarty` cannot be removed as package `piwigo` still depends on it; `piwigo` needs to be removed as well.



(d) Package `libghc-hbro-dev` cannot migrate as it has unsatisfied dependencies; it probably depends on a package that has been updated, and should be recompiled.



(e) Package `libghc-darcs-dev` in *testing* (resp. *unstable*) depends on package `libghc-text-dev` in *testing* (resp. *unstable*), hence both packages must migrate together.

Fig. 5. Examples of full explanations.

replacement repository for *testing* that lies in-between *testing* and *unstable*, and we need to check whether it contains violations of the second class of quality constraints described in Subsection 2.3: new co-installability issues and new packages that are not installable (old packages that become non installable are an instance of co-installability issues). The verification is performed independently on each of the thirteen architectures supported by Debian, and all the issues found are then used to learn new clauses which are added to the constraints known to the Boolean solver. We detail below the approach taken for each of the two classes of constraints.

New co-installability issues The `coinst-upgrades` tool is used to find all co-installability issues introduced in the candidate repository with respect to *testing*. This tool is extensively described in [23], and we just recall here that it takes as input an old and a new repository and computes what is called a *cover* of *broken sets* of packages, which concisely subsumes all co-installability issues introduced in the new repository. A *broken set* is a set of packages which are co-installable in the old repository but no longer in the new one. A *cover* is a collection of *broken sets* such that any healthy installation of packages from the old repository that cannot be successfully upgraded contains at least one of these sets of packages.

All healthy installations can be successfully upgraded if and only if there exists a (unique) empty cover, in which case the check is successful. Otherwise, each broken set is analysed in order to generate a clause to be added to the set of constraints known to the Boolean solver. This analysis was not performed by `coinst-upgrades` and had to be implemented for `comigrate`. As a first step, we extract a small graph of dependencies and conflicts, called *full explanation* in [23], that summarises why the set of packages becomes non co-installable in the candidate repository. Some examples of these explanations, drawn from actual migrations in Debian, are given in Figure 5; they were produced using our tool, with the `--excuses` directive (see Section 4.2 for details). The captions indicate what the tool learns, then how each issue can be fixed by the distribution maintainers. In these graphs, colored packages are the elements of the broken set. Dependencies are represented by arrows, and conflicts by lines connecting two packages. Packages, dependencies and conflicts are drawn with different styles: solid lines indicate an object present in both the old and new repositories, dashed lines indicate an object present only in the new repository, and dotted lines indicate an object present only in the old one. Thus, for instance, on the first graph, `evince` depends in *unstable* on package `libevview3-3` which is only in *unstable*. On the second graph, `mysql-common` is present both in *testing* and *unstable*, but the version in *unstable* conflicts with `mysql-server-core-5.1` both in *testing* and *unstable*.

These small graphs explain why the proposed migration creates a co-installability issue, but do not indicate which parts of the proposed migration are the root cause of the problem. For example, looking at the second graph, we know that the boolean solver has proposed to mi-

grate both `mysql-common` and `mysql-server-core-5.1`, and that this creates an issue. But it would be inefficient to just conclude that they cannot migrate together and learn the clause `mysql-common`∨`mysql-server-core-5.1`. Indeed, one can see that the actual reason for this problem is the migration of `mysql-common`, as it conflicts with all versions of `mysql-server-core-5.1`, and we can learn the much more informative clause `mysql-common`, which also happens to be a Horn clause, while the first one was not.

To extract a clause from an explanation, we proceeds by iteratively relaxing the constraints on the individual packages in the explanation (allowing some packages to come from either *testing* or *unstable* rather than just from the candidate repository) until we have a minimal set of constraints (packages forced to come from *testing* or *unstable*) that still make the co-installability issue appear. Then, we know that for the co-installability issue to disappear, at least one of the corresponding packages should not take part in the migration. This gives us the Boolean clause we will pass back to the Boolean solver.

If we apply this process on the first four graphs shown in Figure 5, we learn that `evince`, `mysql-common`, `smarty` and `libghc-hbro-dev` should not migrate, which all provide unit clauses. In the last graph, where the proposed migration contains the new version of `libghc-darcs-dev` and the old version of `libghc-text-dev`, the clause is already minimal, and we learn that either `libghc-darcs-dev` should not migrate or `libghc-text-dev` should migrate.

Since the verifier finds co-installability issues caused by the proposed migrations, each Boolean clause contains at least one positive literal, which leads to removing the corresponding package from the migration candidates (a variable set to true means that the corresponding package cannot migrate). If there is a single positive literal, the clause is a Horn clause, and can be passed back to the Boolean solver as is. If there are more than one positive literal (examples are given in Section 7.1), we approximate the clause by only keeping a single positive literal: we know that several packages cannot migrate together and make the decision to keep one back arbitrarily, which may lead to suboptimal solutions.

In order to reduce the risk of making suboptimal choices, we delay any choice resulting in information loss by ignoring non-Horn clauses as long as the analysis of the architecture produces at least a Horn clause.

New non installable packages A SAT solver is used to decide which new packages of the candidate repository are not installable. From the output of this solver, we can produce for each non-installable package an explanation of the same shape as the ones produced by `coinst-upgrades`. Then, Boolean clauses can be computed in exactly the same way as described previously.

5.3 Speeding Things Up

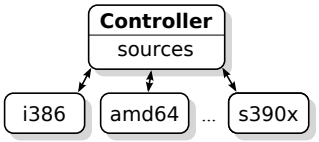


Fig. 6. Parallel structure of `comigrate`.

Since the verifiers work independently on 13 different architectures, performance can be greatly improved by using multiple processes as depicted in Figure 6. A main process handles source packages and runs the Horn clause solver. There is one secondary process per architecture. It parses the per-architecture binary package description files and sends to the main process the constraints corresponding to these packages; after that, it behaves as an external verifier for its specific architecture, reading the candidate solution from the main process and sending back clauses to be

learnt when issues are found.

6 Extensions

The algorithm described up to now produces the largest possible migration candidates, except possibly in the rare cases where one non-Horn clause was produced by the external verifiers. It ensures the absence of *co-installability* issues. This core algorithm has been extended to accommodate two additional needs, described below, that lead to interesting variations in the results obtained.

6.1 Clustering Migrating Packages

Instead of writing out the whole set of packages that can migrate as a single huge `easy` hint, an effort is made to cluster packages in sets of packages which can migrate independently of one another, in any order. This approach makes it easier for a human being to understand what a particular migration does; smaller sets are also more resilient to last-minute changes to the repositories while the migration is computed.

Each source package define a group of packages: the source package and its associated binary packages. We need a way to find out which groups need to migrate together to avoid co-installability issues. The idea is to encode in package dependencies all possible configurations of group migrations, and then use an approximation of the `coinst-upgrade` algorithm to find out which configurations may result in co-installability issues. Package dependencies are thus annotated with special literals that indicate whether they hold when a group migrates, or when it does not. For instance the dependency $\texttt{g/old} \vee \texttt{h/new} \vee d$ says that the dependency d must be satisfied when group `g` does not migrate (we have the old version of the group) and group `h` does (we have the new version). We know from [23] that to have a co-installability issue, one must have either a new conflict or a set of new dependencies (which did not exist in *testing*) connected through conflicts. Thus, to avoid co-installability issues, it is enough to put together the groups of packages connected by a new conflict as well as those which occur in pairs of new dependencies connected through a conflict.

This gives us a collection of independent migrations that correspond to the original global migration. We do not claim the result to be minimal (grouping together packages as we do is a sufficient condition, but not a necessary condition to avoid co-installability issues), although in practice the results are quite satisfactory.

As seen in Section 4, `comigrate` can output its results in *hint* format so that it can be used as an external migration solver to help out `britney`. A so-called `easy` hint lists source packages that `britney` should attempt to migrate simultaneously together with their associated binary packages[8].

6.2 Preserving Just Installability

It is possible to run the `comigrate` with an option that makes it preserve just package installability instead of co-installability, even though this is not a good idea in general, as the users of the repository may face serious problems when co-installability is not preserved (see Section 7).

Due to the architecture of the tool, this is just a matter of replacing the external verifier based on `coinst-upgrades`, which computes a collection of sets of packages which are no longer co-installable, with an external verifier that computes a collection of singleton sets of packages which are no longer installable, using for instance the now standard `edos-debcheck` tool [14].

[8] A new version of some binary packages are sometimes built on a given architecture without changing their source package; their migration can be specified by giving a pair formed of the source package name and the architecture. We ignore this case here for the sake of clarity.

Checking installability may seem a simpler task than checking co-installability, but in fact, surprisingly, this is not the case, and running `comigrate` in such a mode takes longer than in the default mode. Indeed, non co-installability is a property which is more local than installability. For instance, when a conflict is added between two packages `p` and `q`, it is sufficient to report the broken set $\{\texttt{p}, \texttt{q}\}$ when checking co-installability; on the other hand, when looking for new installability issues, one may need to check the installability of the possibly huge number of packages that depend, directly or not, on these two packages to see if any needs both. So, in our case, it turns out that asking for a less precise analysis will actually take more time, even if not prohibilitively more so.

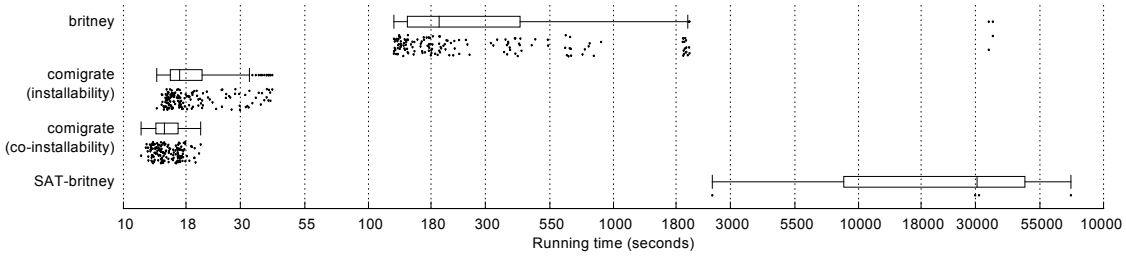


Fig. 7. Performance comparison of the different migration tools, on a logarithmic scale. We use standard box plots: the central vertical bar is the median time, the rectangle spans from the first to the third quartiles (it contains half the data), whiskers denote the largest and smallest data within 1.5 times the interquartile range, outliers are represented as isolated dots. Actual data are plotted just below with some vertical jitter.

7 Evaluation and Validation

We have performed a number of experiments and analyses to assess `comigrate`, comparing it with both `britney`, the official tool used in Debian today, and SAT-britney[8], an experimental tool based on a satisfiability solver.

We wanted to check the following points. First, to replace `britney`, our tool should be at least as good at migrating packages; our systematic approach should ensure that our tool can deal with complex migration situations where `britney`'s heuristics fail, but the fact that we do not use a complete Boolean solver could be a concern. Second, a key novelty of our tool is that it can be used interactively to troubleshoot migration issues; it should be fast enough for this task: less than one minute is fine, more than ten minutes is way too slow. Third, our tool is able to perform more stringent checks than `britney` or SAT-britney, preserving not just package installability but also co-installability; we need to verify that this is indeed useful and that the amount of real issues found outweighs the burden of false positives.

To check all this, we have taken a snapshot of all the information needed for package migration twice a day for about two months, from June 24th, 2013 to September 8th, 2013[9] (which gives 152 samples) and we have run the three tools on these configurations. We have also investigated co-installability issues for a longer period, between January 2010 and June 2012 (start of the freeze period), using the historical information on the state of the Debian repositories which is publicly available through the Debian snapshot archive [9]. We could not reproduce accurately migrations over this longer period as some of the required information (number of bugs, package ages, ...) is not archived.

[9] On June 15th, 2013, after a *freeze* period started in June 2012 during which most migrations were blocked, Debian released its new *stable* repository, and migration were allowed again, so we finally had new, fresh data to analyse.

7.1 Tool Comparison: Migrations

We have compared the quality of the migrations performed by `britney` and `comigrate` for the 152 situations mentioned above. To get a meaningful comparison, `comigrate` has been configured with the option to only check for installability. We have not been able to run SAT-britney on all the architectures simultaneously, as its memory usage exceeded the eight gigabytes of RAM available on our testing machines. Hence, we are not able to provide a meaningful comparison with the output of SAT-britney.

We found out that `comigrate` almost always migrates more packages than `britney`, which failed on eleven occasions to find a suitable set of packages that had to migrate together. In fact, we found a single corner case where `comigrate` migrates less packages than `britney`: when the source used to build a binary package changes, `britney` sometime removes the package from *testing* without waiting for the new version of the package to migrate from *unstable*. In this situation, the extra migration makes the package unavailable for some time, and `comigrate` rightfully refuses to allow it. As a remarkable example, the arrival of KDE 4.10 in *testing* required to migrate 138 packages at once: our tool was able to find automatically which packages had to migrate simultaneously, while a `force-hint` was used by the Debian Release Team to bypass `britney`'s checks and make the packages go through.

We could also verify in all of the 152 runs that `comigrate` never made any suboptimal choice due to the limitations of the solver.

7.2 Tool Comparison: Execution Time

The execution times taken by each tool to compute package migrations, on a 8 GiB desktop computer using an Intel Core i7-870 at 2.93GHz, are shown in Figure 7. The `comigrate` tool takes reliably well less than a minute to compute possible package migrations when only checking for installability issues across upgrades, and is quite faster in the default mode that also checks for co-installability. The `britney` tool is usually one order of magnitude slower, but can occasionally take much longer to complete: we observed a maximum of eight hours over the two month period. We could run SAT-britney on only seven of the thirteen architectures. It is much slower than the other tools, even when running on this limited subset of the distribution, to the point that we aborted the experiment after just a few runs.

7.3 Relevance of Preserving Co-installability

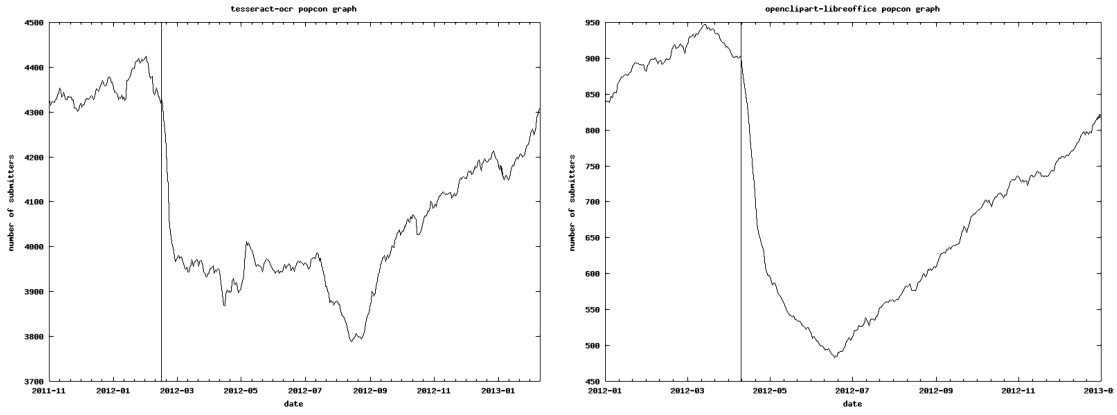


Fig. 8. Impact of not preserving co-installability.

We have compared the output of `comigrate` while checking for co-installability or just installability in the 152 situations mentioned above. We comment shortly on the result. A few packages

(libphonon4, liboss4-salsa-asound2 and libgl1-mesa-swx11) are incompatible with a huge number of other packages, and there is no point in keeping them co-installable with other packages. A break directive can be added once and for all to deal with them. Often, when switching to a new version of a library, the packages of the old version are kept for some time in *testing*. If there is any incompatibility between these packages, comigrate will report many harmless conflicts during the transition. This can be avoided by using break directives to indicate that the old library is deprecated. We encountered five such transitions. In some case, lots of conflicts are introduced. For instance, on September 10th, there were 16 packages depending on libopenmpi1.6 and 172 packages depending on the conflicting package libopenmpi1.3. It is thus not possible to use any of the 16 packages with any of the other 172 packages. Temporarily introducing incompatibilities this way can sometimes be justified in order to ease package migration, but britney does not provide any way to control precisely what incompatibilities are introduced, and member of the Release Team are sometimes not even aware of them. Genuine issues can justify new conflicts between packages. We counted ten of them. For instance, package quantum now conflicts with python-cjson as the latter is a seriously buggy JSON encoder/decoder; package parallel conflicts with moreutils as they both provide a binary of the same name but with a different semantics (though, in fact, it is not clear this is allowed at all in Debian); package lxsession now replaces package lxpolkit and therefore conflicts with it. Sometimes, maintainers can introduce bugs when attempting to fix this kind of issue: the maintainer of colord meant to add a conflict with previous versions of colorhug-client as the packages contained a common file, which is not allowed, but he added a conflict with the current version as well. In fact, we encountered three instances of overly strong dependencies or conflicts such as the colord one. In four cases, the conflicts preventing migration was with a broken or obsolete package that should be removed: mingw-w64 conflicts with libpthreads-mingw-w64, kde-window-manager conflicts with kwin-style-dekorator. Finally, in six cases, a package migrated when it should have waited for another package remaining in *unstable*. In particular, package bandwidthd, meant to be used with a Web server, is in conflict with the version of Apache in *testing*; the FreeBSD kernel kfreebsd-image-9.1-1-amd64 (kfreebsd-amd64 architecture) is incompatible with the bootloader grub-common; package scrapbook, a Firefox extension, is incompatible with Firefox.

