Actes des deuxièmes journées nationales du

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

Université de Pau
10 au 12 Mars 2010

Actes des deuxièmes journées nationales du

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

Université de Pau
10 au 12 Mars 2010

Editeurs :

Eric Cariou
Laurence Duchien
Yves Ledru

Impression : Centre de reprographie de l'Université de Pau et des Pays de l'Adour
Site de Pau
Avenue du Doyen Poplawski
BP 1160
64013 PAU CEDEX
Tél : 05 59 40 73 11
E-mail : reprographie.pau@univ-pau.fr

Photographies : Eric Cariou

Table des matières

Préface **7**

Comités **9**

Conférenciers invités **11**

Ivica Crnkovic : *A Classification Framework for Component Models* 11

Pierre-Etienne Moreau : *Combiner Java et réécriture, c'est possible et utile* 11

Gérard Berry : *Qualité du logiciel : certification vs. vérification* 12

Action AFSEC **13**

Marc Pouzet (LRI / Univ. Paris-Sud 11) et Pascal Raymond (VERIMAG)
Modular Static Scheduling of Synchronous Data-flow Networks – An efficient symbolic representation 15

Marc Boyer et David Doose (ONERA)
Collaboration entre méthode d'ordonnancement et calcul réseau 21

Jean-Luc Scharbarg, Jérôme Ermont (Univ. Toulouse / IRIT / ENSEEIHT / INPT), Henri Bauer (Univ. Toulouse / IRIT / ENSEEIHT / INPT, Airbus France) et Christian Fraboul (Univ. Toulouse / IRIT / ENSEEIHT / INPT)
Analyse des délais de bout en bout pire cas dans des réseaux avioniques 37

Groupe de travail COSMAL **53**

Jannik Laval, Alexandre Bergel, Stéphane Ducasse et Romain Piers (INRIA Lille Nord Europe / USTL / CNRS)
Matrice de dépendances enrichie 55

Floréal Morandat, Roland Ducournau (LIRMM / Univ. Montpellier 2) et Jean Privat (UQAM – Canada)
Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique . 71

Mohamed Messabihi, Pascal André et Christian Attiogbé (LINA / CNRS / Univ. Nantes)
Correction d'assemblages de composants impliquant des données et assertions 87

Groupe de travail FORWAL **103**

Alexander Heußner, Jérôme Leroux, Anca Muscholl et Grégoire Sutre (LaBRI /Univ. Bordeaux)
Reachability Analysis of Communicating Networks with Pushdowns 105

Action IDM **109**

Sagar Sen, Naouel Moha, Benoît Baudry et Jean-Marc Jézéquel (INRIA Rennes Bretagne Altantique)
Meta-model Pruning 111

Pierre-Alain Muller, Frédéric Fondement (Univ. Haute-Alsace) et Benoît Baudry (INRIA Rennes Bretagne Altantique)
Modeling Modeling 113

Jean-Philippe Babau (Univ. Brest) et Sylvain Robert (CEA)
Retours sur l'école de printemps Model-Driven Development for Distributed Realtime Embedded Systems 115

Groupe de travail LaMHA **117**

Mathias Bourgoin, Benjamin Canou, Emmanuel Chailloux, Adrien Jonquet et Philippe Wang (LIP6 / CNRS / Univ. Paris 6)
OC4MC : Objective Caml for Multicore Architectures 119

Frédéric Gava (LACL / Univ. Paris-Est) et Ilias Garnier (LIST / CEA Saclay)
New Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons 137

Mohamad Al Hajj Hassan et Mostafa Bamha (LIFO / Univ. Orléans)
A Scalable Parallel Algorithm for Join Queries Evaluation on Heterogeneous Distributed Systems 141

Groupe de travail LTP **145**

Benoît Montagu et Didier Rémy (INRIA Paris-Rocquencourt)
Types Abstraits et Types Existentiels Ouverts 147

Louis Mandel, Florence Plateau et Marc Pouzet (LRI / Univ. Paris-Sud 11 / INRIA Saclay)
Lucy-n : une extension n-synchrone de Lustre 149

Evelyne Contejean (LRI / Univ. Paris-Sud / CNRS), Pierre Courtieu (Cédric / CNAM), Julien Forest (Cédric / ENSIIE), Andrei Paskevich (LRI / Univ. Paris-Sud / CNRS), Olivier Pons (Cédric / CNAM) et Xavier Urbain (LRI / Univ. Paris-Sud / CNRS, Cédric / ENSIIE)
A3PAT, an Approach for Certified Automated Termination Proofs 153

Groupe de travail MFDL **155**

Thomas Bochot (Airbus France, ONERA / DTIM), Pierre Virelizier (Airbus France), Hélène Waeselynck (LAAS / CNRS / Univ. Toulouse) et Virginie Wiels (ONERA / DTIM)
Application du Model Checking aux commandes de vol : l'expérience Airbus 157

Abderrahman Matoussi, Frédéric Gervais et Régine Laleau (LACL / Univ. Paris-Est)
Définition d'une sémantique Event-B pour les patrons de raffinement de buts KAOS . . 167

Groupe de travail MTV2 **187**

Johan Oudinet (LRI / Univ. Paris-Sud / CNRS)
Exploration aléatoire de modèles 189

Amel Mammar, Ana Cavalli, Willy Jimenez (Télécom SudParis / CNRS / SAMOVAR), Edgardo Montes de Oca (Montimage), Shanai Ardi, David Byers et Nahid Shahmehri (Linköpings universitet – Suède)
Modélisation et Détection Formelles de Vulnérabilités Logicielles par le Test Passif . . . 205

Omar Chebaro (CEA / LIST, LIFC / Univ. Franche-Comté), Nikolai Kosmatov (CEA / LIST), Alain Giorgetti (LIFC / Univ. Franche-Comté, INRIA Nancy Grand Est) et Jacques Julliand (LIFC / Univ. Franche-Comté)
Combining Frama-C and PathCrawler for C Program Debugging 217

Groupe de travail RIMEL **219**

An Phung-Khac, Jean-Marie Gilliot et Maria-Teresa Segarra (Télécom Bretagne)
Une architecture de composants répartis adaptables 221

Eric Cariou, Nicolas Belloir, Franck Barbier et Nidal Djemam (LIUPPA / Univ. Pau)
OCL contracts for the verification of model transformations 237

Groupe de travail Transformations **253**

Marc Pantel (IRIT / Univ. Toulouse), Nassima Izerrouken (Continental, IRIT /Univ. Toulouse), Adrien Champion (IRIT / Univ. Toulouse), Jean-Charles Dalbin (Airbus SAS) et Frédéric Pothon (ACG Solutions)
A pragmatic structural approach for the specification and verification of model transformations 255

Jérôme Delatour, Matthias Brun, Guillaume Savaton, Jonathan Ilias-Pillet et Cédrick Lelionnais (ESEO)
Les plates-formes d'exécution dans l'IDM : quelles modélisations pour quelles utilisations ? 269

Défis pour le Génie de la Programmation et du Logiciel **273**

Nicolas Anquetil, Simon Denier, Stéphane Ducasse, Jannik Laval, Damien Pollet (INRIA), Roland Ducournau, Rodolphe Giroudeau, Marianne Huchard, Jean-Claude König et Abdelhak-Djamel Seriai (LIRMM / CNRS / Univ. Montpellier 2)
Software (re)modularization : Fight against the structure erosion and migration preparation 275

Gabriela Arévalo (LIFIA / UNLP – Argentine), Zeina Azmeh, Marianne Huchard, Chouki Tibermacine (LIRMM / CNRS / Univ. Montpellier 2), Christelle Urtado et Sylvain Vauttier (LGI2P / Ecole des Mines d'Alès)
Component and Service Farms 281

Patrick Albert (IBM), Mireille Blay-Fornarino (I3S), Philippe Collet (I3S), Benoit Combemale (IRISA), Sophie Dupuy-Chessa (LIG), Agnès Front (LIG), Anthony Grost (ATOS), Philippe Lahire (I3S), Xavier Le Pallec (LIFL), Lionel Ledrich (ALTEN Nord), Thierry Nodenot (LIUPPA), Anne-Marie Pinna-Dery (I3S) et Stéphane Rusinek (Psitec)
End-User Modelling 285

Charles Consel (INRIA / Univ. Bordeaux)
Towards Disappearing Languages 289

Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé, Pierre Cointe, Rémi Douence et Mario Südholt (Ecole des Mines de Nantes / INRIA / LINA)
Vers une réification de l'énergie dans le domaine du logiciel – L'énergie comme ressource de première classe 291

Posters et démonstrations **295**

Alexandre Cortier
Projet ANR SPaCIFY : Ingénierie Dirigée par les Modèles et Méthodes Formelles pour les Systèmes Embarqués Critiques 297

François Fages et Martin Julien
Modelling Search Strategies in Rules2CP 298

Arnaud Gotlieb
EUclide is a Constraint Language based on Imperative DEfinitions 300

Nassima Izerrouken, Marc Pantel, Xavier Thirioux et Olivier Ssi Yan Kai
Integrated Formal Approach for a Qualified Critical Code Generator 301

Nour Alhouda Aboud, Philippe Aniorté, Eric Cariou et Eric Gouardères
Integration of Agent and Component approaches by Service Oriented vision using Model Driven Engineering 302

Sabina Akhtar, Stephan Merz et Martin Quinson
Extending PlusCal : A Language for Describing Concurrent and Distributed Algorithms . 303

Vincent Aranega
Traceability Mechanism in Transformations Chains Dedicated to Model and Transformation Debugging 306

Iyad Alshabani, Joyce El Haddad, Nikolaos Georgantas, Tarek Melliti, Lynda Mokdad et Pascal Poizat
Pervasive Service Composition (Projet ANR PERSO) 307

Thanh Thanh Le Thi
L'Activité de Génération de Codes Dirigée par les Modèles 308

Fatiha Zaidi, Richard Castanet, Ana Cavalli, Edgardo Montes de Oca et Andrey Sadovykh
WebMov : Modélisation, Test et Validation de Services Web 309

Youssef Ridene, Franck Barbier, Nicolas Belloir et Nadine Couture
Définition d'un langage de modélisation spécifique (Domain Specific Modeling Language) au test d'applications embarquées sur téléphones mobiles 310

Sebastien Mosser et Mireille Blay-Fornarino
Taming Orchestration Design Complexity through the ADORE Framework 311

Régine Laleau et Jérémy Milhau
Projet ANR SELKIS : une méthode de développement de systèmes d'information médicaux sécurisés 313

Akram Idani, Mohamed-Amine Labiadh et Yves Ledru
Approche orientée modèles pour une intégration efficace de B et UML 314

Peggy Cellier, Mireille Ducassé, Sébastien Ferré et Olivier Ridoux
Fouille de données pour la localisation de fautes dans les programmes 315

Romain Adeline, Janette Cardoso, Christel Seguin, Sophie Humbert et Pierre Darfeuil
Vers une méthode de validation des modèles formels AltaRica 316

Préface

C'est avec plaisir que je vous accueille aux deuxièmes Journées Nationales du GDR GPL. Ces journées marquent la mi-parcours de l'actuel quadriennal du GDR Génie de la Programmation et du Logiciel (GPL), créé en 2008 pour une durée de 4 ans par le CNRS (GDR 3168). 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.

L'année qui vient de se clôturer fut une année de mutation pour la communauté scientifique des sciences de l'information. Le GDR GPL a eu l'occasion de s'exprimer avec les autres GDR de la section 7 à ce sujet. La réforme a abouti à la création, au sein du CNRS, de l'Institut des Sciences Informatiques et de leurs Interactions auquel le GDR GPL a demandé à être rattaché. Florence Sèdes viendra nous parler de ce nouvel institut au cours de ces journées nationales. Le Génie Logiciel et la Programmation sont au cœur de l'activité informatique et, comme elle, sont en constante évolution. Ces journées nationales seront l'occasion de mener une réflexion sur cette évolution. C'est l'objectif de « l'appel à défis » qui a été lancé à notre communauté. Plusieurs groupes viendront ainsi nous présenter leur vision des principaux défis que le Génie Logiciel et les Langages de Programmation devront relever d'ici l'horizon 2020. Cette réflexion est animée par Laurence Duchien et fera l'objet d'une table ronde où Bertrand Braunschweig nous présentera comment notre domaine est perçu par l'ANR.

La principale force d'animation du GDR GPL est constituée par ses groupes de travail qui organisent des journées de rencontre, ou des événements scientifiques (conférences, ateliers, écoles, ...). Ces journées nationales donnent l'occasion à la communauté du Génie Logiciel et de la Programmation de se retrouver au delà des frontières des groupes de travail. Comme l'an dernier, nous avons demandé à chaque groupe de travail ou action d'organiser une des sessions des journées nationales. Ces sessions sont complétées par trois conférences invitées, la présentation des défis, des posters et des démonstrations. L'objectif que nous nous sommes fixé est de proposer un programme scientifique intéressant, de qualité, et accessible à tous les participants de la communauté GPL. Pour y parvenir, 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. Les sessions posters et démos sont destinées à permettre au plus grand nombre de participer activement à ces journées.

Cette année, les journées du GDR GPL clôturent une semaine qui a également vu l'organisation des conférences CAL, LMO et IDM, ainsi qu'une réunion du groupe de travail COSMAL. Après AFADL l'an dernier, cette co-localisation des journées nationales avec des conférences nationales est l'occasion de favoriser les rencontres avec une communauté scientifique active et dynamique.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit d'Ivica Crnkovic (Université de Mälardalen, Suède) dont la présentation sera commune aux quatre conférences CAL/LMO/IDM/GPL, de Pierre-Etienne Moreau (Ecole des Mines de Nancy, membre du LORIA /INRIA) et de Gérard Berry (INRIA, titulaire de la chaire « Informatique et sciences numériques » au Collège de France et membre de l'Académie des Sciences). Comme l'an dernier, le choix de ces invités illustre les principales thématiques de notre GDR, et notamment les relations entre

Génie Logiciel et Langages, ainsi que la dimension européenne dans laquelle notre communauté doit s'ancrer.

Un autre objectif essentiel du GDR GPL est de préparer l'avenir de la communauté en favorisant le développement des jeunes chercheurs et leur future mobilité. Cette année encore, le nombre de demandes de qualification examinées par le CNU dans les thématiques du Génie Logiciel et de la Programmation est resté très bas, alors que les besoins de jeunes enseignants-chercheurs restent importants dans cette discipline qui est au cœur de l'informatique. Notre communauté se doit de porter l'effort tant sur leur quantité que sur leur qualité. L'Ecole des Jeunes Chercheurs en Programmation, qui est soutenue par le GDR GPL, contribue significativement à cet objectif en participant à la formation des jeunes chercheurs, mais également en leur permettant de se créer un réseau de relations au niveau national. Les journées nationales poursuivent un objectif semblable en mettant en relation ces jeunes chercheurs avec des responsables d'équipes d'autres laboratoires, favorisant ainsi leur mobilité dans les recrutements. Dans cet esprit, une formule d'inscription gratuite aux journées nationales a été proposée, cette année, aux jeunes chercheurs et nous espérons ainsi qu'ils participeront en nombre à cet événement.

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. Nos collègues palois ont relevé le défi d'organiser quatre événements scientifiques dans la même semaine. Parmi eux, je remercie chaleureusement Eric Cariou et Franck Barbier qui n'ont pas ménagé leurs efforts pour cette organisation depuis plusieurs mois.

Yves Ledru
Directeur du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2010 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail, ainsi que des membres du comité scientifique qui ont participé à l'évaluation de l'appel à défis.

Yves Ledru (président), LIG, Université de Grenoble-1
Yamine Ait Ameur, LISI / ENSMA
Jean-Pierre Banâtre, INRIA, Rennes
Franck Barbier, LIUPPA, Université de Pau et des Pays de l'Adour
Mireille Blay-Fornarino, I3S, Polytech'Nice
Eric Cariou, LIUPPA, Université de Pau et des Pays de l'Adour
Pierre Castéran, LABRI, Université de Bordeaux I
Pierre Cointe, LINA, Nantes
Charles Consel, INRIA/LABRI, Bordeaux
Jean-Michel Couvreur, LIFO, Université d'Orléans
Christophe Dony, LIRMM, Montpellier
Catherine Dubois, CEDRIC, ENSIIE
Hubert Dubois, CEA-LIST
Laurence Duchien (présidente de l'appel à défis), LIFL, Univ. des Sciences et Technologies de Lille
Jacky Estublier, LIG, Grenoble
Jean-Marie Favre, LIG, Université de Grenoble-1
Sébastien Gérard, CEA-LIST
Jean-Louis Giavitto (président du jury des posters), IBISC, CNRS
Arnaud Gotlieb, IRISA, INRIA
Gaétan Hains, LACL, Créteil
Valérie Issarny, INRIA, Paris-Rocquencourt
Claude Jard, IRISA, ENS-Cachan en Bretagne
Thomas Jensen, IRISA, CNRS
Olga Kouchnarenko, LIFC, Université de Franche-Comté
Philippe Lahire, I3S, Université de Nice
Frédéric Loulergue, LIFO, Université d'Orléans
Dominique Méry, LORIA, Nancy
Pierre-Etienne Moreau, LORIA, INRIA, Nancy
Mourad Oussalah, LINA, Université de Nantes
Marie-Laure Potet, Vérimag, INP Grenoble
Olivier H. Roux, IRCCyN, Université de Nantes
Salah Sadou, VALORIA, Université Bretagne-Sud
Christel Seguin, ONERA Centre de Toulouse

Fatiha Zaïdi, LRI, Université Paris-Sud XI
Mikal Ziane, LIP6, Université Paris V

Comité scientifique du GDR GPL

Jean-Pierre Banâtre (IRISA, Rennes)
Pierre Cointe (LINA, Nantes)
Charles Consel (LABRI, Bordeaux)
Christophe Dony (LIRMM, Montpellier)
Jacky Estublier (LIG, Grenoble)
Paul Feautrier (LIP, Lyon)
Marie-Claude Gaudel (LRI, Orsay)
Gaétan Hains (LACL, Créteil)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christine Paulin (LRI, Orsay)

Comité d'organisation

Eric Cariou (président), LIUPPA, Université de Pau et des Pays de l'Adour
Franck Barbier, LIUPPA, Université de Pau et des Pays de l'Adour
Nicolas Belloir, LIUPPA, Université de Pau et des Pays de l'Adour
Jean-Michel Bruel, IRIT, IUT de Blagnac

Conférenciers invités

A Classification Framework for Component Models

Auteur : Ivica Crnkovic, université de Mälardalen, Suède

Résumé :

The essence of component-based software engineering is embodied in component models. Component models specify the properties of components and the mechanisms of component interactions. In the last decade a large number of different component models have been developed, with different aims and using different principles and technologies. This has resulted in a number of models which have many similarities, but also principal differences, and in many cases unclear concepts. Component-based development has not succeeded in providing standard principles, as has, for example, object-oriented development. In order to increase the understanding of the concepts, and to differentiate component models more easily, this presentation discusses fundamental principles of component models, and defines a Component Model Classification Framework which includes these principles. Further, the principles of several component models are presented using this framework.

Combiner Java et réécriture, c'est possible et utile

Auteur : Pierre-Etienne Moreau, Ecole des Mines de Nancy, LORIA

Résumé :

Le système Tom est une extension de Java permettant de programmer par filtrage, en utilisant des règles et des stratégies. La notion de signature et de terme algébrique permet de définir de manière abstraite les structures de données. Dans ce cadre, l'utilisation de règles de réécriture et de motifs permet de définir, de manière élégante et sûre, des transformations à effectuer. Tom permet donc de programmer aussi bien en Java que par réécriture. Ce qui se prête particulièrement bien à l'enseignement et à la transformation de structures arborescentes telles que les termes ou les documents XML par exemple. Une des particularités du système est de fournir du filtrage modulo l'associativité avec élément neutre. Cette théorie permet de manipuler aisément des structures de liste par exemple. Une autre originalité de Tom est de ne pas imposer de structure de donnée particulière pour représenter les informations à transformer. Le lien entre la structure concrète et le signature algébrique devient un paramètre du programme. Cette souplesse permet d'utiliser la notion de filtrage algébrique pour effectuer des transformations de graphes par exemple. Enfin, la notion de stratégie est un moyen efficace et élégant pour contrôler la façon dont les transformations sont appliquées. L'exposé présentera le système Tom, ses particularités et ses applications.

Qualité du logiciel : certification vs. vérification

Auteur : Gérard Berry, INRIA, Collège de France, Académie des Sciences

Résumé :

La certification est un processus classique pour certains systèmes embarqués critiques (avionique, carte à puce, etc.) ; elle est certainement amenée à se généraliser. Elle se fonde encore essentiellement sur une analyse et une revue du flot de développement et de test. Cette approche reste indispensable, car tout programme mal développé et testé est évidemment dangereux. Mais, avec l'évolution vers un outillage de haut niveau avec synthèse de code embarqué et vérification formelle, d'autres approches plus scientifiques deviennent indispensables. Elles doivent être fondées sur des techniques réellement formelles et réellement implémentées. Nous étudierons l'état de ces techniques, leur évolution potentielle, leurs limites, et leur impact indirect sur les méthodes de conception.

Session de l'action AFSEC

Approches Formelles des Systèmes Embarqués Communicants

Modular Static Scheduling of Synchronous Data-flow Networks

An efficient symbolic representation ⋆

Marc Pouzet[1] and Pascal Raymond[2]

[1] LRI, Université Paris-Sud 11, 91405 Orsay, cedex, France.
Marc.Pouzet@lri.fr.
[2] VERIMAG, 2 avenue de Vignate, 38610 Gières, France.
Pascal.Raymond@imag.fr.

1 Overview

This work addresses the question of producing modular sequential code from synchronous data-flow networks. Precisely, given a system with several input and output flows, how to decompose it into a minimal number of classes executed atomically and statically scheduled without restricting possible feedback loops between input and output?

Though this question has been identified by Raymond in the early years of Lustre, it has almost been left aside until the recent work of Lublinerman, Szegedy and Tripakis. The problem is proven to be intractable, in the sense that it belongs to the family of optimization problems where the corresponding decision problem — there exists a solution with size c — is NP-complete. Then, the authors derive an iterative algorithm looking for solutions for $c = 1, 2, ...$ where each step is encoded as a satisfiability (SAT) problem.

Despite the apparent intractability of the problem, our experience is that real programs do not exhibit such a complexity. Based on earlier work by Raymond, this paper presents a new symbolic encoding of the problem in terms of input/output relations. This encoding *simplifies* the problem, in the sense that it rejects solutions, while keeping all the optimal ones. It allows, in polynomial time, (1) to identify nodes for which several schedules are feasible and thus are possible sources of combinatorial explosion; (2) to obtain solutions which in some cases are already optimal; (3) otherwise, to get a non trivial lower bound for c to start an iterative combinatorial search. The method has been experimented on concrete industrial examples.

The solution applies to a large class of block-diagram formalisms based on atomic computations and a *delay* operator, ranging from synchronous languages such as Lustre or Scade to modeling tools such as Simulink.

2 Static Scheduling

The synchronous block-diagram or *data-flow* formalism is now preeminent in a variety of design tools for embedded systems. Sequential code generation of synchronous block-diagrams have been considered in the early years of Lustre [3] and Signal [1] and is provided by industrial tools such as Scade [3] and RtBuilder [4] for almost fifteen years.

We focus here on the problem of generating imperative, sequential code, implementing the functional behavior of a parallel data-flow network.

Consider the data-flow network given on Figure 1, left. This network is made of *instantaneous nodes* (f,j,h) that require all their inputs before producing their outputs. It also contains a *delay* node (D) which is able to produce its output before reading its input. For sequential code generation, these nodes are important because they reverse the data-dependencies. They typically can be

⋆ This document is an extended abstract of [6].

[3] http://www.esterel-technologies.com/scade/

[4] http://www.tni-software.com

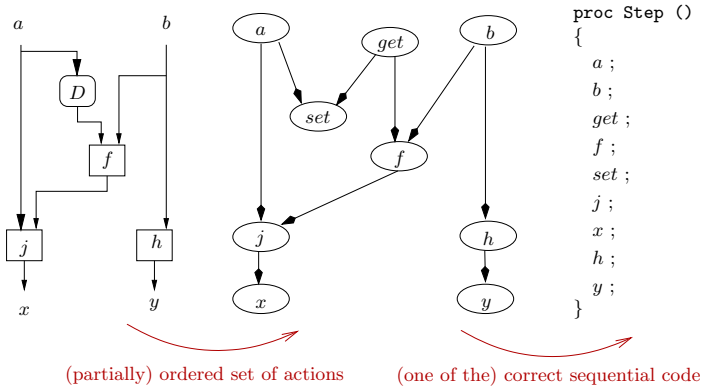


Fig. 1. A data-flow network and the corresponding ordered set of actions

implemented by two imperative actions: *get* produces the output, and must be performed before *set* which memorizes a new value for the sequel (Figure 1, center).

For static scheduling, the network can then be seen as a set of imperative actions with a dependency relation (Figure 1, center). This relation must be a partial order, otherwise the network contains a combinatorial loop and cannot be scheduled. A sequential code can then be produced by selecting a particular total order compatible with the dependencies (Figure 1, right).

Producing such a monolithic imperative procedure for a network is called the *black-boxing* approach: this code can be modularly used when compiling a network that uses this block, but the whole block would be considered as instantaneous, whatever are the actual dependencies between its inputs and its outputs.

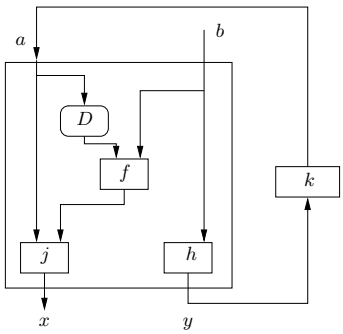


Fig. 2. A correct feedback use of a parallel component

2.1 Feedback Loops and Grey-boxing

In a data-flow network, it makes sense to feedback an output to an input as soon as the output does not depend combinatorially on this input. We say that such a feedback is causality correct. This is illustrated by the example of Figure 2. If we allow such an external loop, it is not possible to use a single, monolithic code, such as the one given on the right of Figure 1.

In order to accept all correct feedback, a straightforward solution consists in recursively inline all the blocks in a network, in order to schedule only atomic nodes (either combinatorial or delays). This approach is called the *white-boxing*, but, indeed, it forbids modular compilation.

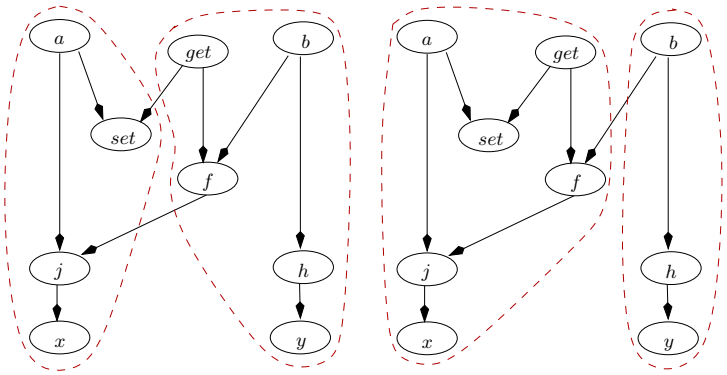


Fig. 3. Two possible "grey-boxing" and optimal scheduling

However, it is not always necessary to fully inline the blocks. Whatever be the actual use of the block, some internal nodes can be gathered without preventing causally correct feedback loops. For instance, in our example, whatever be the calling context, f can be computed:

- together with h, as soon as b is provided;
- together with (and before) j in order to provide the output x.

The same reasoning holds for the delay D:

- $D.get$ is clearly associated to the computation of f;
- $D.set$ requires the input a, and for this reason, is related to the action j.

Finally, this intuitive reasoning shows that the node can be abstracted into two blocks of code:

- the "class" of h with input b and output y;
- the "class" of j with input a, output x, and which must always be computed after the class for `h`.

The frontier between these two classes is not strict: some internal computations can be performed on one side or the other, like g and f. Figure 3 shows two particular solutions, each of them with two classes. Note that this number of class is *optimal*, since it is impossible to produce only one class without forbidding the correct feed-back from y to a.

The problem of partitioning a data-flow program into a (minimal) number of sequential blocks is called the *(Optimal) Static Scheduling Problem*. Figure 4 summarizes this principle on our example: an optimal partition of the actions is chosen (left), a simplified ordered set of (macro) actions is derived (center), and a sequential procedure is produced for each macro-action (right).

3 Theoretical Complexity

The problem of finding an optimal static scheduling has been proven to be NP-complete in [5], through a reduction of the *partition in k cliques* problem, also known as *clique cover* [2]. They then propose an iterative algorithm which tries to solve (with the help of a Sat-solver) the corresponding decision problem (does it exist a solution with n classes? for $n = 1, 2, \cdots$).

However, our experience is that such a complexity is only achieved for unrealistic programs with a huge number of inputs/outputs.