We have also checked all co-installability issues introduced each week over two years and a half from January 2010 to June 2012. We can find these issues thanks to our previous tool coinst-upgrades [23], and have assessed their impact. They were most often due to britney only checking for installability, rather than being introduced by the Release Team by forcing the migration of a package: we verified this by checking that the culprit packages could migrate without help. We observed that they were real issues in the large majority of cases, where britney migrated a package that did not break installability of any other package, and yet made an application unusable. We found about fifteen occurrences of a package migrating in isolation when it should have migrated together with a new version of one or more other packages. In some cases, packages remained kept back in *unstable* for some time, with a disruptive impact on Debian users. We could verify the importance of this impact by analysing the statistics of Debian package installations automatically collected from the users of the popularity-contest package, and for which an historical archive is available. The two graphs in Figure 8 show how many users of the package popularity-contest have the packages tesseract-ocr and openclipart-libreoffice installed, over a period of time before and after the disruptive migration takes place. The vertical bar indicates the date of migration of a new version of respectively tesseract-ocr-eng (as well as other similar packages) and openclipart-libreoffice. These packages conflicts with the versions of their companion packages tesseract-ocr and libreoffice-common in *testing*, while the new versions of these latter packages remained kept back in *unstable*. One can clearly see that this wrong migration caused tesseract-ocr and openclipart-libreoffice to be removed on a large fraction of the user base. As a consequence, the application tesseract that is split in a common core and a series of language packs became unusable; similarly the Open Clip Art Library could not be used in integration with libreoffice.

Overall, checking for co-installability is a significant improvement: while there is some additional work when new conflicts are justified, it does not harm to double check them, and one gains

better control on which incompatibilities are introduced, allowing to find a significant number of bugs.

7.4 Validation with the Debian Release Team

A close connection with the Debian Release Team has been established over the past year, and `comigrate` has been shown to consistently provide valid results, and very useful information to track down the reasons why large package sets cannot migrate (the case shown in Section 3 is an actual example). It is now being seriously considered for inclusion in the Debian release management process.

8 Related Work

Ensuring the correct behaviour of components when composed into an assembly is a fundamental concern and has been extensively studied. One may detect automatically behavioural incompatibilities from the component source code [15,16], or deploy and upgrade such systems [5]. Inter-module dependencies have been used to predict failures [17,26,17,18], identifying *sensitive* components [26] or used as a guideline for testing component-based systems [24,25].

Maintaining large collections of interrelated software components, and in particular GNU/Linux distributions, poses new challenges, that only recently become subject of research. We know how to identify what other components a package will always need [1] and what pairs of packages are incompatible [11]; sophisticated algorithms allow to answer all the questions about package co-installability [22]. We can identify the component upgrades that are more likely to break installability in a repository [3], and efficiently identify the co-installability issues introduced in a new version of a repository [23]. Package migration is encoded as a PMAX-SAT instance in the `SAT-Britney` tool [8], which only ensures preservation of installability.

9 Conclusions

We developed `comigrate`, an efficient tool able to compute migration candidates for evolving the Debian repositories according to the stringent quality constraints imposed by the policy, ensuring that no co-installability issues arise, and providing concise and precise explanations when some packages cannot migrate. The unique combination of speed and explanations enable for the first time to perform interactive tuning of the migration process, even when hundreds of interrelated packages are involved. Extensive testing and validation has been performed on real-world problem instances, in collaboration with the Debian release team.

The core architecture of `comigrate` is particularly suitable for handling migrations. It should be straightforward to adapt it to any other integration process where the quality requirements are composed of a mixture of simple constraints and complex conditions verified by external tools.
Source Code is at `http://coinst.irill.org/comigrate`

References

1. P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli. Strong dependencies between software components. In *ESEM*, pages 89–99. IEEE Press, Oct. 2009. `doi:10.1109/ESEM.2009.5316017`.
2. P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Dependency solving: a separate concern in component evolution management. *Journal of System and Software Science*, 85(10):2228 – 2240, 2012. Automated Software Evolution. URL: `http://www.sciencedirect.com/science/article/pii/S0164121212000477`, `doi:10.1016/j.jss.2012.02.018`.
3. P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. Learning from the Future of Component Repositories. In *15th International ACM SIGSOFT Symposium on Component Based Software Engineering (CBSE-2012)*, Bertinoro, Italie, June 2012. ACM. `doi:10.1145/2304736.2304747`.

4. P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli. A modular package manager architecture. *Information and Software Technology*, 55(2):459 – 474, 2013. Special Section: Component-Based Software Engineering (CBSE), 2011. URL: http://www.sciencedirect.com/science/article/pii/S0950584912001851, doi:10.1016/j.infsof.2012.09.002.
5. S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *ECOOP*, pages 452–476, 2006. doi:10.1007/11785477_26.
6. Apache Software Foundation. Guide to uploading artifacts to the central repository. [Online; accessed 1-March-2013], 2013. URL: http://maven.apache.org/guides/mini/guide-central-repository-upload.html.
7. D. L. Berre and A. Parrain. On sat technologies for dependency management and beyond. In *SPLC (2)*, pages 197–200, 2008.
8. J. Breitner. Tackling the testing migration problem with SAT-solvers, 2012. arXiv:1204.2974.
9. Debian Project. The debian snapshot archive. [Online; accessed 1-March-2013], 2013. URL: http://snapshot.debian.org/.
10. Debian Project. Debian "testing" distribution. [Online; accessed 1-March-2013], 2013. URL: http://www.debian.org/devel/testing.
11. R. Di Cosmo and J. Boender. Using strong conflicts to detect quality issues in component-based complex systems. In *ISEC '10: Proceedings of the 3rd India software engineering conference*, pages 163–172, New York, NY, USA, 2010. ACM. doi:10.1145/1730874.1730905.
12. W. F. Dowling and J. H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *J. Log. Program.*, 1(3):267–284, 1984. doi:10.1016/0743-1066(84)90014-1.
13. Eclipse Foundation. Eclipse marketplace. [Online; accessed 1-March-2013], 2013. URL: http://marketplace.eclipse.org/.
14. F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen. Managing the complexity of large free and open source package-based software distributions. In *Automated Software Engineering (ASE)*, pages 199–208, 2006. doi:10.1109/ASE.2006.49.
15. S. McCamant and M. D. Ernst. Predicting problems caused by component upgrades. In *ESEC / SIGSOFT FSE*, pages 287–296, 2003. doi:10.1145/940071.940110.
16. S. McCamant and M. D. Ernst. Early identification of incompatibilities in multi-component upgrades. In *ECOOP*, pages 440–464, 2004. doi:10.1007/978-3-540-24851-4_20.
17. N. Nagappan and T. Ball. Using software dependencies and churn metrics to predict field failures: An empirical case study. *ESEM 2007*, pages 364–373, 2007. doi:10.1109/ESEM.2007.13.
18. S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of CCS 2007*, pages 529–540, 2007. doi:10.1145/1315245.1315311.
19. R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving sat and sat modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(t). *J. ACM*, 53(6):937–977, 2006.
20. P. Trezentos, I. Lynce, and A. L. Oliveira. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Automated Software Engineering (ASE)*, pages 427–436, 2010.
21. C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. Opium: Optimal package install/uninstall manager. In *International Conference on Software Engineering (ICSE)*, pages 178–188, 2007.
22. J. Vouillon and R. Di Cosmo. On software component co-installability. In *SIGSOFT FSE*, pages 256–266, 2011. doi:10.1145/2025113.2025149.
23. J. Vouillon and R. Di Cosmo. Broken sets in software repository evolution. In *International Conference on Software Engineering (ICSE)*, 2013. URL: http://www.pps.univ-paris-diderot.fr/~vouillon/publi/upgrades.pdf.
24. I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Direct-dependency-based software compatibility testing. In *Automated Software Engineering (ASE)*, pages 409–412, New York, NY, USA, 2007. ACM. doi:10.1145/1321631.1321696.
25. I.-C. Yoon, A. Sussman, A. Memon, and A. Porter. Effective and scalable software compatibility testing. In *Proceedings of ISSTA '08*, pages 63–74, New York, NY, USA, 2008. ACM. doi:10.1145/1390630.1390640.
26. T. Zimmermann and N. Nagappan. Predicting defects using network analysis on dependency graphs. In *International Conference on Software Engineering (ICSE)*, pages 531–540. ACM, 2008. doi:10.1145/1368088.1368161.

On Analyzing the Topology of Commit Histories in Decentralized Version Control Systems

Marco Biazzini
INRIA — Bretagne Atlantique
marco.biazzini@inria.fr

Martin Monperrus
Université de Lille
martin.monperrus@univ–lille1.fr

Benoit Baudry
INRIA — Bretagne Atlantique
benoit.baudry@inria.fr

***Abstract*—Empirical analysis of software repositories usually deals with linear histories derived from centralized versioning systems. Decentralized version control systems allow a much richer structure of commit histories, which presents features that are typical of complex graph models. In this paper we bring some evidences of how the very structure of these commit histories carries relevant information about the distributed development process. By means of a novel data structure that we formally define, we analyze the topological characteristics of commit graphs of a sample of GIT projects. Our findings point out the existence of common recurrent structural patterns which identically occur in different projects and can be consider building blocks of distributed collaborative development.**

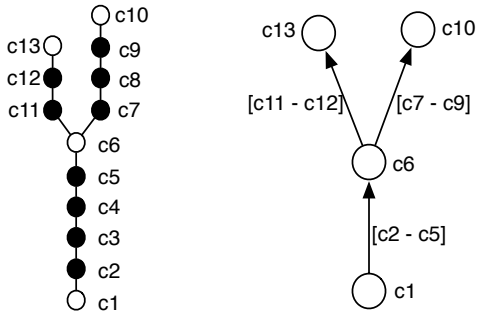


Fig. 1. Example of commit history to *Metagraph* transformation

I. INTRODUCTION

Traditional version control systems (CVCS) such as CVS or SVN have a rather sequential history. There is a linear history of committed change sets (or commits), stored on a central server. Branching makes up for some parallelism, but its capability to improve successful collaboration has been thoroughly questioned [1].

On the contrary, the history of Decentralized Version Control Systems (DVCSs) such as Mercurial or GIT is much richer. In decentralized version control systems, each developer has the full commit history of the codebase locally available. This allows for a much more flexible way to handle different concurrent branches. The extreme ease of branch manipulation and combination results in software histories that are much more complex than what is typically found in a traditional centralized systems software.

Bird and colleagues have shown that the analysis of DVCS presents several issues for traditional mining repository techniques and metrics [2], In DVCSs, there are concurrent branches that start from a common ancestor, run in parallel, split, merge together or with other branches, then are finally merged in a commit which gives a "current stable version" of the software product. Such a canopy of development lines demands to redefine concepts such as "developer collaboration", "current state of the code", "amount of contribution", which are fairly easy to establish for CVCSs.

In this paper, we propose to use graph concepts and metrics to characterize modern decentralized version control systems such as GIT. We consider commit histories of DVCSs as directed acyclic graphs (DAGs) and talk about software development using general graphs characterization (topology, patterns, metrics, etc.). The structure of these DAGs is given by the way developers use branches. The topological structures reflect the way the software is modified, improved, fixed, in one word, evolved.

Thus there is information hidden in the very structure of the commit history graph, which is relevant to understand key characteristics of the development process typical of DVCSs. In this paper we show that the topology of commit graphs presents recognizable patterns, whose recurrence is to the nature of the underlying development process itself. Such patterns are not purposely designed by developers, who rather pursuit goals related to the state of their codebase. They thus *emerge* from the topologically rich structure of DVCSs and characterize the way the codebase evolve over time.

To sum up, this paper makes the following contributions:

- The definition of a novel data structure, repository *Metagraphs*, that captures all the relevant information about the DVCSs repositories.
- A classification of the elements of a DVCS commit histories, based on their topological relevance in their history graph.
- An analysis of topological properties of the commit history of several opensource GIT–based project repositories.

The rest of the paper is organized as follows. Section II presents and formally defines our contribution. Section III describes the settings of our experiments. Section IV discusses our findings and motivates their relevance. Section V presents related works and Section VI draws our conclusion.

II. TOPOLOGICAL ANALYSIS OF COMMIT GRAPHS

The commit history of a decentralized repository can be modeled by a directed acyclic graph whose nodes are commits

and edges are parent/child relations between them.

As the history of a project evolves, parallel branches are created and joined in a distributed process of incremental manipulation of the code base. In general (as backed up by statistics we present in the following), the majority of commits belongs to single branches, while fewer others are those where the creation or the join of the branches themselves occur.

For the purpose of understanding the global structure of the commit history, i.e. the topology of the commit graph, these latter commits are much more relevant than the former.

We propose to classify, from the standpoint of their topological interest, the commits populating a history graph as follows:

– **Terminal**. Commits having no parent or no child.

– **Sequential**. Commits having exactly one parent and one child.

– **Structural**. Non-terminal commits whose union set of parents and children has more than 2 elements.

These categories are devised so that (i) they capture the whole set of commits in a GIT history and (ii) a given commit can belong to one and only one of them. To bridge these definitions with the standard GIT terminology, terminal commits are either first commits or HEADs; sequential commits are non-merging and non-branching ones, excluding first commits and HEADs; structural commits are merging and/or branching commits, which are not HEADs.

Terminal commits are thus the boundaries of the graph, while structural commits are the nodes which define its structure. Sequential commits do not play any role in determining the topology of the graph, being just enqueued in development lines (the branches) which lead from a merge (or branch) commit to another.

Sequential commits are clearly important from the standpoint of the developer contribution to the codebase. They should anyway be bypassed, when analyzing how branches feed each other and the parallel development gets organized in the repository, in order to get rid of topologically irrelevant information, which would bias any metrics.

Thus, a way must be devised to meet conflicting goals:

- Avoiding the overhead and the information noise of sequential commits while analyzing the topology of a commit history graph, in order to be able to obtain meaningful results
- Saving the information about the developer contributions that the sequential commits incorporate, along with their position in the commit history, in order to be able to relate the results of the topological analysis with those derived by other kind of data mining, which focus on code metrics or process metrics.

We define in the following a data structure and a set of properties that allow to positively solve both issues.

TABLE I. THE GIT REPOSITORIES UNDER STUDY — PART I. THE COLUMNS REPORT, IN ORDER, THE NAME OF THE PROJECT, THE NUMBER OF COMMITS IN ITS HISTORY, THE NUMBER OF EDGES AND NODES OF ITS *Metagraph*.

Project	commits	edges	nodes
OpenLeague	43	36	25
BroadleafCommerce	4722	784	496
Fridgemagnet	67	12	10
AutoSave	124	15	12
OryzoneBoilerplateBundle	60	20	14
ketama	18	10	7
Edaha	185	11	9
Tolmey	61	29	21
Ai_Class_Octave_Functions	23	13	10
WPide	152	24	17
sexpistol	37	14	10
ace	3220	1139	693
contracted	72	10	8
gitpython	163	37	25
tcesp	81	25	18
bempp	1530	375	246
FReD	211	45	31
testlol	77	12	9
configs	40	18	14
pool	42	14	10
q4wine	966	81	54
pylibemu	59	10	8
sarah	312	19	14
ConcurrenTree	632	91	59
ProWiC	61	30	22
java	95	47	33
flour	209	45	29
Telephus	272	65	43
arkilloid	177	40	27
RLLVMCompile	87	20	15
pants	643	57	40
pelm	415	103	68
jump	49	32	23
a4a	206	68	46
PySynergy	278	23	16
caveman	211	19	14
cocoagit	756	86	57
stsh	239	37	26
zap	28	10	7
iamhanchang	600	202	123
Locke	29	10	8
ConfigServiceProvider	43	16	12
Twittia	86	14	11
MiniCart	124	15	11

A. The Repository Metagraph

We define an original data structure, based on this classification of GIT commits. This structure, called *Metagraph*, encompasses a commit history, focusing on the topologically relevant commits. The *Metagraph* retains all the information from in the commit history. It aims at simplifying structural analyses performed on the repository commit logs of a GIT repository.

Def. *— Metagraph*
A *Metagraph* Mg of a commit history graph $G = \langle V, E \rangle$ is a multi–graph $Mg = \langle Ve, Me \rangle$, where Ve is the set of nodes which correspond to terminal and structural commits in V and Me is a set of *metaedges* (see definition 2).

We call *root* the node in Ve associated to the first commit of G.

Def. *— metaedge*

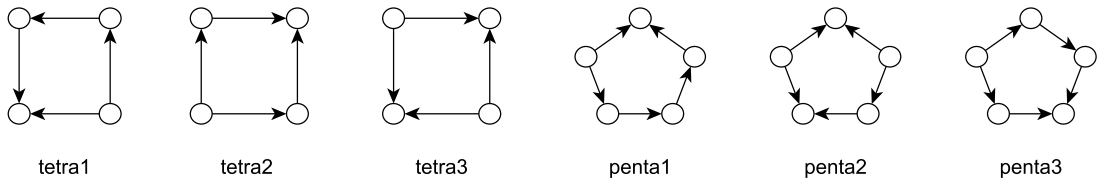


Fig. 2. Patterns of size 4 and 5.

TABLE II. THE GIT REPOSITORIES UNDER STUDY — PART II. THE COLUMNS REPORT, IN ORDER, THE NAME OF THE PROJECT, THE NUMBER OF COMMITS IN ITS HISTORY, THE NUMBER OF EDGES AND NODES OF ITS *Metagraph*.

Project	commits	edges	nodes
nmrglue	132	12	9
vimclojure	346	18	13
robut	167	58	38
chrome_collector	74	43	29
satellite	161	15	11
PJsonCouch	25	10	8
mailchimp	50	15	11
gtalkjsonproxy	60	10	8
owper	140	15	12
mxn	507	201	121
crab	50	12	11
CRIOJO	273	84	57
n2n	203	75	51
migen	486	15	11
pinocchio	18	13	10
gokogiri	566	124	88
zamboni	13800	1334	865
MailCore	279	69	44
pod	149	20	15
jquery_add_more	32	23	15
blur	997	118	78
hsnews	89	16	11
easyXDM	761	176	116
spidr	635	10	8
pyrocms	14228	7176	4187
statsd	142	62	38
spark	198	21	15
ri_cal	322	18	14
asset_sync	301	109	70
bamboo	55	10	8
perl5i	792	256	164
andthekitchensink	176	18	14
connect	2515	650	410
eksamensnotater	136	65	42
MvvmCrossConference	92	10	8
Newscoop	10721	3742	2208
xyzcmd	594	18	13
tradinggame	69	25	17
CMSC858E	65	34	23
tinycon	53	32	21
cloakstream	56	23	17
iTetrinet	981	167	112
Tir	146	25	17

Given a *Metagraph* Mg of a commit history graph G, a *metaedge* is a directed edge between commits in the set of nodes Ve of Mg.

A *metaedge* me between two nodes Ve_1 and Ve_2 exists in Mg if and only if a path in G exists between the corresponding nodes V_1 and V_2, which is composed uniquely by sequential commits. The list of (sequential) commits composing this path in V is associated (as metadata) to the *metaedge*.

We call Ve_1 (resp. Ve_2) first (resp. last) node of me.

Figure 1 illustrates a *Metagraph* (right) built from a commit history graph (left). Our definition of *metaedges* serves two main purposes: (i) it simplify the topology of a commit history by excluding sequential commits from its structure, while preserving the information they carry and (ii) it makes the *Metagraph* a model that focuses on parallel development events (occurrences of branching or merging in the commit history graph G).

Our analysis of *Metagraphs* extracted from GIT repositories is based on the following properties:

Def. *— Metagraph's layer*
For any $v_i \in Ve$, let $dist(v_i) : Ve \to \mathbb{N}$ be a function which maps each node of Mg to its maximum distance from the root of Mg, measured in number of *metaedges*.
A layer L_n in a *Metagraph* Me is defined as
$L_n \subseteq Ve \in Me : l_i \in L_n \iff dist(l_i) = n.$

Layer L_n is thus the set of all nodes having the same maximum distance n from the root. We call n the *index* of the layer. By construction, roots are the only nodes belonging to L_0.

Supposing the example in Figure 1 represents a complete *Metagraph*, we have $L_0 = c1$, $L_1 = c6$ and $L_2 = c10, c13$. The set of layers of a *Metagraph* is a convenient partitioning of its nodes, which groups commits according to their distance from the first commit, measured in terms of number of parallel development events, as captured by the *metaedges*.

Def. *— Width of a layer.*
The width of a layer L_n is equal to its cardinality $|L_n|$.

Def. *— Density of a layer.*
We call density of a layer L_n the number of *metaedges* whose last node is in $L_m, m \geq n$.

The density of a layer L_n is thus the number of *metaedges* terminating either in L_n or in a layer which is farther from the root. By definition, the density of $L_0 = 0$.

Let c_n be a cut of Mg partitioning its nodes in two sets: $Ve_1 = v_i : dist(v_i) < n$ and $Ve_2 = v_j : dist(v_j) \geq n$. Then, the density of L_n is the number of *metaedges* crossing c_n.

Def. *— Diameter of a Metagraph.*
The diameter of a *Metagraph* is equal to the length of the longest path in the *Metagraph*. By construction, it is equal to the greatest layer index of the *Metagraph*.

B. Development Patterns

Since the *Metagraph* captures the structural properties of a commit graph, we now look for evidence of the presence (or lack thereof) of recurrent development patterns in repository metagraphs.

A way to spot such patterns is to find recurring subgraphs in a *Metagraph* over different projects. Since any subgraph of a *Metagraph* can only be composed by merging and/or branching commits of the project history, the fact that the same subgraph occurs several times in a project *Metagraph* would denote the emergence of the same collaborative pattern at different moments of the project history.

Moreover, the fact that the same subgraph repeatedly occurs in *Metagraphs* which model the history of different projects would point out *emergent patterns* characterizing collaborative development itself, rather than the collaboration habits of a specific community of developers.

The problem of finding all subgraphs of a given graph is in general *NP-hard*. Even limiting the size of the subgraphs to target, the feasibility of such a task heavily depends on the structural complexity of the *Metagraph*. Several efficient algorithms exist, though, to find single subgraphs within larger graphs. By restricting the number of targets and narrow down the search to each one of them, we can find all occurrences of specific patterns of a given size (number of nodes).

A way to choose the target patterns is needed. We are interested in enlightening the very structure of the *Metagraphs*. Terminal nodes are the ever-changing boundaries of the *Metagraphs* and thus not really relevant for our purposes. We can focus on structural nodes and on the *metaedges* that start from or arrive to structural nodes.

Let us therefore consider the subgraph $Sg = \langle Vs, Ms \rangle$ of a *Metagraph* Mg, composed by the set Vs of all structural nodes of Mg, along with the set Ms of all *metaedges* of Mg whose first and last nodes are both in Vs. Recalling that, by construction, any *metaedge* in Ms must either start form or end to a node in Vs, it is easy to see that all subgraphs of Sg are either isomorphic to simple polygons or to polygons composed by adjacent polygons of smaller size. Figure 3 exemplifies our point by showing two isomorphic graphs.

Thus the quest for patterns becomes the search for all polygon subgraphs in Sg. Finding simple polygons is enough, because it is easy to spot out composed polygons, by tracking common nodes among simple (smaller) polygons. Such a task is accomplished by finding induced subgraphs in Mg, which are isomorphic to simple polygons.

We thus propose the following definition.

Def. *— Pattern*
We call pattern any induced subgraph of a *Metagraph*, which is isomorphic to a polygon graph.

As *metaedges* are always directed, there are several different (*i.e.*, non–isomorphic) patterns of any given size $s, s > 3$. Thus the relevance of a pattern may not be given only by its size, but also by its very topology. Since we are interested in characterizing the way collaborative development shapes commit histories, we propose a simple way to discriminate between patterns of *incremental development* and patterns of *code integration*. Clearly both aspects are always present in any pattern, but their relative relevance may differ, as we exemplify in the following.

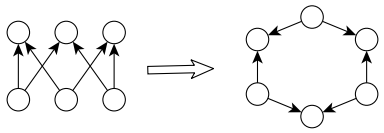


Fig. 3. Isomorphic simple acyclic digraphs of degree 2.

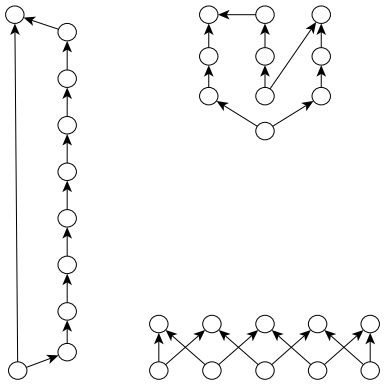


Fig. 4. Patterns of size 10 showing a different slenderness.

Being our patterns simple acyclic digraphs of degree 2, each pattern is composed by two or more walks. The lengths of these walks fall in the interval $]0, 1, \ldots, N[$, where N is the size of the pattern. Considering the topology of a pattern, the presence (resp. absence) of few longer walks indicates a greater (resp. smaller) relevance of incremental developing w.r.t. code integration. Figure 4 intuitively shows how three identically sized patterns can be different from this standpoint.

In order to classify patterns of different size w.r.t. the kind of collaboration they entail, and, more generally, to compare them w.r.t. this criterion, we propose to compute their *slenderness*, defined as follows:

Def. *— Pattern's slenderness*
The slenderness of a pattern is a real number in $[0 \ldots 1[$, computed as the number of vertexes which are internal to its walks, divided by the size of the pattern.

Intuitively, a higher (resp. lower) slenderness denotes a pattern where incremental development (resp. code integration) dominates over its counterpart. For instance, the patterns in Figure 4 have a slenderness of $8/10 = 0.8$ (left), $6/10 = 0.6$ (top right) and $0/10 = 0$ (bottom right), which well denotes the fact that the first pattern is strongly incremental, while the second balance increments and integration (with a slight dominance of the first) and the third pattern is all about integration.

The results presented in the following show that recurring patterns indeed characterize commit histories of a sample of GIT–based projects.

TABLE III. THE PATTERNS WE CONSIDER.

Name	Size	How many
tri	3	1
tetra*	4	3
penta*	5	3
exa*	6	8
epta*	7	9
octa*	8	20
enna*	9	28
deca*	10	53

III. EXPERIMENTAL DESIGN

We present an experiment that aims at identifying development patterns in a sample of GIT repositories, collected from GITHUB. For each repository, we build a *Metagraph* modeling its commit history. We then detect patterns in the topology of the *Metagraphs* and analyze their characteristics.

A. Experimental Data

The experimental data comes from GITHUB, an Internet hosting service for open source. According to FLOSSmole [3] (Free Libre OpenSource Software) statistics, GITHUB had 191765 repositories publicly available in May 2012. In order to obtain a statistically representative sample of GITHUB hosted projects, we order these projects according to the number of watchers. To discard outliers and less significant entries, we decide to cut off the extrema of the range, *i.e.* projects whose number of watchers is less than 2 or more than 1000. Then we select 1% of the projects in each of three subsets:

- Projects that had from 2 to 9 watchers (total: 30236; sampled: 303)
- Projects that had from 10 to 99 watchers (total: 3554; sampled: 36)
- Projects that had from 100 to 999 watchers (total: 286; sampled: 3)

This way, we obtain a set of 342 GIT repositories. We then discard those having a too poor structure, which we define as less than 10 *metaedges* in their *Metagraph*. We finally obtain a set of 87 projects, listed in Tables I and II.

B. Patterns and Metagraph Properties

As explained above, we define patterns as polygonal induced subgraphs of our *Metagraphs*. We limit our analysis to patterns from size 3 to size 10, because small patterns are trivial and large patterns are too costly to compute. There are 125 non–isomorphic patterns within this size range. For instance, there is only one pattern of size 3, which corresponds to a triangle. Figure 2 shows all patterns of size 4 and 5. To ease the references to each pattern in the following, we use some arbitrary nicknames, composed by the greek name of numbers from three to ten and by a numerical index. Table III summarizes the information about the "families" of patterns we consider. The third column reports the number of existing non–isomorphic instances of each pattern.

It is important to recall that any pattern can only occur in a *Metagraph* of a commit history where two or more parallel lines of development exist, since patterns are composed by structural nodes, which by themselves imply the existence of more than one branch. Thus, whenever a pattern is found in a *Metagraph*, it is implied that those nodes whose in–degree and out–degree is equal to 1 *within* the pattern are actually linked to other nodes of the *Metagraph*, which are not part of the same pattern.

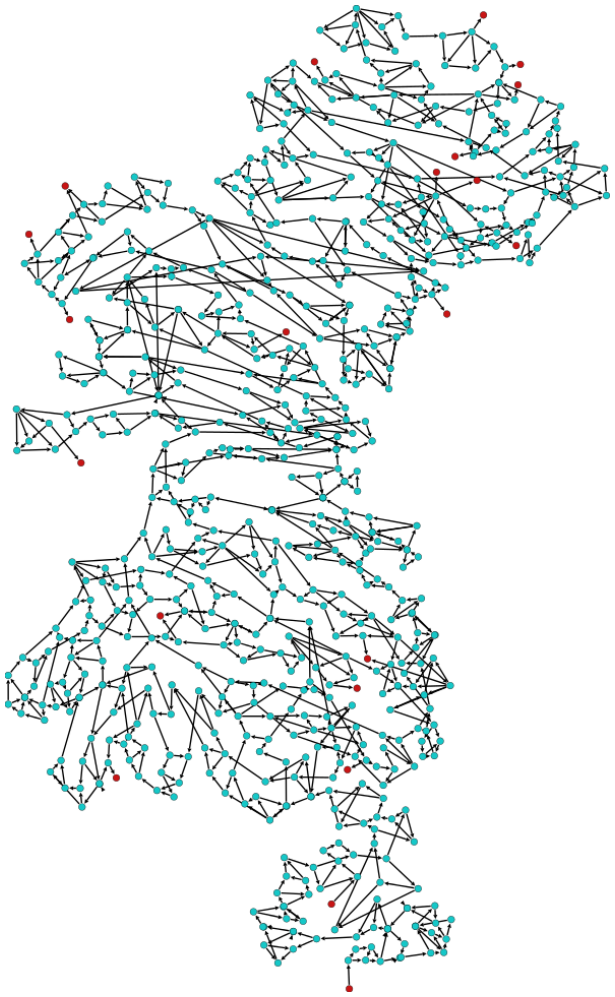


Fig. 5. *Metagraph* of the `ace` project. Root at the bottom; other red nodes are branches' heads.

In order to investigate if and how the properties of the *Metagraph* are associated with occurrences of patterns, we analyze the relations between patterns and the structural properties of the *Metagraph* (diameter, layer width and density).

We then define two categories of patterns, namely *increment* and *integration*. All patterns having a slenderness greater or equal to 0.5 are assigned to the first category, while the others fall into the second. We thus analyze the presence of each category of patterns in our repository sample.

C. Implementation

In order to perform our analysis, we developed a toolset called GitWorks, as a pure Java application. GitWorks uses JGit[1] to extract information from GIT repositories, then build

[1] http://eclipse.org/jgit/

Metagraphs and extract several features from the collected data. GitWorks is freely available on GITHUB[2].

In order to extract the occurrences of patterns in the *Metagraph*, we use the Grochow–Kellis subgraph detection algorithm, based on symmetry breaking [4]. The implementation of the algorithm has been provided by the authors.

D. Research Questions

R.Q. 1 *— Can we find significant recurring structures in DVCS histories?*
Our purpose is to see whether there are patterns in commit histories which repeatedly occur. If yes, they probably characterize the development process and we will try to analyze them.

R.Q. 2 *— Does the structure of DVCS evolve over time?*
If structural patterns can be detected, we want to understand how they change along the history of a project, i.e. whether some collaboration patterns are more frequent at the beginning of a project or in its maturity. If a commit history be morphologically characterized by specific patterns at specific times of its evolution, we would be able to tell something about the social facet of projects, about how people collaborations change over time.

R.Q. 3 *— Is the occurrence of patterns correlated with increased concurrent development activities?*
Whenever patterns appear, they may simply signal a particular increase of activity in the project. Patterns which tend to appear whenever activity increase or decrease may then be correlated to the strengthening of the project in terms of developer or code base.

R.Q. 4 *— Are there structural similarities among DVCS histories?*
We aim at finding evidences of the existence (or the lack thereof) of same subgraphs in unrelated projects. Such a finding would clearly indicate that there are "universal" patterns of development collaboration. This would then highlight common problems or strong points of different development communities.

IV. EXPERIMENTAL RESULTS

In this section, we present the outcomes of our analysis and discuss their relevance. We first present results related to the properties of the *Metagraphs*. Then we describe the results related to the pattern detection. Finally, we discuss the outcomes.

A. Metagraph Properties

The *Metagraphs* which model the commit graphs of our sample are indeed quite diverse, from very simple parallel organization to very complex structures. Figure 5 depicts a topologically rich *Metagraph*.

Table IV shows the correlation between each couple of *Metagraph* properties listed in Tables I and II, plus the number of authors per repository, measured over the whole ensemble of repositories.

[2]https://github.com/marbiaz/GitWorks

TABLE IV. METAGRAPH FEATURES CORRELATIONS.

	nodes	edges	commits
edges	0.99978	—	—
commits	0.86953	0.86000	—
authors	0.91235	0.91190	0.77873

The correlation between the number of authors and the number of commits is not particularly high. More interestingly, *Metagraph* features are more correlated to the dimension of the developer community than to their total amount of contributions. This suggests that the *Metagraph* does highlight patterns of collaboration in its very structure.

Figure 7 shows the distributions of the time span of the layers (boxplots) and the diameter (tiles) of each *Metagraph*. The boxplots extend for the standard interquartile range, while the whiskers cover all data points. The maximum and minimum values found are reported along the vertical axes.

Layers present high variability in terms of time range: from few seconds to hundreds of days elapse between two consecutive layers. There seems to be a mild inverse correlation between minimum time range of layers and the *Metagraph* diameter. Figure 7 shows that lower whiskers gets to smaller values, from left–most to right–most boxplots, while they are plotted in ascending order according to their diameter values.

B. Patterns

Our results show that indeed different commit histories feature the same patterns. There are therefore structures of distributed development which marks portion of the commit graphs and characterize their evolution.

Such patterns would not be detectable on "raw" commit history graphs, because sequential commits "overwhelm" their structure destroying relevant topology properties. Thanks to the transformation to *Metagraphs*, the very structure of commit graphs can be successfully mined from a topological standpoint, revealing the otherwise concealed patterns of developer collaboration.

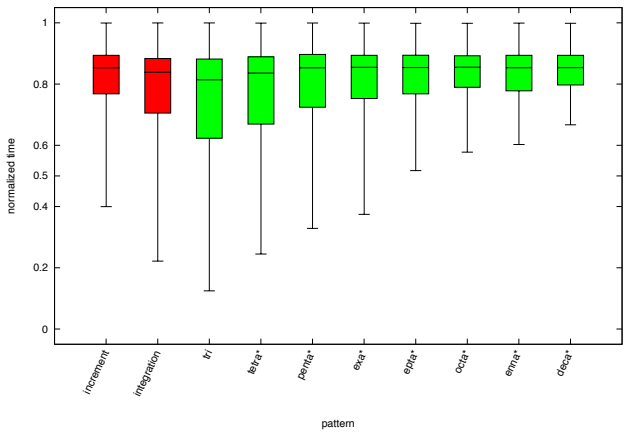


Fig. 8. Distribution of timestamps of structural commits per pattern.

Patterns occurs at different moments of projects histories. Figure 8 shows the distribution of timestamps of *Metagraphs* vertexes composing patterns in each category we defined.