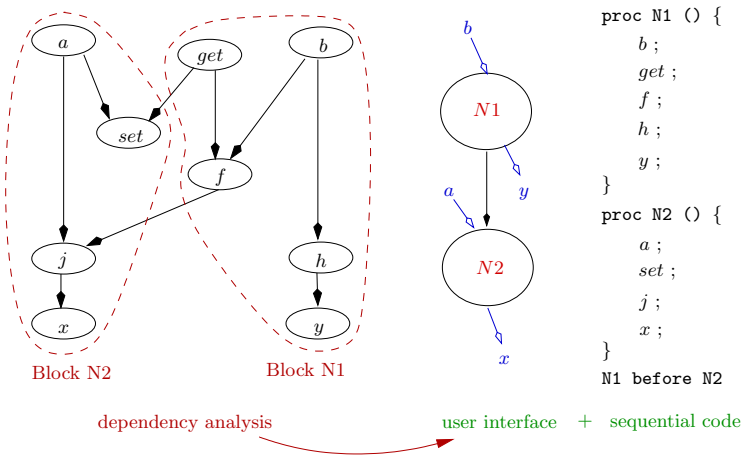


Fig. 4. Grey-boxing: orderer blocks plus sequential code

4 Input/Output Analysis

This section summarizes the results presented in [6], see the paper for details and proofs. We note $(A, I, O, \preceq)$ the data of the problem: A is the set of nodes, partially ordered by $\preceq$, $I \subseteq A$ is the subset of inputs, $O \subseteq A$ is the subset of outputs. We define:

- $\mathcal{I}(x) = \{i \in I, i \preceq x\}$ the inputs of x,
- $\mathcal{O}(x) = \{o \in O, x \preceq o\}$ the outputs of x,

Note that the inclusion of inputs (resp. reverse inclusion of outputs) are relations that give more precise information on possible scheduling than the "raw" dependency relation $\preceq$:

- if $\mathcal{I}(x) \subseteq \mathcal{I}(y)$ then x can always be computed before y,
- if $\mathcal{O}(x) \supseteq \mathcal{O}(y)$ then x can always be computed before y.

An even more precise information on scheduling is obtained by combining both reasoning, for instance inputs according outputs:

- $\mathcal{I}_{\mathcal{O}}(x) = \{i \in I \; O(x) \subseteq O(i)\}$

This function is interesting because it captures the notion of optimal scheduling when considering inputs and outputs only: two input/output can be gathered together if and only if they have the same $\mathcal{I}_{\mathcal{O}}$ value. This result gives a non-trivial lower bound for the number of classes in any optimal solution: the number of values of $\mathcal{I}_{\mathcal{O}}(x)$ for $x \in I \cup O$. We say that the input/output "keys" (the set of $\mathcal{I}_{\mathcal{O}}(i)$ and $\mathcal{I}_{\mathcal{O}}(o)$) are *mandatory*.

5 Input-key encoding (KI-enc)

We propose an alternative definition of the static scheduling problem based on the analysis of input/output relations. The idea is to associate to each node in the network a *key* which is the subset of inputs that will actually be computed (i.e. read) before the computation of the node. We search for key-encoding $\mathcal{K} : A \mapsto 2^I$ satisfying:

- (KI-1) $\forall x \in I \cup O, \;\; \mathcal{K}(x) = \mathcal{I}_{\mathcal{O}}(x)$
- (KI-2) $\forall x, y, \;\; x \preceq y \;\Rightarrow\; \mathcal{K}(x) \subseteq \mathcal{K}(y)$

Moreover, $\mathcal{K}$ is said optimal if its image set is minimal (i.e. the mapping uses a minimal number of keys).

Indeed, this problem is still NP-complete, but it is proven to be "simpler" than the original one: it has strictly less (non-optimal) solutions, but it captures all the optimal ones.

However, the problem can be simplified (typically by bounds propagation), in order to check if it admits some "trivial" solution. For instance, the particular mapping $\mathcal{K}^{\top} : x \rightarrow \mathcal{I}_{\mathcal{O}}(x)$ is a solution. Moreover, if it appears that each $\mathcal{I}_{\mathcal{O}}(x)$ are mandatory (that is, they are all the key of some input or output), than $\mathcal{K}^{\top}$ is proven optimal, without the help of any exponential decision procedure.

The simplification of the system gives another candidate solution, called $\mathcal{K}^{\bot}$: intuitively, $\mathcal{K}^{\top}$ corresponds to the heuristic "compute as late as possible", while $\mathcal{K}^{\bot}$ corresponds to the heuristic "compute as soon as possible". We do not detail here the definition of $\mathcal{K}^{\bot}$ (see the full paper for details [6]).

6 Experimentation

We have developed a prototype tool in OCAML [4] that:

- extract dependencies information from SCADE programs,
- build the KI-enc system, simplify it and search for trivial solutions (all in polynomial time):
  - if an optimal solution is found, the system is considered as simple,
  - otherwise, the system is encoded into a satisfiability problem, and a third-party decision tool is iteratively called in search of an optimal solution

We have considered 3 benchmarks: the standard SCADE library, and two real-size industrial applications from Airbus. The whole experiment concerns more than 350 block diagrams, and it takes less than 0.4 seconds to achieve on a laptop (CoreDuo II, 2.8Gh, Mac OS X). The programs form the SCADE library are clearly designed to be reused, and then, are good candidates for modular compilation. Unsurprisingly, these library programs are very small and then are polynomially solved. Benchmark 2 and 3 are not really representative for modular compilation, since these programs are not intended to be reused: they are tasks and sub-tasks of a big control application, and then, they are used exactly once. However, our goal is not to justify modular compilation, but rather to find some example big enough to challenge our method. We finally found one program that were not polynomially solved by the prototype. Note that this counter-example would have been solved by a simple heuristic that was not yet implemented.

	SCADE lib		Bench 2		Bench 3	
	simples	others	simples	others	simples	others
n. of programs	223	0	27	0	124	1
n. of classes	1 to 2	-	1 to 4	-	1 to 4	3 or 4
n. of in/out	2 to to 9	-	2 to 19	-	2 to 26	4+2
n. of nodes	2 to 64	-	2 to 48	-	7 to 600	25
av. size	12	-	18	-	70	-

Results show the ranges of the number of classes (i.e. the size of the optimal solution); the programs in a particular benchmark are of various sizes, we give the ranges of the number of inputs/outputs and of internal nodes, and also the average size (in internal nodes).

Fig. 5. Experimental results: almost 100% of the programs are polynomially solved.

7 Conclusion

This work addresses the static scheduling problem of a synchronous data-flow network. Precisely, how to decompose a network into a minimal number of classes executed atomically without restricting possible feedback loops between input and output? Though this optimization problem is intractable in the general case and can be tackled with general combinatorial methods, our experience is that real programs do not reveal such a complexity. This calls for a specific algorithm able to identify programs which can be solved in polynomial time. Based on the notion of input/output dependencies, we build a symbolic representation (KI-enc) of the problem. This representation *simplifies* the problem in the sense that it has strictly less solutions but it contains *all* optimal ones. This representation gives a non trivial lower bound on the number of classes, and two particular solutions $\mathcal{K}^{\perp}$ and $\mathcal{K}^{\top}$. It can then be checked whether $\mathcal{K}^{\perp}$ and/or $\mathcal{K}^{\top}$ are optimal or not by comparing their number of classes with the lower bound. In most programs we have encountered, at least one of these solution is optimal. Otherwise, the non trivial bound on the number of classes is used to start a combinatorial search with a SAT-solver.

References

1. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
2. M. R. Garey and D. S. Johnson. *Computers and Intractability - A guide to the Theory of NP-completeness*. Freeman, New-York, 1979.
3. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
4. Xavier Leroy. The Objective Caml system release 3.11. Documentation and user's manual. Technical report, INRIA, 2009.
5. R. Lublinerman, C. Szegedy, and S. Tripakis. Modular Code Generation from Synchronous Block Diagrams — Modularity vs. Code Size. In *ACM Principles of Programming Languages (POPL)*, 2009.
6. Marc Pouzet and Pascal Raymond. Modular static scheduling of synchronous data-flow networks: An efficient symbolic representation. In *ACM International Conference on Embedded Software (EMSOFT'09)*, Grenoble, France, October 2009.

Collaboration entre méthode d'ordonnancement et calcul réseau

Marc Boyer, David Doose

ONERA

Résumé Dans cet article, nous faisons collaborer deux méthodes de calcul du monde temps réel : l'ordonnancement de tâches et le calcul réseau. L'ordonnancement de tâches cherche à savoir si, sur un processeur partagé, un ensemble de tâches (temps-réel) respectera ses échéances. Les différentes techniques d'ordonnancement peuvent souvent trouver exactement le pire temps de réponse, mais avec une complexité algorithmique assez importante. Le calcul réseau, lui, fait souvent des sur-approximations, mais avec une complexité algorithmique plus faible (souvent linéaire) qui lui permet de traiter des systèmes de très grande taille. Nous nous proposons ici de faire collaborer ces deux techniques dans un système embarqué communiquant en appliquant l'ordonnancement dans les calculateurs pour évaluer au plus près le flots de données qui en est issu dans un format adapté au calcul réseau.

1 Introduction

Les systèmes embarqués temps réels sont de nos jours constitués de dizaines voire centaines de calculateurs, hébergeant chacun des dizaines ou centaines d'applications temps réels. Pour garantir la correction de ces applications, il faut non seulement borner le temps de réponse des systèmes, mais aussi les délais subis par les données échangées à travers le réseau.

Pour calculer des temps de réponse, les techniques d'ordonnancement sont appliquées sur les calculateurs. Elles permettent souvent de calculer l'exact pire temps de réponse, au prix d'une forte complexité algorithmique. Cela n'est pas trop gênant pour les systèmes embarqués, où le nombre d'applications par calculateur reste raisonnable (un système trop complexe à analyser serait trop complexe à ordonnancer par le système).

En ce qui concerne le réseau, ce sont des dizaines de milliers de flux qu'il faut prendre en compte dans un avion comme l'A380, sur une topologie contenant une dizaine de commutateurs et une centaine de calculateurs. Pour borner le temps de traversée d'un tel réseau, il faut des méthodes avec une complexité algorithmique plus faible. Le calcul réseau [8] est capable d'utiliser des algorithmes linéaires, au prix d'approximation pessimistes. Il a été utilisé pour garantir les délais du réseau de l'A380 [6].

Pour garantir des délais sur un réseau partagé, le calcul réseau a besoin de contrats sur les trafics entrant (dits *courbes d'arrivée*) et aussi sur les capacités des commutateurs (dits *courbes de service*).

Notre idée est d'utiliser les informations d'ordonnancement pour calculer au plus juste le trafic issu de chaque calculateur. Grossièrement, lorsque le calcul réseau estime le trafic émis par une tâche périodique, il considère la somme des tailles maximales de trames émises par période, et en déduit un débit crête (une rafale) égal à la somme des tailles de trames, et un débit moyen qui est la rafale divisée par la période. Pour prendre en compte le trafic émis par plusieurs applications hébergées sur le même calculateur, il fait la somme des débits individuels. Mais ce faisant, il considère que toutes les applications peuvent émettre leur rafale en même temps, ce qui est impossible, une seule application s'exécutant à un moment donné sur un processeur. Notre contribution consiste à considérer uniquement les ordonnancements possibles, d'en déduire les trafics générés, puis d'en déduire un contrat de trafic le plus juste possible.

Nous commencerons par faire une rapide présentation du calcul réseau, au paragraphe 2, à la fin de laquelle nous présenterons l'intuition qui fonde notre approche (paragraphe 2.3). Nous présenterons ensuite une notre méthode d'ordonnancement pour le calcul de courbes de trafic (paragraphe 3). Des exemples permettant d'illustrer le gain de la méthode sont présentés au paragraphe 4.

2 Le calcul réseau

L'objectif du calcul réseau est de permettre de calculer des bornes supérieures garanties pour les délais subis par des flux dans des réseaux, ainsi que pour les quantités de mémoire utilisées par ces flux dans les éléments de réseau. Pour ce faire, il modélise un contrat de trafic par des « courbes d'arrivée », et modélise le serveur par une « courbe de service » [4,5,8,3]. Nous n'entrerons pas dans les détails de cette théorie, mais uniquement les parties nécessaires pour cet article. On pourra se reporter à [2] pour une introduction en français.

2.1 Algèbre (min,plus)

Un des intérêts du calcul réseau est que les notions qu'il manipule s'expriment très bien dans le dioïde $(\min, +)$, dont la première loi est le minimum, et la seconde loi la somme.

Les flux de données sont représentés par leur fonction de cumul. Nous manipulons donc des fonctions croissantes ($\mathcal{G}$), nulles sur les négatifs ($\mathcal{F}$) et nulles en 0 ($\mathcal{F}_0$).

$$\mathcal{G} \stackrel{\text{def}}{=} \{f : \mathbb{R} \to \mathbb{R} \cup \{+\infty\} \mid x < y \implies f(x) \leq f(y)\}$$

$$\mathcal{F} \stackrel{\text{def}}{=} \{f \in \mathcal{G} \mid x < 0 \implies f(x) = 0\} \qquad \mathcal{F}_0 \stackrel{\text{def}}{=} \{f \in \mathcal{F} \mid f(0) = 0\}$$

Trois opérateurs sur les fonctions sont utilisés, la convolution ($*$), la déconvolution ($\oslash$) et la clôture sous-additive ($\cdot^*$).

$$(f * g)(t) \stackrel{\text{def}}{=} \inf_{0 \le s \le t} f(t-s) + g(s) \qquad (f \oslash g)(t) \stackrel{\text{def}}{=} \sup_{s \ge 0} f(t+s) - g(s)$$

$$f^* \stackrel{\text{def}}{=} \delta_0 \wedge f \wedge (f * f) \wedge (f * f * f) \wedge \cdots$$

2.2 Bases de calcul réseau

En calcul réseau, on modélise un *flux* par sa courbe de trafic cumulé : $A \in \mathcal{F}_0$, où $A(t)$ représente le nombre de bits émis par le flux sur l'intervalle $[0, t)$. Par convention, $A(0) = 0$.

Mais A représente le flux "réel", ce qui va transiter sur le réseau, et qui est inconnu au moment de l'analyse. En pratique, on va travailler avec une abstraction du flux, un contrat de trafic, appelé « courbe d'arrivée ». On dit que A admet une courbe α pour courbe de service (noté $A \prec \alpha$) ssi

$$\forall t, s \in \mathbb{R}_{\ge 0} : A(t+s) - A(t) \le \alpha(s) \iff A \le A * \alpha \tag{1}$$

On voit qu'il s'agit d'une sur-approximation : on cherche une enveloppe pour un phénomène réel A, et plusieurs pourront convenir, capturant plus ou moins précisément le réel.

Un *serveur* S va être une application de $\mathcal{F}$ dans $\mathcal{F}$, qui associe à un flux d'entrée un flux de sortie. De même que pour les flux, on ne manipule pas directement les serveurs, mais un contrat de service. On dit qu'un serveur S offre un service de courbe β ssi

$$\forall A, A' : A \xrightarrow{S} A' \implies A' \ge A * \beta \tag{2}$$

On peut borner le délai subi par un flux dans un serveur par la déviation horizontale $h(\alpha, \beta)$ entre la courbe d'arrivée α et la courbe de service β, comme illustré sur la figure 1. On y retrouve quelques intuitions du monde réel si la courbe d'arrivée croît plus vite que la courbe de service, cela signifie que le système est surchargé. Il est important d'arriver à modéliser les phénomènes au plus juste, c'est à dire, pour les courbes d'arrivée, d'avoir la courbe la plus petite possible.

Une propriété particulièrement intéressante pour cet article est le résultat suivant : soit A un flux, alors $A \oslash A$ est la meilleure courbe d'arrivée possible pour le flux A[1]. Par monotonie de la déconvolution, si on a deux bornes A^M et A^m telles que $A^m \le A \le A^M$, alors, $A^M \oslash A^m \ge A \oslash A$ est une courbe d'arrivée de A.

$$A^m \le A \le A^M \implies A \prec A^M \oslash A^m \tag{3}$$

Le principe de notre démarche est d'arriver, par des méthodes d'ordonnancement, à construire un encadrement (A^m, A^M) afin de calculer une courbe d'arrivée.

[1] Formellement, si α est une courbe d'arrivée pour A, alors $\alpha \ge A \oslash A$.

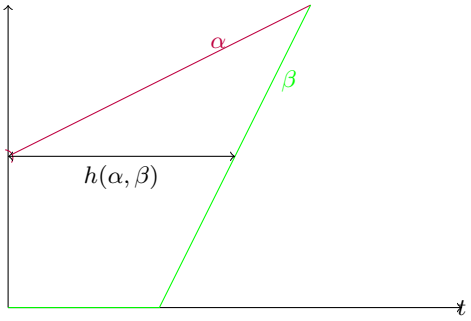


Fig. 1. Exemple de courbe d'arrivée, de service et déviation horizontale

2.3 Gains attendus pour le calcul réseau

Prenons ici un exemple simplifié pour illustrer le gain attendu et le principe de la méthode. Considérons un système périodique de période T. Dans chaque période, b bits sont émis. Si l'on ne connaît rien sur les moments d'émission de ces messages, on doit considérer les pires cas, dont celui où toutes les données sont émises en début de période, ce qui conduit à la courbe d'arrivée A de la figure 2. De plus, certains auteurs préfèrent éviter les courbes en escalier, et manipulent en fait la sur-approximation linéaire γ. Si on sait par contre que les données sont émises de façon régulière, un message de $\frac{b}{5}$ bit étant émis tous les $\frac{T}{5}$ unités de temps par exemple, on obtient la courbe en escalier B, beaucoup plus lisse. Supposons que le service soit celui de la figure β, il apparaît clairement que la borne de délai (la déviation horizontale entre courbe d'arrivée et de service) calculée pour le flux B est trois fois plus petite que celle du flux A.

2.4 Des courbes de trafic aux courbes d'arrivée

Nous ne présenterons pas dans cet article, par manque de place, le détail technique du passage des courbes de trafic (A^m, A^M) calculées au paragraphe 3 aux courbes de service. Deux point sont à traiter. Premièrement, il s'agit de calculer une déconvolution entre courbes en escalier. Nous pourrions reprendre les algorithmes généraux de [1], mais dans ce cas particulier, une résolution directe est aussi simple et plus rapide à implanter. Le second point consiste à étudier le passage d'une étude d'ordonnancement sur un horizon fini (une hyper-période) à une courbe d'arrivée à horizon infini.

3 Méthode d'ordonnancement pour le calcul de courbe d'arrivée

Comme présenté dans l'équation (3), le but des méthodes à base d'ordonnancement est de calculer un encadrement (A^m, A^M) du flux de données produits par un ensemble d'applications.

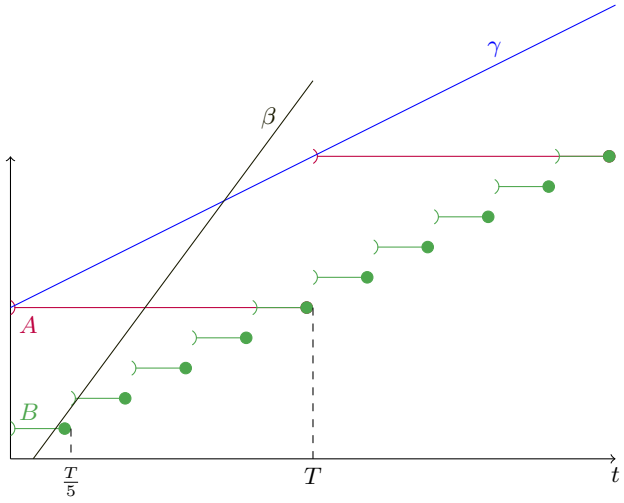


Fig. 2. Trafics plus ou moins lissés

Le modèle de tâches que nous allons considérer est le suivant : des tâches périodiques, avec un ordonnancement à priorités fixes. Chaque tâche τ émet un unique message de taille variable en fin d'exécution[2]. Comme on va devoir étudier les émissions de message au plus tôt et au plus tard, on a besoin d'avoir le meilleur et le pire temps d'exécution.

Si on sait calculer ainsi les dates de fin d'exécution au plus tôt et au plus tard, on peut encadrer le débit réel : en effet, la courbe de trafic cumulé la plus grande A^M correspond au cas où les tâches finissent le plus tôt et émettent les plus grandes trames ; inversement, la courbe de trafic cumulé la plus petite correspond aux exécutions au plus tard et aux données les plus petites.

Ainsi, chaque tache τ_i sera caractérisée par sa date de réveil R_i, sa priorité P_i (en supposant une unique tâche par priorité), son meilleur (resp. pire) temps d'exécution C_i^m (resp. C_i^M), sa période T_i et son échéance D_i. Le message émis est compris entre une taille minimale S_i^m et une maximale S_i^M.

Nous notons respectivement Θ_i^m la date d'émission au plus tôt, et Θ_i^M celle au plus tard. L'ensemble des tâches de plus haute priorité que τ_i est noté $hp(\tau_i)$:

$$hp(\tau_i) = \{\tau_j \mid P_j < P_i\}$$

Pour simplifier l'étude, nous supposons aussi que :

Hyp. 1 les tâches sont indépendantes (par de relation de précédence, pas de ressource partagée) ;

[2] C'est une hypothèse réaliste dans le monde temps réel embarqué, où une tâche périodique commence généralement par lire des données d'état, puis calcule une information, et émet le résultat de ses calculs en fin d'exécution. Si une tâche émet plusieurs message en fin d'exécution, cela revient du point de vue du débit à un seul message dont la taille est la somme des tailles individuelles.

Hyp. 2 des date de réveil nulles[3] ($R_i = 0$) ;

Hyp. 3 des échéances plus petites que les périodes ($D_i \leq T_i$).

Avec les hypothèses 2 et 3, nous pouvons borner l'étude sur l'hyper-période $T = ppcm(T_i)$ [11]. Et l'hypothèse 2 implique que le pire (resp. meilleur) temps d'exécution est égal au pire (resp. meilleur) temps de réponse Θ_i^m (resp. Θ_i^M).

3.1 Évaluation du bag

La méthode classique pour calculer une courbe d'arrivée est de diviser la taille maximale des messages par la durée minimale entre deux émissions (bag). Le délai maximal (resp. minimal) entre deux émissions du même message correspond à l'intervalle de temps entre la fin au plus tôt (resp tard) d'une instance et sa terminaison suivante au plus tard (resp. tôt). La figure 3 illustre ces deux situations.

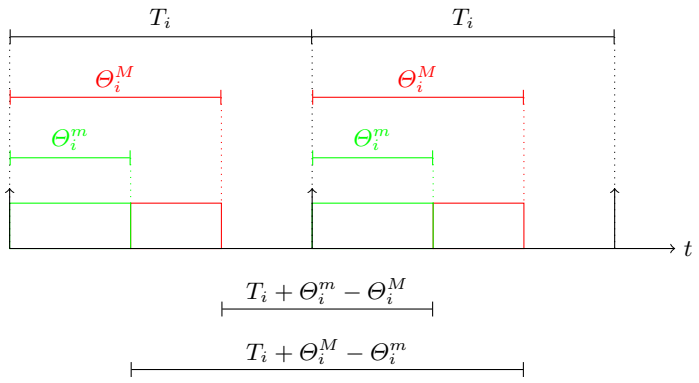


Fig. 3. Déduction du bag à partir des informations issues de l'étude d'ordonnançabilité

Nous pouvons en déduire un valeur optimale (i.e. maximale) pour bag_i :

$$bag_i = T_i + \Theta_i^m - \Theta_i^M \tag{4}$$

3.2 Approximation des traffics cumulés réels utilisant l'ordonnancement des tâches

Le pire temps de réponse [7,9,12] de la tâche τ_i dépend de son pire temps d'exécution C_i^M et des interactions des tâches plus prioritaires ($hp(\tau_i)$). L'ex-

[3] Des travaux non encore publiés montrent que les dates non nulles répartissent la charge dans l'hyper-période, rendent le système encore plus lisse, et améliorent le gain de la méthode.

pression mathématique du pire temps de réponse est la suivante :

$$\Theta_i^M = C_i^M + \sum_{j \in hp(\tau_i)} \left\lceil \frac{\Theta_i^M}{T_j} \right\rceil . C_j^M \tag{5}$$

Le meilleur temps de réponse est la notion duale au pire temps de réponse. Il est calculé en considérant que la tâche τ_i a le temps d'exécution le plus court et qu'elle ne subit aucune interaction de la part des instances plus prioritaires.

$$\Theta_i^m = C_i^m \tag{6}$$

À l'aide du calcul du pire et meilleur temps de réponse d'une tâche nous pouvons déterminer une approximation du traffic cumulé minimal et maximal (A^m et A^M). La courbe A^m (resp. A^M) est définie par une fonction en escalier correspondant à l'émission du message de taille minimale S_i^m (resp. maximale S_i^M) au plus tard Θ_i^M (resp. tôt Θ_i^m) ; ce motif se répétant à chaque période de la tâche. La figure 4 illustre la construction des fonctions A^m et A^M.

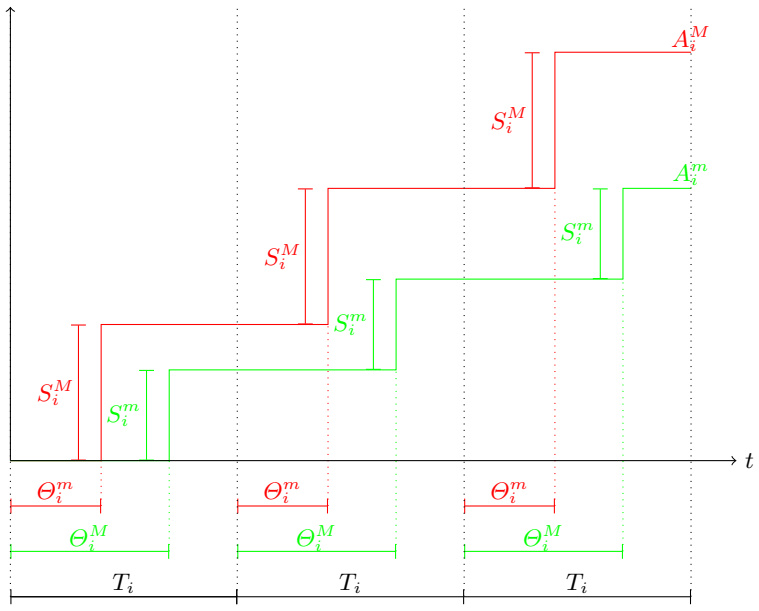


Fig. 4. Représentation graphique des fonctions A^m et A^M.

L'expression mathématique des fonctions A_i^m et A_i^M pour une tâche τ_i est donnée par les deux équations suivantes :

$$A_i^m(t) = \left\lfloor \frac{t}{T_i} \right\rfloor .S_i^m + 1_{t-\left\lfloor \frac{t}{T_i} \right\rfloor .T_i > \Theta_i^M} .S_i^m \tag{7}$$

$$A_i^M(t) = \left\lfloor \frac{t}{T_i} \right\rfloor .S_i^M + 1_{t-\left\lfloor \frac{t}{T_i} \right\rfloor .T_i > \Theta_i^m} .S_i^M \tag{8}$$

avec 1_x la fonction de test qui vaut 1 si son argument est vrai et 0 sinon.

L'approximation des trafics cumulés minimaux et maximaux est défini de la manière suivante :

$$A^m(t) = \sum_{\tau_i} A_i^m(t) \tag{9}$$

$$A^M(t) = \sum_{\tau_i} A_i^M(t) \tag{10}$$

3.3 Meilleurs et pires temps de réponse des instances

L'expérimentation de la méthode « par tâche » présentée au paragraphe précédent sur de petits exemples nous apparue un peu décevante[4]. Après réflexion, il apparut qu'en ne considérant que les tâches on génère un trafic qui fait comme si toutes les instances d'une même tâche pouvaient émettre en même temps dans une hyper-période. Nous avons donc développé une méthode qui s'intéressent aux instances de tâches ($\tau_{i,j}$ est la $j^{\text{ième}}$ instance de la tâche τ_i).

Notations Introduisons quelques notations nécessaires utiles à la compréhension de cette partie.

$[a, b]$ désigne une partie fermée de $\mathbb{R}$, $\cup$ l'union, $\setminus$ la différence, $\sup(\mathcal{X})$ la limite supérieure de $\mathcal{X}$ et $\overline{\mathcal{X}}$ la fermeture de $\mathcal{X}$. Nous définissions aussi trois opérateurs :

1. l'opérateur d'élargissement d'une partie fermée de f $\mathbb{R}$: $\triangleright$ défini par

$$\mathcal{X} \triangleright \delta \equiv \mathcal{X} \cup [sup(\mathcal{X}), sup(\mathcal{X}) + \delta] \tag{11}$$

2. la longueur d'un sous-ensemble de $\mathbb{R}$ définie comme l'intégrale de la fonction de test :

$$l(\mathcal{X}) \equiv \int_{\mathcal{X}} 1_{x \in \mathcal{X}} \tag{12}$$

   Exemples : $l([2, 3]) = 1$, $l([2, 3] \cup [10, 12]) = l([2, 3]) + l([10, 12]) = 3$

3. la différence non vide de deux parties fermées de $\mathbb{R}$

$$\mathcal{X} \ominus \mathcal{Y} \equiv \overline{(\mathcal{X} \setminus \mathcal{Y})} \cup [sup(\mathcal{X}), sup(\mathcal{X})] \tag{13}$$

[4] On trouvera au paragraphe 4.1 un petit cas d'étude illustratif.