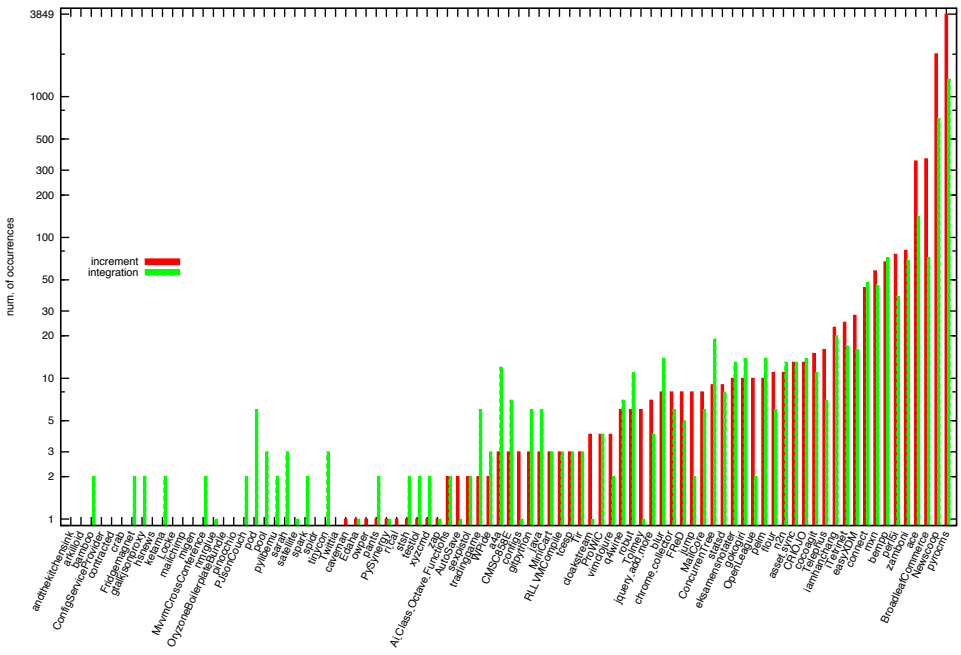


Fig. 6. Occurrences of increment and integration patterns.

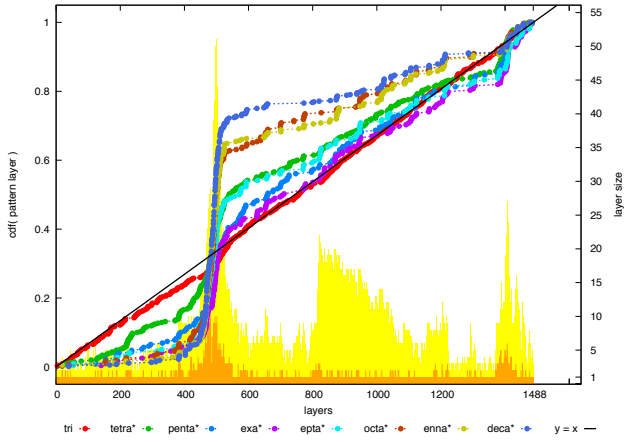


Fig. 9. Cumulative distribution of patterns in the Newscoop metagraph. The cdf values of pattern occurrences, grouped per size, are given on the left axis, while the right axis gives the values of the layer width (orange vertical bars) and density (yellow vertical bars). The black line traces the slope of a linearly uniform distribution across the *Metagraph*'s layers.

Data are aggregated over all the *Metagraphs*. The boxplots are standard interquartile plus whiskers to cover 95% of data points. Outliers are not shown, to improve legibility.

We see how patterns of higher size tend to occur later than smaller ones. The skew towards the top is somehow biased by the fact that we compare project of different age and size, thus, in our case, the normalization of outcomes of smaller project tends to give high values. Nonetheless, it is true that larger patterns are definitely more numerous in the second half of projects' life.

The first two boxplots show that increment patterns tend to occur substantially later than integration pattern.

Table V presents the results of a comparative analysis of the number of authors and commits among *metaedges* which belong to patterns and which do not. In order to summarize a large amount of data, we present these results as dominance scores of the main aggregates. Each triplet reports the number of projects where a given aggregate (*i.e.*, minimum, median or maximum) measured in a pattern is ⟨`less_than-equal_to-greater_than`⟩ the same aggregate measured in *metaedges* non belonging to a pattern. For instance, the first case on the left tells that the minimum number of distinct authors in *metaedges* belonging to increment patterns is greater than the minimum of *metaedges* non belonging to any patterns in 40 projects out of 59. Only projects where the considered patterns appear are taken into account, thus the triplets sum up to less than 87.

We see that pattern *metaedges* consistently dominate the others w.r.t. the minimum and the median number of distinct authors, while they usually carry less sequential commits.

TABLE V. DOMINANCE COUNTS FOR AUTHORS AND COMMITS IN PATTERNS.

Patterns	**Authors**			**Commits**		
	min	med	max	min	med	max
increment	1–18–**40**	10–**33**–16	**34**–19–6	21–**35**–3	**57**–1–1	**56**–1–2
integration	1–24–**46**	7–**34**–30	**41**–20–10	26–**44**–1	**65**–4–2	**63**–2–6
tri	1–24–**46**	9–**33**–29	**40**–19–12	26–**44**–1	**64**–5–2	**63**–2–6
*tetra**	1–16–**38**	9–**30**–16	**31**–18–6	18–**35**–2	**54**–1–0	**53**–0–2
*penta**	0–9–**26**	7–**15**–13	**24**–9–2	9–**26**–0	**32**–2–1	**33**–0–2
*exa**	0–7–**25**	4–**17**–11	**19**–8–5	8–**23**–1	**30**–1–1	**31**–1–0
*epta**	0–5–**16**	2–**11**–8	**15**–4–2	5–**16**–0	**21**–0–0	**21**–0–0
*octa**	0–4–**13**	2–7–**8**	**12**–3–2	4–**13**–0	**16**–1–0	**17**–0–0
*enna**	0–4–**15**	2–6–**11**	**14**–4–1	4–**15**–0	**17**–0–2	**18**–0–1
*deca**	0–2–**13**	2–4–**9**	**12**–2–1	2–**13**–0	**14**–0–1	**15**–0–0

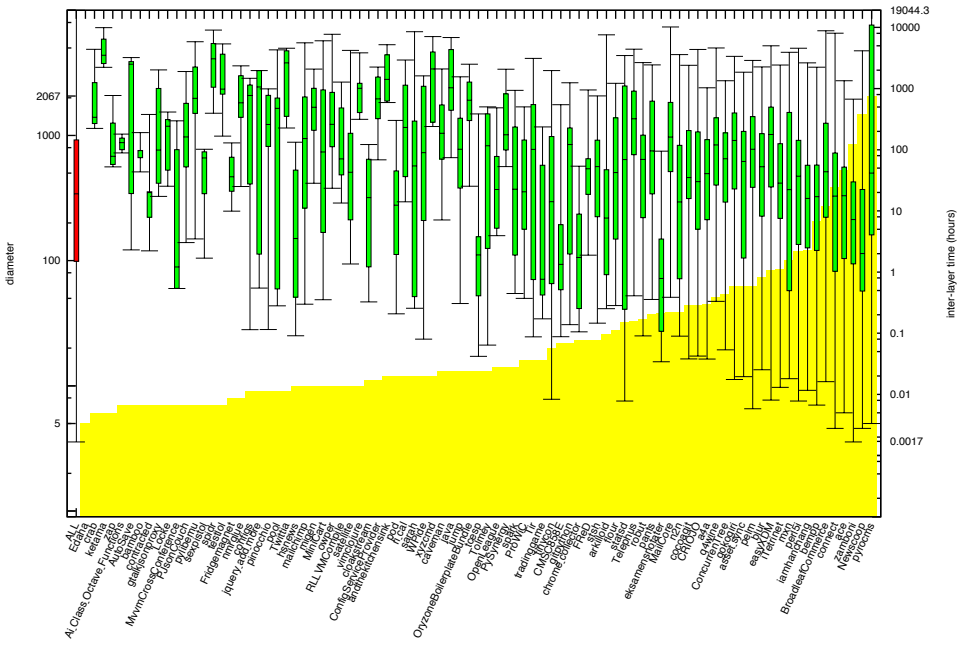


Fig. 7. Distribution of time spans of the layers (boxplots) and metagraphs' diameters (tiles). The first boxplot shows aggregates over the whole ensemble.

C. Discussion

As said, the definition of *Metagraphs* as models of commit graphs allows a novel kind of topological analysis of repositories.

Several properties of the *Metagraphs* offer interesting standpoints for further analysis of single projects. The results we present aggregate measures of diverse repositories and are not meant to offer a precise description of any of them. They rather point out interesting features which shall be further investigated both at the aggregate level and project by project, in order to reveal their specific importance.

In the following, we discuss our results by recalling and answering our research questions.

R.Q. 1 — *Can we find significant recurring structures in DVCS histories?*
Answer — The patterns we discover are the very evidence of concurrent development in a GIT repository. As Figure 9 shows, in general patterns, and most of all the smaller ones (size 3 to 5), are quite uniformly distributed across *Metagraph* layers, which means that they are not related to the "development stage" of the software, but rather to the parallel development process itself.

The same figure also gives a synopsis of layers densities and widths. All patterns, but `tri` and to some extent `tetra*`, seem to be particularly sensitive to layer properties, thus clustering more where layers are denser.

These findings are common to several of the largest *Metagraphs*, with some exception, mostly among the smaller projects. The main information we can recollect is that while some patterns emerge consistently as "building blocks" of the evolution of the commit history, others are associated to specific properties of the *Metagraph* layers.

R.Q. 2 — *Does the structure of DVCS evolve over time?*
Answer — The time span of layers, presented in Figure 7 do not present a strong correlation with either the diameter of the *Metagraph* or its distribution of layer widths. This suggests that the pace of integration depends on something different than the number of times it occurred in the past (captured by the diameters), or the number of concurrent development lines at a given moment (given by the layer widths). It is anyway visible that repositories with a longer diameter do feature mostly short–time layers. Anyway several exceptions appears even in our small sample.

Figure 8 shows that patterns mostly appear later in time, if not "in space". We find an interesting distinction between the timing of codebase evolution and the changes in its commit topology.

We cannot propose a generalization because of the limited size of our sample and because of what we can consider an "internal bias" of several large repository. We found that many of them have been automatically converted to GIT repository from a pre–existing CVCS. They thus initially exhibit a mostly linear topology which, *e.g.* for long–lasting successful projects, can encompass a significant portion of their total age.

Wherever the topology starts becoming complex and the *Metagraphs* denser, we see that layers do not subscribe to any regularity w.r.t. the pace of collaboration. We find layers whose time span is of few seconds and others which encompass several months. This fact may suggest that automatic merging is used in the first case. But, depending on the number of separate branches belonging to the layers, it may simply reveal highly synchronous development activity.

Thus, if we can see that larger patterns do tend to occur later in time w.r.t. smaller ones, we cannot answer so far an interesting question: whether is there a feed–back loop between

the structural complexity of a commit history (the *Metagraph* getting longer and denser) and the behavior of the developers (the fact that they branch and merge within shorter time).

We think that deepening this kind of analysis may reveal specific practices of integration and development which may affect the quality of the code, either positively or negatively.

R.Q. 3 *— Is the occurrence of patterns correlated with increased concurrent development activities?*
Answer — We found some apparently conflicting evidences in our outcomes, summarized in Table V.

On the one hand, we see that pattern *metaedges* present a higher concentration of authors w.r.t. most of other *metaedges*. On the other hand, they do not seem to carry a particularly high number of internal commits. Based on the first observation, we can say that patterns do identify portions of the commit history where authors collaboration becomes more evident. At the same time, quite clearly higher collaboration does not entail a higher number of commits per branch.

Our hypothesis — which will need a more thorough analysis of the single projects to be confirmed — is that commits that brings most novelties to a codebase are made by fewer authors, who work in branches which do not directly end up in pattern, but merge with other "local integration" branches, which then lead the modifications to the main integration structures.

Thus, *metaedges* which comprise higher numbers of authors do not include higher numbers of commits. This is an interesting finding that open a novel perspective on how to determine the strength and the extent of author collaboration in a distributed project. By combining the analysis of author presence in *metaedges* with, for instance, code–based analysis about authors modifying the same files or components, it is possible to deepen the comprehension of collaboration practices in developer communities.

R.Q. 4 *— Are there structural similarities among DVCS histories?*
Answer — We can positively answer this question and from two diverse perspectives.

First, as Table IV shows, the structural properties of commit histories are more correlated with the size of the developer community than with the number of commits they produce. Thus it is reasonable to think that projects whose community size is close would exhibit structures of similar richness, at least from the standpoint of aggregate measuring.

Then, we find the same patterns in otherwise very diverse software repositories. Figure 6 shows that increment patterns tend to occur more, but repositories with a richer structure tend to have anyway a higher number of occurrences of both categories.

Patterns that rarely occur in our repository sample are interesting as well. Their occurrence may be related to a particular necessity or may happen only in specific topologies. Did further analysis find that they present some drawback, it would be interesting to understand which structural configuration let them emerge, in order to devise better practices.

The occurrence of same patterns in unrelated software repositories suggests that development practices of different teams produce similar topologies in the repositories. Common patterns across repositories may reveal common vantage points or flaws in the way the software is developed, which may impact the quality of the software product itself.

As a final remark, we underline that the present work is far from exhausting the space of possibilities. *Metagraphs* and pattern analysis can be driven much further in many directions, either deepening the understanding of the relations between the various measures we presented, or by exploring the space of pattern combination and inter–dependence. This paper aims at introducing this quite promising topic and showing the evidence of relevant findings.

D. Threats to validity

The reasons behind the detection of patterns in commit histories may be questioned, given the absence of evidences that support their relevance. In this paper, our main effort is giving some empirical evidence of the fact that these common sub-structures occur in complex development histories, though they may be hidden in a standard commit graph topology.

We present results which concern patterns of maximum size equal to 10. It is certainly feasible and interesting to scale up, though the computation of all possible polygons of a given size gets exponentially harder as the size increases. Anyway, the main purpose of the present work is to show relevant characteristics that, to some extent, are common to pattern of different size and are mainly due to their topological properties.

Our repository sample has been chosen with no known bias towards any specific topological structure. Due to the novelty of the analysis, though, it is possible that we are not aware of specific features of the *Metagraph* topology, which somehow affect the emergence of the patterns.

Our repository sample is of limited size, thus some of our results may not generalize on larger ensembles.

Finally, the toolset we used to perform the computations has been developed by us and manually tested on repositories presenting high variability. We cannot exclude the presence of unnoticed bugs which could affect the correctness of our results.

The subgraph detection algorithm has been kindly provided by its author and has not been thoroughly tested by us.

V. RELATED WORK

The works which use graph–based techniques to analyze repositories focus either on characterizing the sourcecode, or on finding patterns in the social networks of developers.

In the first group, Posnett *et alii* propose models derived from ecological inference to analyze code at different hierarchical levels (files, packages, modules) [5]. They show that properties inferred at a given level may not hold at a different one or often be mistakenly considered to hold in general.

Demeyer *et alii* [6] propose metrics to detect refactoring. Valverde e Solé [7] show how large scale software architectures can be modeled by dynamic logarithmic networks.

These and similar works focus on the content of contributions, rather than on the topology of commit histories.

In the second group, Posnett *et alii* propose graph–based models to infer developer focus [8], showing correlation with bug occurrences and issue solving performance.

Bhattacharya *et alii* combine the analysis of code organization and developer networks. They characterize both in terms of graph metrics and they show how it is possible to predict several indicators of software quality by using these metrics.

In [9] the authors analyze how different social organizations of developers impact the quality of the development in opensource communities.

Several relevant works target opensource software development [10], [11]. A recent paper [12] analyze the social structure of developer networks on a large sample of GITHUB projects.

These works apply complex network theory to the analysis of interactions among developers and to the characterization of their contribution to projects.

A thorough analysis of GITHUB–based software development, with a deep comparative analysis of centralized and decentralized versioning systems is given in [13].

Though recent publications pay more and more attention to GIT–based repositories and their distributed development paradigm [1], [14], to the best of our knowledge no study focuses on the characterization of the topology of commit histories.

VI. CONCLUSION

Decentralized version control systems produce structurally rich commit graphs. The commit history of projects that use these systems is thus a highly non–linear graph, whose topology can be analyzed by means of techniques, which are typical of complex graph analysis.

As for many other domains, rich complex structures incorporate patterns that reveal similarities among otherwise unrelated projects. Such patterns characterize the development process from a purely topological standpoint.

We discuss in this paper the reasons why detecting these patterns is relevant and the challenges of such a quest.

We define a novel data structure, the *Metagraph*, that makes it possible to perform a topological analysis of commit history graphs, in order to detect recurrent subgraphs hidden in their structure.

These subgraphs are network *patterns*, building blocks of the very topology of the *Metagraphs* that model the commit histories of different projects.

Thanks to our contributions, we find empirical evidences of the recurring presence of several patterns in a sample of opensource GIT repositories available on GITHUB.

We analyze the characteristics of these patterns and their relations with the properties of the *Metagraphs*. We propose a simple partition in categories which help investigating their role as functional blocks of a distributed development process.

Our findings open several different research directions. We plan to dig further the relation between patterns and properties of commit graphs, such as number of commits, authors and size of commit changes.

We think it is possible to devise novel models to characterize developer collaboration, which are based on common contributions to the same *metaedges* or patterns of a given *Metagraph*.

The novel kind of analysis we propose is not an alternative, but rather a complement to all existing techniques. By focusing entirely on topological features, it can guide to new insights on the emergent properties of distributed collaborative development.

REFERENCES

[1] E. Shihab, C. Bird, and T. Zimmermann, "The effect of branching strategies on software quality," in *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*. ACM, 2012, pp. 301–310.

[2] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu, "The promises and perils of mining git," in *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*. IEEE, 2009, pp. 1–10.

[3] J. Howison, M. Conklin, and K. Crowston, "Flossmole: A collaborative repository for floss research data and analyses," *International Journal of Information Technology and Web Engineering*, vol. 1, pp. 17–26, 07/2006 2006.

[4] J. A. Grochow and M. Kellis, "Network motif discovery using subgraph enumeration and symmetry-breaking," in *Research in Computational Molecular Biology (RECOMB07)*, ser. Lecture Notes in Computer Science, vol. 4453. Springer-Verlag, 2007, pp. 92–106.

[5] D. Posnett, V. Filkov, and P. T. Devanbu, "Ecological inference in empirical software engineering," in *ASE*, 2011, pp. 362–371.

[6] S. Demeyer, S. Ducasse, and O. Nierstrasz, "Finding refactorings via change metrics," in *ACM SIGPLAN Notices*, vol. 35/10. ACM, 2000, pp. 166–177.

[7] S. Valverde and R. Sole, "Logarithmic growth dynamics in software networks," *EPL (Europhysics Letters)*, vol. 72, p. 858, 2005.

[8] D. Posnett, R. D'Souza, P. Devanbu, and V. Filkov, "Dual ecological measures of focus in software development," in *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 2013, pp. 452–461.

[9] S. Valverde and R. V. Solé, "Self-organization versus hierarchy in open-source social networks," *Phys. Rev. E*, vol. 76, p. 046118, Oct 2007.

[10] A. Mockus, R. T. Fielding, and J. Herbsleb, "A case study of open source software development: the apache server," in *Proceedings of the 22nd international conference on Software engineering*, ser. ICSE00. New York, NY, USA: ACM, 2000, pp. 263–272.

[11] C. Rodriguez-Bustos and J. Aponte, "How distributed version control systems impact open source software projects," in *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*. IEEE, 2012, pp. 36–39.

[12] F. Thung, T. Bissyande, D. Lo, and L. Jiang, "Network structure of social coding in github," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 323–326.

[13] C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig, "How do centralized and distributed version control systems impact software changes?" in *ICSE'14*, 2014 (to appear).

[14] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in github: transparency and collaboration in an open software repository," in *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*. ACM, 2012, pp. 1277–1286.

Table ronde : 'Génie de la Programmation et du Logiciel' et Agilité : un mariage heureux

Table ronde : 'Génie de la programmation et du logiciel' et Agilité : un mariage heureux ?

Animateurs

Mireille Blay-Fornarino (I3S, U. Nice Sophia-Antipolis)
Jean-Michel Bruel (IRIT, U. Toulouse)

Résumé de la table ronde
Les méthodes de développement changent et l'agilité est aujourd'hui un buzzword. Pourtant derrière ce terme se trouvent des besoins, des techniques et aussi des défis. L'objectif de cette table ronde est sur la base de retours industriels et académiques de mettre en avant les difficultés rencontrées et les solutions éventuelles pour que les principes du GLP s'adaptent à ses nouveaux usages sans perdre les fondements acquis. Au travers de questions préparées et de discussions avec la salle, cette table ronde vise à engager une discussion entre les différentes communautés de GPL et les industriels.

Participants

Agusti Canals est Directeur d'Unité Fonctionnelle (Technique) à CS Communication & Systèmes http ://www.c-s.fr, son domaine d'expertise est le génie logiciel (Certifié N3/3 en UML par l'OMG) et l'ingénierie système (Certifié N2/4 en SysML par l'OMG), Il est l'un des fondateurs, en 2004, des journées Neptune (dont le thème récurrent est le MDE)http ://neptune.irit.fr, et du forum des méthodes formelles (cycle de conférences) http ://projects.laas.fr/IFSE/FMF/. Depuis plusieurs années il expérimente les approches agiles, et notamment SCRUM, aujourd'hui largement utilisée à CS sous le nom « Agile CS »

Yves Ledru est Professeur à l'Université Joseph Fourier de Grenoble, où il enseigne le Génie Logiciel. Il mène ses recherches dans le domaine des Méthodes Formelles appliquées au Génie Logiciel. De 2008 à 2011, il était directeur du GDR GPL.

Frédéric Bonnot travaille chez Thales Avionics depuis 1998. Il a participé en tant que développeur et chef de projet au développement de logiciels critiques embarqués dans de nombreux projets en appliquant ou non les méthodes agiles. Actuellement responsable d'un service de développement logiciel embarqué dans les cockpits d'aéronefs et en charge de l'amélioration de la compétitivité des développements logiciels pour la division cockpit, il nous partagera son expérience.

Patrice Petit (Coach Agile) a un doctorat en mathématiques appliquées et enseigne l'Agile dans plusieurs universités. Fondateur de l'Agile Tour, avec plus de 20 ans d'expériences en développements logiciels, il a crée la première entreprise de Coaching et de Formation dédiée à l'Agile en France, AgilBee. Il réalise du conseil en organisation, des formations certifiantes sur Scrum/Agile et de l'accompagnement sur la transformation Agile des équipes et organisations. Il donnera le point de vue d'un coach agile dans le contexte de la table ronde à la fois par sa connaissance du domaine et les difficultés rencontrées qui peuvent relever de la recherche.

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

Prix du GDR GPL : Une Approche Basée sur les Lignes de Produits Logiciels pour Sélectionner et Configurer un Environnement d'Informatique dans les Nuages

Auteur : Clément Quinton (Université Lille 1, CRIStAL, Inria & Politecnico di Milano)

Résumé :

Résumé : L'informatique dans les nuages est devenu une tendance majeure dans les environnements distribués, permettant la virtualisation des logiciels sur des environnements d'exécution configurables. Ces multiples environnements fournissent de nombreuses ressources hautement configurables à des niveaux de fonctionnalité différents, pouvant mener à des erreurs lors de la configuration si cette dernière est effectuée manuellement. Cette thèse propose donc une approche basée sur les principes des lignes de produits logiciels, reposant sur des modèles de caractéristiques dédiés à la description des similitudes et de la variabilité entre ces environnements de nuage. Nous proposons notamment trois contributions essentielles. En premier lieu, nous fournissons un méta-modèle permettant de décrire des modèles de caractéristiques étendus avec des attributs, des multiplicités et des contraintes s'appliquant sur ces extensions. Ce type de modèle de variabilité est nécessaire à la description de la variabilité des environnements de nuage. En fournissant un modèle abstrait, nous sommes donc indépendant de toute implémentation et permettons ainsi aux approches existantes de s'appuyer sur ce modèle pour étendre leur support. Deuxièmement, nous proposons un support automatisé pour maintenir la cohérence des modèles de caractéristiques étendus avec des multiplicités. En effet, lorsque ceux-ci évoluent, des incohérences peuvent survenir dues aux différentes multiplicités définies et aux contraintes sur ces multiplicités. Les détecter peut alors être une tâche complexe, surtout lorsque la taille du modèle de caractéristiques est grande. Enfin, nous fournissons comme troisième contribution SALOON, une plateforme pour automatiser la sélection et la configuration des environnements de nuage basée sur les principes des lignes de produits logiciels. SALOON s'appuie notamment sur notre méta-modèle pour décrire les environnements de nuage en tant que modèles de caractéristiques, et fournit un support automatisé pour générer des fichiers de configurations et scripts exécutables, permettant ainsi une configuration fiable desdits environnements.

Biographie :
Clément Quinton a réalisé son doctorat d'informatique au Laboratoire d'Informatique Fondamentale de Lille (LIFL) et à l'Inria Lille-Nord Europe, au sein de l'équipe Spirals (Self-adaptation for distributed services and large software systems, anciennement ADAM) sous la direction du Pr. Laurence Duchien. Ses travaux ont porté sur l'élaboration d'une approche basée sur des lignes de produits logiciels pour automatiquement sélectionner et configurer un environnement de nuage en fonction des besoins de l'application à déployer. Il est actuellement post-doctorant à Politecnico di Milano (Italie), au sein du groupe Deep-Se (Dependable evolvable pervasive Software engineering), où il étudie l'évolution des lignes de produits logiciels dynamiques et le maintien de la cohérence du système généré par de telles lignes de produits lors de son adaptation à l'exécution.

Accessit : Static Analysis by Abstract Interpretation and Decision Procedures

Auteur : Julien Henry (Université de Grenoble-VERIMAG & Université du Wisconsin-Madison)

Résumé :
L'analyse statique de programme a pour but de prouver automatiquement qu'un programme vérifie certaines propriétés. L'interprétation abstraite est un cadre théorique permettant de calculer des invariants de programme. Ces invariants sont des propriétés sur les variables du programme vraies pour toute exécution. La précision des invariants calculés dépend de nombreux paramètres, en particulier du domaine abstrait et de l'ordre d'itération utilisés pendant le calcul d'invariants. Dans cette thèse, nous proposons plusieurs extensions de cette méthode qui améliorent la précision de l'analyse.

Habituellement, l'interprétation abstraite consiste en un calcul de point fixe d'un opérateur obtenu après convergence d'une séquence ascendante, utilisant un opérateur appelé élargissement. Le point fixe obtenu est alors un invariant. Il est ensuite possible d'améliorer cet invariant via une séquence descendante sans élargissement. Nous proposons une méthode pour améliorer un point fixe après la séquence descendante, en recommençant une nouvelle séquence depuis une valeur initiale choisie judiscieusement. L'interprétation abstraite peut également être rendue plus précise en distinguant tous les chemins d'exécution du programme, au prix d'une explosion exponentielle de la complexité. Le problème de satisfiabilité modulo théorie (SMT), dont les techniques de résolution ont été grandement améliorée cette décennie, permettent de représenter ces ensembles de chemins implicitement. Nous proposons d'utiliser cette représentation implicite à base de SMT et de les appliquer à des ordres d'itération de l'état de l'art pour obtenir des analyses plus précises.

Nous proposons ensuite de coupler SMT et interprétation abstraite au sein de nouveaux algorithmes appelés Modular Path Focusing et Property-Guided Path Focusing, qui calculent des résumés de boucles et de fonctions de façon modulaire, guidés par des traces d'erreur. Notre technique a différents usages : elle permet de montrer qu'un état d'erreur est inatteignable, mais également d'inférer des préconditions aux boucles et aux fonctions.

Nous appliquons nos méthodes d'analyse statique à l'estimation du temps d'exécution pire cas (WCET). Dans un premier temps, nous présentons la façon d'exprimer ce problème via optimisation modulo théorie, et pourquoi un encodage naturel du problème en SMT génère des formules trop difficiles pour l'ensemble des solveurs actuels. Nous proposons un moyen simple et efficace de réduire considérablement le temps de calcul des solveurs SMT en ajoutant aux formules certaines propriétés impliquées obtenues par analyse statique.

Enfin, nous présentons l'implémentation de Pagai, un nouvel analyseur statique pour LLVM, qui calcule des invariants numériques grace aux différentes méthodes décrites dans cette thèse. Nous avons comparé les différentes techniques implémentées sur des programmes open-source et des benchmarks utilisés par la communauté.

Biographie :
Julien Henry est post-doctorant à l'Université du Wisconsin-Madison depuis octobre 2014, avec Thomas Reps, où il travaille sur des méthodes de SAT/SMT-solving distribué. Il a obtenu un doctorant de l'Université de Grenoble en 2014 au laboratoire VERIMAG, sous la direction de David Monniaux et Matthieu Moy. Sa thèse porte sur l'analyse statique de programme par combinaisons d'interprétation abstraite et de procédures de décision SMT.

Démonstrations et Posters

Improving Reuse of Tools by means of Model Migrations

Paola Vallejo, Mickaël Kerboeuf, Jean-Philippe Babau
Univ. Bretagne-Occidentale, Lab-STICC, Brest, France

Context :

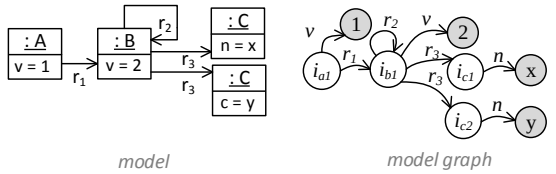
Legacy code of a dedicated tool handling domain specific data gathers valuable expertise. However, in many cases, this code must be rewritten in order to make it apply to semantically equivalent but incompatible data. This update can be complex and error-prone.

How to improve the reuse of legacy tools?

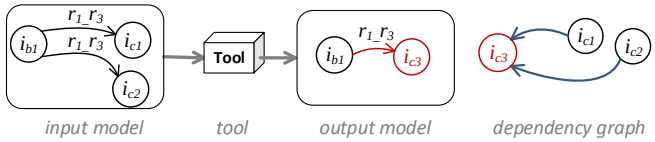
Approach : Automatic adaptation of models, instead of rewriting or adapting the tool itself

- Based on co-evolution operators (rename, remove, flatten, hide, etc.).
- Refactoring at metamodel-level.
- Migration round-trip at model-level.
  - Graph based model semantics.
  - Asymmetrical onward and reverse migrations.
  - Tool characterized by a dependency graph.

Model graph:



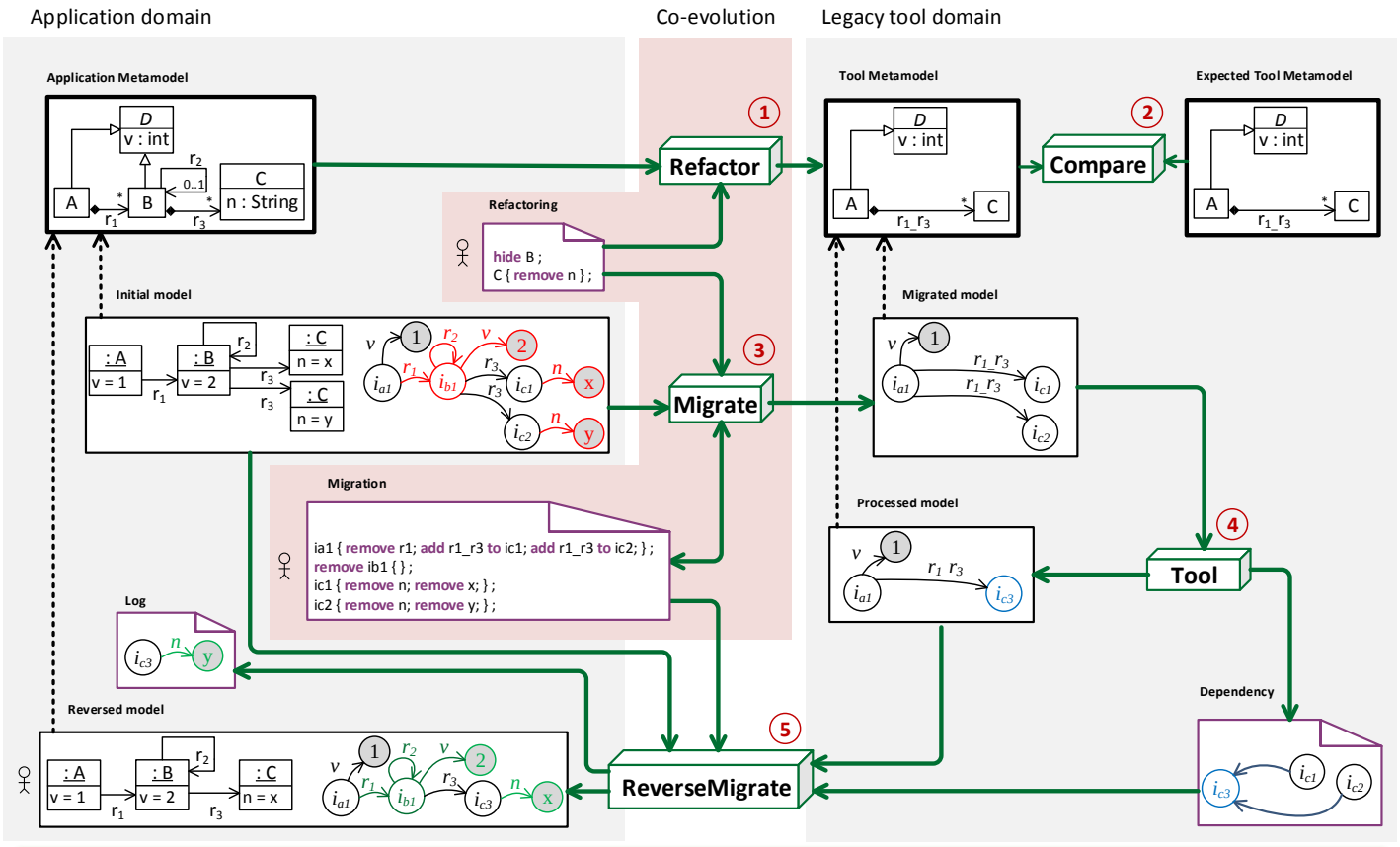
Dependency graph



Notation:

- instance #i
- scalar value
- reference
- attribute

Round-trip and Example:



Notation:

Publications:

- J.-P. Babau and M. Kerboeuf. Domain Specific Language Modeling Facilities. In ME@MoDELS, pages 1–6, 2011.
- M. Kerboeuf and J.-P. Babau. A dsml for reversible transformations. In Proceedings of the 11th OOPSLA Workshop on Domain-Specific Modeling, pages 1–6, 2011.
- M. Kerboeuf, P. Vallejo, and J.-P. Babau. Formal framework of recontextualization by means of dependency graphs. Research report, Lab-STICC UBO CACS MOCS, 2015.
- P. Vallejo, M. Kerboeuf, and J.-P. Babau. Specification of a legacy tool by means of a dependency graph to improve its reusability. In ME@MoDELS, pages 80–87, 2013.
- P. Vallejo, M. Kerboeuf, and J.-P. Babau. Adaptable model migrations. In MODELSWARD 2015 - Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development, 2015.

Self-configuration of the Number of Concurrently Running MapReduce Jobs in a Hadoop Cluster

Bo Zhang, Filip Křikava, Romain Rouvoy, Lionel Seinturier

University of Lille 1 - UMR CRIStAL / Inria Lille

1 Introduction

There is a trade-off between the number of concurrently running MapReduce (MR) jobs and their corresponding map and reduce tasks within a node in a Hadoop cluster. Leaving this trade-off statically configured to a single value can significantly reduce job response times leaving only suboptimal resource usage. This phenomenon is caused by two problems: *under-allocation* and *over-allocation* for the MR jobs. The former limits the number of MR jobs that can run in parallel, resulting in cluster node underutilization by not having enough MR tasks to run. The latter, on the other hand, allocates too much memory for MR job application master (the job supervising process) which leads to insufficient memory to process the individual MR map and reduce tasks. Moreover, in the case of time-varying workloads, a statically configured threshold negatively impact the performance as it trades one type of MR jobs (one job size) for another. A MR job size is the ratio between the maximum concurrent resource requirements of the job and the total amount of resources provided by the Hadoop cluster.

To overcome this problem, we propose a feedback control loop based approach that dynamically adjusts the Hadoop resource manager configuration based on the current state of the cluster. The preliminary assessment based on workloads synthesized from real-world traces shows that the system performance could be improved by about 30% compared to default Hadoop setup.