La période d'activité [10] d'une instance $\tau_{i,j}$ est la durée entre sa date de réveil $R_{i,j}$ et la date de la fin de son exécution. Selon les notations précédentes la pire (resp. la meilleure) période d'activité de l'instance $\tau_{i,j}$ notée $\mathcal{I}_{i,j}^{M}$ (resp. $\mathcal{I}_{i,j}^{m}$) est définie par

$$\mathcal{H}_{i,j}^{M} = \bigcup_{\tau_{k,l} \in hp(\tau_{i,j})} \mathcal{I}_{k,l}^{M} \tag{14}$$

Périodes d'activité des instances Dans ce paragraphe, nous introduisons une nouvelle méthode pour calculer les pires et les meilleures dates de fin d'exécution d'instance. Cette technique consiste à calculer le pire (resp. le meilleur) période d'activité de chaque instance, puis à déterminer son pire (resp. le meilleur) temps d'exécution $\Theta_{i,j}^{M} = sup(\mathcal{I}_{i,j}^{M})$ (resp. $\Theta_{i,j}^{m} = sup(\mathcal{I}_{i,j}^{m})$).

L'algorithme suivant calcule la pire période d'activité de l'instance $\tau_{i,j}$:

Etape 1 Calculer la remière approximation de période d'activité en prenant en compte uniquement l'exécution de l'instance, sans aucune interaction avec d'autres.

$$\mathcal{I}_{i,j}^{M,0} = [R_{i,j}, R_{i,j} + C_{i,j}^{M}] \tag{15}$$

Etape 2 Calculer l'approximation $\mathcal{I}_{i,j}^{M,n+1}$ en utilisant $\mathcal{I}_{i,j}^{M,n}$. Le principe consiste à appliquer l'opérator d'élargissement ($\triangleright$) afin d'ajouter du temps de calcul utile pour que l'instance puisse finir son exécution sans interaction ($C_{i,j}^{M} - l(\mathcal{I}_{i,j}^{M,n})$). Finalement les interactions des instances plus prioritaires sont ajoutées à l'aide de l'opérateur de différence non-vide.

$$\mathcal{I}_{i,j}^{M,n+1} = \left(\mathcal{I}_{i,j}^{M,n} \triangleright \left(C_{i,j}^{M} - l(\mathcal{I}_{i,j}^{M,n})\right)\right) \ominus \mathcal{H}_{i,j} \tag{16}$$

Etape 3 Déterminer si l'approximation de la période d'activité est le résultat.

- Si la date de fin d'exécution de l'instance dépasse son échéance, alors le système est non-ordonnançable.

$$sup(\mathcal{I}_{i,j}^{M,n+1}) > D_{i,j} \tag{17}$$

- L'approximation correspond à la période d'activité effective si et seulement si :

$$\mathcal{I}_{i,j}^{M,n+1} = \mathcal{I}_{i,j}^{M,n} \tag{18}$$

- Sinon retourner à **l'étape 2**.

La figure 5 illustre le calcul de la période d'activité de l'instance $\tau_{3,1}$. La première trace d'exécution représente la première apporximation de la période d'activité (i.e. l'exécution de l'instance sans intéraction). La seconde montre la période d'activité des instances plus prioritaires $\mathcal{H}_{i,j}^{M}$. La trace suivante, met en évidence les effets des instances de plus fortes priorités. La dernière trace d'exécution montre l'utilisation de l'opérateur d'élargissement.

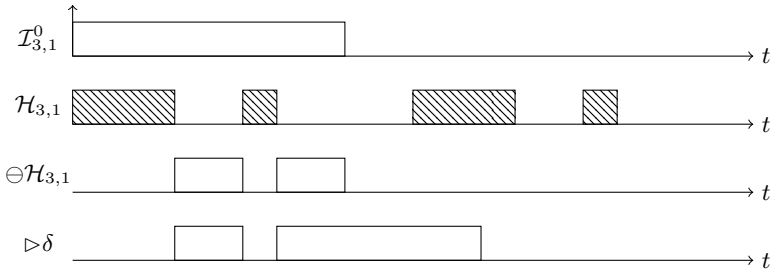


Fig. 5. Calcul de la période d'activité

3.4 Approximation des traffics cumulés réels utilisant l'ordonnancement des instances

A l'aide des pires périodes d'activité des instances nous en déduisons la meilleure approximation du traffic cumulé produit par l'instance $\tau_{i,j}$:

$$A^m_{i,j}(t) = 1_{t>\Theta^M_{i,j}}.S^m_i \tag{19}$$

L'approximation du traffic cumulé de la tâche τ_i correspond à la somme des flux produits par ses instances.

$$A^m_i(t) = \sum_{j=1}^{\infty} A^m_{i,j}(t) \tag{20}$$

Par conséquent les approximations des traffic cumulés minimal et maximal de la tâche τ_i sont définis de la manière suivante :

$$A^m_i(t) = \left\lfloor \frac{t}{T} \right\rfloor . \frac{T}{T_i}.S^m_i + \sum_{j=1}^{\frac{T}{T_i}} 1_{t>\Theta^M_{i,j}}.S^m_i \tag{21}$$

$$A^M_i(t) = \left\lfloor \frac{t}{T} \right\rfloor . \frac{T}{T_i}.S^M_i + \sum_{j=1}^{\frac{T}{T_i}} 1_{tT>\Theta^m_{i,j}}.S^M_i \tag{22}$$

4 Exemples

Pour illustrer cette méthode, trois exemples seront présentés : un premier illustre la méthode sur un petit exemple (section 4.1) ; un second illustre le gain sur la taille de rafale, ainsi que l'effet de la variation du temps d'exécution et du nombre de tâches ; un troisième présente des résultats à partir de configuration réalistes générées aléatoirement.

4.1 Études de cas illustrative

Dans ce paragraphe, nous présentons une étude de cas simples à des fins d'illustration. Prenons un système temps réel composé de quatre tâches.

Afin de faciliter l'étude et l'analyse des résultats, certaines simplifications ont été faites : le meilleur et le pire temps d'exécution sont les mêmes pour de petites tâches ($C_i^m = C_i^M$), la taille des messages est constante ($S_i^m = S_i^M$). Le tableau suivant montre les paramètres des tâches et du flux :

Task	P_i	C_i^m	C_i^M	T_i	D_i	S_i^m	S_i^M
τ_1	1	1	1	10	10	10	10
τ_2	2	1	1	10	10	10	10
τ_3	3	2	3	20	20	60	60
τ_4	4	2	5	20	20	70	70

La figure 6 représente les courbes d'arrivée. La courbe rouge est obtenue avec la technique "classique" du calcul réseau : à partir de tailles de trame et du bag, on déduit la pire rafale et le débit long terme (cf paragraphe 2.3). La courbe bleue utilise une technique (non présentée ici) qui ne prend pas en compte les instances de tâche mais seulement les tâches. La courbe verte est calculée en prenant en compte le comportement des instances de tâches. Les trois courbes noires correspondent à trois charges réseaux différentes envisagées dans cet exemple.

On voit clairement le gain de la nouvelle méthode : le trafic généré est beaucoup plus lisse, et la pire rafale est bien plus petite. En ce qui concerne les délais (déviation horizontale entre la courbe d'arrivée et de service), le gain dépend bien sûr non pas seulement de la courbe d'arrivée mais aussi de celle du service. Mais là aussi, les gains sont conséquents, même s'ils diminuent avec la charge (en effet, sur un système peu chargé, c'est surtout les rafales qui génèrent le délai).

4.2 Efficacité

L'objectif de ce paragraphe est d'évaluer l'avantage des techniques proposées en fonction de plusieurs paramètres : le nombre de tâches et la variation du temps de calcul. Dans cette étude, la technique basée sur les comportements des instances est la seule considérée.

La notion d'*efficacité* introduite ici pour caractériser le gain de notre technique est calculée comme suit :

$$\textit{efficiency} = \frac{\text{rafale classique} - \text{rafale par instance}}{\text{rafale classique}} \tag{23}$$

Pour chaque couple $(n, \Delta) \in [1, 100] \times [0, 1]$, nous générons cent configurations. Dans chaque configuration, les tâches $(\tau_i)_{i\in[1,n]}$ sont générées, avec la même période T, un délai d'exécution maximal C_i^M choisi au hasard entre 1 et $\frac{T}{n}$ et une période minimale $C^m = \Delta C_i^M$.

L'expérimentation met en évidence deux points :

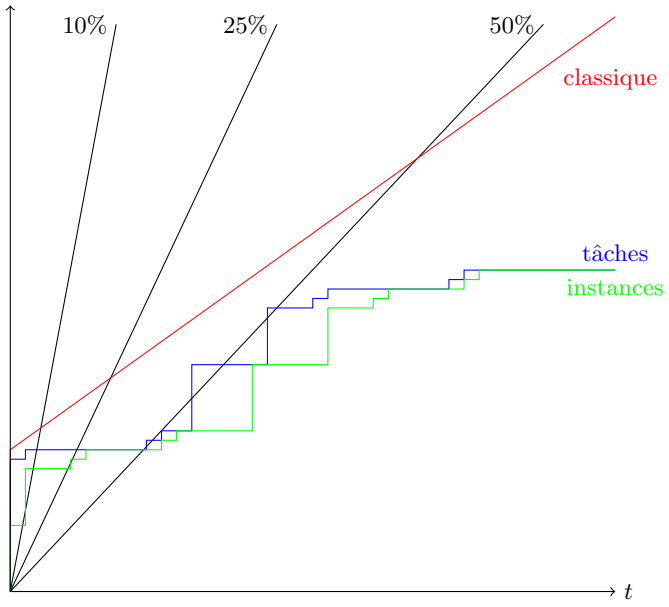


Fig. 6. Exemple

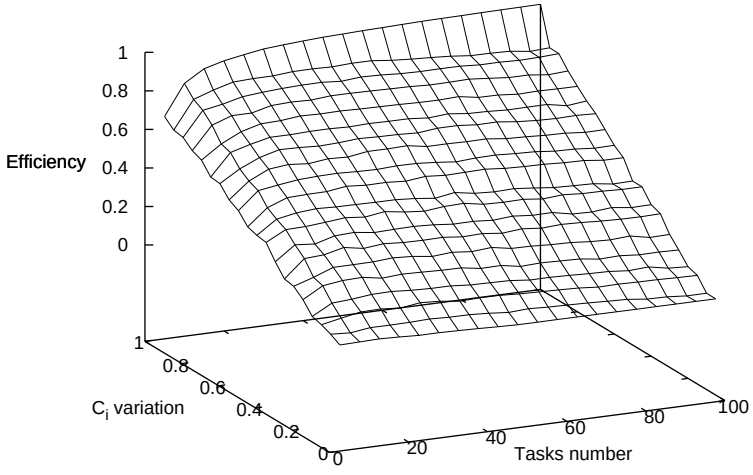


Fig. 7. Efficacité de la méthode

1. L'efficacité augmente rapidement avec le nombre de tâches (presque égale 1 avec 20 tâches). En effet, la technique classique estime que toutes les tâches peuvent produire leurs messages en même temps. Cette approximation est très pessimiste. Avec notre méthode, plus il y a de tâches, moins elles peuvent émettre simultanément.
2. L'efficacité décroît rapidement avec la variation du temps d'exécution des tâches. Ce résultat est dû au fait que, lorsque C^m/C^M tend vers 0, le temps d'exécution C^m tend aussi vers 0, et toutes les instances peuvent effectivement se terminer au même instant 0.

4.3 Exemple réaliste

Nous allons faire le même genre d'analyse que dans les sections 4.1 et 4.2, c'est à dire évaluer le gain de la méthode pour trois charges réseau (10 %, 25 %, 50 %), mais avec des configurations réalistes, en augmentant progressivement la taille de la configuration étudiée.

Dans chaque configuration, n tâches sont générées (avec n un multiple de trois), également reparties en trois classes (chacune de taille $n/3$). Dans chaque classe, le pire temps d'exécution de chaque tâche τ_i est $C_i^M = \max(\lfloor \frac{T_i}{n} \rfloor, 1)$ et la taille du message est constante, égale à la période.

La première classe, "petite et forte priorité", se compose de tâches τ_i avec une période T_i choisie au hasard dans $\{10, 20, 30, 40, 50\}$, et un ratio $\frac{C^m}{C^M} = 0,9$.

Dans la deuxième classe "moyenne", les périodes sont choisies au hasard dans $\{50, 100, 150, 200, 250\}$ et le ratio $\frac{C^m}{C^M} = 0,75$. La troisième classe de priorité "grande et faible priorité" a des périodes dans $\{300, 400, 500\}$ et un ratio $\frac{C^m}{C^M} = 0,5$.

Ces classes font des hypothèse plutôt réalistes pour les systèmes temps-réel, où les tâches à haute fréquence font quelques petits calculs, générant de petits messages urgents avec une priorité élevée, souvent pour les fonctions de contrôle/commande, alors que de tâches de priorité plus faibles effectuent des tâches de fond, comme la supervision, l'enregistrement de l'activité, et génèrent de plus gros messages.

Pour chaque valeur de n dans $[3, 30]$, 100 configurations ont été générées, et les gains moyens sont tracés dans la figure 8. Sont tracés le gain pour la rafale (*burst*) et le gain en délai pour différentes charges. Comme prévu, le gain diminue avec la charge du réseau. Mais les résultats restent proches. Mais le principal résultat est que, avec des configurations réalistes, le gain augmente avec la taille des systèmes, croît très rapidement au dessus de 50 %, et atteint près de 80 % avec 30 tâches.

5 Conclusion

Dans cet article, nous avons fait collaborer deux méthodes du monde temps réel, l'ordonnancement et le calcul réseau, en demandant à l'ordonnancement de prendre en compte des mécanismes que le calcul réseau ne peut pas traiter.

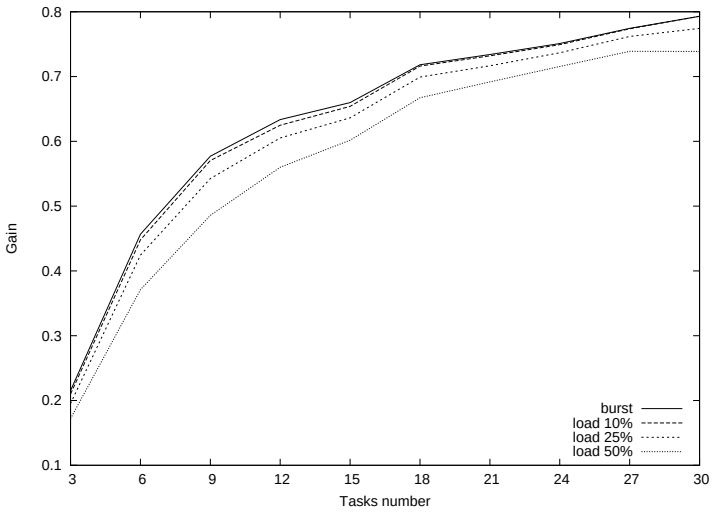


Fig. 8. Gain avec des configurations réalistes

Les premiers résultats sont extrêmement prometteurs, d'autant qu'ils grandissent avec le nombre de tâches du système.

Références

1. Anne Bouillard and Éric Thierry. An algorithmic toolbox for network calculus. *Discrete Event Dynamic Systems*, 17(4), october 2007. http ://www.springerlink.com/content/ 876x51r6647r8g68/.
2. Marc Boyer, Laurent Jouhet, and Anne Bouillard. Notations pour le calcul réseau. *Journal Européen des Systèmes Automatisés*, 43(7–9) :921–935, 2009. Actes du 7ème colloque francophone sur la Modelisation des Systemes Reactifs (MSR'09).
3. Cheng-Shang Chang. *Performance Guarantees in communication networks*. Telecommunication Networks and Computer Systems. Springer, 2000.
4. Rene L. Cruz. A calculus for network delay, part I : Network elements in isolation. *IEEE Transactions on information theory*, 37(1) :114–131, January 1991.
5. Rene L. Cruz. A calculus for network delay, part II : Network analysis. *IEEE Transactions on information theory*, 37(1) :132–141, January 1991.
6. Jérôme Grieu. *Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques*. PhD thesis, Institut National Polytechnique de Toulouse (INPT), Toulouse, Juin 2004.
7. M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5) :390–395, 1986.
8. Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus*, volume 2050 of *LNCS*. Springer Verlag, 2001. http ://lrcwww.epfl.ch/PS_files/NetCal.htm.

9. J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm : exact characterization andaverage case behavior. *Real Time Systems Symposium, 1989., Proceedings.*, pages 166–171, 1989.

10. JP Lehoczky. Fixed priority scheduling of periodic task sets with arbitrarydeadlines. *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 201–209, 1990.

11. Joseph Y. T. Leung and Jennifer Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2(4), 1982.

12. K. Tindell. An extendible approach for analyzing fixed priority hard real-time tasks. Technical Report YCS189, 1992.

Analyse des délais de bout en bout pire cas dans des réseaux avioniques

Jean-Luc Scharbarg[*] — Jérôme Ermont[*] — Henri Bauer[*,] — Christian Fraboul[*]**

[*] *Université de Toulouse - IRIT/ENSEEIHT/INPT - 2, rue Camichel - 31000 Toulouse*
{Jean-Luc.Scharbarg,Jerome.Ermont,Henri.Bauer,Christian.Fraboul}@enseeiht.fr
[**] *Airbus France - 316 Route de Bayonne 31300 Toulouse, France*

RÉSUMÉ. *L'AFDX (standard ARINC 664) est aujourd'hui utilisée pour absorber l'augmentation des échanges de données dans les systèmes avioniques. La certification impose de borner les délais de communication entre ces systèmes. Ce papier présente et compare différentes méthodes pour le calcul d'une borne supérieure garantie du délai de bout en bout sur un réseau AFDX. Le calcul réseau est à la base de l'outil utilisé pour la certification du réseau AFDX de l'A380. Les bornes supérieures obtenues sont le plus souvent pessimistes, entrainant une sous-utilisation du réseau. La méthode des trajectoires, plus récente, peut s'appliquer à l'AFDX. Les premièrs résultats montrent qu'elle permet de s'approcher davantage de la borne supérieure exacte. Une approche utilisant la vérification de modèle permet d'obtenir une borne supérieure exacte sur des configurations réseaux dont la taille est limitée, en raison du problème d'explosion combinatoire. Les méthodes sont comparées sur une configuration industrielle réaliste.*

ABSTRACT. *AFDX (ARINC 664 standard) is currently used to cope with the increasing data exchange needs between avionic systems. The certification imposes to upper bound the end-to-end communication delay between these systems. This paper presents and compares approaches for the computation of a guaranteed upper bound of end-to-end delays on an AFDX network. The network calculus approach has been used for the certification of the AFDX network on the A380. Obtained upper bounds are often pessimistic, leading to an under-utilization of the network. The more recent trajectory approach can be applied in the context of AFDX. First results are less pessimistic than those obtained by the network calculus approach. A model checking approach gives an exact upper bound on network configurations which size is limited by a combinatorial explosion problem. Approaches are compared on a realistic industrial configuration.*

MOTS-CLÉS : *AFDX, délais pire cas, calcul réseau, trajectoires, vérification de modèles*

KEYWORDS: *AFDX, worst end-to-end delay, network calculus, trajectories, model checking*

1. Introduction

L'AFDX (ARI, 2002-2005) est devenue la technologie de communication de référence pour l'avionique civile, permettant le multiplexage de flux de communication sur un réseau Ethernet commuté full duplex. Cette technologie élimine l'indétermisme lié aux collisions dans les réseaux Ethernet de base. Cependant, l'indéterminisme est présent au niveau des commutateurs, où différents flux partagent des ressources (politique de service FIFO au niveau des ports de sortie des commutateurs). Pour maîtriser cet indéterminisme, le standard AFDX impose des contraintes, en particulier au niveau de la définition des flux : chaque flux est défini statiquement et son débit doit être borné (rafale, débit moyen).

La certification du réseau, indispensable dans le domaine de l'avionique, nécessite de prouver que le délai de communication de bout en bout est borné pour tout flux transmis sur le réseau. Pour l'A380, la certification s'est appuyée sur une approche fondée sur le calcul réseau. D'autres approches sont envisageables, utilisant en particulier la méthode des trajectoires et la vérification de modèles. Un autre axe d'analyse concerne le calcul, pour chaque flux d'une borne probabiliste qui peut être dépassée avec une probabilité donnée. Ce type de borne est intéressant, car où les fonctions avioniques sont conçues pour résister à la perte d'une (petite) proportion de trames. Une telle borne peut être obtenue par calcul réseau probabiliste (Scharbarg *et al.*, 2009).

Ce papier s'intéresse au premier axe d'analyse, i.e. le calcul d'une borne déterministe pour chaque flux. L'objectif est de présenter l'application des différentes méthodes envisageables au contexte de l'AFDX, puis de comparer les résultats qu'elles permettent d'obtenir sur une configuration AFDX industrielle (degré de pessimisme des bornes obtenues, taille de configuration analysable par l'approche).

2. L'analyse des délais de bout en bout dans un réseau AFDX

L'AFDX (Avionics Full DupleX Switched Ethernet) (ARI, 2002-2005) est un réseau Ethernet commuté prenant en compte les contraintes du contexte avionique. La figure 1 montre un petit exemple de configuration AFDX avec quatre commutateurs

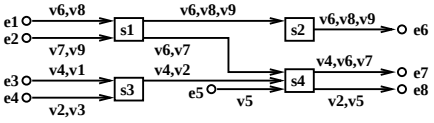


Figure 1. *Un petit réseau AFDX*

$c1 \ldots c4$. Une file d'attente FIFO est associée à chaque port de sortie. Des modules d'interface ($mi1 \ldots mi8$ sur la figure 1) constituent les entrées et les sorties du réseau. Chaque module d'interface est connecté à exactement un port d'un commutateur et

chaque port d'un commutateur est connecté à au plus un module d'interface. Tous les liens sont full duplex.

Le trafic sur le réseau est caractérisé par des *Virtual Links* (VLs) qui permettent la définition statique des flux entrant sur le réseau. Un VL définit une connexion logique unidirectionnelle entre un module d'interface source (flux mono-émetteur) et un ou plusieurs modules d'interface destination. Sur l'exemple de la figure 1, $v4$ est un VL unicast (chemin $mi3 - c3 - c4 - mi7$), tandis que $v6$ est un VL multicast (chemins $mi1 - c1 - c2 - mi6$ et $mi1 - c1 - c4 - mi7$). Le routage de chaque VL est défini statiquement. En outre, un VL v est défini par un *Bandwidth Allocation Gap* ($BAG(v)$) et un longeur minimum ($s_{min}(v)$) et une longueur maximum ($s_{max}(v)$) de trame. Le BAG est le délai minimum entre deux trames consécutives du VL associé. La définition de ces trois paramètres dépend des données applicatives véhiculées par le VL. On considèrera dans la suite le modèle de VL suivant. Soit un VL vi qui transmet un ensemble de données applicatives $data(vi)$. Au début de chacun de ses BAGs, un VL donné transmet une trame de longueur comprise entre $s_{min}(vi)$ (le plus petit sous-ensemble de données prêtes) et $s_{max}(vi)$ (l'ensemble des données prêtes) ou ne transmet rien(aucune donnée prête). La figure 2 montre un exemple de séquence de trames pour un VL vi ayant $s_{min}(vi) = 200\ octets$) et ($s_{max}(vi) = 500\ octets$).

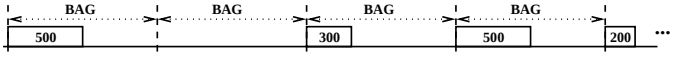


Figure 2. *Exemple de séquence de trames d'un VL*

Soit un chemin p_x d'un VL v et une trame F_v de v transmise sur ce chemin p_x. Le délai de bout en bout F_v sur p_x est appelé $D(F_v, p_x)$. Il est défini par :

$$D(F_v, p_x) = DL(F_v, p_x) + DC(F_v, p_x) + DP(F_v, p_x) \quad [1]$$

$DL(F_v, p_x)$ est le délai de transmission sur les liens full duplex (sans collision). Le délai de transmission sur un lien est donc $t_{octet} \times s_{F_v}$, t_{octet} étant le temps de transmission d'un octet et s_{F_v}, la longueur de la trame F_v. Si tous les liens du réseau ont le même débit, on a donc pour un chemin p_x contenant nbl_{p_x} liens :

$$DL(F_v, p_x) = nbl_{p_x} \times (t_{octet} \times s_{F_v}) \quad [2]$$

$DC(F_v, p_x)$ est le délai de commutation entre les ports d'entrée et de sortie des commutateurs. Dans le contexte de ce papier, ce délai prend la valeur constante dt. On a donc pour un chemin p_x contenant nbs_{p_x} commutateurs :

$$DC(F_v, p_x) = nbs_{p_x} \times dt \quad [3]$$

$DP(F_v, p_x)$ représente le temps passé par la trame dans les files d'attente des commutateurs et des modules d'interface. Ce temps dépend de la charge de chacun des ports de sortie à l'instant où la trame F_v atteint ce port. On a :

$$DP(F_v, p_x) = DP(F_v, p_x, mi_v) + \sum_{c_k \in \Psi_{p_x}} DP(F_v, p_x, c_k) \quad [4]$$

où mi_v est le module d'interface source du VL v, Ψ_{p_x} est l'ensemble des commutateurs contenus dans p_x, $DP(F_v, p_x, xx)$ est le délai dans la file d'attente du port de sortie du composant xx.

$D(F_v, p_x)$ comprend donc une partie fixe $DL(F_v, p_x) + DC(F_v, p_x)$ qui peut être calculée statiquement et une partie variable $DP(F_v, p_x)$. Cette dernière est fonction de données dynamiques, en particulier la séquence des trames émises par chaque VL (longueur de chaque trame) et le phasage entre les différents VLs, i.e. l'instant d'émission de la première trame de chaque VL.

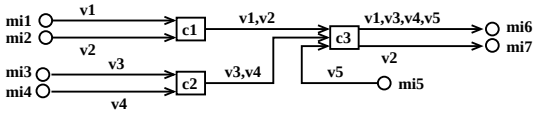


Figure 3. *Exemple de configuration AFDX*

A titre d'exemple, considérons la configuration AFDX de la figure 3. Celle-ci inclut 5 VLs unicast $v1 \ldots v5$, de chemins respectifs $p1 \ldots p5$, avec $BAG(vi) = 4\ ms$, $s_{min}(vi) = 300$ octets et $s_{max}(vi) = 500$ octets pour tous les VLs. Chaque lien a un débit de 100 Mb/s ($t_{octet} = 0,08\ \mu s$). La figure 4 montre trois scénarios possibles de transmission de trames sur le réseau avec un délai de commutation $dt = 0$. Pour chacun des scénarios de la figure 4, un seul BAG est visualisé. On s'intéresse au délai de bout en bout de la trame $F1$ du VL $v1$ (chemin $p1 = m1 - c1 - c3$). Lorsque la longueur de cette trame vaut $s_{max}(vi)$ (500 octets), le délai de transmission sur les liens $DL(F1, p1)$ vaut $3 \times (0,08 \times 500) = 120\ \mu s$. Lorsque la longueur de cette trame vaut s_{min}, ce même délai vaut 72 μs.

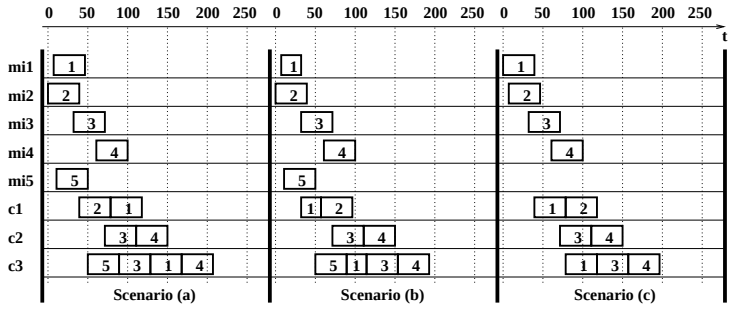


Figure 4. *3 scénarios possibles de séquences de trames*

Dans le scénario **a**, chaque VL génère une trame de longueur maximale. Le délai de bout en bout de la trame de $v1$ est donc 160 μs (120 μs de transmission sur les liens et 80 μs d'attente dans les ports de sortie). Dans le scénario **b**, les instants de génération des trames sont les mêmes que dans le scénario **a**, mais la longueur de la

trame générée par $v1$ passe de 500 octets à 300 octets. Le délai de bout en bout de la trame de $v1$ devient 107 μs (72 μs de transmission sur les liens et 35 μs d'attente dans les ports de sortie). La longueur de la trame émise peut donc influer sur son temps d'attente dans les ports de sortie. Dans le scénario **c**, $v1$, $v2$, $v3$ et $v4$ génèrent une trame de longueur maximale, tandis que $v5$ ne génère pas de trame. Par rapport aux deux scénarios précédents, les instants de génération des trames de $v1$ et $v2$ sont permutés. Dans ce scénario, la trame de $v1$ n'attend pas dans les ports de sortie des commutateurs $c1$ et $c3$. Son délai de bout en bout est donc 120 μs (les temps de transmission sur les liens). Ce scénario montre que pour un VL donné, son phasage par rapport aux autres et le fait que ces autres VLs émettent ou non des trames impacte le délai de bout en bout du VL considéré.

L'analyse du délai de bout en bout d'un chemin p_x d'un VL v donné doit prendre en compte tous les scénarios possibles. Elle doit permettre de déterminer entre autres les caractéristiques suivantes de ce délai de bout en bout.

– La valeur minimale de ce délai, qui correspond au cas où le VL génère une trame de longueur minimale ($s_{min}(v)$), qui n'attend dans aucun des ports de sortie qu'elle traverse. D'après les équations 1, 2 et 3, cette valeur minimale est calculée par $D_{min}(v, p_x) = nbl_{p_x} \times (t_{octet} \times s_{min}(v)) + nbs_{p_x} \times dt$

– La valeur maximale de ce délai, correspondant au scénario pire cas et indispensable pour la certification. Dans le cas général, la recherche de ce scénario nécessite l'énumération de tous les scénarios possibles, ce qui n'est pas réalisable sur des configurations industrielles. Nous présenterons dans la suite deux approches permettant de déterminer une borne supérieure pessimiste du délai de bout en bout pour un chemin d'un VL d'une configuration quelconque, puis une approche permettant de calculer la valeur maximale exacte du délai de bout en bout pour un chemin d'un VL d'une configuration de petite taille.

3. Une borne supérieure garantie du délai par calcul réseau

Le calcul réseau (Chang, 2000)(Le Boudec *et al.*, 2001) a permis d'obtenir, pour chaque chemin de chaque VL, une borne supérieure garantie de son délai de bout en bout. Ces bornes ont permis la certification du réseau AFDX. Nous présentons succinctement le calcul réseau dans le cadre de l'AFDX.

Dans le cas général, le calcul réseau donne des bornes supérieures déterministes sur le délai et la gigue d'un flux transmis sur un réseau. Cette théorie est fondée sur le dioïde $(\min, +)$, dans lequel la convolution $\otimes$ et la déconvolution $\oslash$ sont définies par $(f \otimes g)(t) = \inf_{0 \leq u \leq t} (f(t-u) + g(u))$ et $(f \oslash g)(t) = \sup_{0 \leq u} (f(t+u) - g(u))$

Un flux est représenté par une fonction R, telle que $R(t)$ est le nombre total de bits émis par le flux à l'instant t. Une fonction α est une courbe d'arrivée associée à R si et seulement si $R \leq R \otimes \alpha$. Dans le contexte de l'AFDX, la courbe d'arrivée de chacun des flux (VL) est un seau percé $\gamma_{r,b}$ de capacité b (la rafale maximale) et de débit r. Pour chaque VL v, on a $b = s_{max}(v)$ et $r = \frac{s_{max}(v)}{BAG(v)}$, comme l'illustre la figure 5.

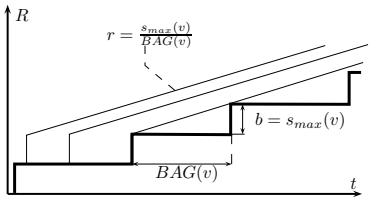


Figure 5. *Courbe d'arrivée $\gamma_{r,b}$ d'1 VL*

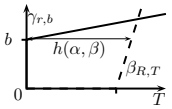


Figure 6. *Délai max $h(\alpha, \beta)$*

Un serveur a une courbe de service β si et seulement si, pour tout flux qu'il traite, on a $R' \geq R \otimes \beta$, R et R' correspondant respectivement aux flux d'entrée et de sortie. Dans ce cas, $\alpha' = \alpha \oslash \beta$ est une courbe d'arrivée pour R'. Un commutateur AFDX est considéré comme un ensemble de ports de sortie. Chaque port de sortie offre une courbe de service de la forme $\beta_{R,T} = R\,[t - T]^+$. R est le débit du lien sortant (c dans notre cas) et T est la latence technologique maximale (dt dans notre cas).

Un réseau AFDX peut donc être modélisé comme un ensemble de flux ayant des courbes d'arrivée de la forme $\alpha = \gamma_{r,b}$ à l'entrée du réseau, chacun de ces flux traversant un ensemble de ports de sortie offrant des courbes de service de la forme $\beta = \beta_{R,T}$. Pour calculer les bornes supérieures des délais, on considère les courbes d'arrivée et de service, et non pas les instants d'arrivée exacts et les taux de service exacts. Le délai subi par un flux R contraint par une courbe d'arrivée α traversant seul un nœud offrant une courbe de service β est borné par la différence horizontale maximale entre les courbes α et β. Cette différence, illustrée sur la figure 6, est définie par :

$$h(\alpha, \beta) = \sup_{s \geq 0} \left(\inf \left\{ \tau \geq 0 \mid \alpha(s) \leq \beta(s + \tau) \right\} \right) \quad [5]$$

Chaque port de sortie du réseau AFDX est partagé par plusieurs flux. La borne sur le délai dans un port de sortie donné est obtenu en sommant en obtenu en sommant les courbes d'arrivée des différents flux partageant ce port.

La généralisation du calcul pour un flux traversant plusieurs ports de sortie s'effectue en déterminant la courbe de sortie du flux pour chacun des ports qu'il traverse. Chacune de ces courbes de sortie devient la courbe d'entrée du port suivant sur le chemin du flux. Cette propagation est simple dans la mesure où il n'y a pas de dépendance circulaire dans les configurations AFDX industrielles. La courbe de sortie d'un flux pour un port de sortie donné est : $\alpha' = \alpha \oslash \delta_{gigue}$. α est la courbe d'arrivée du flux dans le port, $gigue$ est la gigue maximale subie par le flux dans le port et δ_{gigue} est une courbe de service de type délai garanti : on a $\delta_d(t) = 0$ si $t \leq d$, ∞ sinon. Graphiquement, cela revient à décaler la courbe d'arrivée α vers la gauche, de la valeur de la gigue maximale. La gigue maximum dans un port de sortie correspond à la différence entre le délai minimum et le délai maximum dans ce même port. On a montré à la section 2 que le délai minimum correspond aux scénarios où la trame

n'attend pas dans la file associée au port de sortie. La gigue maximale dans un port de sortie correspond donc au temps maximum d'attente dans la file associée au port.

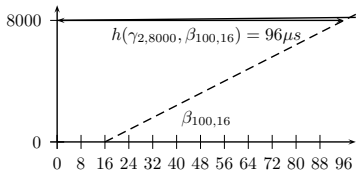


Figure 7. *Délai max en sortie de* $c1$

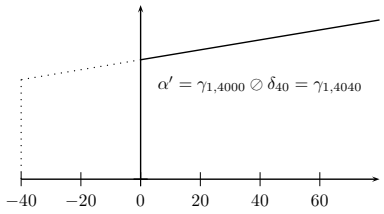


Figure 8. $v1$ *en sortie de* $c1$

La méthode de calcul est illustrée sur les VLs $v1$ et $v2$ de la figure 3. Ces deux VLs ont comme courbes d'arrivée $\alpha_1 = \alpha_2 = \gamma_{1,4000}$. En effet, ces deux VLs ont un BAG de $4000\ \mu s$ et une longueur de trame de 4000 bits (500 octets). On a donc $r = \frac{s_{max}(vi)}{BAG(vi)} = 1\ Mb/s$ et $b = s_{max}(vi) = 4000\ bits$. La courbe d'arrivée globale pour l'unique port de sortie du commutateur $c1$ est donc $\alpha_1 + \alpha_2 = \gamma_{2,8000}$. La courbe de service de ce port de sortie est $\beta_{100,16}$. En effet, le lien de sortie a un débit de $100\ Mb/s$ et on considère une latence technologique maximale de $16\ \mu s$. Le délai maximum dans ce port de sortie est donc borné par la différence horizontale maximale entre $\gamma_{2,8000}$ et $\beta_{100,16}$. Comme le montre la figure 7, la borne supérieure de ce délai vaut $h(\gamma_{2,8000}, \beta_{100,16}) = 96\ \mu s$. Ce délai maximum inclut, d'une part le délai minimum (sans attente dans la file associé au port) qui comprend la latence technologique de $16\ \mu s$ et le temps de transmission de $40\ \mu s$, d'autre part le temps d'attente dans la file, qui est borné par $40\ \mu s$ (chaque trame d'un des deux VLs $v1$ ou $v2$ peut être retardée par au plus une trame de l'autre VL). La gigue maximum subie par $v1$ ou $v2$ dans le port de sortie de $c1$ est donc $40\ \mu s$. La courbe de sortie α'_1 de $v1$ (respectivement α'_2 de $v2$ est donc obtenue en décalant α_1 (respectivement α_2 de $40\ \mu s$ vers la gauche ($\alpha'_1 = \alpha_1 \oslash \delta_{40} = \gamma_{1,4040}$), comme le montre la figure 8. $v1$ et $v2$ traversent ensuite le commutateur $c3$. $v2$ est seul sur un port de sortie de $c3$. La courbe d'arrivée des flux traversant à ce port de sortie est donc la courbe d'arrivée de $v2$ sur $c3$, i.e. $\gamma_{1,4040}$. La borne supérieure du délai subi par $v2$ dans $c3$ est obtenu de la même manière que pour $c1$, en déterminant la différence horizontale maximale entre $\gamma_{1,4040}$ et $\beta_{100,16}$, la courbe de service d'un port de sortie de $c3$. Cette borne est donc de $56,4\ \mu s$. La borne supérieure du délai de bout en bout de $v2$ est obtenue en sommant le délai de transmission sur le lien $m1 - c1$ ($40\ \mu s$) et les bornes supérieures des délais subis par $v2$ dans $c1$ et $c3$ ($96\ \mu s$ et $56,4\ \mu s$).

$v1$ partage l'autre port de sortie de $c3$ avec $v3$, $v4$ etse $v5$. $v3$ et $v4$ ont exactement les mêmes caractéristiques que $v1$ et $v2$ jusqu'à leur arrivée dans $c3$. Elles ont donc les mêmes courbes d'arrivée que $v1$ et $v2$ dans le port de sortie de $c3$, i.e. $\gamma_{1,4040}$. Quant à $v5$, sa courbe d'arrivée sur ce même port de sortie est $\gamma_{1,4000}$. La courbe d'arrivée globale de l'ensemble des flux sur ce port de sortie est donc $3 \times \gamma_{1,4040} + \gamma_{1,4000} = \gamma_{4,16120}$. La borne supérieure du délai subi par $v1$ dans $c3$ est donc $177,2\ \mu s$. La borne supérieure du délai de bout en bout de $v1$ est donc $40 + 96 + 177,2 = 313,2\ \mu s$.

Le tableau 1 (colonne *calcul réseau de base*) donne la borne obtenue pour chacun des VLs de la figure 3.

VL	Vérif	Calcul réseau de base	Calcul réseau avec groupage	Trajectoire de base	Trajectoire avec groupage
$v1$	$272\ \mu s$	$313,2\ \mu s$	$273,62449\ \mu s$	$312\ \mu s$	$272\ \mu s$
$v2$	$192\ \mu s$	$192,4\ \mu s$	$192,4\ \mu s$	$192\ \mu s$	$192\ \mu s$
$v3$, $v4$	$272\ \mu s$	$313,2\ \mu s$	$273,62449\ \mu s$	$272\ \mu s$	$272\ \mu s$
$v5$	$176\ \mu s$	$217,2\ \mu s$	$177,62449\ \mu s$	$216\ \mu s$	$176\ \mu s$

Tableau 1. *Bornes supérieures des délais de bout en bout par les différentes approches*

La méthode de calcul somme donc, pour un port de sortie, l'ensemble des courbes d'arrivée des flux traversant ce port. Elle ne tient donc pas compte du fait que des flux arrivant via un même lien sont sérialisés (deux trames ne peuvent pas arriver en même temps via un lien donné). La courbe d'arrivée résultante présente donc une rafale maximale qui ne peut jamais être atteinte. Ainsi, sur l'exemple de la figure 3, la courbe d'arrivée calculée pour le port de sortie de $c3$ partagé par $v1$, $v3$, $v4$ et $v5$ est $\gamma_{4,16120}$. Sa rafale maximale (16120 bits) correspond au cas où 4 trames, une pour chaque VL, arrivent en même temps sur le port de sortie. Cela n'est pas possible, dans la mesure où $v3$ et $v4$ arrivent via le même lien. La technique du groupage (Grieu, 2004) permet de prendre en compte cette sérialisation des flux dans le calcul de la courbe d'arrivée pour un port de sortie. Cette technique procède en deux étapes. Dans la première étape, une courbe d'arrivée par lien est calculée. Elle résulte du minimum entre, d'une part, la courbe obtenue en sommant les courbes d'arrivée des flux partageant le lien, d'autre part la courbe bornant la rafale maximum à la plus grande rafale parmi les courbes d'arrivée des différents flux et le débit au débit du lien. La figure 9 illustre cette première étape pour un lien partagé par deux flux ayant des courbes d'arrivée γ_{r_1,b_1} et γ_{r_2,b_2}. Dans la deuxième étape, on somme les courbes d'arrivée par lien obtenues lors de la première étape. Le gain procuré par cette technique de groupage est dû au fait que la rafale maximale est réduite.

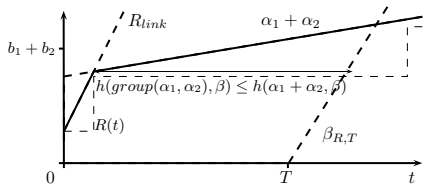


Figure 9. *Courbe d'arrivée pour un lien partagé par 2 flux de courbes γ_{r_1,b_1} et γ_{r_2,b_2}*

Le tableau 1 (colonne *calcul réseau avec groupage*) donne la borne obtenue avec cette méthode pour chacun des VLs de la figure 3.

4. Une borne supérieure garantie du délai par la méthode des trajectoires

La méthode des trajectoires (Martin *et al.*, 2006)(Migge, 1999) calcule des bornes supérieures garanties sur les temps de réponse de flux sporadiques dans les systèmes distribués. L'application de cette méthode à l'AFDX s'appuie sur les résultats de (Martin *et al.*, 2006).

Un système distribué se compose d'un ensemble de nœuds de calcul interconnectés. Le chemin suivi par chaque flux à travers ce système est défini statiquement. La méthode des trajectoires suppose que, pour tous flux τ_i et τ_j suivant des chemins $\mathcal{P}_i$ et $\mathcal{P}_j$ non disjoints ($\mathcal{P}_j \cap \mathcal{P}_i \neq \emptyset$), τ_j ne rejoint pas τ_i après l'avoir quitté.

L'algorithme d'ordonnancement des flux dans les nœuds est FIFO. Chaque flux τ_i est défini par une durée inter-paquets minimum T_i et une gigue maximale J_i sur son nœud source, une valeur maximum acceptable D_i pour le temps de réponse de bout en bout et une durée d'exécution pire cas C_i^h pour chaque nœud h de $\mathcal{P}_i$.

La durée de transmission d'un paquet sur un lien est comprise entre L_{min} et L_{max}. Il n'y a, ni collision, ni perte de paquets sur les liens.

Le temps de réponse de bout-en-bout d'un paquet est la somme des délais de transmission sur les liens (au plus L_{max} par lien) et des durées passées dans chacun des nœuds traversés. Le temps passé par un paquet m dans un nœud h dépend des paquets en attente dans h au moment où m y arrive (du fait de l'ordonnancement FIFO, ils seront traités avant m). Le problème est donc de déterminer une borne supérieure du temps passé par le paquet considéré dans l'ensemble des nœuds qu'il visite.

L'approche par trajectoire est fondée sur la notion de période active. Une période active de niveau $\mathcal{L}$ est un intervalle $[t, t')$ tel que t et t' sont des instants de repos de niveau $\mathcal{L}$ et il n'y a pas d'instant de repos de niveau $\mathcal{L}$ dans (t, t'). Un instant de repos t de niveau $\mathcal{L}$ est tel que tous les paquets ayant une priorité supérieure ou égale à $\mathcal{L}$ et générés avant t ont été traités à l'instant t. Avec l'ordonnancement FIFO, aucun paquet de la période active de niveau correspondant à la priorité de m ne peut être arrivé après m sur le nœud considéré.

Pour un paquet m d'un flux τ_i généré à l'instant t, l'approche par trajectoires identifie pour chaque nœud traversé par m la période active et les paquets retardant m, du dernier nœud visité par m vers le premier. Cette décomposition permet le calcul de l'instant de début au plus tard de m sur ce dernier nœud. Cet instant de début peut être calculé récursivement et conduit au pire temps de réponse du flux τ_i.

Dans le contexte de l'AFDX, chaque nœud du système correspond à un port de sortie de commutateur, incluant le lien de sortie. Chaque lien du système correspond au mécanisme de commutation entre les ports d'entrée et les ports de sortie d'un commutateur. Chaque flux correspond à un chemin d'un VL. Toutes les hypothèses de l'approche par trajectoire sont vérifiées (politique de service FIFO en sortie des commutateurs, pas de collision ou de perte de paquet, délai de commutation borné, routage statique des VLs qui ne se croisent jamais deux fois). Les paramètres des

VLs correspondent à la définition des flux sporadiques du système : $T_i = BAG(v_i)$, $C_i^h = \frac{s_{max}(v)}{R}$, $J_i = 0$. Tous les liens ont le même débit $R = 100\ Mb/s$. On a donc $C_i^h = C_i$ pour tout nœud h du chemin de v considéré.

L'approche par trajectoire appliquée à l'AFDX est maintenant illustrée sur l'exemple de la figure 3. La figure 10 montre un ordonnancement possible de trames des VLs traversant les commutateurs $c2$ et/ou $c3$. Sur la figure, la trame i correspond au VL vi. L'instant d'arrivée de la trame 3 sur le nœud $m3$ (a_3^{m3}) est arbitrairement choisi comme instant de départ. La trame 3 est traitée par le nœud $m3$ puis, après le délai de commutation de 16 μs, elle arrive dans le nœud $c2$ à l'instant $a_3^{c2} = 56$. La trame 4 arrive dans le nœud $c2$ au même instant $a_4^{c2} = 56$ et elle est arbitrairement traitée avant la trame 3. La trame 4 arrive donc dans le nœud $c3$ à l'instant $a_4^{c3} = 112$ et elle y est immédiatement traitée. Les trames 1 et 5 arrivent dans ce même nœud $c3$ aux instants $a_1^{c3} = 127$ et $a_5^{c3} = 136$. La trame 3, qui est la dernière à être traitée dans la période active bp^{c3}, arrive dans le nœud $c3$ à l'instant $a_3^{c3} = 152$.

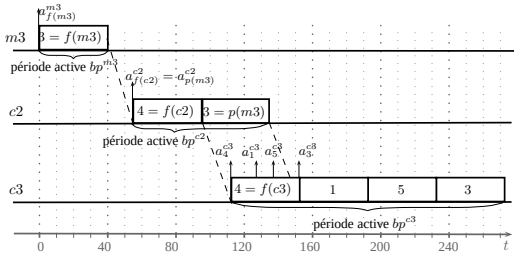


Figure 10. *Un ordonnancement possible des trames*

La trame 3 traverse donc trois périodes actives (bp^{e3}, bp^{c2} et bp^{c3}) sur sa trajectoire. Examinons de plus près bp^{c3}, la période active dans laquelle la trame 3 est traitée dans le nœud $c3$. Soit $f(c3)$, la première trame de priorité supérieure ou égale à celle de la trame 3 traitée dans bp^{c3}. Les flux ne suivant pas nécessairement le même chemin, la trame $f(c3)$ peut ne pas venir du même nœud que la trame 3. Nous définissons donc $p(c2)$, la première trame traitée entre $f(c3)$ et 3 et venant du même nœud que 3. Sur l'exemple de la figure 10, $p(c2) = f(c3)$ est la trame 4 provenant du nœud $c2$. La trame $p(c2)$ a été traitée dans le nœud $c2$ au cours d'une période active bp^{c2} de niveau correspondant à la priorité de $p(c2)$. $f(c2)$ est défini comme la première trame de priorité supérieure ou égale à celle de $p(c2)$. Sur la figure 10, nous avons $f(c2) = p(c2)$. La même convention de nommage est appliqué jusqu'au premier nœud du chemin du VL considéré, i.e. le nœud $m3$ pour l'exemple de la figure 10. L'instant de début de la trame 3 sur le nœud $c3$ est exprimé en additionnant des parties des périodes actives considérées.

Il a été montré dans (Martin *et al.*, 2006) que l'ordonnancement de la figure 10 conduit au pire délai de bout en bout pour une trame de $v3$. Cet ordonnancement est construit en décalant, dans chaque nœud, l'instant d'arrivée des trames rencontrant la

trame 3 pour la première fois (donc venant d'un autre nœud) de manière à retarder l'instant de début de traitement de la trame 3 dans le dernier nœud de son chemin. Ainsi, dans le nœud $c3$, les trames 1 et 5 arrivent entre les trames 4 et 3. Clairement, si la trame 1 ou la trame 5 arrivait avant la trame 4, la période active bp^{c2} débuterait plus tôt et le traitement de la trame 3 serait avancé d'autant. De la même manière, si la trame 1 ou la trame 5 arrivait après la trame 3, celle-ci serait traitée avant la trame 1 ou la trame 5. Des observations similaires peuvent être faites pour le nœud $c2$.

On peut donc déduire de la figure 10 que le délai maximal de bout en bout pour une trame de $v3$ est : $C_3^{m3} + L + C_4^{c2} + L + C_4^{c3} + C_1^{c3} + C_5^{c3} + C_3^{c3} = 272\ \mu s$ (sur cet exemple, $C_m^h = 40\ \mu s$ pour tout flux m sur tout nœud h et $L = 16\ \mu s$).

Un calcul similaire peut être effectué pour tous les VLs de la configuration. Les délais pire cas obtenus sont présentés dans le tableau 1 (colonne *trajectoires de base*). Pour tous les VLs de cet exemple, l'approche par trajectoire donne de meilleurs résultats que l'approche par calcul réseau de base. En revanche, l'approche par calcul réseau avec groupage est meilleure pour les VLs $v1$ et $v5$. Ce sont les seuls VLs de la configuration pour lesquels la prise en compte de la sérialisation des flux diminue la borne obtenue par calcul réseau.

Ces résultats ne sont pas surprenants. En effet, l'application de la méthode des trajectoires à l'AFDX, telle que présentée au-dessus, ne tient pas compte de la sérialisation des flux. Considérons à titre d'exemple le VL $v5$ de la figure 3. La figure 11 montre l'ordonnancement considéré par l'approche par trajectoire pour le calcul du délai pire cas de $v5$. Dans ce scénario, les trames des flux $v1$, $v3$, $v4$ et $v5$ arrivent au même instant dans le nœud $c3$ et la trame de $v5$ est arbitrairement traitée en dernier. Le délai de bout en bout de la trame de $v5$ est alors de $216\ \mu s$. Il s'avère que ce scénario n'est pas possible. En effet, les trames des flux $v3$ et $v4$ proviennent du même lien $c2 - c3$ et leurs arrivées dans le nœud $c3$ sont donc espacées d'au moins $40\ \mu s$. Un ordonnancement pire-cas pour une trame de $v5$ correspond donc au scénario où la trame de $v5$ arrive dans $c3$ au même instant que les trames de $v1$ et $v3$ ou $v4$ (mais pas les deux). La dernière trame (de $v3$ ou $v4$) arrive dans $c3$ au moins $40\ \mu s$ avant ou après la trame de $v5$. La figure 12 montre l'ordonnancement correspondant à un tel scénario. Le délai maximal de bout en bout pour une trame de $v5$ est donc $176\ \mu s$.

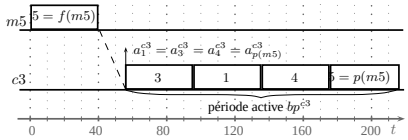


Figure 11. *Pire cas théorique pour* $v5$

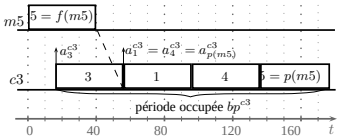


Figure 12. *Pire cas faisable pour* $v5$

Plus généralement, la sérialisation des flux est intégrée dans l'approche par trajectoire en agrégeant, pour chaque nœud, les flux par lien entrant. Pour chaque lien, la fonction du flux agrégé est la somme des fonctions des flux transmis sur ce lien, bornée par un seau percé dont la rafale correspond à la plus grande trame transmise sur le

lien et le débit est le débit maximum du lien. Les résultats obtenus sur la configuration de la figure 3 sont présentés dans le tableau 1 (colonne trajectoires avec groupage).

5. Le calcul du délai pire cas exact par vérification de modèle

L'approche considérée est fondée sur une modélisation du système en automates temporisés. Ceux-ci ont été introduits par Alur et Dill (Alur *et al.*, 1994) pour décrire le comportement de systèmes en incluant le temps. Un automate temporisé est un automate d'états finis auquel est associé un ensemble d'horloges, qui sont des variables réelles positives augmentant uniformément avec le temps. A chaque transition de l'automate sont associées une condition sur les valeurs des horloges(la garde), des actions à exécuter et des affectations de valeurs aux horloges. Le produit synchronisé permet la composition des automates temporisés. Les communications utilisent un mécanisme de rendez-vous : la synchronisation entre deux automates ne peut avoir lieu que si les deux automates peuvent exécuter la même action au même instant. Le franchissement d'une transition s'effectue en temps nul. En revanche, le temps s'écoule dans les nœuds. Un invariant est associé à chaque nœud. Il s'agit d'une condition booléenne sur des horloges. Un nœud ne peut être occupé que si son invariant est vrai.

Plusieurs extensions des automates temporisés ont été proposées. L'une d'elles concerne les variables entières partagées qui sont consultées et mises à jour par n'importe quel automate modélisant l'application (Burgueño Arjona, 1998).

Un système modélisé par des automates temporisés peut être vérifié en utilisant l'analyse d'accessibilité. La propriété à vérifier est codée en terme d'accessibilité d'un nœud donné de l'un des automates. La propriété est vérifiée si le nœud est atteignable depuis la configuration initiale. L'accessibilité est décidable pour les automates temporisés de base et il existe des algorithmes (Larsen *et al.*, 1997). Elle ne l"est pas pour les automates temporisés avec variables entières partagées. Nous utilisons les variables entières partagées pour identifier le message présent dans chaque position de la file d'attente. Dans ce cas particulier, la méthode de dépliage permet de transformer ces variables en nœuds d'un automate temporisé. L'accessibilité est alors décidable.

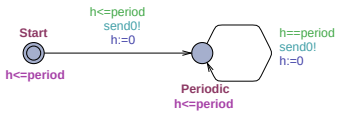


Figure 13. *Automate associé à un VL*

L'AFDX est modélisé en construisant un automate pour chaque VL et un automate pour chaque port de sortie d'un commutateur. La figure 13 montre l'automate temporisé associé à un VL de BAG $period$. Cet automate génère un premier message $send_i$ ($send_0$ sur l'exemple) à l'issue d'une durée comprise entre 0 et $period$, puis génère ensuite périodiquement un nouveau message $send_i$ (la période vaut le BAG du VL,

i.e. $period$). Cet automate modélise donc un VL purement périodique avec une phase comprise entre 0 et son BAG.

La figure 14 montre un exemple d'automate modélisant un port de sortie de commutateur. Chaque nœud de l'automate modélise un emplacement de la file d'attente FIFO associée au port. Le nombre de nœuds de l'automate est donc égal à la taille de cette file (trois sur l'exemple de la figure 14). Toute transition d'un nœud $Position_i$ vers un nœud $Position_{i+1}$ modélise l'arrivée d'une trame dans la file d'attente du port de sortie, tandis que toute transition d'un nœud $Position_{i+1}$ vers un nœud $Position_i$ modélise la fin de transmission d'une trame depuis ce port. L'automate de la figure 14 considère deux flux (i.e. deux VLS) reçus via les signaux $send_0$ et $send_1$ et transmis via les signaux $send_4$ pour $send_0$ et $send_5$ pour $send_1$. $delay$ est la durée de transmission d'une trame sur le lien. Sur l'exemple considéré, toutes les trames ont la même longueur. $pos1$, $pos2$ et $pos3$ précisent les flux (0 ou 1) auxquels correspondent les trames en attente dans chacune des cases de la file.

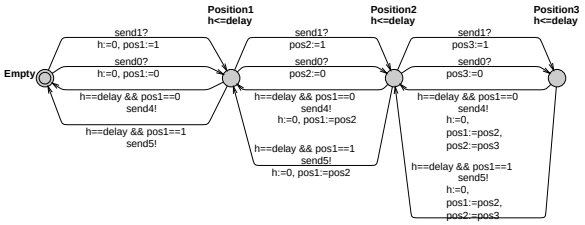


Figure 14. *Automate associé à un port de sortie de commutateur*

Le système global est obtenu en composant les automates des VLs et des ports de sortie des commutateurs. Pour l'exemple de la figure 3 qui comprend 5 VLs et 4 ports de sortie, le modèle comprend donc 9 automates, comme l'illustre la figure 16. A titre d'exemple, le VL $v2$ est modélisé par l'automate $v2$, qui génère le signal $send1$. celui-ci est reçu par l'automate ***p1-1*** qui modélise l'unique port de sortie du commutateur $c1$. Le cheminement de $v2$ se poursuit via les signaux $send5$ et $send10$.