2 Approach

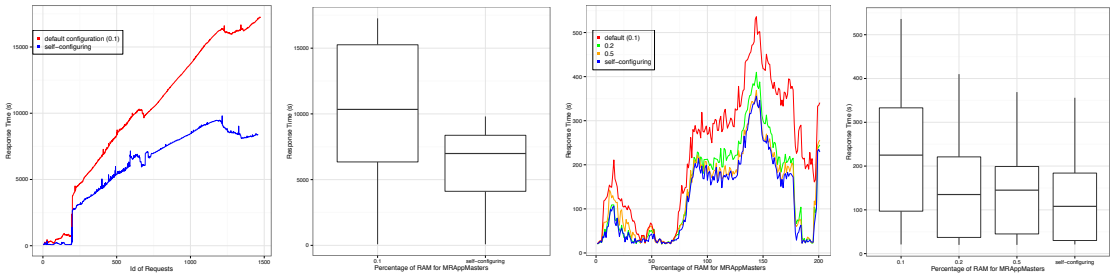
YARN separates the responsibilities between running Hadoop MR jobs[1] and their concrete map and reduce tasks into individual entities: a global, cluster-wide `ResourceManager`, a per node slave `NodeManager`, a per job MR application master (running within a `NodeManager`)—*i.e.* `MRAppMaster`, and a per MR task container (also running within a `NodeManager`)—*i.e.* `YarnChild`. Each `NodeManager` therefore runs both MR jobs (`MRAppMaster`) spawned by `ResourceManager` and a number of map and reduce task containers (`YarnChild`) that were spawned by the running `MRAppMasters`.

The objective of this work is to prevent both under- and over-allocation of the MR application masters in a Hadoop cluster. The `ResourceManager` capacity scheduler has a configuration parameter that controls the maximum concurrently running applications (C).

Our approach is based on a basic closed feedback control loop that consists of *monitoring*, *controlling* and *reconfiguration* components:

- *Monitor* periodically (every t seconds) measures the amount of memory used by both `MRAppMaster` application masters (M^{AM}) and `YarnChild` task containers (M^{YC}), as well as the number of idle MR jobs waiting to be scheduled by the `ResourceManager` (n_{idle}). The memory usage is gathered from each cluster node i together with the total available memory M^T and the results are summed up.
- *Controller* is responsible to derive a new value for the scheduler configuration parameter. At this point we use a simple algorithm that incrementally increases or decreases the value depending whether the system is in under- or over-allocation case respectively. It works as follows:
  (1) if the normalized amount of memory ($\frac{\sum M_i^{AM} + \sum M_i^{YC}}{\sum M_i^T}$) is less then a given threshold T_1 and there the number of idle jobs is increasing (n_{idle} at t_n is less then n_{idle} at t_{n+1}), the system is *under-allocated* and thus C will be incremented by a single step s that defaults to 0.05.

[1] A distributed application in general since YARN is not limited to just MR.



(a) W1: job response time (b) W1: job response time distribution (c) W2: job response time (d) W2: job response time distribution

Fig. 1. Preliminary results

(2) if the system is not under-allocated and the normalized amount of memory used by `YarnChild` is $(\frac{\sum M_i^{YC}}{\sum M_i^T})$ is less then a given threshold T_2, the system is *over-allocated* and thus C will be reduced by the same single step s.

- *Reconfigure* component simply modifies YARN setting with the new value. It does that by changing the capacity scheduler configuration file and issuing a command that makes YARN reload its settings.

3 Preliminary Results

A preliminary assessment of our work has been done on two experiments that use two different MR workloads. Both workloads were generated by the SWIM, a tool that generates representative test workloads by sampling historical MapReduce cluster traces from Facebook [**?**]. In the first experiment, the workload consisted of 1450 simple MR jobs of one map and one reduce task. In the second experiment, the workload contained 200 jobs with the size ranging from 2 to 8 tasks (1 map / 1 reduce to 7 maps / 1 reduce). The testbed used to evaluate our approach was a cluster of 5 hosts each with 7GB RAM, 2x4 cores Intel CPU 2.83GHz. The experiments results are shown in Figure **??**.

In Figure **??** the two lines represent the variations of MR jobs response time in the first experiment. The red line captures the behavior of a vanilla Hadoop configuration (*i.e.* $C = 0.1$) while the blue line shows the behavior using our approach. It shows that even if the submission rate keeps increasing (*e.g.* a load peak around the request Id 230), the response time will be reduced by our approach. Furthermore, after the load peak, the difference between the two lines widens and is even more apparent since our approach lets more MR jobs be run in parallel. In summary, the average response time of the self-configuring C is reduced about 30% compared to the vanilla configuration (*cf.* Figure **??**).

In Figure **??** we show the results of the second experiment. This time we compare our approach not only to a vanilla Hadoop configuration, but we make several runs for different values of $C \in \{0.1, 0.2, 0.5\}$. Again, we can observe, that our approach performs much better than a vanilla Hadoop configuration. With increasing C the response time decreases in most cases, but it still remains statically set and thus does not respond to to the workload dynamics. The self-configuring approach, however, keeps changing the settings and thus adjusts better to the varying runtime condition.

Acknowledgments This work is partially supported by the Datalyse project `www.datalyse.fr`.

Validation Conjointe en UML et B de Modèles de Sécurité en SI

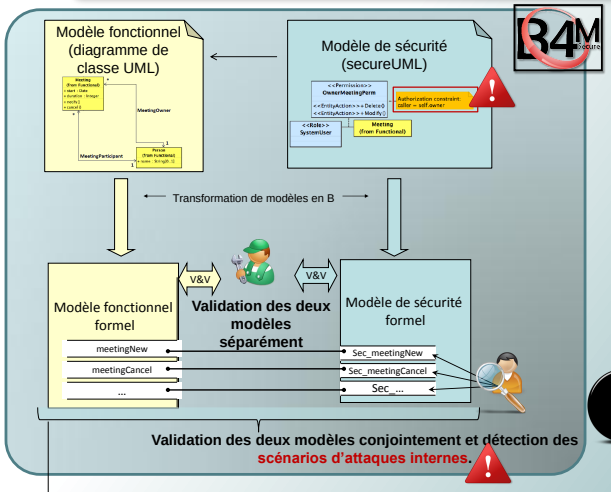
Par: Amira RADHOUANI
Sous la direction de : Yves LEDRU
Akram IDANI
Narjes BEN RAJEB

OBJECTIF

Validation formelle de politiques de contrôle d'accès sur deux plans:

- Vérification et génération automatique de cas d'usage normaux.
- Extraction des scénarios d'usage malicieux.

CONTEXTE



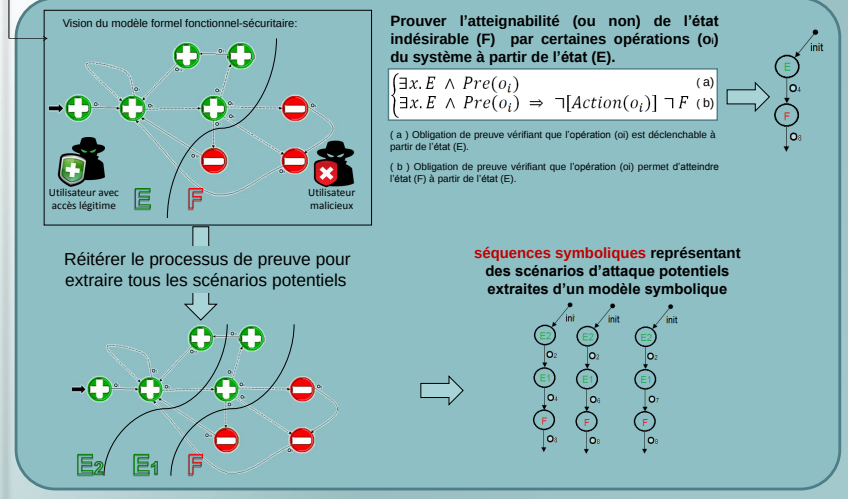
1) La validation de certaines propriétés de sécurité (comme les contraintes d'autorisation) dépend à la fois du modèle fonctionnel et du modèle de sécurité, d'où la nécessité de la prise en compte des deux modèles conjointement lors de la validation de la politique de sécurité.

2) L'extraction de scénarios d'attaques revient à trouver les chemins menant vers les états indésirables. Il s'agit, donc, d'un problème d'atteignabilité. Les techniques les plus connues pour résoudre ce genre de problème comme le model-checking souffrent du problème de l'explosion combinatoire de l'espace d'états quand le système est d'une grandeur importante.

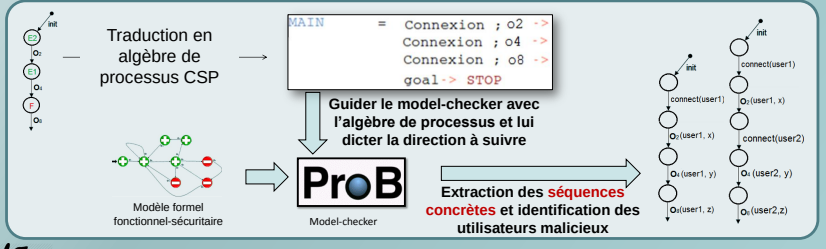
Un **scénario d'attaque interne** est l'exécution d'une séquence d'opérations par un utilisateur malicieux **ayant le droit d'accès au système**, dans le but de s'octroyer de **nouveaux droits** et réaliser des opérations auxquelles il n'avait pas droit initialement.

SOLUTION

1. Recherche par la preuve dans un modèle symbolique



2. Recherche par model-checking guidé dans le modèle concret



L'approche a permis l'extraction de scénarios d'attaque sur 3 cas d'étude. Les résultats sont très prometteurs pour l'expérimenter sur des cas réels.

PERSPECTIVES

- ❑ Application de l'approche de détection de scénarios d'attaque pour la génération et la validation de scénarios nominaux.
- ❑ Identification d'autres types d'attaques: élévation de privilège, déni de service, etc.
- ❑ Automatisation de l'approche et expérimentation en vraie grandeur.

REFERENCES

- ❑ Amira Radhouani, Akram Idani, Yves Ledru, Narjes Ben Rajeb. **Symbolic search of insider attack scenarios from a formal information system modeling.** LNCS Transactions on Petri Nets and Other Models of Concurrency (ToPNoC). Special issue of selected revised papers of FMS'2014 . (Accepté)
- ❑ Akram Idani, Yves Ledru, Amira Radhouani. **Modélisation graphique et validation formelle de politiques RBAC en Systèmes d'Information**. Ingénierie des Systèmes d'Information 19(6): 33-61 (2014). Numéro spécial « Sécurité des SI ».

"Validation de Systèmes, Composants et Objets logiciels"

Dedukti : un vérificateur de preuves universel

Ronan Saillard

Centre de recherche en informatique - MINES ParisTech

ronan.saillard@mines-paristech.fr

Contexte et objectif

- **Dedukti** est un vérificateur de type pour $\lambda\Pi$-calcul modulo.
- Le **$\lambda\Pi$-calcul modulo** est une extension du λ-calcul avec des **types dépendants** et une relation de conversion étendue par des **règles de réécriture**.
- Dedukti est capable de vérifier les preuves provenant de **prouveurs de théorèmes** (Zenon, iProver) et d'**assistants de preuves** (Coq, HOL, Focalize).
- On montre à travers plusieurs exemples que l'utilisation de la réécriture caractéristique du $\lambda\Pi$-calcul modulo permet d'obtenir des **preuves plus petites** et **plus rapides** à vérifier.

Traducteurs

Pour vérifier une preuve avec **Dedukti**, il faut d'abord l'exprimer dans le **$\lambda\Pi$-calcul modulo**. Pour cela, on utilise des **traducteurs** spécialisés :

- **Holide** traduit les preuves de l'assistant de preuve **HOL Light**.
- **Coqine** traduit les preuves de l'assistant de preuve **Coq**.
- **Zenonide** traduit les preuves du prouveur de premier ordre **Zenon**.
- **Focalide** traduit les programmes certifiés de **FoCaLiZe**.
- Une extension du prouveur **iProver** permet de traduire ses preuves.

Ces logiciels sont développés au sein de l'équipe **Deducteam** de **INRIA** par Ali Assaf, Guillaume Burel, Raphaël Cauderlier, Frédéric Gilbert et Pierre Halmagrand.

Règles de typage

$$\textbf{(Empty)}\ \frac{}{\emptyset\ wf} \qquad \textbf{(Dec)}\ \frac{\Gamma \vdash A : s \quad x \notin \Gamma}{\Gamma(x : A)\ wf}$$

$$\textbf{(Rw)}\ \frac{\Gamma\Delta \vdash l : T \quad \Gamma\Delta \vdash r : T \quad FV(r) \cap \Delta \subset FV(l)}{\Gamma([\Delta]l \hookrightarrow r)\ wf}$$

$$\textbf{(Type)}\ \frac{\Gamma\ wf}{\Gamma \vdash \textbf{Type} : \textbf{Kind}} \qquad \textbf{(Var/Cst)}\ \frac{\Gamma\ wf \quad (x : A) \in \Gamma}{\Gamma \vdash x : A}$$

$$\textbf{(Abs)}\ \frac{\Gamma \vdash A : \textbf{Type} \quad \Gamma(x : A) \vdash t : B \quad B \neq \textbf{Kind}}{\Gamma \vdash \lambda x^A.t : \Pi x^A.B}$$

$$\textbf{(Prod)}\ \frac{\Gamma \vdash A : \textbf{Type} \quad \Gamma(x : A) \vdash B : s}{\Gamma \vdash \Pi x^A.B : s}$$

$$\textbf{(App)}\ \frac{\Gamma \vdash t : \Pi x^A.B \quad \Gamma \vdash u : A}{\Gamma \vdash tu : B[x \backslash u]} \qquad \textbf{(Conv)}\ \frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A \equiv_{\beta\Gamma} B}{\Gamma \vdash t : B}$$

Règles de typage du $\lambda\Pi$-calcul modulo.

Mise en œuvre

- Un millier de lignes de code **OCaml** pour le vérificateur de types du $\lambda\Pi$-calcul modulo.
- Une machine de réduction **call-by-need** pour la **β-réduction** et les **règles de réécriture** ajoutées.
- Des règles de réécriture **linéaires ou non**.
- De la **réécriture modulo β**.
- La compilation des **règles de réécriture** en **arbres de décision** pour une réécriture efficace.
- Une **licence libre** : CeCILL-B.

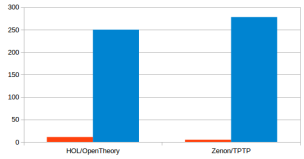
Expérience

On compare, sur deux jeux de tests différents, des encodages avec règles de réécriture et sans règle de réécriture.

- Le premier test concerne 86 fichiers de la bibliothèque de preuves et théorèmes **OpenTheory** tels qu'encodés par **Holide**.
- Le deuxième test concerne 520 fichiers correspondant à des preuves de théorèmes de la bibliothèque **TPTP**, obtenus grâce au prouveur **Zenon**. La bibliotèque TPTP est le jeu de test standard des prouveurs automatiques.

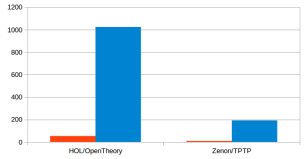
Les tests ont été réalisés sur un ordinateur Linux muni d'un processeur Intel Core i7-3520M CPU @ 2.90GHz x 4 et de 16GB de Ram.

Résultats



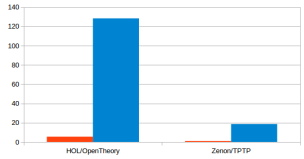
La vérification des preuves encodées en utilisant les règles de réécriture est **23 fois plus rapide** pour OpenTheory et est **55 fois plus rapide** pour TPTP.

Temps de vérification (en secondes)



La taille des fichier obtenus après encodage est environ **20 fois inférieure** lorsqu'on utilise des règles de réécriture.

Taille des fichiers (en mégaoctets)



Le nombre de tests de conversion effectués lors de la vérification des preuves de **OpenTheory** est **23 fois inférieur** pour l'encodage avec règles de réécriture et **18 fois inférieur** pour TPTP.

Nombre de tests de conversion (en millions)

Légende : avec réécriture - sans réécriture

Interprétation : Un encodage avec réécriture est nettement plus rapide a vérifier. Cela s'explique, d'abord par la taille réduite des termes obtenus par ces encodage, et ensuite par le nombre moins important de tests de conversion nécessaires à leur vérification.

Vers l'interopérabilité

- Les systèmes de preuves actuels souffrent d'un manque d'**interopérabilité**. Il est difficile de réutiliser une théorie d'un système dans un autre sans refaire toutes les preuves.
- La traduction de différents systèmes dans un formalisme commun et facilement vérifiable permettra de **combiner** leurs preuves pour construire des théories plus larges.