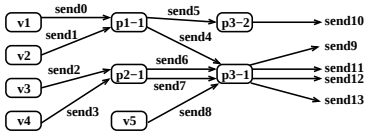


Figure 15. *Modèle global du système*

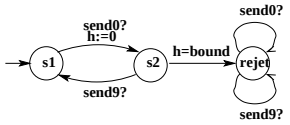


Figure 16. *Automate de test de* $v1$

Partant de ce modèle, le délai pire cas pour chaque VL est obtenu en utilisant la méthode de l'automate de test (Bérard ***et al.***, 2001; Burgueño Arjona, 1998). L'automate de test correspondant au VL $v1$ est présenté sur la figure 16. Il modélise la propriété

"délai de $v1$ inférieur à $bound$". Lorsque le signal $send0$ est reçu, on passe dans l'état $s2$. On attend alors le signal $send9$ (transmission de $v1$ en sortie du commutateur $c3$. S'il n'est pas arrivé à l'issue d'une durée $bound$, on passe dans l'état $rejet$. Cela signifie qu'il existe au moins un scénario où $v1$ dépasse la borne $bound$. Le principe consiste donc à déterminer la plus petite valeur de $bound$ pour laquelle l'état $rejet$ est atteignable. On obtient en outre un scénario correspondant à cette valeur maximale de délai de bout en bout. L'outil UPPAAL est utilisé. La vérification s'effectue en moins d'une seconde sur une station Linux munie d'un Pentium 4 et de 2 GO de RAM.

Les délais pire-cas exacts obtenus avec cette approche de vérification de modèle pour les VLs de la figure 3 sont donnés dans le tableau 1 (colonne vérif).

6. Analyse d'une configuration industrielle réaliste

La configuration d'un réseau AFDX industriel analysée dans ce paragraphe comprend deux sous-réseaux AFDX redondants de huit commutateurs chacun interconnectant plus de cent modules d'interface. Un millier de VLs est transmis sur chaque sous-réseau, pour un total de plus de 6000 chemins (Charara *et al.*, 2006). Une telle configuration ne peut pas être analysée par l'approche de vérification de modèle, en raison de l'explosion combinatoire liée à ce type de méthode. L'analyse est donc menée en utilisant l'approche par calcul réseau avec ou sans groupage et l'approche par trajectoires avec ou sans groupage. Des outils mettant en œuvre ces différentes approches ont été développés. Une synthèse des résultats obtenus est présentée dans le tableau 2. La borne supérieure du délai a été calculée avec chacune des quatre approches pour chaque chemin de chaque VL de la configuration. Le tableau 2 donne pour chaque approche la moyenne de l'ensemble des bornes obtenues, ainsi que la plus petite et la plus grande borne. Les résultats montrent l'apport important de la sérialisation des flux dans le contexte de l'AFDX, que ce soit pour le calcul réseau (la moyenne des bornes passe de $6036,9\ \mu s$ à $4532,4 \mu s$) où pour la méthode des trajectoires (la moyenne des bornes passe de $4869,5\ \mu s$ à $4009,8\ \mu s$). Ces résultats montrent aussi que les bornes obtenues par l'approche par trajectoire avec groupage sont inférieures en moyenne de plus de 10 % à celles obtenues par l'approche par calcul réseau avec groupage.

7. Conclusion

Ce papier présente plusieurs alternatives pour borner les délais de bout en bout sur un réseau de type AFDX pour des flux homogènes (même qualité de service). Seul la vérification de modèle permet d'exhiber une borne atteignable, mais ne peut être actuellement appliqué que sur des configurations restreintes. Les méthodes d'abstraction pourraient permettre d'analyser des configurations plus complexes, sans toutefois aller jusqu'à une configuration industrielle. L'analyse du pire cas obtenu permet néanmoins de mieux comprendre le comportement d'un tel réseau. L'approche par calcul réseau est appliquée sur des configurations industrielles pour des besoins de certification. Les

bornes obtenues restent cependant pessimistes, malgré l'amélioration significative obtenue par la prise en compte de la sérialisation des flux. L'approche par trajectoire est elle-aussi applicable sur des configurations industrielles. Elle nécessite cependant une généralisation en y intégrant la sérialisation des flux pour obtenir des gains de l'ordre de 10 % par rapport à l'approche par calcul réseau. Un axe de recherche important concerne la généralisation des méthodes au cas de flux ayant des contraintes de qualité de service hétérogènes, dans le cadre de l'évolution de l'AFDX.

	Vérif	Calcul réseau de base	Calcul réseau avec groupage	Trajectoire de base	Trajectoire avec groupage
Minimum	-	$474,4\ \mu s$	$386,2\ \mu s$	$406,2\ \mu s$	$308,6\ \mu s$
Moyenne	-	$6036,9\ \mu s$	$4532,4 \mu s$	$4869,5\ \mu s$	$4009,8\ \mu s$
Maximum	-	$17868,1\ \mu s$	$13194,2\ \mu s$	$12553,8 \mu s$	$10892,1\ \mu s$

Tableau 2. *Bornes supérieures obtenues sur une configuration industrielle*

8. Bibliographie

Alur R., Dill D. L., « Theory of Timed Automata », *Theoritical Computer Science*, vol. 126, n° 2, p. 183-235, 1994.

ARI, ARINC Specification 664 : Aircraft Data Network, Parts 1,2,7, Technical report, Aeronotical Radio Inc., 2002-2005.

Bérard B., Bidoit M., Finkel A., Laroussinie F., Petit A., Petrucci L., Schnoebelen Ph., *Systems and Software Verification. Model-Checking Techniques and Tools*, Springer, 2001.

Burgueño Arjona A., Vérification et synthèse de systèmes temporisés par des méthodes d'obervation et d'analyse paramétrique, PhD thesis, Ecole Nationale Supérieur de l'Aéronautique et de l'Espace, Toulouse, France, 1998.

Chang C.-S., *Performance Guarantees in Communication Networks*, Springer-Verlag, London, UK, 2000. isbn = {1852332263}.

Charara H., Scharbarg J.-L., Ermont J., Fraboul C., « Methods for Bounding end-to-end Delays on an AFDX Network », *Proceedings of the 18th ECRTS*, Dresde, Germany, July, 2006.

Grieu J., Analyse et évaluation de techniques de commutation Ethernet pour l'interconnexion des systèmes avioniques, PhD thesis, INP-ENSEEIHT, France, September, 2004.

Larsen K. G., Pettersson P., Yi W., « UPPAAL in a Nutshell », *International Journal on Software Tools for Technology Transfer*, vol. 1, n° 1–2, p. 134-152, 1997.

Le Boudec J.-Y., Thiran P., *Network Calculus : A Theory of Deterministic Queuing Systems for the Internet*, vol. 2050 of *Lecture Notes in Computer Science*, Springer-Verlag, 2001. ISBN : 3-540-42184-X.

Martin S., Minet P., « Schedulability analysis of flows scheduled with fifo : application to the expedited forwarding class », *IPDPS*, 2006.

Migge J., L'ordonnancement sous contraintes temps-réel : un modèle à base de trajectoires, PhD thesis, INRIA, Sophia Antipolis France, November, 1999.

Scharbarg J.-L., Ridouard F., Fraboul C., « A probabilistic analysis of end-to-end delays on an AFDX network », *IEEE transactions on industrial informatics*, February, 2009.

Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures
et Langages

Matrice de dépendances enrichie

Jannik Laval, Alexandre Bergel, Stéphane Ducasse, Romain Piers

RMoD Team, INRIA - Lille Nord Europe - USTL - CNRS UMR 8022, Lille, France
firstname.lastname@inria.fr

Résumé. Les matrices de dépendance (DSM - Dependency Structure Matrix), développées dans le cadre de l'optimisation de processus, ont fait leurs preuves pour identifier les dépendances logicielles entre des packages ou des sous-systèmes. Il existe plusieurs algorithmes pour structurer une matrice de façon à ce qu'elle reflète l'architecture des éléments analysés et mette en évidence des cycles entre les sous-systèmes. Cependant, les implémentations de matrices de dépendance existantes manquent d'informations importantes pour apporter une réelle aide au travail de réingénierie. Par exemple, le poids des relations qui posent problème ainsi que leur type ne sont pas clairement présentés. Ou encore, des cycles indépendants sont fusionnés. Il est également difficile d'obtenir une visualisation centrée sur un package. Dans ce papier, nous améliorons les matrices de dépendance en ajoutant des informations sur (i) le type de références, (ii) le nombre d'entités référençantes, (iii) le nombre d'entités référencées. Nous distinguons également les cycles indépendants. Ce travail a été implémenté dans l'environnement de réingénierie open-source *Moose*. Il a été appliqué à des études de cas complexes comme le framework *Morphic UI* contenu dans les environnements Smalltalk open-source *Squeak* et *Pharo*. Les résultats obtenus ont été appliqués dans l'environnement de programmation *Pharo* et ont mené à des améliorations.

Note : cet article a été publié l'an passé à LMO 09, revue RNTI L-3, éditions Cépaduès

1 Introduction

Comprendre la structure de grosses applications est un défi mais aussi une tâche importante pour la maintenance (Demeyer et al., 2002). Plusieurs approches présentent des informations à propos des packages et de leurs relations, par la visualisation d'artefacts logiciels, de leur structure, de leur évolution ou encore de métriques. Les métriques logicielles peuvent être difficiles à comprendre puisqu'elles dépendent fortement de l'application analysée. Distribution Map (Ducasse et al., 2006) résoud ce problème en montrant comment les propriétés se répartissent dans une application. Langelier et al. (2005) caractérisent l'évolution des packages et de leurs métriques. Package Surface Blueprint (Ducasse et al., 2007) révèle la structure interne d'un package et les relations avec les autres packages — le concept de surface représente les relations entre le package analysé et les packages auxquels il accède. Dong et Godfrey (2007) ont travaillé sur la présentation de systèmes par des graphes de dépendances objets de haut niveau pour aider à la compréhension des systèmes au niveau de leurs structures de packages.

Les matrices de dépendance (DSM - Dependency Structure Matrix) sont une solution éprouvée pour révéler les problèmes de cycles dans une structure (Sangal et al., 2005). Développées à l'origine pour l'optimisation des processus, elles mettent en évidence des problèmes de dépendances entre tâches. La technique a été appliquée avec succès pour l'identification de dépendances applicatives (Steward (1981); Sullivan et al. (2001); Lopes et Fiadeiro (2005)). MacCormack et al. (2006) ont appliqué les matrices de dépendance pour analyser le niveau de modularité de l'architecture de Mozilla et Linux.

Cependant, appliquéesà la réingénierie logicielle, elles n'offre pas une solution optimale : elles ne fournissent pas suffisamment d'informations exploitables. Les DSMs utilisées de nos jours permettent juste de faire un état des lieux, sans réellement offrir une aide à la re-ingéniérie. Par exemple, les implémentations actuelles de DSM ne fournissent pas d'informations à propos des types de liens entre les entités. Certains algorithmes (comme l'élévation à la puissance des matrices d'adjacences) n'affichent pas tous les cycles indépendamment les uns des autres. Il est également difficile de détecter les cycles propre à une entité précise. De plus, les matrices de dépendance ne prennent pas en compte les extensions de classes, construction élémentaire dans les languages dynamiquement typés[1].

Notre contribution est double : d'une part, nous identifions les faiblesses des matrices de dépendance traditionnelles (Section 2), d'autre part, nous répondons à ces faiblesses (Section 3). Nous appliquons notre outil sur le package *Morphic UI* afin de le remodulariser et l'intégrer à *Pharo*, une nouvelle version de Squeak. Nous proposons des matrices de dépendances enrichies (EDSM). Nous enrichissons les cellules des matrices de dépendance avec des informations contextuelles en affichant (i) les types de références existantes (héritage, accès aux classes, ...), (ii) la proportion d'entités (classes/méthodes) origines, (iii) la proportion d'entités cibles. Nous distinguons également les cycles indépendants et utilisons des informations colorimétriques pour cela. Finalement nous offrons une vision par niveau des cycles entre les entités.

Le document est organisé de la façon suivante : la Section 2 présente les DSM et leurs limites actuelles. La Section 3 présente notre solution. La Section 4 montre un exemple de mise en œuvre sur l'application *Morphic UI*. La Section 5 présente les limites de notre solution. La Section 6 conclue le document.

2 Limites des matrices de dépendance

Les matrices de dépendances (DSM) sont efficaces pour détecter des cycles entre composants logiciels. Bien que ces résultats soient pertinents pour vérifier l'indépendance de différents composants logiciels (Sangal et al., 2005), les DSM sont moins utiles pour la re-ingénierie.

Nous avons identifié un certain nombre de limites aux DSM : la fusion de certains cycles indépendants (Section 2.1), le manque de précision de la visualisation (Section 2.2), l'absence d'information de cycles pour une entité précise (Section 2.3) et le manque de support des extensions de classes (Section 2.4).

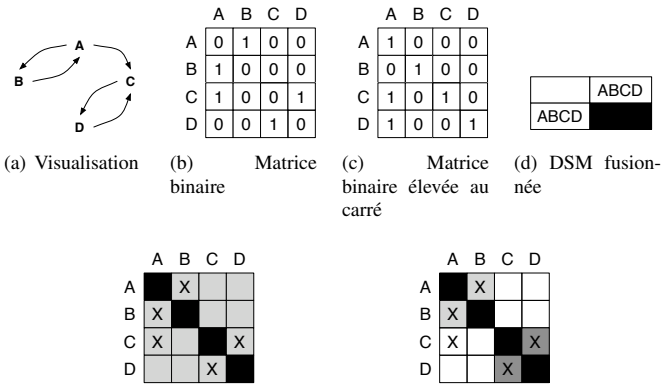
[1] Une extension de classe est le fait qu'une méthode peut être définie dans un autre package que celui de sa classe. Les langage Objective-C, Smalltalk, C# 3.0 offrent différents degrés d'extensions de classes.

2.1 Cycles flous avec la méthode d'élévation à la puissance

Un des algorithmes utilisés pour le calcul des cycles est basé sur l'élévation à la puissance de la matrice d'adjacence. Le principe de cette approche est d'élever une DSM binaire à sa énième puissance pour indiquer quels éléments peuvent emprunter un chemin revenant sur eux-mêmes en n étapes et donc constituer un cycle (Yassine et al., 1999). Cet algorithme est limité car les cycles de même longueur ne sont pas différenciés.

Le formalisme utilisé pour la lecture des DSM est le suivant : les éléments en tête de colonne font appel aux éléments en tête de ligne. Par exemple, sur la Figure 1(b), A fait appel à B et à C, B fait appel à A, C fait appel à D et D fait appel à C.

Sur la Figure 1(a), les éléments A et B constituent un cycle direct et C et D en constituent un autre. Mais si nous élevons la DSM binaire (Figure 1(b)) au carré, une valeur différente de zéro apparaît dans la diagonale de chacun des éléments (Figure 1(c)). Ces valeurs signifient que chacun des éléments A, B, C et D est impliqué dans au moins un cycle direct mais ces valeurs ne montrent pas la répartition des éléments dans les cycles directs. Ainsi l'algorithme fusionne ces 4 éléments (Figure 1(d)), ce qui signifie que ces éléments apparaissent comme un seul cycle (Figure 1(e)) – notons que la zone grise représente les cycles. Ainsi, la matrice de partitionnement fournit une information erronée en indiquant un cycle unique (la zone grise dans la Figure 1(e)) alors que la matrice devrait montrer 2 cycles directs comme dans la Figure 1(f).



(a) Visualisation (b) Matrice binaire (c) Matrice binaire élevée au carré (d) DSM fusionnée

(e) DSM partitionnée avec la méthode de la matrice d'adjacence (f) DSM partitionnée idéale avec 2 cycles distincts

FIG. 1 – *Limite de la méthode de puissance de matrice d'adjacence*

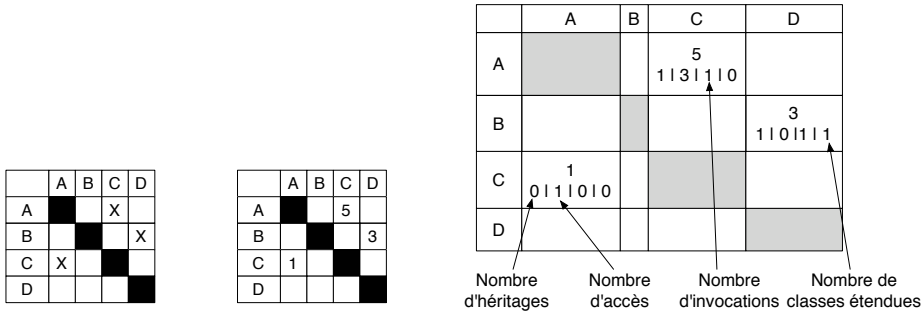
L'élévation à la puissance de la matrice d'adjacence ne permet pas de déterminer précisément les cycles différents. Cependant, nous utilisons cet algorithme combiné à une méthode de recherche de chemin afin d'identifier les différents cycles de même niveau.

2.2 Manque de précision de la visualisation

Une DSM traditionnelle offre un aperçu général mais sans information précise de la situation qu'elle décrit. Nous identifions deux sortes de faiblesses : le manque d'information sur les

causes des dépendances d'une part, et sur leur importance d'autre part.

Manque d'information sur les causes. Les dépendances peuvent avoir plusieurs causes : des accès aux classes, des extensions de classes (voir la Section 2.3), des relations d'héritage et des invocations de méthodes. Les coûts d'élimination d'un cycle varient en fonction du type de lien choisi pour suppression : changer une référence directe à une classe est souvent plus simple que de changer une relation d'héritage. C'est pourquoi il n'est pas suffisant d'indiquer les dépendances dans une DSM avec un simple marqueur *X* (Figure 2(a)) ou même avec un nombre représentant le nombre de dépendances qui existent (Figure 2(b)). Nous pensons qu'indiquer au moins un nombre pour chaque type de dépendance (Figure 2(c)) donne une indication plus précise et aide ainsi à une meilleure compréhension de la situation.



(a) DSM avec des marqueurs

(b) DSM avec des nombres

(c) DSM avec les types de dépendance

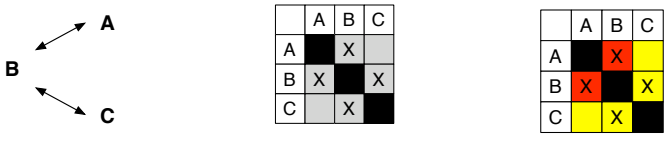
FIG. 2 – *Exemple de références dans une DSM*

Manque d'information sur les impacts. De plus, certaines informations importantes ne sont pas données et nous n'avons aucune idée des problèmes potentiels. Par exemple, le fait qu'un package ait 72 références sur un autre package (package MorphicExtra-Demo sur Morphic-Kernel présenté dans la Figure 12) est une information importante. Mais de telles références peuvent être faites par un grand nombre de classes ou bien un petit groupe et ces 72 références peuvent faire référence à un petit sous-ensemble ou à un grand nombre de classes. La même remarque peut être faite pour les méthodes. Ces informations supplémentaires sont pertinentes et importantes. Par exemple pour le package MorphicExtra-Demo, 6 classes et 45 méthodes sont concernées et elles appellent 2 classes et 40 méthodes du package Morphic-Kernel. En plus, avoir accès à ces informations sans avoir à regarder chaque classe permet de gagner du temps.

2.3 Manque de précision sur les cycles d'une entité

Les cycles sont vus dans le contexte du système complet. Il est difficile dans une DSM standard de comprendre le cycle (et pas son niveau) dans lequel un package donné est impliqué. En particulier, quand des cycles de même niveau sont fusionnés, nous obtenons des informations

imprécises. Par exemple, considérons un package A impliqué dans un cycle direct avec un package B. Ce package B est impliqué dans un cycle direct avec un package C (Figure 3(b)). Nous avons donc A dans un cycle qui inclut également C mais la longueur du cycle entre A et C n'est pas le même que la longueur du cycle entre le package A et le package B. Voir ces différences de longueurs de cycle (Figure 3(c)) est une information importante parce que les méthodes utilisées pour casser des cycles de différentes longueurs ne seront pas les mêmes. Dans la Figure 3(c) les cycles incluant l'entité A sont montrés : les cellules rouges montrent les cycles directs avec A, alors que les cellules jaunes montrent les cycles indirects.



(a) Visualisation (b) Cycles d'un système (c) Cycles d'une entité

FIG. 3 – *L'importance des cycles appliqués à une entité précise*

2.4 Extensions de classes non prises en compte

Une extension de classe est une méthode qui est définie dans un package différent du package où est définie la classe (Bergel et al., 2005). Les extensions de classes existent en Smalltalk, CLOS, Objective-C, Ruby et maintenant en C#. Elles offrent une manière adéquate de modifier incrémentalement les classes existantes quand la construction de sous-classes est inappropriée (Figure 4).

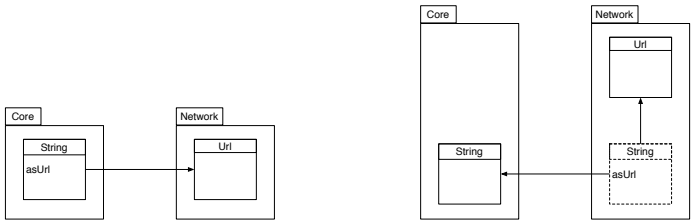
La construction de sous-classes peut être inappropriée lorsque l'on ne peut pas mettre à jour les clients d'une classe pour faire référence à une nouvelle sous-classe ou que le code source de la classe est inaccessible. Ainsi, l'extension de classe offre une bonne solution au dilemme qui apparaît lorsque l'on veut modifier ou étendre le comportement d'une classe existante sans modifier le code source de la classe en question.

Dans la Figure 4, plutôt que de créer une méthode dans le package *Core* auquel on n'a pas forcément accès, on étend la classe *String* dans le package *Network* avec la méthode asUrl. Le lien entre les deux packages est par conséquent inversé.

Cette dernière caractéristique est particulièrement importante dans l'analyse des cycles car elle offre la possibilité de supprimer un cycle en inversant la dépendance, sans pour autant devoir casser le lien qui existe entre les deux packages.

3 Une DSM enrichie

Notre approche définit une matrice enrichie (EDSM). Nous prenons en compte les limites décrites précédemment et ajoutons plusieurs fonctionnalités qui ne sont pas offertes par des logiciels de DSM industriels comme Lattix (Sangal et al., 2005) : (i) des informations enrichies dans les cellules (Section 3.1), (ii) une différenciation des cycles indépendants (Section 3.2), et (iii) le focus sur un package avec coloration des niveaux de cycle (Section 3.3).



(a) Dépendance sans extension de classe (b) Dépendance avec extension de classe

FIG. 4 – *Principe d'extension de classe*

L'outil est implémenté dans l'environnement de réingénierie *Moose*. Ainsi, l'outil profite d'une indépendance vis à vis des langages et travaille sur un modèle basé sur *FAMIX*.

3.1 Cellules enrichies

EDSM enrichit les informations contenues dans une cellule (Figure 5). Ces nouvelles informations contextuelles montrent les éléments suivants :

– *Force de la dépendance.*
Sur la première ligne, nous montrons le nombre de références du package source vers le package cible. Ce nombre nous donne la force du lien qui existe entre ces packages. C'est une information brute telle que les DSM la présentent.

– *Type de dépendance.*
La deuxième ligne montre les types de référence : Héritage (H), Accès (A), Invocation (I) et Extension de classe (E), et combien de références existent pour chaque type. Ces nombres nous donnent une information plus raffinée des relations entre les packages.

– *Source et diffusion de la dépendance.*
Enfin, nous montrons comment les références sont distribuées dans les deux packages source et destination. La troisième ligne montre le nombre de classes de chaque package. Cette information indique la taille des packages impliqués dans la dépendance. La quatrième ligne indique le nombre de classes et de méthodes qui sont actuellement référençantes et référencées pour le package faisant la référence et le package référencé. La dernière ligne exprime la même idée mais avec des pourcentages. Cette information est importante parce qu'elle montre deux choses : le taux de classes sources des références et le taux de diffusion de ses références dans le package cible. Nous pouvons donc obtenir une idée de la diffusion et des flux entre les deux packages. Par exemple, si le taux de classes d'un package source des dépendances est faible, nous orientons l'effort de réingénierie pour enlever ces dépendances sur la source, alors que si ce taux est élevé, nous orienterions plutôt l'effort sur le package cible.

Ces informations contextuelles enrichies permettent d'identifier plus facilement la situation en analysant rapidement la matrice. Il est difficile d'afficher l'ensemble de la matrice à l'écran en conservant la lisibilité du texte, ainsi nous recourons au zoom. En zoom arrière, la vue permet toujours d'avoir toute la matrice affichée à l'écran.

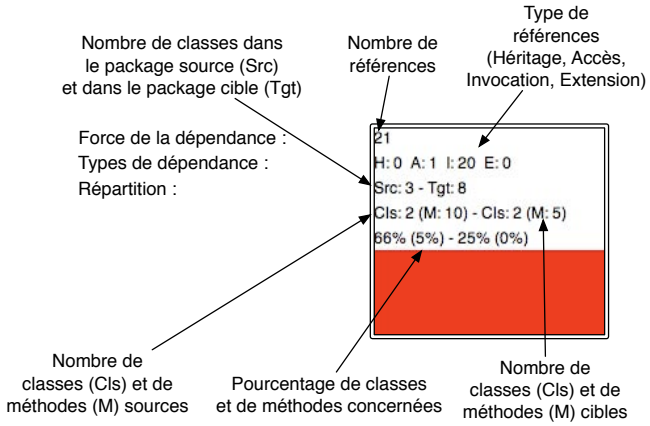


FIG. 5 – *Cellule enrichie de DSM*

Exemple. La Figure 5 montre un exemple de cellule enrichie. Il y a 21 liens dont 1 accès (A) et 20 invocations (I), il y a 2 classes (10 méthodes) sources et 2 classes (5 méthodes) cibles. Cela représente 66% des classes (5% des méthodes) du package source et 25% des classes (0% des méthodes : le chiffre 0 apparaît en raison d'un arrondi, cela signifie que le package cible a un nombre important de méthodes) du package cible.

3.2 Meilleure détection des cycles

Dans le contexte des DSM, une alternative à l'algorithme d'élévation à la puissance de la matrice d'adjacence est un algorithme de recherche de chemin (Gebala et al., 1991). Cet algorithme nous permet de détecter séparément tous les cycles indépendants. Ainsi, comme exprimé dans la Section 2.1, deux packages, A et B, peuvent être en cycle direct et les packages C et D dans un autre cycle direct. Avec l'algorithme de recherche de chemin, ces deux cycles sont clairement identifiés. A partir de ce constat, notre approche améliore la matrice traditionnelle en proposant : la distinction des cycles , l'identification d'emboîtement des cycles , et des indications pour aider à la réingénierie des cycles.

Distinctions des cycles. Notre approche distingue les cycles indépendants. Elle est basée sur un algorithme de recherche de chemin. Avec cette méthode, deux cycles indépendants sont détectés séparément et peuvent être ainsi isolés l'un de l'autre dans la DSM (Figure 6).

Emboîtement des cycles. Nous avons ajouté des informations colorimétriques dans les cellules de la DSM pour donner des informations à propos des cycles. Ainsi, comme montré dans la Figure 7(a), après un partitionnement, les cellules de la DSM impliquées dans un cycle ont une couleur rouge ou jaune. La couleur rouge signifie que les deux packages concernés sont impliqués dans un cycle direct alors que la couleur jaune signifie qu'ils sont impliqués dans un cycle indirect. La surface bleu pâle entoure l'ensemble des packages qui sont en cycle (comme

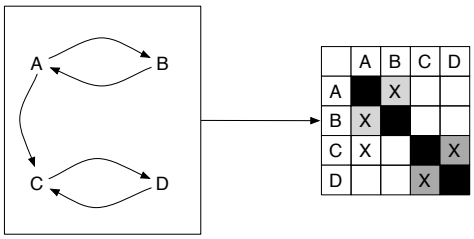
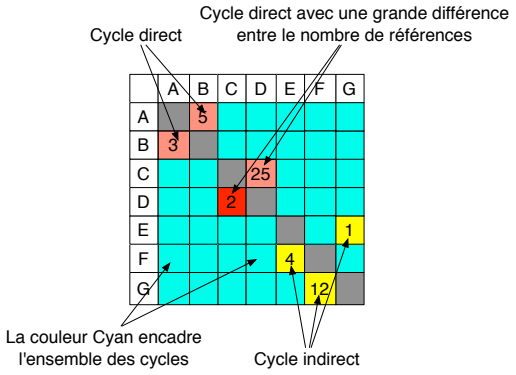


FIG. 6 – *Distinction des cycles indépendants en teinte de gris différente*

montré dans la Figure 8), alors que les lignes et les colonnes blanches représentent les packages sans cycle.

Indication pour la réingénierie des cycles. Dans le cas de cycles directs, s'il y a une grande différence entre le nombre de références de chacun des deux packages impliqués (nous utilisons un rapport de 3 comme seuil), la cellule qui contient le moins de références a une couleur rouge vif (Figure 7(a)). Cette information permet à l'utilisateur de centrer son attention sur les références qui peuvent être résolues a priori. Comme montré dans la Figure 8, une telle indication est vraiment utile pour repérer les packages présentant des cycles.



(a) Représentation de cycles

FIG. 7 – *Coloration des cycles directs et indirects et indication de problèmes.*

La Figure 7(a) illustre comment interpréter les cellules colorées. Elle montre qu'il y a deux cycles directs : entre A et B et entre C et D, car les cellules sont rouges. Les cycles sont distincts car il n'y a pas de cellules rouges au croisement des lignes A ou B et des colonnes C ou D. Nous voyons que D peut être la source du problème pour le cycle avec C car il fait deux références sur C alors que C fait 25 références sur D. Puis il y a un cycle indirect entre E, F et G car il y a des cellules jaunes entre ces trois packages.

Exemple. Nous avons appliqué notre approche au framework Morphic de Squeak. Notons que Morphic n'a jamais été créé de façon modulaire. Par conséquent il présente un nombre important de dépendances cycliques. Nous observons dans la Figure 8 les packages impliqués dans des cycles (représenté par l'espace bleu pâle). Nous détaillons cet exemple plus loin. Notez que nous pouvons concentrer notre attention sur les cellules rouge vif qui indiquent les dépendances pouvant probablement disparaître pour éliminer les cycles directs.

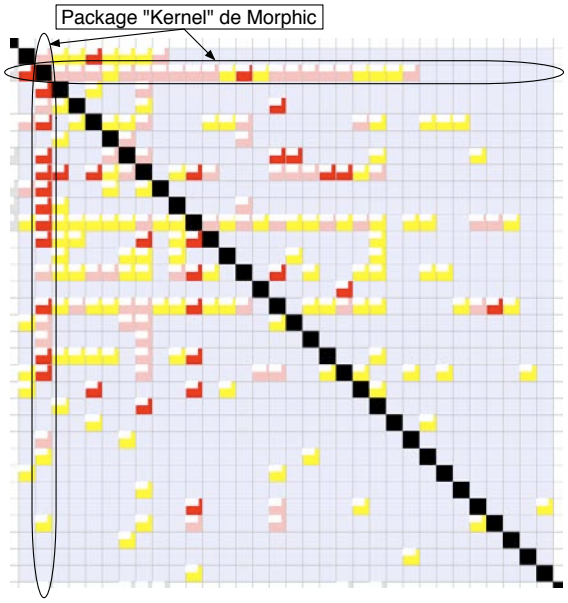


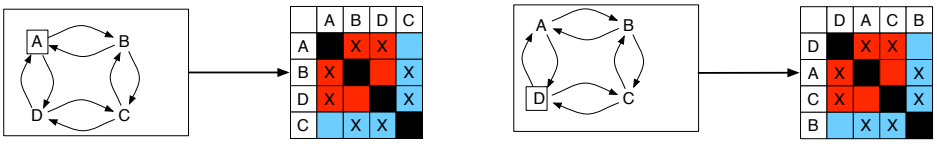
FIG. 8 – *Exemple de détection de cycle.*

3.3 Visualisation focalisée sur un package précis

Nous avons vu dans la section précédente que notre approche isole les cycles indépendants et leur ajoute des informations grâce à une coloration des cellules pour distinguer les cycles directs et indirects. Cependant, s'il est facile de voir les différents cycles qui existent à un niveau donné quand il y a peu de packages, cela devient plus difficile avec un plus grand nombre de packages. Ainsi, nous ne pouvons pas facilement connaître la longueur d'un cycle entre deux packages. En effet, ce problème apparaît particulièrement lorsqu'un élément appartient à plusieurs cycles qui n'ont pas la même longueur. Il est difficile de montrer la longueur de tous les cycles auxquels appartient cet élément. De plus, lors d'une phase de remodularisation, l'ingénieur doit souvent porter son attention sur un package bien particulier.

Pour résoudre ce problème, nous avons ajouté la notion de package cible. Ainsi, la longueur d'un cycle entre un élément et un autre est relative à l'élément ciblé. Par exemple, dans la Figure 9, nous pouvons voir que l'élément ciblé est A, les éléments B et D sont invoqués dans

un cycle direct avec A, et l'élément C est en cycle indirect avec A (Figure 9(a)). Mais si nous prenons comme élément cible le package D, les éléments A et C sont impliqués dans un cycle direct avec D et l'élément B qui est en cycle indirect avec D (Figure 9(b)).



(a) Situation avec l'élément ciblé A

(b) Situation avec l'élément ciblé D

FIG. 9 – *Visualisation des différents cycles relatifs aux éléments ciblés*

3.4 Niveaux de cycle

Pour aider la visualisation des différents cycles impliquant le package cible, nous introduisons une couleur différente pour chaque niveau de cycle. En effet, comme dans la Figure 10, les éléments impliqués dans le premier niveau de cycle avec l'élément ciblé (c-à-d les éléments qui sont en cycle avec l'élément ciblé et dont la longueur du cycle est le plus court) sont de couleur rouge. Ce processus est répété pour chaque élément qui est en cycle avec l'élément ciblé : les éléments qui appartiennent au même niveau de cycle sont de la même couleur. Grâce à ces informations colorimétriques, nous pouvons d'une part facilement voir si un élément est dans un cycle avec le package cible, et d'autre part nous pouvons compter le nombre d'éléments dans chaque niveau de cycle.

4 Expériences

Pour expérimenter et valider notre approche, nous l'avons appliquée à deux études de cas non triviales : l'environnement de réingénierie Moose (10 packages - 180 classes), et Morphic (61 packages - 346 classes) (voir Figure 8). Pour l'application Moose, nous avons découvert un certain nombre de cycles que les développeurs de Moose ont corrigé en conséquent.

Nous rapportons ici un exemple tiré de l'analyse de Morphic. Supprimer les cycles dans Morphic est un travail conséquent. Parfois des mécanismes d'enregistrements ou des plugins manquent et empêchent d'éliminer certaines dépendances. Morphic est clairement une application dont le packaging est à revoir car elle contient beaucoup de cycles (49 cycles directs).

Prenons le package Morphic-Kernel. Ce package, comme son nom l'indique, est le noyau de Morphic. Il ne devrait normalement avoir aucun cycle avec d'autres packages. Or, comme le montre la Figure 8, Morphic-Kernel est en cycle direct avec 17 packages (voir Figure 10). Avant d'étudier et de corriger d'autres packages, il nous est apparu intéressant d'examiner celui-ci. Par exemple, la Figure 12 montre un cycle entre Morphic-Kernel et MorphicExtras-Demo (appelé respectivement Kernel et Demo par la suite). La Figure 12 est faite de la façon suivante : on observe en fond une partie de la DSM appliquée sur Morphic. Les cases représentant les liens entre les packages Kernel et Demo sont entourées d'un cercle et agrandies. Les

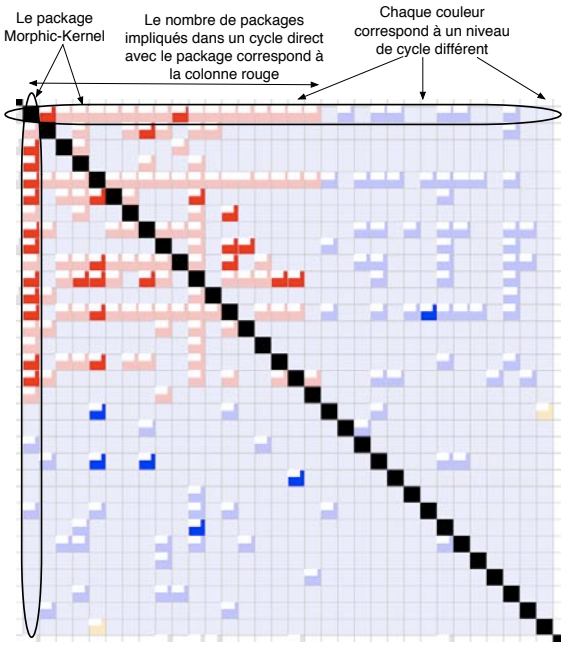


FIG. 10 – *Visualisation de différents niveaux de cycles pour le package Morphic-Kernel*

liens ayant pour origine Kernel sont représentés par la case rouge vif et les liens ayant pour origine Demo sont représenté par la case rouge claire. La structure du contenu des cases est expliquée dans une section précédente. Par exemple, pour la case rouge claire, il y a 72 liens dont 3 héritage (H) et 69 invocations (I), il y a 6 classes (45 méthodes) sources et 2 classes (40 méthodes) cibles. Cela représente 27% des classes (34% des méthodes) de Demo et 25% des classes (3% des méthodes) de Kernel. Les boîtes jaunes sont des popups apparaissant lorsqu'on survole la case, elles représentent la même information en version détaillée.

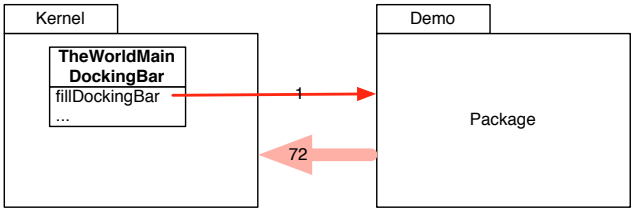


FIG. 11 – *exemple de lien avec le package Kernel*

Ces informations nous aident à comprendre la structure de Morphic sans voir le code. En effet, sans connaître le contenu du package, on peut penser par son nom que MorphicExtras-Demo est un supplément à Morphic et qu'en plus il correspond à un package de démonstration. Pourtant, il est en cycle avec Kernel. En analysant notre DSM, on observe par les différences de couleurs rouge que Kernel fait moins accès à Demo que l'inverse. Le contenu de la case de Kernel indique un seul accès à Demo. En survolant la case, une popup s'affiche avec des informations complémentaires : la classe TheWorldMainDockingBar est la classe posant problème. Plus précisément, c'est la méthode fillDockingBar : qui fait l'appel au package Demo. Nous pouvons faire la même analyse avec le package Demo, mais elle n'est pas nécessaire dans ce cas là : Kernel n'est pas censée faire un appel à ce package.

Cette analyse nous place devant trois possibilités : soit la référence est faite par du code mort et on enlève la référence, soit on déplace la classe concernée dans Demo en vérifiant que d'autres packages n'y font pas référence, soit on déplace la classe dans un package plus approprié.

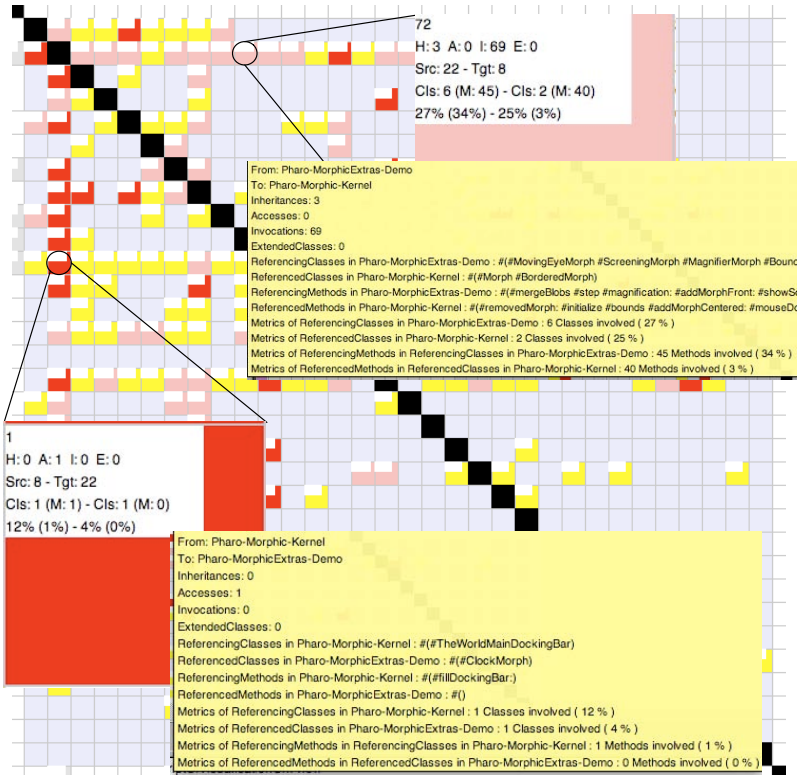


FIG. 12 – *analyse du package Kernel*

L'analyse de la DSM nous montre que ce n'est pas du code mort. La classe TheWorld-

MainDockingBar utilise d'autres packages (par exemple MorphicExtra-Flaps), eux aussi en cycle avec Kernel. Il n'est donc pas intéressant de déplacer cette classe dans Demo, mais cette analyse nous montre qu'elle n'a pas sa place dans Kernel.

Ainsi, notre outil de DSM enrichie permet d'analyser rapidement et efficacement les cycles entre les packages sans regarder le code source et sans vraiment connaître l'application.

5 Limites de l'EDSM

Bien qu'améliorée, l'EDSM présente plusieurs limites pour lesquelles nous souhaitons trouver des solutions afin de rendre l'outil plus efficace dans la remodularisation.

- La première est la difficulté de localiser les cases symétriques. Dans la Figure 12 par exemple, il est difficile d'identifier les deux cases correspondant à un cycle lorsque ces cases sont éloignées de la diagonale. Il est envisagé dans les travaux futurs d'offrir un affichage supplémentaire des cases deux à deux.
- La deuxième est l'affichage des informations dans les cases. L'affichage textuel n'est pas un vecteur idéal pour la transmission d'information. Les valeurs 10% et 90% s'affiche de la même manière actuellement, alors que leur importances sont différentes. Il conviendrait de les différencier. Nous envisageons donc par la suite d'étudier une présentation graphique des informations textuelles.
- La troisième limite est le manque d'informations sémantiques. Cette limite n'est pas spécifique à notre approche mais commune aux approches de visulisations d'information structurelle. Pour l'instant, les informations dans les cases n'utilisent que des informations purement structurelles comme l'héritage et les références. Or dans une phase de réingénierie nous avons besoin de comprendre les fonctionalités, informations plus sémantiques que structurelles. Nous aimerions pouvoir annoter les dépendances avec ce type d'informations, qui pourrait être donné par le réingénieur.

6 Conclusion

Ce papier propose une amélioration des matrices de dépendance (DSM - Dependency Structure Matrix), une approche qui identifie les dépendances entre les packages logiciels. Une DSM organise les références entre les packages dans une table matricielle. Plusieurs algorithmes existent déjà pour réordonner cette matrice et ainsi révéler l'architecture logicielle.

Après avoir dressé une liste de limites des DSM traditionnelles , nous avons proposé l'amélioration de l'information contenue dans une DSM et l'amélioration de la visualisation des DSM. D'une part, le contenu des cellules est enrichi avec le type et la diffusion des références ainsi que le nombre de classes invoquées. D'autre part, plusieurs couleurs sont utilisées pour distinguer les cycles directs et indirects.

Grâce à ces améliorations, la compréhension et l'analyse d'applications logicielles deviennent plus facile et ainsi plus rapide. Des travaux sont en cours pour améliorer encore la visualisation des informations au sein d'une case.

Egalement, dans des travaux futurs, il serait très utile de pouvoir voir directement dans la matrice les conséquences des changements appliqués à l'application sans devoir modifier le

code source. Ainsi, la matrice deviendrait une interface pour l'exécution de restructurations (refactorings) simples comme le déplacement d'une classe vers un autre package.

Remerciements. Le travail présenté a été développé dans le cadre du projet ANR COOK : Réarchitecturisation des applications industrielles objets (JC05 42872).

Références

Bergel, A., S. Ducasse, et O. Nierstrasz (2005). Classbox/J : Controlling the scope of change in Java. In *Proceedings of 20th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05)*, New York, NY, USA, pp. 177–189. ACM Press.

Demeyer, S., S. Ducasse, et O. Nierstrasz (2002). *Object-Oriented Reengineering Patterns*. Morgan Kaufmann.

Dong, X. et M. Godfrey (2007). System-level usage dependency analysis of object-oriented systems. In *ICSM 2007 : IEEE International Conference on Software Maintenance*, pp. 375–384.

Ducasse, S., T. Gîrba, et A. Kuhn (2006). Distribution map. In *Proceedings of 22nd IEEE International Conference on Software Maintenance (ICSM '06)*, Los Alamitos CA, pp. 203–212. IEEE Computer Society.

Ducasse, S., D. Pollet, M. Suen, H. Abdeen, et I. Alloui (2007). Package surface blueprints : Visually supporting the understanding of package relationships. In *ICSM '07 : Proceedings of the IEEE International Conference on Software Maintenance*, pp. 94–103.

Gebala, D., S. Eppinger, et M. Cambridge (1991). Methods for analyzing design procedures. *Design Theory and Methodology*.

Langelier, G., H. Sahraoui, et P. Poulin (2005). Visualization-based analysis of quality for large-scale software systems. In *ASE '05 : Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, New York, NY, USA, pp. 214–223. ACM.

Lopes, A. et J. L. Fiadeiro (2005). Context-awareness in software architectures. In *Proceeding of the 2nd European Workshop on Software Architecture (EWSA)*, Volume 3527 of *Lecture Notes in Computer Science*, pp. 146–161. Springer.

MacCormack, A., J. Rusnak, et C. Y. Baldwin (2006). Exploring the structure of complex software designs : An empirical study of open source and proprietary code. *Management Science 52*(7), 1015–1030.

Sangal, N., E. Jordan, V. Sinha, et D. Jackson (2005). Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pp. 167–176.

Steward, D. (1981). The design structure matrix : A method for managing the design of complex systems. *IEEE Transactions on Engineering Management 28*(3), 71–74.

Sullivan, K. J., W. G. Griswold, Y. Cai, et B. Hallen (2001). The structure and value of modularity in software design. In *ESEC/FSE 2001*.

Yassine, A., D. Falkenburg, et K. Chelst (1999). Engineering design management : an information structure approach. *International Journal of Production Research 37*(13), 2957–2975.

Summary

Dependency Structure Matrix (DSM), an approach developed in the context of process optimization, has been successfully applied to identify software dependencies among packages and subsystems. A number of algorithms help organizing the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies between subsystems. However, the existing DSM implementations often miss important information in their visualization to fully support a reengineering effort. For example, they do not clearly qualify or quantify problematic relationship. It is difficult to get a package centric point of view. Independent cycles are often merged. In this paper we enhanced DSM with enriched cell by showing

contextual information information about (i) the kinds of references made (inheritance, class accesses..), (ii) the proportion of entities referencing, (iii) the proportion of entities referenced. We distinguish independent cycles and stress cycles using coloring information.This work has been implemented on top of the *Moose* open-source reengineering environment and the Mondrian visualization framework. It has been applied to non-trivial case studies such as the *Morphic UI* framework available in open-source Smalltalk *Squeak* and *Pharo*. Results have been implemented in the *Pharo* programming environment.

Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique

Floréal Morandat[1], Roland Ducournau[1], Jean Privat[2]

[1] LIRMM – CNRS et Université Montpellier II
161 rue Ada Montpellier – 34392 Cedex 5 France
{morandat,ducour}@lirmm.fr,
[2] Université du Québec à Montréal
privat.jean@uqam.ca

Résumé La programmation par objets présente une apparente incompatibilité entre trois termes : l'héritage multiple, l'efficacité et l'hypothèse du monde ouvert — en particulier, le chargement dynamique. Cet article présente des résultats d'expérimentations exhaustives comparant l'efficacité de différentes *techniques d'implémentation* (coloration, BTD, hachage parfait, ...) dans le contexte de différents *schémas de compilation* (de la compilation séparée avec chargement dynamique à la compilation purement globale). Les tests sont effectués avec et sur le compilateur du langage Prm. Ils confirment pour l'essentiel les résultats théoriques antérieurs. Les schémas d'optimisation globale démontrent un gain significatif par rapport à la coloration qui fait fonction de référence, tandis que le chargement dynamique rend réel le surcoût de l'héritage multiple. Enfin, ces tests confirment l'intérêt du hachage parfait pour les interfaces de Java.

1 Introduction

L'hypothèse du monde ouvert (OWA pour *Open World Assumption*) représente certainement le contexte le plus favorable pour obtenir la *réutilisabilité* prônée par le génie logiciel. Une classe doit pouvoir être conçue, programmée, compilée et implémentée indépendamment de ses usages futurs et en particulier de ses sous-classes. C'est ce qui permet d'assurer la compilation séparée et le chargement dynamique.

Cependant l'héritage multiple — ou ses variantes comme le sous-typage multiple d'interfaces à la Java— s'est révélé difficile à implémenter sous l'hypothèse du monde ouvert. En typage statique, seule *l'hypothèse du monde clos* (CWA) permet d'obtenir la même efficacité qu'en héritage simple, c'est-à-dire avec une implémentation en temps constant nécessitant un espace linéaire dans la taille de la relation de spécialisation. Les deux langages les plus utilisés actuellement illustrent bien ce point. C++, avec le mot-clef `virtual` pour l'héritage, procure une implémentation pleinement réutilisable, en temps constant — bien que pleine d'ajustements de pointeurs — mais qui nécessite un espace cubique dans le nombre de classes (dans le pire des cas). En Java, où l'héritage multiple est restreint aux interfaces, il n'existe pas à notre connaissance d'implémentation des interfaces en temps constant [Alpern *et al.*, 2001a; Ducournau, 2008]. En typage dynamique, même l'héritage simple pose des problèmes et nous nous restreindrons ici au typage statique.

Dans cet article, nous distinguons l'implémentation qui concerne la *représentation* des objets et la compilation qui calcule cette représentation. Au total, l'exécution des programmes à objets met en jeu une ou plusieurs *techniques d'implémentation* dans le contexte de différents *schémas de compilation*. L'implémentation est concernée par les trois mécanismes de base des langages objets — accès aux attributs, invocation de méthode et test de sous-typage. Le schéma de compilation constitue la chaîne de production de l'exécutable : génération de code, édition de liens, chargement.

L'objectif de ce travail est d'évaluer de manière réaliste et objective l'impact effectif des schémas de compilation et des techniques d'implémentation sur l'efficacité d'un programme objet significatif en comparant deux à deux les techniques ou les schémas, *toutes choses égales par ailleurs*. Ces expérimentations sont réalisées sur le compilateur de Prm, un langage objet qui supporte l'héritage multiple et qui propose une notion de module et de raffinement de classe [Privat et Ducournau,

2005b]. Grâce à cela, PRM$_\mathcal{C}$ son compilateur autogène (écrit en PRM) est un programme modulaire et remplacer une technique par une autre peut se faire à moindre coût.

Cet article étend les premiers résultats de [Morandat *et al.*, 2009] avec les nouvelles expérimentations ainsi qu'une partie des résultats présentés dans [Ducournau *et al.*, 2009]. Nous commencerons par décrire des techniques d'implémentation des mécanismes objet, puis des schémas de compilation — de la compilation séparée qui permet de compiler le code modulairement à la compilation globale qui permet d'effectuer de nombreuses optimisations, ainsi que certains schémas intermédiaires qui ont été peu étudiés. Nous présentons ensuite une série de tests réalisés sur PRM$_\mathcal{C}$, afin de mesurer l'impact réel des différents schémas et techniques sur de vrais programmes — en l'occurrence PRM$_\mathcal{C}$ lui-même. Enfin nous commenterons ces résultats avant de parler des travaux connexes, puis nous conclurons en détaillant les perspectives de ce travail.

2 Implémentation et compilation

Cette section présente les différentes techniques d'implémentation et les schémas de compilation que nous considérons dans cet article. Le lecteur intéressé est renvoyé à [Ducournau,] pour une synthèse plus générale.

2.1 Techniques d'implémentation

Nous présentons d'abord la technique de référence de l'héritage simple puis les différentes techniques d'implémentation de l'héritage multiple que nous considérons ici. Nous parlerons principalement de l'appel de méthode, l'implémentation des deux autres mécanismes pouvant généralement se déduire de celui-ci.

Technique de base : héritage simple et typage statique Dans le cas de l'héritage simple, le graphe de spécialisation d'un programme est une arborescence. Il n'existe donc qu'un chemin reliant une classe (sommet dans le graphe) à la classe racine. En typage statique, cette propriété donne lieu à une implémentation simple et efficace de l'héritage simple. Il suffit pour cela de concaténer les méthodes introduites par chacune des classes dans l'ordre de spécialisation pour construire les tables de méthodes. L'opération similaire appliquée aux attributs permet de représenter les instances (Figure 1-a). Le test de sous-typage de Cohen [1991] s'intègre dans la table de méthodes suivant le même principe.

Nous considérons cette implémentation comme celle de référence car elle rajoute le minimum d'information nécessaire à l'exécution d'un programme dans les diverses tables. De plus, elle vérifie deux invariants : *de position*, chaque méthode (resp. attribut) a une position dans la table des méthodes (resp. l'objet) invariante par spécialisation, donc indépendante du type dynamique du receveur ; *de référence*, la référence à un objet est indépendante du type statique de la référence.

En héritage simple, il suffit de connaître la super-classe pour calculer l'implémentation d'une classe (OWA). La taille totale des tables est *linéaire* dans le cardinal de la relation de spécialisation, donc, dans le pire des cas, *quadratique* dans le nombre de classes.

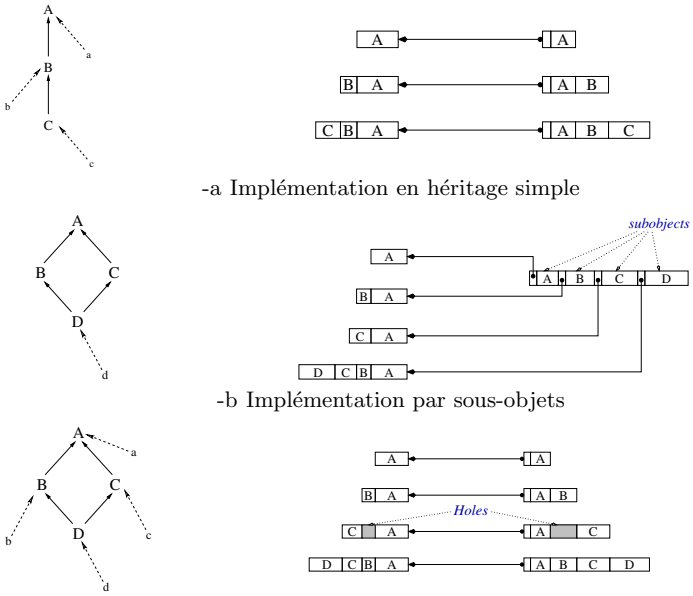
Sous-objets (SO) Dans le cas de l'héritage multiple, ces invariants sont trop forts car deux classes incomparables, peuvent avoir une sous-classe commune (par exemple B et C dans la figure 1-b). La technique de base ne peut donc plus être utilisée.

L'implémentation de C++ détaillée dans Ellis et Stroustrup [1990] repose entièrement sur les types statiques des références. La représentation d'un objet est faite par concaténation des *sous-objets* de toutes les super-classes de sa classe (elle-même comprise). Pour une classe donnée, un sous-objet est un pointeur sur sa table de méthodes suivi des attributs *introduits* par cette classe. Chaque table de méthodes contient l'ensemble des méthodes *connues* par le type statique du sous-objet. Une référence à un objet pointe sur le sous-objet correspondant au type statique de la référence et toutes les opérations polymorphes sur un objet nécessitent un ajustement de

pointeur. Cet ajustement est dépendant du type dynamique de l'instance pointée, donc la table des méthodes doit contenir aussi les décalages à utiliser. Dans le pire des cas, la taille totale des tables est ainsi *cubique* dans le nombre de classes. Cette implémentation reste compatible avec le chargement dynamique (OWA).

En C++, cette implémentation suppose que le mot-clef `virtual` annote chaque super-classe. L'absence de `virtual` dans les clauses d'héritage se traduit par une implémentation simplifiée qui entraîne un risque d'héritage répété. La majorité des programmes utilisent peu ce mot-clef et ils ne souffrent donc pas du surcoût entraîné par l'héritage multiple et le chargement dynamique, mais c'est au prix d'une réutilisabilité limitée (CWA).

Coloration (MC/AC) La coloration a été initialement proposée par Dixon *et al.* [1989], Pugh et Weddell [1990] et Vitek *et al.* [1997] pour chacun des trois mécanismes de base. Ducournau [2006] montre l'identité des trois techniques et fait la synthèse de l'approche. L'idée principale est de maintenir en héritage multiple les invariants de l'héritage simple, mais sous l'hypothèse du monde clos. Comme les éléments doivent avoir une position invariante par spécialisation, il faut laisser certains trous dans les tables, par exemple dans celle de C (Figure 1-c). Pour obtenir un résultat efficace, il faut minimiser la taille des tables (ou le nombre de trous).