Référence

R. Saillard, Towards Explicit Rewrite Rules in the $\lambda\Pi$-Calculus Modulo, IWIL, Stellenbosch, 2013.

R. Saillard, Rewriting Modulo β in the $\lambda\Pi$-Calculus Modulo, Submitted, 2015.

Dedukti et les différents **traducteurs** sont disponibles à l'adresse suivante : `https://www.rocq.inria.fr/deducteam/software.html`

INCREMENTAL PATTERN-BASED MODELING AND SAFETY/SECURITY ANALYSIS FOR CORRECT BY CONSTRUCTION SYSTEMS DESIGN

Anas MOTII
CEA, LIST,
Laboratory of Model-Based Engineering for Embedded Systems (LISE), and
IRIT, University of Toulouse

Keywords: pattern-based design, model-based security risk analysis, concrete and abstract security patterns,

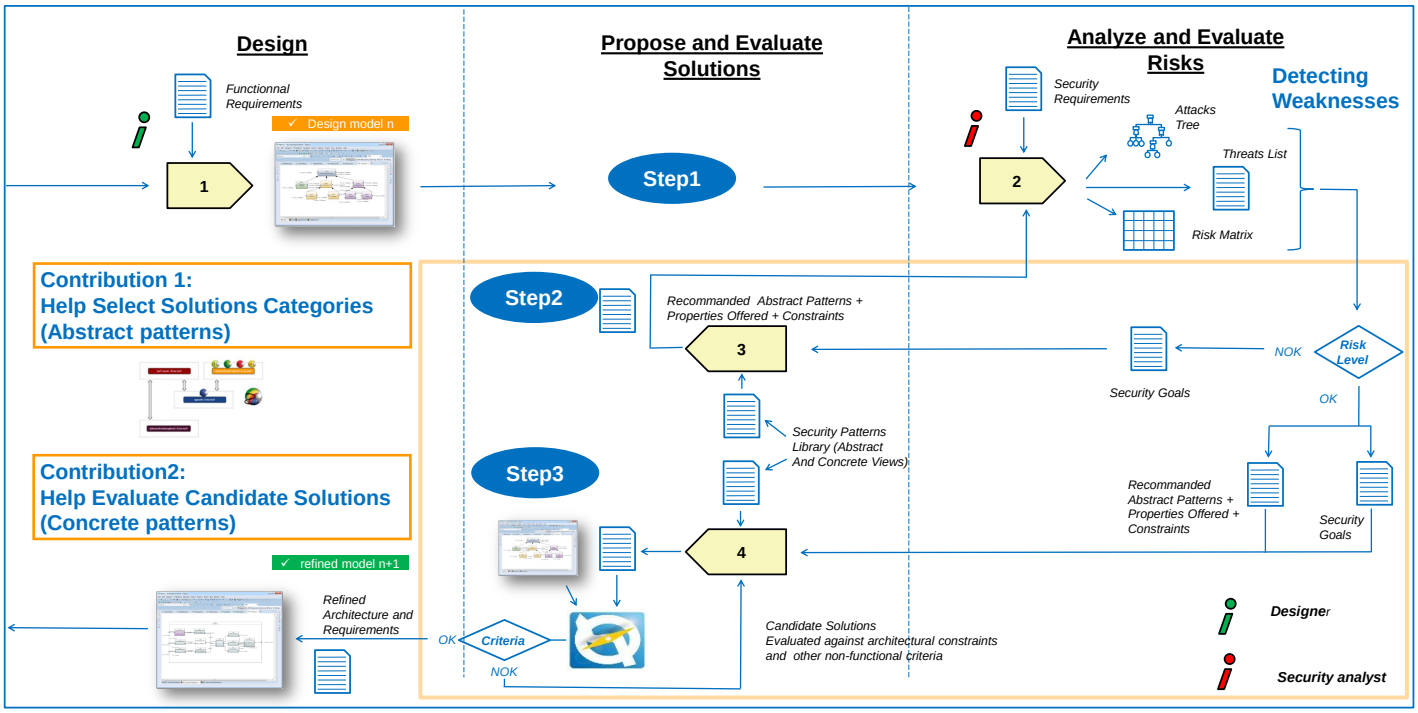
Context and Problem

- Secure software and systems design must follow strict certification prescriptions.
- Very costly process (money and time)
  - because of late treatment of errors.
  - because of complexity due to lack of separation of concerns.
- Requires expertise and continuous improvement

Objectives

- Objective 1. Consider security from the beginning of software and systems development.
- Objective 2. Exploit the benefits of Model Driven Engineering (MDE): separation of concerns and refinement support (through model transformations).
- Objective 3. Exploit security expertise and foster reuse and exchange between designers and security analysts using artifacts called "security patterns".

Approach



Contributions and Future work

- Contributions:
  - Pattern modeling and integration.
  - Pattern selection assistance with STRIDE attacks.
  - Validation of pattern integration (integration analysis).
  - Proof of concept of the process using a SCADA (Supervisory Control And Data Acquisition) system example.
- Future work:
  - Extend SOPHIA framework with security risk analysis.
  - Extend SEMCO framework with pattern modeling and integration.
  - Compare our process with forefront processes: Microsoft's SDL, OWASP's CLASP and McGraw's Touchpoints.

Contacts:
Anas MOTII, anas.motii@cea.fr
Brahim HAMID, brahim.hamid@irit.fr
Agnès LANUSSE, agnes.lanusse@cea.fr
Jean-Michel BRUEL, bruel@irit.fr

This work is supported by a doctoral grant from CEA LIST and is

Binary analysis for security purposes: approaches based on model inference

Franck de Goër

Laboratoire d'Informatique de Grenoble & Vérimag

1 PhD - Overview

1.1 Context

An increasing number of applications are shared over the internet, through smartphone application markets, web browser extensions, and download links on a website. Therefore, the potential impact of vulnerabilities in widely-deployed applications, either intentional (in case of malwares) or accidental (a bug being exploited) is considerable. Our research addresses two directions in the analysis of software artifacts based on binary code:

- vulnerability detection: accurate methods may not be scalable to large programs and fast analyses often suffer from both false-positive and false-negative results, inducing a lack of confidence in results.
- malware detection: a few efficient techniques exist to identify known malwares, but very few results are known about malware detection.

This PhD thesis places itself in this context of software behavior analysis, trying to combine static and dynamic analysis to gain information about binaries.

1.2 Overall directions

The goal of this research is to develop a new approach which combines existing methods in order to infer information about a binary whose source code is not available. The kind of information we try to infer is the behavior of the software w.r.t. its environment (including shared memory and key resources). For instance, we would try to retrieve an embedded allocator and possibly its allocating strategy, but do not attempt to analyze each instruction. Therefore, implementation-related vulnerabilities (such as buffer-overflows and use-after-frees) are not directly our target ; however retrieving allocating and liberating functions provide valuable information to detect them. The approach we propose is a combination of existing techniques. Mainly, we want to use *inference of state based models*, combined with *static* and *dynamic analysis techniques*.

2 First step

According to the global context of the PhD, we work on analyzing a compiled binary, possibly stripped, without using source code (which may be available or not). Our first step is to retrieve function prototypes and function dataflow coupling. Our approach is based on binary instrumentation using `Pin`.

2.1 Approach

Prototypes From a (stripped) binary, we want to retrieve functions, embedded or linked, and more precisely:

- entry point and number of arguments
- number of dynamic calls

– type of each argument and return value

Existing research already addresses prototype recovery from a (stripped) binary. However, as our work takes place in a larger approach including several interconnected steps, we do not aim to be sound nor complete from a theoretical point of view at this particular step (though we claim to obtain results that are accurate). Therefore, we can develop heuristics to make the analysis faster than other methods that try to be sound or complete (or both).

Dataflow coupling In addition to prototypes, we also try to infer relations between functions. In particular, we find couplings between the output of a function and a parameter of another one.

2.2 Rationale

The motivation of the inference of this information is to guide the model inference. First, to be able to extract a model from a binary, we need to interact with it, that is why we need function locations and prototypes. Second, knowing how data flows from the output of a function to an input of another function is essential for further steps of analysis.

3 Preliminary results

We present here the first results we obtained with the approach proposed in section 2.

3.1 Inference of function prototypes

Relatively to prototype inference, we tested our implementation on twenty functions from `C Standard Library`, with one or several parameters and different types (address, float, integer). We retrieve correctly the arity of each function tested. We also retrieve types of each parameter and return value. Figure 1 shows part of the results we obtain.

3.2 Coupling

On small programs that use either the `C Standard Library` allocator or a custom allocator, we retrieve the couple `malloc -> free` (or equivalent) and the couple `fopen -> fclose` when those two functions are used. We also retrieve other couplings, but they need a deeper understanding of the binary under analysis to be explained. Figure 2 shows our results, where the value returned by function f is passed as i^{th} parameter of function g.

address	name	prototype
0x400beb	calloc	void *f(int, int)
0x400c73	fclose	int f(void *)
0x400c8d	fopen	void *f(void *, int)
0x400cd4	free	int f(void *)
0x400d35	malloc	void *f(int)
0x400d8e	realloc	void *f(void *, int)

Fig. 1. Arity inference on some `libc` functions

f	g[i]	ρ
fopen	fclose[1]	1
fopen	fputc[2]	1
malloc	free[1]	0.96
malloc	realloc[1]	1
realloc	free[1]	0.96

Fig. 2. Coupling correlation factor ρ on `libc`

$\mathcal{G}$rimm : un assistant à la conception de méta-modèles par génération d'instances

Adel Ferdjoukh, Eric Bourreau, Annie Chateau, Clémentine Nebut
{ferdjoukh,bourreau,chateau,nebut}@lirmm.fr

Introduction et Problématique

En Ingénierie Dirigée par les Modèles, concevoir des méta-modèles est une tâche importante et récurrente. Néanmoins, très peu d'outils permettent d'aider à l'accomplissement de cette tâche. Nous proposons dans [1, 2] une approche basée sur les *Problèmes de Satisfaction de Contraintes* (CSP) et un outil dont le but est d'assister le designer d'un méta-modèle en générant des instances visualisées graphiquement et modifiables. Un processus de génération d'instances de méta-modèles efficace doit respecter certaines **caractéristiques**:

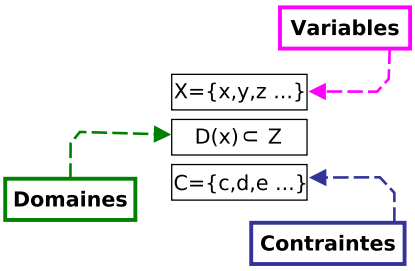
- Automatisation
- Passage à l'échelle
- Prise en charge d'OCL
- Modèles réalistes
- Diversité des modèles

Aperçu sur les CSP

CSP est un paradigme issu de l'intelligence artificielle permettant la résolution de problèmes combinatoires (Emploi du temps, Sudoku ...).

Un CSP est définit par Freuder: "*Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.*."

Formellement un CSP est triplet (X, D, C) où:



Une **solution** d'un CSP est une instanciation de toutes les variables respectant toutes les contraintes simultanément.

References

[1] Ferdjoukh, A., Baert, A.-E., Bourreau, E., Chateau, A., Coletta, R., and Nebut, C. Instantiation of Meta-models Constrained with OCL: a CSP Approach. In *MODELSWARD* (2015), pp. 213–222.

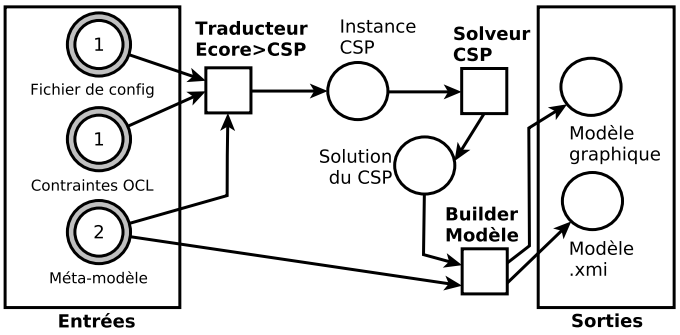
[2] Ferdjoukh, A., Baert, A.-E., Chateau, A., Coletta, R., and Nebut, C. A CSP Approach for Metamodel Instantiation. In *IEEE ICTAI* (2013), pp. 1044–1051.

Télécharger $\mathcal{G}$rimm

L'outil est disponible ici:
`www.lirmm.fr/~ferdjoukh`

Processus de Génération de Modèles

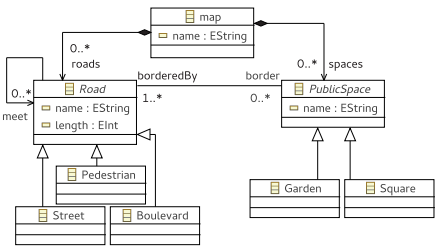
- Processus de génération de modèles par CSP.



- Processus totalement automatique. L'utilisateur fournit le méta-modèle et ses contraintes OCL. Un fichier de configuration sert à paramétrer l'outil (Nombre d'instances par classe, domaines des attributs).
- Traduction des classes, attributs, références et contraintes OCL d'un méta-modèle en CSP.
- Pour une modélisation en CSP efficace et qui passe à l'échelle: (1) peu de variables et contraintes, (2) Optimisation des références bidirectionnelles, (3) Contraintes globales, (4) Modélisation spécifique pour OCL.
- Les **caractéristiques** importantes de $\mathcal{G}$**rimm**:

- 100% automatique
- Visualisation graphique et xmi
- Support partiel d'OCL
- Paramétrisation totale
- Gestion des symétries

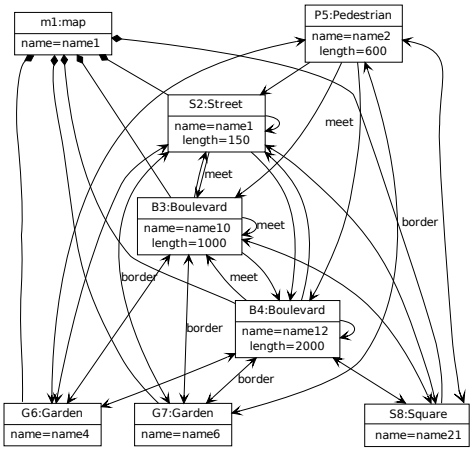
Exemple Concret



OCL Constraints:

Context Road inv: length >= 0

Context map inv: spaces->forall(n1,n2 | n1 <> n2 implies n1.name <> n2.name)



- La figure de gauche représente un méta-modèle décrivant des cartes de villes et deux contraintes OCL que ses instances doivent respecter.
- La figure de droite est une instance possible du méta-modèle, elle est générée automatiquement par $\mathcal{G}$rimm en respectant les deux contraintes OCL données.

Conclusion & Perspectives

- ok Automatisation — Grimm est 100% automatique
- ok Passage à l'échelle — Test sur Gros modèles et méta-modèles
- ok Prise en charge d'OCL — Support automatique partiel d'OCL
- ... Modèles réalistes — Proche de la réalité / à améliorer
- ... Diversité des modèles — Distance de Hamming / Définir sa propre distance

COMUE L'UNAM
École Doctorale STIM
Sciences et Technologies de l'Information et Mathématiques

Spécialité : Informatique
Laboratoire : Inria, LINA
Équipe : ASCOLA

Walid Benghabrit
walid.benghabrit@mines-nantes.fr

Making the cloud more accountable with AccLab framework

Director : Jean-Claude Royer
Co-supervisor : Hervé Grall

Context

The cloud :

Set of services

Availability Scalability

Fault tolerance

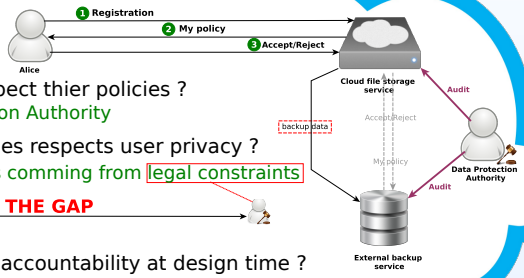
Personal data environment :

- Interconnected systems with many different technologies which implies many security breaches

Your personal information are already on the cloud !

Problematic

Motivating exemple :



- What forces them to respect thier policies ?
  - Audit by a Data Protection Authority
- How to ensure that policies respects user privacy ?
  - Must respect obligations comming from legal constraints

THE GAP

How to handle accountability at design time ?

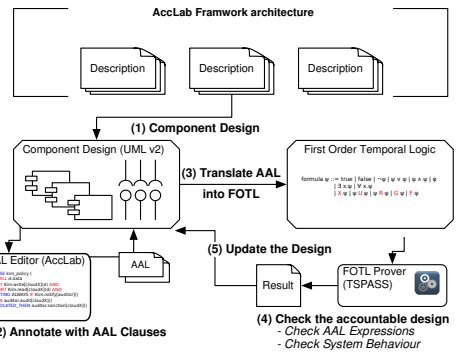
Approach

- Static system analysis
- Static policies complience
- Dedicated policy language AAL
  (Abstract Accountability Language)

AccLab
Accountability Laboratory

The framework

Steps :



(1) We design the architecture using a component diagram.

(2) We annotate the diagram with actors policies using an abstarct accountability language AAL.

(3) The diagram and policies are automatically translated into temporal logic formula FOTL.

(4) Policies compliance and system behavior are automatically checked using a logical prover.

(5) We update the design and the policies according to prover's result.

Ongoing work

- Synthesizing reference monitors for each actors
  (Control the behavior of actors to check if they respect their policy)
- Simulation platform (Describe actors behavior and observe the system's reaction)
- Policy enforcement (Translating AAL policies into executable policies)

CLOUD ACCOUNTABILITY PROJECT
Inria
MINES
lina
l'université nantes angers le mans

Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava, Julien Tesson

LACL, Université Paris Est

The BSP Model

In the BSP model [1], a computer is a set of **p** uniform processor-memory pairs and a communication network. A BSP program is executed as a sequence of *super-steps* (Fig. 1), each one divided into three successive disjoint phases:

1) Each processor only uses its local data to perform sequential computations and to request data transfers to other nodes;
2) The network delivers the requested data;
3) A global synchronisation barrier occurs, making the transferred data available for the next super-step.

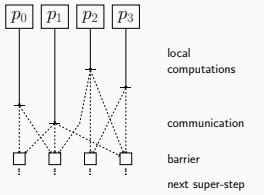


Figure 1: A BSP super-step

The Multi-BSP Model

The MULTI-BSP model [2] is another *bridging model* as the original BSP, but adapted to *clusters of multicores*. The MULTI-BSP model introduces a vision where a *hierarchical architecture* is a *tree* structure of *nested components* (*sub-machines*) where the lowest stage (*leaf*) are processors and every other stage (*node*) contains memory. A node executes some codes on its nested components (*aka* "*children*"), then waits for results, do the communication and synchronised the sub-machine.

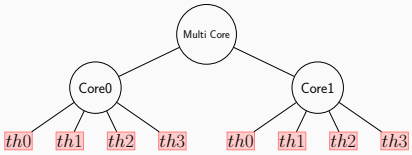


Figure 2: A MULTI-BSP view of a multi-core architecture

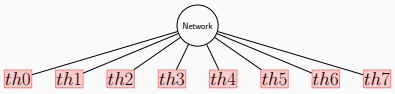


Figure 3: A BSP view of a multi-core architecture

For a multicore architecture it is possible to distinguish all the level thanks to MULTI-BSP (Fig. 2). On the contrary, the BSP model (Fig. 3) flattens the architecture.