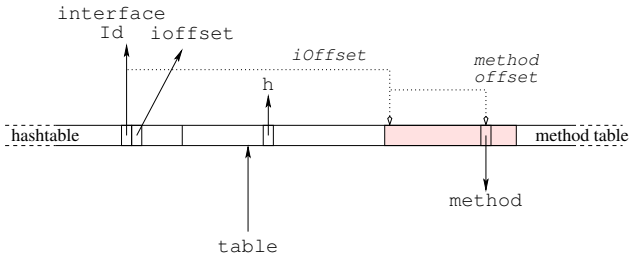
A chaque exemple de hiérarchie de classes $(A, B, C, \ldots)$ est associée l'implémentation correspondante d'instances $(a, b, c, \ldots)$. Les objets sont à droite, les tables de méthodes à gauche, inversées pour éviter les croisements. Dans les tables, les étiquettes de classe désignent le groupe de méthodes ou d'attributs *introduits* par la classe.

Fig. 1. Différentes techniques d'implémentation

La technique de la coloration peut être utilisée pour les trois mécanismes objet et le code généré pour chacun de ces mécanismes ne dépend que de la couleur de l'entité accédé. Seul le calcul des couleurs nécessite de connaître la hiérarchie de classes (CWA). Dans la suite, MC désigne l'application de la coloration aux méthodes et au test de sous-typage et AC l'application à la représentation des instances. D'autres techniques de compression de tables existent, par exemple *row displacement* [Driesen, 2001], mais elles ne s'appliquent pas à la représentation des objets.

Simulation des accesseurs (AS) La simulation des accesseurs, proposée par Myers [1995] et Ducournau [], permet de réduire l'accès aux attributs à l'envoi de message, en rajoutant une indirection par la table des méthodes. Les attributs sont groupés par classe d'introduction et la position de chaque groupe est incluse dans la table des méthodes comme si cela en était une. La simulation des accesseurs s'applique à n'importe quelle technique d'implémentation des méthodes — elle suppose juste qu'un attribut soit introduit par une classe unique, comme en typage statique. Nous la considérons ici couplée à la coloration de méthodes comme alternative à la coloration d'attributs qui peut engendrer un nombre de trous trop important dans certaines classes, voir [Ducournau, 2006].

Arbres binaires d'envoi de messages (BTD) Toutes les techniques présentées jusqu'à présent utilisent des tables pour invoquer les méthodes. L'idée des *binary tree dispatch* (BTD), proposée par Zendra *et al.* [1997], est de remplacer les tables par une série de tests d'égalité de types qui déterminent la méthode à appeler. L'appel de méthode lui-même est implémenté comme un arbre binaire équilibré de comparaison de types dont les feuilles sont les appels statiques de méthodes. En pratique, grâce au cache d'instructions et aux prédictions de branchements des processeurs modernes, si la profondeur de l'arbre n'excède pas trois niveaux — donc avec huit feuilles au maximum — le nombre moyen de cycles pris pour ces comparaisons est inférieur au temps d'accès par l'indirection d'une table de méthodes. On notera par la suite BTD_i un BTD de profondeur i. Les BTD_0 correspondent aux appels statiques.



L'objet pointe (par `table`) à l'indice 0 de la table de méthodes. Les indices positifs contiennent les adresses des méthodes dans l'implémentation de l'héritage simple, où les méthodes sont groupées par interface d'introduction. Les indices négatifs contiennent la table de hachage dont le paramètre `h` est aussi dans la table. Une entrée de la table est formée de l'identifiant de l'interface (`interfaceId`) pour le test de sous-typage et de la position relative (`iOffset`) du groupe de méthodes introduites par l'interface.

Fig. 2. Hachage parfait en Java

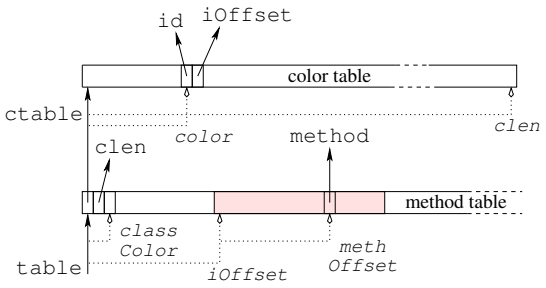
Hachage parfait (PH) Le hachage parfait est une optimisation en temps constant des tables de hachage pour des ensembles immutables [Czech *et al.*, 1997]. Ducournau [2008] propose de l'utiliser pour le test de sous-typage et l'envoi de message, dans le cas particulier des interfaces en Java (Figure 2). Cette technique nécessite une indirection supplémentaire avant l'appel de la méthode. A notre connaissance, cette technique est la seule alternative à l'implémentation par sous-objets de C++ qui soit en temps constant et incrémentale (OWA). Cependant sa constante de temps pour l'appel de méthode est à peu près le double de celle de l'héritage simple. Deux fonctions de hachage ont été étudiées : modulo, le reste de la division entière (notée `mod`) et la conjonction bit-à-bit (notée `and`). Les deux ne nécessitent qu'une instruction machine mais, alors que `and` prend un seul cycle d'horloge, la latence de la division entière peut attendre 20 ou 25 cycles, en particulier sur les processeurs comme le Pentium où la division entière est prise en charge par l'unité de calcul flottant. Par ailleurs, les expérimentations effectuées sur PH-`and` et PH-`mod` montre que le premier

nécessite des tables beaucoup plus grandes que le second. Le hachage parfait semble donc conduire à un choix espace-temps difficile.

Le hachage parfait peut être utilisé pour l'accès aux attributs grâce à la simulation des accesseurs, pour cela la table de hachage doit contenir des entrés triples[3].

Coloration incrémentale (IC) Une version incrémentale de la coloration a été proposée par Palacz et Vitek [2003] pour implémenter le test de sous-typage en Java. Pour les besoins de la comparaison, une application à l'invocation de méthode basée sur les mêmes principes que l'appel de méthode du hachage parfait a été proposée dans [Ducournau, 2008].

La coloration nécessitant l'hypothèse du monde clos (CWA), la coloration incrémentale peut entraîner certains recalculs durant le chargement. De ce fait, la structure de donnée utilisée ajoute une indirection dans une zone mémoire différente, ce qui a pour conséquence d'augmenter le risque de défaut de cache (Figure 3). Le lecteur intéressé est renvoyé à [Ducournau, 2008] pour les détails de l'implémentation.



L'implémentation ressemble à celle du hachage parfait sauf que la position d'une interface dépend d'un `load` au lieu d'être le résultat d'une fonction de hachage. La position d'une couleur peut être stockés dans la table des méthodes de la classe cible ou dans une table séparée. Dans tous les cas, le `load` nécessite un accès à une région mémoire différente et peut conduire à trois défauts de cache supplémentaire
De plus, comme la table des couleurs doit pouvoir être recalculée, elle doit nécessairement être disjointe de la table des méthodes et nécessite donc une indirection supplémentaire ainsi qu'un test de borne sur la valeur `clen` — et donc un `load` supplémentaire.

Fig. 3. Coloration incrémentale pour les interfaces de Java

Caches et recherche (CA) Une technique d'implémentation souvent appliquée dans les langages à typage dynamique ou aux interfaces de Java consiste à coupler une technique d'implémentation (qui est supposée, ici, être plutôt naïve et inefficace) à un cache qui mémoïse le résultat de la dernière recherche [Alpern *et al.*, 2001a; Alpern *et al.*, 2001b; Click et Rose, 2002].

Par exemple, pour les interfaces de Java, en utilisant le hachage parfait (PH) ou la coloration incrémentale (IC) comme technique sous-jacente, la table des méthodes pourrait *cacher* l'identifiant de l'interface ainsi que l'offset du groupe de méthode du dernier accès réussi. La séquence de code pour l'invocation de méthode et le test de sous-typage est relativement plus longue que l'implémentation sous-jacente. De plus, le pire des cas est plutôt inefficace car il rajoute la gestion du cache au temps nécessaire pour l'appel sous-jacent. Quant au meilleur des cas, il est légèrement plus efficace que PH-`and` pour l'invocation de méthode et identique au test de Cohen [1991] pour le test de sous-typage. Nous pourrions attendre du cache qu'il dégrade PH-`and` mais qu'il améliore PH-`mod`.

Ce cache peut être utilisé avec toutes les techniques d'implémentation basées sur des tables et pour chacun des trois mécanismes, au prix de *cacher* les trois données qui sont : l'identifiant de

[3] Pour respecter l'alignement, il vaut mieux faire des entrés quadruples.

la classe, l'offset du groupe de méthodes ainsi que celui du groupe d'attributs introduits par cette classe. De plus, le cache peut être commun aux trois mécanismes ou dédié à chacun d'eux. A l'instar de la coloration incrémentale et à l'inverse de toutes les autres techniques, le cache nécessite que les tables de méthodes soient accessibles en écriture — donc allouées dans un segment mémoire de données.

L'amélioration proposée par le cache est une question de statistiques. Celles présentées dans [Palacz et Vitek, 2003] montrent qu'en fonction des benchmarks les défauts de cache varient de 0.1% à plus de 50%. Le taux de réussite du cache peut être amélioré en utilisant plusieurs caches, ce qui augmente encore la taille du code par rapport à l'implémentation sous-jacente. Les classes (ou interfaces) sont statiquement partitionnées en n ensembles — par exemple en hachant leur nom — et la structure du cache est répliquée n fois dans la table des méthodes. Quand n augmente, le taux de défaut de cache doit asymptotiquement tendre vers 0 — néanmoins, le meilleur et le pire des cas restent inchangés. De plus, avec cette approche, les tables de la structure sous-jacente peuvent se restreindre à contenir les identifiants de classe (ou d'interface) pour lesquelles il demeure une collision dans leur propre cache.

Dans nos tests, nous considérerons aussi le hachage parfait lorsqu'il est couplé à un cache.

2.2 Schémas de compilation

Les schémas de compilation constituent la chaîne de production d'un exécutable. Elle est composée de plusieurs phases distinctes qui incluent de façon non limitative la compilation et le chargement des unités de code ainsi que l'édition de liens. La compilation nécessite au minimum le code source de l'unité considérée et des informations sur toutes les classes qu'elle utilise (superclasses par exemple). Ces informations sont contenues dans le *modèle externe*, qui est l'équivalent des fichiers d'en-têtes (.h) en C et peut être extrait automatiquement comme en Java. La distinction entre tous ces schémas n'est pas toujours très tranchée. En particulier les compilateurs adaptatifs [Arnold *et al.*, 2005] font appel à des techniques globales (cf. 2.2) dans un cadre de chargement dynamique (cf. 2.2) au prix d'une recompilation dynamique quand les hypothèses initiales se trouvent infirmées. Dans cet article nous nous intéressons seulement aux schémas sans recompilation à l'exécution.

Compilation séparée et chargement dynamique (D) La compilation séparée, associée au chargement dynamique, est un schéma de compilation classique illustré par les langages Lisp, Smalltalk, C++, Java. Chaque unité de code est compilée séparément, donc indépendamment des autres unités, puis l'édition de liens rassemble les morceaux de façon incrémentale au cours du chargement (OWA). Ce schéma permet de distribuer des modules déjà compilés sous forme de boites noires. De plus, comme les unités sont traitées de manière indépendante, la recompilation des programmes peut se restreindre aux unités modifiées directement ou indirectement. Malheureusement, si l'on exclut des recompilations dynamiques, ce schéma ne permet aucune optimisation des mécanismes objet sur le code produit — librairies, programmes — et ne permet d'utiliser que peu d'implémentations pour l'héritage multiple — seuls les sous-objets de C++ et le hachage parfait sont compatibles. La coloration a été essayée, dans le cas particulier du test de sous-typage [Palacz et Vitek, 2003], mais le chargement dynamique impose un recalcul et des indirections qui sont coûteuses, aussi bien au chargement qu'à l'exécution.

Compilation globale (G) Ce schéma de compilation est utilisé par Eiffel[4] et constitue le cadre de toute la littérature sur l'optimisation des programmes objet autour des langages Self,

[4] Bien que les spécifications des langages de programmation soient en principe indépendantes de leur implémentation, de nombreux langages sont en fait indissociables de celle-ci. Par exemple, la covariance non contrainte d'Eiffel n'est pas implémentable efficacement sans compilation globale et la règle des *catcalls* n'est même pas vérifiable en compilation séparée [Meyer, 1997]. Il en va de même pour C++ et son implémentation par sous-objets ainsi que pour Java et le chargement dynamique.

Cecil, Eiffel [Zendra *et al.*, 1997; Collin *et al.*, 1997]. Il suppose que toutes les unités de code soient connues à la compilation (CWA) y compris le point d'entrée du programme. Il est donc possible d'effectuer une *analyse de types* [Grove et Chambers, 2001], de déterminer l'ensemble des *classes vivantes* (effectivement instanciées par le programme). Grâce à cela, on peut réduire la taille des exécutables en éliminant le *code mort* et compiler les envois de messages par des BTD — en effet le nombre de types à examiner pour chaque appel devient raisonnable — voire des appels statiques. La coloration peut alors s'utiliser en complément, pour les sites d'appels dont le degré de polymorphisme reste grand. Bien que ce schéma permette de générer le code le plus efficace il présente de nombreux inconvénients. L'ensemble des sources doit être disponible, ce qui empêche la distribution de *modules* pré-compilés. De plus, chaque modification sur le code, même mineure, entraîne une recompilation globale. Inversement, si la modification porte sur le code mort, elle peut ne même pas être vérifiée.

Compilation hybride Il est possible de combiner la compilation séparée (OWA) avec une édition de liens globale (CWA). C'est ainsi que la majorité des programmes utilisent C++. Dans le cas de C++, l'éditeur de liens n'a besoin d'aucune spécificité, mais les approches suivantes nécessitent un calcul spécifique avant l'édition de liens proprement dite. Une alternative que nous ne considérons pas plus serait que la compilation génère le code idoine qui effectuerait ce calcul lors du lancement du programme.

Compilation séparée et édition de liens globale (S) Pugh et Weddell [1990] proposaient initialement la coloration dans un cadre de compilation séparée, où le calcul des couleurs serait fait à l'édition de liens : seuls les modèles externes de toute la hiérarchie sont nécessaires. Comme la coloration est une technique intrinsèquement globale, une fois l'édition de liens effectuée, aucune nouvelle classe ni aucune nouvelle méthode ne peuvent être ajoutées sans recalculer tout ou partie de la coloration. Cette version de la compilation séparée est donc incompatible avec le chargement dynamique mais elle permet d'en conserver de nombreuses qualités — distribution du code compilé, recompilation partielle. En revanche, ce schéma ne s'applique pas aux BTD, car le code du BTD doit être généré à la compilation en connaissant la totalité des classes.

Compilation séparée avec optimisations globales (O) Privat et Ducournau [2004; 2005a] proposent un schéma de compilation séparée, permettant l'utilisation d'optimisations globales à l'édition de liens. C'est une généralisation du schéma précédent, qui nécessite certains artifices supplémentaires. Lorsqu'une unité de code est compilée, le compilateur produit, en plus du code compilé et du *modèle externe*, un *modèle interne* qui contient l'information relative à la circulation des types dans l'unité compilée. La phase d'édition de liens nécessite l'ensemble des unités compilées, y compris le point d'entrée du programme, ainsi que l'ensemble des modèles internes et externes. Grâce à ces modèles, on peut procéder à des analyses de types en se servant du flux de types contenu dans les modèles internes. La majorité des optimisations proposées dans un cadre de compilation globale (analyses de types, BTD, ...) deviennent possibles dans ce schéma hybride. A la compilation, chaque site d'appel est lui-même compilé comme un appel à un symbole unique. Lors de l'édition de liens, le code correspondant est généré suivant les spécificités du site d'appel : appels statiques (site monomorphe), BTD (site oligomorphe), coloration (site mégamorphe) — on appelle cela un *thunk* comme dans l'implémentation de C++ [Ellis et Stroustrup, 1990]. Mais la phase d'édition de liens travaille toujours suivant l'hypothèse du monde clos, et le chargement dynamique reste exclu. Un schéma voisin avait été proposé par Boucher [2000] dans un cadre de programmation fonctionnelle.

Compilation séparée des bibliothèques et globale du programme (H) Une dernière approche consiste à compiler les bibliothèques de manière séparée comme dans le schéma précédent. Le programme lui-même est compilé de manière globale en effectuant toutes les optimisations possibles : schéma global (G) sur le programme et schéma séparé avec optimisations globales (O) sur les bibliothèques.

2.3 Comparaisons et compatibilités

Le tableau 1 résume la compatibilité des schémas de compilation avec les techniques d'implémentation ainsi que l'ensemble des combinaisons testés.

Technique d'implémentation		Schéma de compilation					
		D dynamique	S séparé	O optimisé	H hybride	G global	
sous-objets	SO	◇	◇	∗	∗	∗	
hachage parfait	PH	○	●	∗	∗	∗	
coloration incrémentale	IC	○	●	∗	∗	∗	
caching	CA	○	●	∗	∗	∗	
method coloring	MC	⋆	●	●	◇	●	
binary tree dispatch	BTD	⋆	⋆	●	◇	●	
attribute coloring	AC	⋆	●	●	◇	●	
accessor simulation	AS	○	●	●	◇	●	

● : Testé, ○ : Extrapolé, ◇ : Compatible mais non testé, ∗ : inintéressant, ⋆ : Incompatible

Tab. 1. Compatibilité avec les schémas et efficacité des techniques d'implémentation

Le tableau 2 décrit l'efficacité a priori des techniques en se basant sur les études antérieures [Ducournau,]. L'espace est considéré suivant trois aspects : le code, les tables statiques, la mémoire dynamique (objets). Le temps est considéré à l'exécution, à la compilation et au chargement. La notation va de « − − » à « +++ », « ++ » représentant l'efficacité de la technique de référence en héritage simple. Un des objectifs des expérimentations qui suivent est de vérifier empiriquement les évaluation théoriques.

3 Expérimentations et discussion

Les expérimentations ont été réalisées avec le langage Prm, un langage expérimental modulaire [Privat et Ducournau, 2005b] qui a été conçu en grande partie dans ce but.

Les divers schémas et techniques présentés plus haut ont été testés avec la restriction suivante :

– les spécifications de Prm, en particulier le raffinement de classes, le rendent absolument incompatible avec le chargement dynamique ; le hachage parfait et la coloration incrémentale ont donc été implémentés en compilation séparée avec édition de liens globale (S) mais le code généré est exactement celui qui aurait été généré avec chargement dynamique (D). Seuls les défauts de cache et les recalculs liés à la coloration incrémentale sont sous-estimés.

3.1 Expérimentations

Le dispositif d'expérimentation (Figure 4) est constitué d'un programme de test P, qui est compilé par une variété de compilateurs $C_i, i \in [1..k]$ (ou par le même compilateur avec une variété d'options). Le temps d'exécution de chacun des exécutables P_i obtenus est ensuite mesuré sur une donnée commune D. Les C_i représentent différentes versions de $\text{Prm}_{\mathcal{C}}$. Comme le compilateur est le seul programme significatif disponible en Prm, le programme de test P est lui aussi le compilateur, tout comme la donnée D.

Les mesures temporelles sont effectuées avec la fonction Unix `times(2)` qui comptabilise le temps utilisé par un processus indépendamment de l'ordonnanceur système (obligatoire à cause

	Espace			Temps		
	Code	Statique	Dyn.	Exécution	Compilation	Chargement/Édt. de liens
SO	−	− −	− −	−	++	++
IC	−	+	+++	−	++	− −
PH-`and`	−	−	+++	−	++	+
PH-`mod`	−	+	+++	− −	++	+
PH-`and` +AC	− −	− −	+++	− −	++	+
PH-`mod` +AC	− −	−	+++	−	++	+
MC	++	++	+++	++	+	−
$\text{BTD}_{i<2}$	+++	+++	+++	+++	++	− −
$\text{BTD}_{i>4}$	− − −	+++	+++	− − −	−	− −
AC	++	++	+	++	+	−
AS	+	+	+++	−	+	+

+++ : optimal, ++ :très bon, + : bon, − : mauvais, − − : très mauvais, − − − : déraisonnable

Tab. 2. Efficacité attendue

des processeurs multi-cœur). Les mesures du temps et les statistiques dynamiques sont effectuées dans des passes différentes afin de ne pas les influencer réciproquement.

Ces tests ont été réalisés sous Ubuntu 8.4 et `gcc` 4.2.4 sur des processeurs de la famille des Pentium d'Intel. Chaque test est le meilleur temps pris sur au moins dix exécutions. Un compilateur P_i est généré pour chaque exécution pour tenir compte des éventuelles variations dans l'implantation mémoire. Malgré cela, les variations observées ne dépassent pas 1 à 2%. Un test complet représente donc plusieurs heures de calcul. Les tests ont été faits sur différents processeurs de même architecture pour confirmer la régularité du comportement observé.

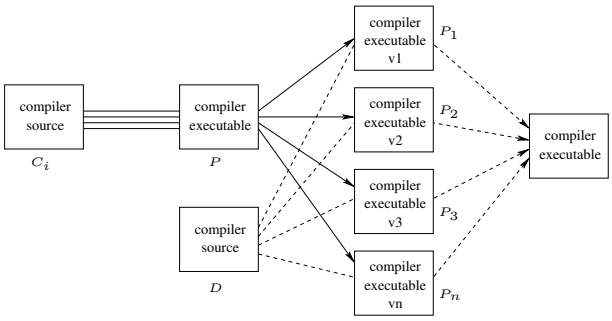


Fig. 4. Protocole d'expérimentation

La table 3 donne des statistiques sur le programme de test (P), d'un point de vue statique (nombre d'éléments du programme) et dynamique (nombre d'accès), lorsqu'il est appliqué à la donnée D.

3.2 Résultats

La table 4 présente les temps d'exécution de différents couples technique-schéma, sous la forme du rapport $(t_T - t_{\text{ref}})/t_{\text{ref}}$, où t_T est le temps de la version T considérée, et t_{ref} est le temps de la version de référence, c'est-à-dire le schéma S avec coloration (MC-AC). La signification de ces résultats dépend étroitement de la signification de ce temps de référence, qui pourrait être grossièrement surévalué, réduisant ainsi artificiellement les différences observées.

nombres de		statique	dynamique
classes	introductions	532	—
	instantiations	6651	35 M
	tests de sous-typage	194	87 M
méthodes	introductions	2599	—
	définitions	4446	—
	appels	15873	1707 M
BTD	0	124875	1134 M
	1	848	61 M
	2	600	180 M
	3	704	26 M
	4..7	1044	306 M
	8	32	228 K
attributs	introductions	614	—
	accès	4438	2560 M

La colonne "statique" représente le nombre d'éléments du programme (classes, méthodes, attributs) ainsi que le nombre de sites d'appel pour chaque mécanisme. La colonne "dynamique" rapporte le nombre d'invocations de ces sites durant l'exécution (en milliers ou millions). Les appels de méthodes sont comptés séparément en fonction de leur degré de polymorphisme.

Tab. 3. Statistiques sur le programme de test P [Ducournau *et al.*, 2009]

Il faut donc évaluer le niveau général d'efficacité de PRM$_\mathcal{C}$, ce qui réclame une référence externe. Nous avons pris SMART EIFFEL : les deux langages ont des fonctionnalités équivalentes et SMART EIFFEL est considéré comme très efficace (BTD/G). Sur I-5, le temps d'une compilation de SMART EIFFEL vers C sur la machine de test est d'environ 1 minute, soit le même ordre de grandeur que PRM$_\mathcal{C}$ — une comparaison plus fine ne serait pas significative. Comme les deux compilateurs ne font pas appel au même niveau d'optimisation, ces résultats ne sont pas non plus strictement comparables et la principale conclusion à en tirer est que, au total, les ordres de grandeur sont similaires. On peut donc affirmer que PRM$_\mathcal{C}$ est suffisamment efficace pour que les résultats présentés ici soient considérés comme significatifs : dans le rapport différence sur référence que nous analysons ci-dessous, le dénominateur n'est pas exagérément surévalué. Il l'est cependant un peu, par exemple par l'utilisation de *ramasse-miettes* conservatif de Boehm, qui n'est pas aussi efficace que le serait un *ramasse-miettes* dédié au modèle objet original de PRM. La gestion mémoire n'utilisant pas de mécanismes objet, son surcoût ne pèse que sur le dénominateur.

3.3 Discussion

La première conclusion à tirer de ces résultats concerne les schémas de compilation. Malgré la limitation des optimisations globales effectivement implémentées dans ces tests, le schéma G produit un code nettement plus efficace — ce n'est pas une surprise. Le fort taux de sites d'appels monomorphes explique ce résultat, puisque la seule différence entre MC-S et MC-BTD$_0$-G est la *mise en ligne* des appels monomorphes.

En revanche, le schéma O ne semble être qu'une légère amélioration du schéma S. Il s'agit cependant d'un résultat positif : même avec une faible optimisation, les sauts rajoutés par les *thunks* sont compensés par les gains sur les appels monomorphes. C'est une confirmation de l'analyse abstraite des processeurs modernes dont l'architecture en *pipe-line* est effectivement censée annuler le coût des sauts statiques, modulo les défauts de cache bien entendu [Driesen, 2001].

Dans les deux cas, la différence devrait se creuser avec des optimisations plus importantes comme par exemple une analyse de types plus sophistiquée que la simple CHA [Dean *et al.*, 1995] utilisée pour ces tests.

Du point de vue des techniques d'implémentation, les conclusions sont multiples. Le surcoût de la simulation des accesseurs (AS) est réel mais il est suffisamment faible avec la coloration de

processeur			S-1	UltraSPARC III		I-2	Xeon Prestonia		I-4	Xeon Irwindale		I-5	Core T2400	
fréquence				1.2 GHz			1.8 GHz			2.8 GHz			2.8 GHz	
L2 cache				8192 K			512 K			2048 K			2048 K	
année			123.2s	2001		87.4s	2001		43.3s	2006		34.8s	2006	
technique	scheme		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$	RTA	G	-22.6	-11.5	14.4	-10.9	-2.2	9.7	-13.3	-1.0	14.1	-8.9	2.9	13.0
MC-BTD$_2$	RTA	G	-22.2	-10.9	14.6	-11.7	-3.8	10.6	-13.7	-1.3	14.4	-10.2	-0.8	10.5
MC-BTD$_\infty$	CHA	O	***	***	***	-2.9	1.4	4.5	-3.0	4.9	8.2	2.7	19.4	16.2
MC-BTD$_2$	CHA	O	***	***	***	-5.4	-2.8	2.8	-5.9	2.3	8.7	-2.7	14.2	17.3
MC		S	0	9.8	9.8	0	5.7	5.7	0	7.9	7.9	0	11.1	11.1
IC		D	13.7	34.1	17.9	5.3	14.5	8.7	4.3	16.6	11.8	7.7	28.5	19.2
PH-`and`		D	13.4	35.4	19.5	2.5	14.5	11.6	4.2	19.0	14.2	6.2	31.4	23.8
PH-`mod`		D	81.1	226.0	80.0	28.6	104.3	58.8	55.2	172.0	75.2	24.4	106.3	65.8
PH-`mod+CA`$_4$		D	33.1	121.0	66.1	21.1	81.2	49.6	28.1	98.2	54.7	21.4	82.7	50.5
processeur			A-6	Athlon 64		A-7	Opteron Venus		I-8	Core2 T7200		I-9	Core2 E8500	
fréquence				2.2 GHz			2.4 GHz			2.0 GHz			3.16 GHz	
L2 cache				1024 K			1024 K			4096 K			6144 K	
année			34.0s	2003		32.8s	2005		30.4s	2006		18.5s	2008	
technique	scheme		AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC	AC	AS	AS/AC
MC-BTD$_\infty$	RTA	G	-12.5	2.9	17.6	-13.7	9.9	27.3	-9.4	7.5	18.6	-10.3	0.6	12.1
MC-BTD$_2$	RTA	G	-12.5	1.1	15.5	-13.1	10.7	27.3	-9.7	7.0	18.5	-9.7	0.6	11.5
MC-BTD$_\infty$	CHA	O	5.8	27.8	20.8	4.0	23.3	18.6	1.1	12.1	10.8	1.8	15.0	12.9
MC-BTD$_2$	CHA	O	3.4	27.3	23.1	2.0	22.8	20.4	-1.8	10.1	12.2	-2.5	13.0	16.0
MC		S	0	15.8	15.8	0	15.2	15.2	0	11.3	11.3	0	10.3	10.3
IC		D	14.9	40.1	21.9	12.6	38.9	23.3	7.7	29.7	20.5	8.0	29.7	20.1
PH-`and`		D	15.2	52.4	32.2	16.8	57.3	34.7	5.1	30.0	23.7	6.2	30.1	22.6
PH-`mod`		D	80.1	235.4	86.2	73.4	221.5	85.4	19.5	108.7	74.7	17.6	85.7	57.9
PH-`mod+CA`$_4$		D	40.4	141.2	71.8	38.3	129.3	65.8	19.6	81.3	51.6	20.8	74.6	44.5

Chaque sous-table détaille les résultats pour un processeur particulier, en donnant d'abord ses caractéristiques et le temps de compilation de référence (en secondes). Tous les autres chiffres sont des pourcentages. Chaque ligne décrit une technique d'invocation de méthode et de test de sous-typage dans un schéma particulier. La référence est la coloration de méthodes et d'attributs (MC-AC) dans le schéma séparé (S). Les deux premières colonnes représentent le surcoût vis-à-vis de la référence, respectivement avec la coloration d'attributs (AC) ou la simulation des accesseurs (AS). La troisième colonne donne le surcoût de AS par rapport à AC.