Benchmarks

Fig. 4 shows the results of our experimentations. We can see that the efficiency on small list is poor but as the list grows, MULTI-ML exceeds BSML. This difference is due to the fact that BSML communicates through the network at every super steps; while MULTI-ML focusing on communications through local memories and finally communicates through the distributed level.

	100_000		1_000_000		3_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	125.3	430.7	...	...
16	0.5	0.8	68.1	331.5	1200.0	...
32	0.3	0.5	11.3	122.2	173.2	...
48	0.5	0.4	5.5	88.4	69.3	...
64	0.3	0.3	4.1	56.1	51.1	749.9
96	0.3	0.38	3.9	30.8	38.1	576.1
128	0.5	0.45	4.7	24.3	30.6	443.7

Figure 4: Execution time of Eratosthenes (naive) using MULTI-ML and BSML.

Fig. 5 gives the computation time of the simple scan using a summing operator. We can see that MULTI-ML introduce a small overhead due to the level management; however it is as efficient as BSML and concord to the estimated execution times.

	5_000_000			
	MULTI-ML	BSML	*Pred*_MULTI-ML	*Pred*_BSML
8	2.91	2.8	3.44	1.83
16	1.42	1.4	1.72	0.92
32	0.92	0.73	0.43	0.46
48	0.84	0.75	0.28	0.31
64	0.83	0.74	0.21	0.23

Figure 5: Execution time and predictions of scan (sum of integers)

BSP Programming in ML : BSML

BSML [3] uses a *small set of primitives* and is currently implemented as a library for the ML programming language OCAML. A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its ML type is **'a par**. A vector expresses that each of the **p** processors *embeds* a value of any type **'a**. The BSML primitives are summarized in Fig. 6 :

Primitive	Level	Type	Informal semantics
≪e≫	g	'a **par** (if e:'a)	$\langle \mathsf{e}, \ldots, \mathsf{e} \rangle$
pid	g	int **par**	A predefined vector: i on processor i
v	l	'a (if v: 'a **par**)	v_i on processor i, assumes $\mathsf{v} \equiv \langle \mathsf{v}_0, \ldots, \mathsf{v}_{\mathbf{p}-1} \rangle$
proj	g	'a **par**→(int→'a)	$\langle x_0, \ldots, x_{p-1} \rangle \mapsto (\textbf{fun}\ i \to x_i)$
put	g	(int→'a)**par**→(int→'a)**par**	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\textbf{fun}\ i \to f_i\ 0), \ldots, (\textbf{fun}\ i \to f_i\ (\mathbf{p}-1)) \rangle$

Figure 6: The BSML primitives

An example of a parallel vector construction using the BSML toplevel :

```
#let vec = ≪ "HLPP"≫ in ≪ $vec$^"␣proc␣"^(string_of_int $pid$) ≫ ;;
  val vec : string par = <"HLPP␣proc␣0", "HLPP␣proc␣1", "HLPP␣proc␣2">
```

The Multi-ML language

MULTI-ML is based on the idea of executing a BSML-like code on every stage of the MULTI-BSP architecture, that is on every sub-machine. For this, we add a *specific syntax* to ML in order to code special functions, called *multi-functions*, that recursively go through the MULTI-BSP tree. At each stage, a multi-function allows the execution of any BSML code. The main idea of MULTI-ML is to structure parallel codes to control all the stage of a tree: we generate the parallelism by allowing a node to call recursively a code on each of its sub-machines (children). When leaves are reached, they will execute their own codes and produce values, accessible by the top node using a vector. The data are distributed on the stages (toward leaves) and results are gathered on nodes toward the root node as shown in Fig. 7. Let us consider a code where, on a node, ≪ e ≫ is executed. As shown in Fig. 8, the node creates a vector containing, for each sub-machine i, the expression **e**. As the code is run asynchronously, the execution of the node code will continue until reaching a barrier.

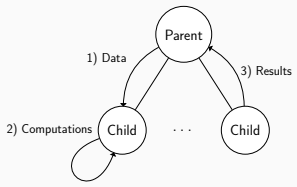


Figure 7: Code propagation

Fig. 9 shows the MULTI-ML primitives (without recall the BSML ones); their authorised level of execution and their informal semantics.

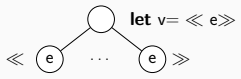


Figure 8: Data distribution

Primitive	Level	Type	Informal semantics
§e§	m	'a **tree**	Build $\wr \mathsf{e} \wr$, a tree of **e**
t	s	'a	In a §e§ code, t_n on node/leaf n of the tree $\wr t \wr$
v (if v: 'a **tree**)	b	'a	v_n on node n of tree $\wr v \wr$,
v	l	'a	In the ith component of a vector, $\mathsf{v}_{n,i}$ on node/leaf n of the tree $\wr v \wr$
gid	m	id	The predefined tree of nodes and leaves ids
≪...f...≫	l	'a	In a component of a vector, recursive call of the multi-function
#x#	l	'a	In a component of a vector, reading the value x at upper stage (id)
mkpar f	b	'a **par**	$\langle v_0, \ldots, v_{p_n} \rangle$, where $\forall i, \mathsf{f}\ \mathsf{i} = v_i$, at id n of the tree
finally v_1 v_2	b,s	'a	Return value v_1 to upper stage (id) and keep v_2 in the tree
this	b,l,s	'a option	Current value of the tree if exists, None otherwise

Figure 9: The MULTI-ML primitives

An example of a tree construction using the MULTI-ML toplevel :

```
#let multi f n =
  where node =
    let _=≪f ($pid$+ #n#+ 1)≫ in
    finally ~up:() ~keep:(gid^"=>"^n)
  where leaf=finally ~up:() ~keep:(gid^"=>"^n);;
val f : int→string tree = <multi-fun>
```

```
#f 0
o "0→ 0"
o "0.0→ 1"
    → "0.0.0→ 2"
    → "0.0.1→ 3"
o "0.1→ 2"
    → "0.1.0→ 3"
    → "0.1.1→ 4"
```

References

[1] L. G. Valiant. "A Bridging Model for Parallel Computation". In: *Comm. of the ACM* 33.8 (1990), pp. 103–111.
[2] L. G. Valiant. "A bridging model for multi-core computing". In: *J. Comput. Syst. Sci.* 77.1 (2011), pp. 154–166.
[3] Louis Gesbert et al. "Bulk Synchronous Parallel ML with Exceptions". In: *Future Generation Computer Systems* 26 (2010), pp. 486–490.

SysDICE: Integrated Conceptual Design Evaluation of Mechatronics Systems using SysML

Mohammad Chami and Jean-Michel Bruel

IRIT/University of Toulouse

Université de Toulouse — MACAO — IRIT — CNRS - INPT - UPS - UT1 - UTM

Contact:
M.Sc. Mohammad Chami
PhD Student at University of Toulouse
IRIT Research Laboratory (MACAO team)
Mohammad.Chami@irit.fr

1. Selected Problems in Mechatronic Design

- Complex system structure, various levels of abstraction, wide range of models
- It is agreed that there is no "one accepted design and development methodology"
- Sophisticated IT-tools within the different domains, but hardly cross-domain IT-support
- Weak collaboration across the domain specific phases
- An early evaluation adhering to adaptable design requirements is still missing
- Lack of knowledge about the counterparts domain
- Distributed development: different departments, companies, countries, ...

2. Research Objective and Questions

2.1 Research Objective:

- The research scope concerns mainly the usability of model based system engineering approaches and artificial intelligence techniques for supporting the mechatronic design.
- It explores an integrated conceptual design evaluation approach for mechatronic systems based on SysML, which we have called SysDICE.
- Notably, SysML is used as the common language between the discipline engineers and the system engineers to form a system model. Later on, this system model is formalized and transferred accordingly in order to make it executable and applicable for artificial intelligence algorithms aiming at evaluating design criteria with different configurations.

2.2 Research Questions:

- **RQ_1**: How to achieve an integrated multidisciplinary mechatronic system design model?
  - What type of models are required and which methods can be used to capture the exchanged information?
  - How to provide and assure models integration between domain specic models and domain independent models?
- **RQ_2**: How to execute the model attained from RQ_1 while assuring consistency and managing increasing complexity?
  - Namely, which meta-models and model transformations are necessary and what type of mathematical formulations can be employed?
- **RQ_3**: How to support system engineers during the conceptual design evaluation?
  - Notably, which evaluation goals and what kind of support is required while assuring a multi-alternative design mechanism, standardization and enhancing traceability?

3. Research Hypothesis

- **RH_1**: RQ_1 is concerned with the integration challenges. Hereby, SysML is adopted to model the interdisciplinary system model. SysML diagrams shall capture the various interdisciplinary elements and their relationships. A SysDICE profile shall be developed to identify the evaluation goals and their relation to other model elements.
- **RH_2**: RQ_2 deals with the transformation of the interdisciplinary model into an executable version. It is very related to RH_1 and RH_3. Therefore, the generated SysML model must be based on a predefined meta-model in order to transfer it and be able to execute it under the use cases of RH_3.
- **RH_3**: RQ_3 requires computer interpretable techniques for the design evaluation. A tool shall be implemented to support system engineers (i.e.: SysML-tools plugins) with the interdisciplinary information and being able to apply their constraints, variations and configurations on the system level and evaluate it during their decision making process. Moreover, a SysDICE method will be developed to described the tasks to answer RQ_1 and RQ_3.

4. SysDICE Approach

SysDICE Methodology

1. **System Model Generation**:
   (a) Adopting SysML for capturing different domains information (requirements, functional, structural, behavior, parametric)
   (b) Mapping the domain-specific models to SysML models (domain-independent)
   (c) Using SysML for managing the interdisciplinary model and providing traceability
2. **System Model Transformation**:
   (a) Transferring the SysML model to an executable model version
   (b) Working on the mapping between SysML and its execution engine
   (c) Assuring consistency and validating the SysML model
3. **System Model Evaluation**:
   (a) Modeling the design evaluation criteria (the evaluation goals)
   (b) Applying the necessary configurations to the model
   (c) Evaluating the conceptual design

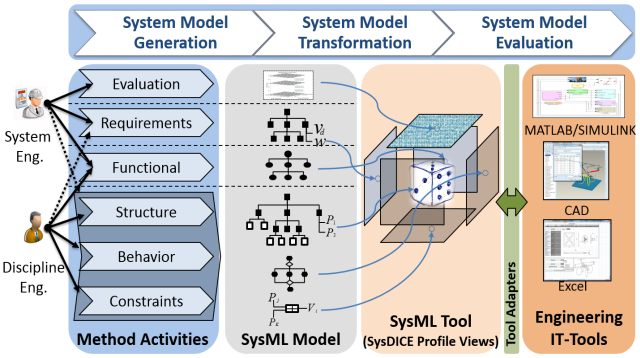


Figure 1: *SysDICE overall framework*

5. Preliminary Work and Application Example

The design of a two wheel differential drive robot illustrates the application SysDICE for modeling the robot with SysML (via MagicDraw tool) and applying the mathematical formulation (via MATLAB) to find the optimal combination of components alternatives for a specific evaluation goals with different design focus configurations:

- High speed and operation time
- Low price and moderate operation time
- Low price and moderate speed

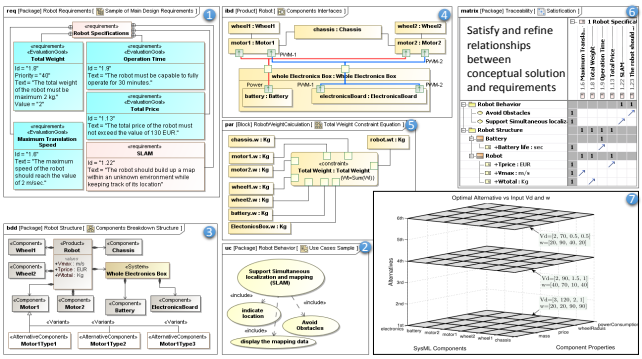


Figure 2: *SysML Diagrams and results of three different evaluation goals configurations (Vd=[weight,price,speed,time])*

Mathematical Formulation (Gaussian Processes Application)

- Desired values of a set of k requirements: $\mathbf{v}_d = [v_d^{(1)}, \ldots, v_d^{(k)}]^T \in \mathbb{R}^{k\times 1}$
- Priorities of each requirement: $\mathbf{W}_{k,k} = diag(\mathbf{w})$
- Output of the constraint equations: $\mathbf{v} = [v_1, \ldots, v_k]$
- Likelihood for a desired value to occur is define as:

$$p(v_d^{(i)}|v^{(i)}; \sigma^2, w^{(i)}) = \prod_{i=1}^{k} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2\sigma^2} w_{i,i} (v_d^{(i)} - v^{(i)})^2\right) \quad (1)$$

- Maximizing the natural logarithm of Equation 1 leads to minization problem:

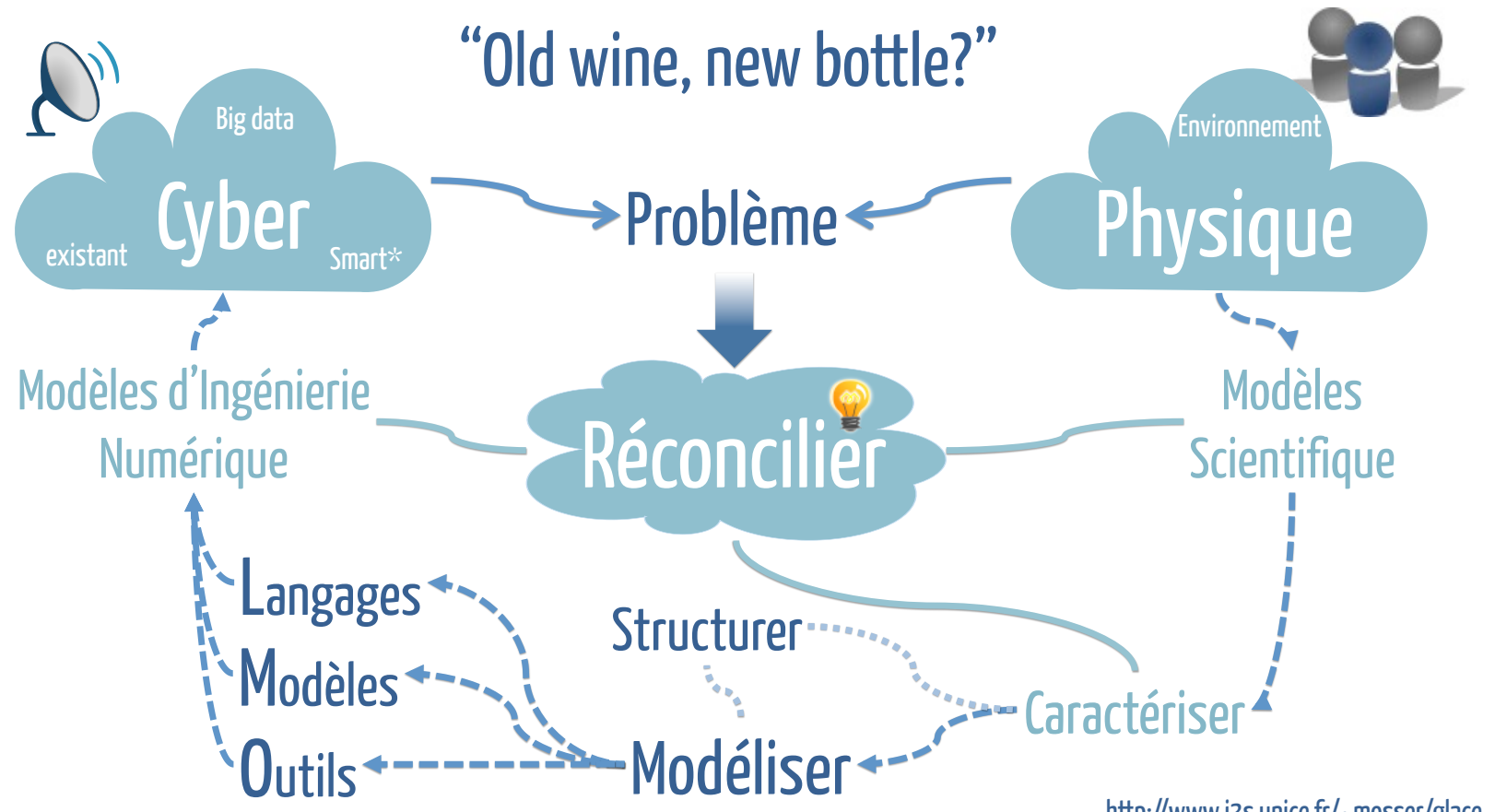
$$\min_{\mathbf{v}} \frac{1}{2} [\mathbf{v} - \mathbf{v}_d]^T \mathbf{W} [\mathbf{v} - \mathbf{v}_d] \quad (2)$$

- The approximated functions are then substituted in Equation 2, to generate a new minimization problem defined by the following cost function:

$$\min_{P} J(P) = \frac{1}{2} \sum_{i=1}^{k} w_{i,i} (\mathcal{GP}_i(P) - v_d^{(k)})^2, \quad (3)$$

where $P = p_1 \otimes p_2 \cdots \otimes p_N$, with N being the number of components, and J representing the cost function.

GL/\CE : Génie Logiciel pour les systèmes Cyber-physiquEs



http://www.i3s.unice.fr/~mosser/glace

Sébastien Mosser (I3S), Romain Rouvoy (CRIStAL), Benoit Combemale (IRISA), Francisco Javier Acosta Padilla (IRISA)

GL/\CE est un groupe de travail du GDR GPL, ce poster est une reflexion menée lors de la 1ère journée de travail du groupe

Ce document contient les actes des Septièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées au LaBRI à l'Université de Bordeaux du 10 au 12 juin 2015.

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.