Tab. 4. Temps de compilation suivant les techniques, schémas et processeurs [Ducournau *et al.*, 2009]

méthodes (MC) pour que ce ne soit pas un handicap si les trous de la coloration des attributs en sont un.

Les résultats des tests sur le hachage parfait sont très intéressants par leur caractère marqué. PH-`and` apparaît très efficace, dépassant presque nos espérances, quand il est couplé avec la coloration d'attributs. Il faut donc vraiment l'envisager pour implémenter les interfaces de JAVA, d'autant qu'il serait utilisé beaucoup moins intensément dans ce cadre.

De son côté, PH-`mod` entraîne un surcoût qui le met vraisemblablement hors jeu sur ce type de processeur et sa combinaison avec la simulation des accesseurs est absolument inefficace. La simulation des accesseurs implique une séquence plus longue ce qui réduit le parallélisme. PH-`mod` aggrave la situation car le passage à l'unité de calcul flottant présente en plus l'inconvénient de vider le *pipeline*.

Ces conclusions s'appliquent peu ou prou aux processeurs considérés ici. Si l'on compare les processeurs, qui sont, dans les tables 4, classés de gauche à droite dans l'ordre chronologique, il est difficile d'en tirer des conclusions régulières. L'absence de points de comparaison "toutes choses égales par ailleurs" ne permet pas de juger de l'effet de l'augmentation parallèle de la taille du cache.

Esquisse de prescription. Certes les conclusions de ces expériences ne nous permettent pas de conclure définitivement pour tous les processeurs et pour tous les programmes, le choix d'une implémentation particulière restant dépendant des besoins fonctionnels du langage, par exemple le chargement dynamique. Cet article n'a pas la prétention de livrer une prescription sur comment implémenter les langages objets. Cependant, en partant du compromis entre modularité, efficacité et expressivité, ces premiers résultats peuvent être formulés de manière plus prescriptive.

- Si le point essentiel est l'efficacité, par exemple pour les petits systèmes embarqués, la compilation globale (G) avec un mélange de BTD — éventuellement restreint à BTD_1 si le processeur n'est pas équipé de prédiction de branchement — et de coloration est définitivement la solution. En effet, dans ce cadre le surcoût de l'héritage multiple est insignifiant.
- Si l'essentiel est l'expressivité et que le chargement dynamique n'est pas requis, la coloration en compilation séparée (S) représente certainement le meilleur compromis entre la modularité et l'efficacité — il peut être amélioré avec des optimisations globales lors de l'édition de liens (O). Le schéma hybride (H) qui combine la compilation séparée des librairies et la compilation globale du programme permet un bon compromis entre la flexibilité — recompilations rapides — et l'efficacité du code produit. Dans ce cadre, l'utilisation conjointe des BTD et de la coloration assure le code le plus compact et le plus efficace.
- Si l'essentiel est la modularité, C++ est une solution éprouvée dans le cadre de l'héritage multiple et du chargement dynamique. De plus, par rapport à Java, le surcoût des sous-objets est en *pratique* contre-balancé par l'implémentation hétérogène des génériques, par le fait que les types primitifs ne sont pas intégrés au système de type objet et que les programmeurs utilisent beaucoup moins les fonctionnalités objet. Cependant, le passage à l'échelle de cette implémentation est douteux, dans le pire des cas, la taille des tables est cubique dans le nombre de classes et le nombre d'entrées ajoutées par le compilateur dans les diverses tables peut être élevé. De ce fait, ces conclusions ne s'appliquent sûrement qu'à des programmes de taille moyenne, comme le compilateur Prm. En revanche, le sous-typage multiple — à la Java ou C#— est un compromis intéressant entre l'expressivité et l'efficacité. Actuellement cette efficacité vient principalement du compilateur JIT, cependant PH-`and` devrait être considéré par les concepteurs de machines virtuelles comme une implémentation sous-jacente pour les interfaces — dans le cas où le compilateur JIT n'arrive pas à la court-circuiter.

4 Travaux connexes, conclusion et perspectives

Dans nos travaux précédents, nous avons réalisé des évaluations abstraites [Ducournau, ; Ducournau, 2006; Ducournau, 2008] ou des évaluations concrètes reposant sur des programmes artificiels [Privat et Ducournau, 2005a]. Cet article présente les premiers résultats d'expérimentation permettant de comparer des techniques d'implémentation et des schémas de compilation, de façon à la fois systématique et aussi équitable que possible. Le protocole d'expérimentation, basé sur le *bootstrap* du compilateur n'est pas nouveau — il avait été utilisé, entre autres, pour Smart Eiffel. Cependant, Smart Eiffel imposait pour l'essentiel un schéma de compilation (G) et une technique d'implémentation (BTD) et ne les comparait qu'avec un compilateur Eiffel existant. Des travaux analogues ont aussi été menés autour des langages Self et Cecil [Dean *et al.*, 1996], mais ils concernent à la fois la compilation globale (G) et le typage dynamique, dans un cadre adaptatif. En typage statique, le compilateur Polyglot [Nystrom *et al.*, 2006] possède une architecture modulaire analogue à celle de $\text{Prm}_{\mathcal{C}}$ mais nous ne connaissons pas d'expérimentations comparatives réalisées avec ce compilateur dédié a Java. Dans le cadre de Java, toute une littérature s'intéresse à l'invocation de message et au test de sous-typage lorsqu'ils s'appliquent à des interfaces [Alpern *et al.*, 2001a]. Cependant, le schéma de compilation de Java (D) autorise peu d'implémentations en temps constant — les sous-objets (SO) sont vraisemblablement incompatibles avec les machines virtuelles et le hachage parfait (PH) n'a pas encore été expérimenté.

Les expérimentations présentées ici sont donc, à notre connaissance, les premières à comparer différentes techniques d'implémentation ou schémas de compilation *toutes choses égales par ailleurs*. La plate-forme de compilation de Prm produit un code globalement efficace — si l'on compare les temps de compilation avec ceux de Smart Eiffel— même si l'optimum est loin d'être atteint. Les différences que l'on observe devraient donc rester significatives dans des versions plus évoluées. On peut tirer deux conclusions assez robustes de ces expérimentations : (i) même avec une optimisation globale limitée, le schéma global (G) reste nettement meilleur que le schéma séparé (S) ; (ii) le schéma optimisé (O) est prometteur : le surcoût des *thunks* est bien compensé par les optimisations. S'il ne sera vraisemblablement jamais aussi bon que le global, des

optimisations plus poussées devraient en faire un bon intermédiaire entre S et G. Cependant, des expérimentations complémentaires sont nécessaires pour valider complètement le schéma O : pour des raisons techniques (l'inefficacité de l'analyseur syntaxique de $\text{PRM}_{\mathcal{C}}$), nous n'avons utilisé ici qu'une analyse de types très primitive, CHA, qui ne nécessite pas d'employer des *modèles internes*. Une analyse plus sophistiquée pourrait rendre la compilation trop lente : notez que nous n'avons pas mesuré ici le temps de compilation des différents compilateurs C_i, mais uniquement celui des différentes versions P_i du même compilateur. On peut enfin ajouter à ces conclusions techniques particulières une conclusion méthodologique générale : ces expérimentations confirment pour l'essentiel les analyses abstraites qui ont été menées depuis une dizaine d'années autour d'un modèle de processeur simplifié [Driesen, 2001].

Du côté des techniques d'implémentation, deux conclusions se dégagent : le surcoût de la simulation des accesseurs (AS) est faible quand elle est basée sur la coloration de méthodes (MC). En revanche, sa combinaison avec des techniques plus coûteuses augmente le surcoût. Par ailleurs, cette première implémentation du hachage parfait confirme les analyses abstraites antérieures en séparant nettement PH-`and` et PH-`mod`. Cela confirme l'intérêt de PH-`and` pour les interfaces de Java, d'autant que des résultats récents démontrent que son coût spatial n'est pas si élevé que cela [Ducournau et Morandat, 2009].

Ces tests présentent une limitation évidente. Un seul programme a été mesuré, dans des conditions de compilation différentes. Cette limitation est d'abord inhérente à l'expérimentation elle-même — bien que ces techniques de compilation soient applicables à tout langage (modulo les spécificités discutées par ailleurs à propos de C++ et Eiffel), le langage Prm a été conçu d'abord pour cette expérimentation. C'est ce qui la rend possible mais explique le fait que le compilateur Prm soit le seul programme Prm significatif. Cela dit, le compilateur Prm est un programme objet représentatif, par son nombre de classes et le nombre d'appels de mécanisme objet. On retrouve aussi un taux d'appels monomorphes comparable à ceux qui sont cités dans la littérature. En revanche, d'autres programmes pourraient différer sur le taux d'accès aux attributs par rapport aux appels de méthodes, ce qui pourrait changer les conclusions vis-à-vis de la simulation des accesseurs.

Les perspectives de ce travail sont de deux ordres. Les comparaisons systématiques ne sont pas encore complètes — il nous reste par exemple à intégrer l'implémentation de C++ (SO) et à analyser le temps de compilation (par C_i), les défauts de cache et l'espace mémoire consommé (par P_i). Des expérimentations sur d'autres familles ou architectures de processeurs sont aussi indispensables. Nos expériences sur des Pentiums de diverses générations donnent des résultats assez similaires même si les différences peuvent varier de façon marquée. Il est très possible que des architectures vraiment différentes, RISC par exemple, changent les conclusions. Par exemple, PH-`mod` pourrait très bien revenir dans la course sur un processeur doté d'une division entière native. Le schéma de compilation original de Prm doit être encore amélioré. Il reste des difficultés, par exemple l'élimination du code mort dans un module vivant. Le schéma hybride de compilation séparée des bibliothèques et globale du programme (H), qui représente sans doute un compromis pratique pour le programmeur, reste aussi à tester. Certaines techniques peuvent aussi être améliorées : ainsi la simulation des accesseurs nécessiterait un traitement particulier pour les vrais accesseurs, afin qu'ils ne soient pas doublement pénalisés. Enfin, l'adoption d'un *ramasse-miettes* plus adapté à Prm pourrait augmenter l'efficacité globale et la part des mécanismes objet dans la mesure totale, donc les différences relatives.

Depuis le début de cette recherche, notre démarche vise à développer des optimisations globales à l'édition de liens. Dans une perspective à plus long terme, les techniques qui ont été mises au point doivent aussi pouvoir s'appliquer aux compilateurs adaptatifs des machines virtuelles [Arnold *et al.*, 2005]. Le *thunk* généré au chargement d'une classe pourrait être recalculé lors du chargement d'une nouvelle classe qui infirme les hypothèses antérieures. La plate-forme Prm peut fournir des premières indications : on pourrait par exemple utiliser des *thunks* avec hachage parfait pour tous les appels polymorphes et des appels statiques pour les appels monomorphes. Cela permettrait une première validation de l'utilisation des *thunks* dans un cadre de compilation adaptative mais seule l'expérimentation dans un cadre de chargement dynamique permettrait de mesurer l'effet des recompilations, en particulier sur la localité des accès mémoire. Dans une moindre mesure, cette

limitation de la plate-forme d'expérimentations vaut aussi pour les tests sur le hachage parfait présentés ici.

Références

[Alpern *et al.*, 2001a] B. Alpern, A. Cocchi, S. Fink, et D. Grove. Efficient implementation of Java interfaces : Invokeinterface considered harmless. In *Proc. OOPSLA'01*, SIGPLAN Notices, 36(10), pages 108–124. ACM Press, 2001.

[Alpern *et al.*, 2001b] B. Alpern, A. Cocchi, et D. Grove. Dynamic type checking in Jalapeño. In *Proc. USENIX JVM'01*, 2001.

[Arnold *et al.*, 2005] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, et Peter F. Sweeney. A survey of adaptive optimization in virtual machines. In *Proc. of the IEEE, 93(2), 2005. Special issue on Program Generation, Optimization, and Adaptation*, 2005.

[Boucher, 2000] Dominique Boucher. GOld : a link-time optimizer for Scheme. In *Proc. Workshop on Scheme and Functional Programming. Rice Technical Report 00-368*, éditeur M. Felleisen, pages 1–12, 2000.

[Click et Rose, 2002] C. Click et J. Rose. Fast subtype checking in the Hotspot JVM. In *Proc. ACM-ISCOPE conference on Java Grande (JGI'02)*, pages 96–107, 2002.

[Cohen, 1991] N.H. Cohen. Type-extension type tests can be performed in constant time. *ACM Trans. Program. Lang. Syst.*, 13(4) :626–629, 1991.

[Collin *et al.*, 1997] S. Collin, D. Colnet, et O. Zendra. Type inference for late binding. the SmallEiffel compiler. In *Proc. Joint Modular Languages Conference*, LNCS 1204, pages 67–81. Springer, 1997.

[Czech *et al.*, 1997] Zbigniew J. Czech, George Havas, et Bohdan S. Majewski. Perfect hashing. *Theor. Comput. Sci.*, 182(1-2) :1–143, 1997.

[Dean *et al.*, 1995] J. Dean, D. Grove, et C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP'95*, éditeur W. Olthoff, LNCS 952, pages 77–101. Springer, 1995.

[Dean *et al.*, 1996] J. Dean, G. Defouw, D. Grove, V. Litvinov, et C. Chambers. Vortex : An optimizing compiler for object-oriented languages. In *Proc. OOPSLA'96*, SIGPLAN Notices, 31(10), pages 83–100. ACM Press, 1996.

[Dixon *et al.*, 1989] R. Dixon, T. McKee, P. Schweitzer, et M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *Proc. OOPSLA'89*, pages 211–214. ACM Press, 1989.

[Driesen, 2001] K. Driesen. *Efficient Polymorphic Calls*. Kluwer Academic Publisher, 2001.

[Ducournau,] R. Ducournau. Implementing statically typed object-oriented programming languages. *ACM Comp. Surv.*, 42. (to appear).

[Ducournau *et al.*, 2009] R. Ducournau, F. Morandat, et J. Privat. Empirical assessment of object-oriented implementations with multiple inheritance and static typing. In *Proc. OOPSLA'09*, éditeur Gary T. Leavens, SIGPLAN Notices, 44(10), pages 41–60. ACM Press, 2009.

[Ducournau et Morandat, 2009] R. Ducournau et F. Morandat. More results on perfect hashing for implementing object-oriented languages. Rapport Technique 09-001, LIRMM, Université Montpellier 2, 2009.

[Ducournau, 2006] R. Ducournau. Coloring, a versatile technique for implementing object-oriented languages. Rapport Technique LIRMM-06001, Université Montpellier 2, 2006.

[Ducournau, 2008] R. Ducournau. Perfect hashing as an almost perfect subtype test. *ACM Trans. Program. Lang. Syst.*, 30(6) :1–56, 2008.

[Ellis et Stroustrup, 1990] M.A. Ellis et B. Stroustrup. *The annotated C++ reference manual*. Addison-Wesley, Reading, MA, US, 1990.

[Grove et Chambers, 2001] D. Grove et C. Chambers. A framework for call graph construction algorithms. *ACM Trans. Program. Lang. Syst.*, 23(6) :685–746, 2001.

[Meyer, 1997] B. Meyer. *Eiffel - The language*. Prentice-Hall, 1997.

[Morandat *et al.*, 2009] F. Morandat, R. Ducournau, et J. Privat. Evaluation de l'efficacité des implémentations de l'héritage multiple en typage statique. In *Actes LMO'2009*, éditeurs B. Carré et O. Zendra, pages 17–32. Cépaduès, 2009.

[Myers, 1995] A. Myers. Bidirectional object layout for separate compilation. In *Proc. OOPSLA'95*, SIGPLAN Notices, 30(10), pages 124–139. ACM Press, 1995.

[Nystrom *et al.*, 2006] Nathaniel Nystrom, Xin Qi, et Andrew C. Myers. $\mathcal{JE}$: Nested intersection for scalable software composition. In *Proc. OOPSLA'06*, éditeurs Peri L. Tarr et William R. Cook, SIGPLAN Notices, 41(10), pages 21–35. ACM Press, 2006.

[Palacz et Vitek, 2003] Krzysztof Palacz et Jan Vitek. Java subtype tests in real-time. In *Proc. ECOOP'2003*, éditeur L. Cardelli, LNCS 2743, pages 378–404. Springer, 2003.

[Privat et Ducournau, 2004] J. Privat et R. Ducournau. Intégration d'optimisations globales en compilation séparée des langages à objets. In *Actes LMO'2004* in *L'Objet vol. 10*, éditeurs J. Euzenat et B. Carré, pages 61–74. Lavoisier, 2004.

[Privat et Ducournau, 2005a] J. Privat et R. Ducournau. Link-time static analysis for efficient separate compilation of object-oriented languages. In *ACM Workshop on Prog. Anal. Soft. Tools Engin. (PASTE'05)*, pages 20–27, 2005.

[Privat et Ducournau, 2005b] J. Privat et R. Ducournau. Raffinement de classes dans les langages à objets statiquement typés. In *Actes LMO'2005* in *L'Objet vol. 11*, éditeurs M. Huchard, S. Ducasse, et O. Nierstrasz, pages 17–32. Lavoisier, 2005.

[Pugh et Weddell, 1990] W. Pugh et G. Weddell. Two-directional record layout for multiple inheritance. In *Proc. PLDI'90*, ACM SIGPLAN Notices, 25(6), pages 85–91, 1990.

[Vitek *et al.*, 1997] J. Vitek, R.N. Horspool, et A. Krall. Efficient type inclusion tests. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 142–157. ACM Press, 1997.

[Zendra *et al.*, 1997] O. Zendra, D. Colnet, et S. Collin. Efficient dynamic dispatch without virtual function tables : The SmallEiffel compiler. In *Proc. OOPSLA'97*, SIGPLAN Notices, 32(10), pages 125–141. ACM Press, 1997.

Correction d'assemblages de composants impliquant des données et assertions

Mohamed Messabihi, Pascal André, and Christian Attiogbé

LINA CNRS UMR 6241 - Université de Nantes
2, rue de la Houssinière
F-44322 Nantes Cedex, France
Prenom.Nom@univ-nantes.fr

Résumé La démarche de construction du logiciel en partant de l'architecture, nécessite la prise en compte de la correction à différentes étapes afin d'assurer la qualité du logiciel final. Ainsi la correction est une préoccupation qui doit être prise en compte au niveau des composants et de leurs assemblages pour élaborer l'architecture logicielle. Kmelia est un langage et un modèle à composants multi-service où les composants sont abstraits et formels de façon à pouvoir y exprimer des propriétés et les vérifier. Les services de Kmelia peuvent êtres paramétrés par des données et sont dotés d'assertions (pré/post-conditions opérant sur les données). Dans cet article nous nous intéressons à la correction des modèles à composants en couvrant différents aspects : la correction au niveau des services et la correction des assemblages du point de vue des données présentes dans les interfaces des services. Nous présentons les enrichissements du langage de données de Kmelia permettant de traiter la correction au niveau des services et de l'architecture. Nous illustrons l'étude par un exemple.

Key words: Composants, Services, Architecture Logicielle, Correction, Assertions

Cet article est une version actualisée de [5].

1 Introduction

Les architectures logicielles présentent le grand intérêt de permettre le raisonnement sur des systèmes logiciels complexes à un niveau abstrait, c'est-à-dire en faisant abstraction des détails de conception ou d'implantation. Il est crucial de pouvoir démontrer des propriétés générales (sûreté, vivacité) de ces systèmes, ce qui est très difficile avec le code exécutable. Les propriétés peuvent concerner des composants assemblés dans les architectures ou bien les assemblages eux-mêmes. Elles couvrent aussi bien les aspects données, dynamiques ou structurels. Nous considérons des architectures logicielles à composants [2,17,23,24] décrites par des ADLs (*Architecture Description Language*) qui couvrent les aspects structurels, de données (fonctionnels) et dynamiques. Au niveau abstrait, l'architecture est perçue comme une collection de composants assemblés via des connecteurs

dans des configurations [2]. Les composants offrent et requièrent des services via leurs interfaces. La connexion entre composants se fait selon les modèles par des liaisons d'interfaces, de ports ou directement de services.

Notre travail porte sur la formalisation des architectures permettant la vérification de propriétés et le raffinement. Il se situe donc dans la lignée des approches formelles pour la description abstraite d'architectures logicielles. Dans des travaux précédents, nous avons introduit un modèle et un langage appelé Kmelia pour décrire formellement des assemblages et vérifier des propriétés structurelles et dynamiques [8]. Nous avons aussi abordé dans [3] le problème de la méthodologie de modélisation de ces architectures avec des services hiérarchiques.

Dans cet article, nous nous intéressons au langage de données et à la vérification des propriétés associées. Cet aspect couvre la définition de types de données, les expressions, les assertions, les communications, le typage des composants et des assemblages. Les propriétés d'intérêt sont la sûreté de fonctionnnement des services et la préservation des propriétés des services dans les assemblages. Les principales contributions de ce travail sont : *i)* un langage formel pour les données et assertions qui fait de Kmelia un langage riche pour la spécification d'architectures de composants, *ii)* des techniques de vérification associées.

La suite de l'article est organisée de la façon suivante : la section 2 présente les principales caractéristiques du modèle Kmelia, complété par une partie données décrite dans la section 3. L'ensemble est illustré sur un cas concret extrait du référentiel CoCoME [26] dans la section 4. Dans la section 5 nous traitons la démarche de vérification de propriétés autour de Kmelia et nous illustrons par un petit exemple. La section 6 situe notre approche parmi des travaux similaires. Enfin nous évaluons le travail et indiquons des perspectives dans la section 7.

2 Kmelia : un modèle à composants multi-services

Kmelia est un modèle de spécification de composants basés sur des descriptions de services complexes [8]. Les composants sont *abstraits*, indépendants de leur environnement et par conséquent non exécutables. Kmelia sert à modéliser des *architectures logicielles* et leur *propriétés*. Ces modèles peuvent ensuite être raffinés vers des plate-formes d'exécution. Kmelia sert aussi de modèle commun pour l'étude de propriétés de modèles à composants et services (abstraction, interopérabilité, composabilité). Les caractéristiques principales du modèle Kmelia sont : les composants, les services et les assemblages.

Un **composant** est défini par un espace d'états, des services et une interface I. L'espace d'état est un ensemble de constantes et de variables typées, contraintes par un invariant. Dans l'interface d'un composant on distingue les *services offerts* I_p (resp. *requis* I_r) qui réalisent (resp. déclarent les besoins) des fonctionnalités. Les **services** sont eux-mêmes constitués d'une interface, d'une description d'état et d'assertions (pre-post conditions). Formellement, un service s est défini par un couple $(I_s,\ \mathcal{B}_s)$ où I_s est l'interface du service et $\mathcal{B}_s$ est un éventuel comportement dynamique.

L'interface du service est une abstraction des relations de composition de services, soit horizontale (dépendance) soit verticale (inclusion). En effet Kmelia permet l'expression de services complexes et leur composition. Nous ne détaillons pas ce point dans ce papier et renvoyons le lecteur à [6] qui en détaille la syntaxe et l'utilisation. Les services appelables au sein d'un autre service sont nommés sous-services et sont déclarés dans l'interface du service I_s. Formellement, I_s, l'**interface d'un service** s d'un composant C est spécifiée par un 5-uplet $\langle \sigma,\ P,\ Q,\ V_s, S_s \rangle$ où σ est la signature, P la pré-condition d'appel, Q la post-condition de déroulement, V_s un ensemble de déclarations de variables locales au service et $S_s = \langle sub_s,\ cal_s,\ req_s,\ int_s \rangle$ un quadruplet d'ensembles finis et disjoints de noms de services tels que $req_s \subseteq I_r$ et sub_s (resp. cal_s, req_s, int_s) est l'ensemble des services offerts (resp. les requis de l'appelant, les requis d'un composant quelconque, les offerts en interne) dans le cadre du service s.

Le comportement (dynamique) d'un service est caractérisé par un automate, qui précise les enchaînements d'actions autorisés. Ces actions sont des calculs, des communications (émissions, réceptions de messages), des invocations ou retours de services. Formellement, $\mathcal{B}_s$, le **comportement d'un service** s est un système de transitions étiquetées étendu (ou $eLTS$) spécifié par un sexuplet $\langle S, L, \delta, \Phi, S_0, S_F \rangle$ où S est l'ensemble des états de $\mathcal{B}_s$, $S_0 \in S$ est l'état initial, $S_F \subset S$ est l'ensemble non vide des états finaux (le service se termine toujours), L est l'ensemble des étiquettes des transitions entre les éléments de S. $\delta : S * L \rightarrow S$ est la relation de transition entre les états de S selon les étiquettes. $\Phi : S \leftrightarrow sub_s$ est la relation d'étiquetage des états par des points d'expansion de type optionnel.

Les composants Kmelia peuvent être assemblés ou composés via des liens entre services. Dans un **assemblage**, les services requis par certains composants sont liés (connectés) aux services offerts d'autres composants. Ces liaisons, appelées **liens d'assemblage**, établissent des canaux implicites pour les communications entre services. Les canaux sont point-à-point dans le modèle de base mais bidirectionnels. La figure 1 illustre une vue partielle d'un assemblage pour une application de commerce électronique dans laquelle on se focalise sur le processus de vente `process_sale` pour lequel deux sous-services sont requis (`ask_amount` et `bar_code`). Il n'y a pas de restrictions à la profondeur des sous-services et sous-liens d'un assemblage, elle est liée à la composition verticale des services. Une **composition** est un assemblage encapsulé dans un composant. La continuité des services est mise en œuvre par des **liens de promotion** qui servent à la promotion des services d'un composant vers ceux d'un composite (traits doubles dans la figure 1).

La hiérarchisation des services et des composants est une des caractéristiques de Kmelia qui permet une bonne lisibilité, de la flexibilité et une bonne traçabilité dans la conception des architectures. Cet aspect a été abordé dans [3]. Ce modèle de base a été enrichi d'une couche **protocole** [6] permettant de définir des enchaînements licites de services. Une extension aux services **partagés** (canaux multipoints) et communication multiple est proposée dans [7]. La spécialisation se fait dans les services et non les liens.

3 Le langage de données de Kmelia

Dans Kmelia, on souhaite formaliser des données et des assertions dans les descriptions des services, des composants et des assemblages. Les **données** concernent les états de composants ou de services, les paramètres des interactions. Les **assertions** couvrent les invariants, les pré/post-conditions, les propriétés spécifiques et les prédicats. Ces assertions sont nécessaires pour vérifier des propriétés prescrites. L'expressivité des assertions permet de s'engager à vérifier statiquement aussi bien (*i*) la correction au niveau du services (propriétés fonctionnelles) que (*ii*) la correction au niveau de l'architecture (propriétés d'assemblage).

Pour mettre en œuvre cet aspect **langage de données** [1], qui n'est pas nouveau dans les modèles composants, nous avons du établir un compromis entre l'expressivité (souhaitée) du langage et les contraintes liées notamment à l'intégration des aspects dynamiques des eLTS et des communications, en particulier pour la vérification cohérente de propriétés.

Dans la suite de cette section nous montrons l'extension de la partie données comprenant des déclarations de types, des expressions arithmétiques et logiques ainsi que des prédicats. Les règles de grammaires ci-après sont une représentation très simplifiée de l'analyseur syntaxique spécifié avec ANTLR dans l'outil COSTO [2].

3.1 Types

Kmelia reprend les types de base usuels $Integer, Boolean, Char, String$, et fournit à l'utilisateur un moyen de définir ses propres types en suivant les règles ci-dessous :

$$
\begin{array}{rcl}
TypeDef & :: & TypeName \\
 & | & \text{"struct" "\{"} \ TypeDecl \ (\text{";"} \ TypeDecl)^* \ \text{"\}"} \\
 & | & \text{"array" "["} \ LiteralValue_1 \ \text{".."} \ LiteralValue_2 \ \text{"]" "of"} \ TypeName \\
 & | & \text{"enum" "\{"} \ KmlExpr \ (\text{","} \ KmlExpr)^* \ \text{"\}"} \\
 & | & \text{"range"} \ LiteralValue_1 \ \text{".."} \ LiteralValue_2 \\
 & | & \text{"setOf"} \ TypeName
\end{array}
$$

$$TypeDecl ::= Identifier \ (\text{","} \ TypeDecl)^* \ \text{" :"} \ TypeName$$

3.2 Expressions

Une expression est construite avec des constantes, des variables et des applications d'opérateurs arithmétiques et logiques classiques $(+, *, mod, <, >=, != =$

1. Notons que la partie calcul définit la sémantique des actions dans les composants (initialisation et actions sur les transitions,...) en réutilisant la partie donnée.
2. COmponent Study TOolkit, plugin Eclipse dédié au langage Kmelia.

, ...). Dans la suite, chaque classe d'identificateurs est dénotée par un symbole non-terminal défini comme suit (C : constantes, V : variables, O : opérateurs, T : types). Les identificateurs sont composés de lettres, de chiffres et du caractère "_" suivant les règles usuelles. La troisième règle exprime qu'une expression peut être une opération préfixée (d'arité quelconque) et donc elle inclut les appels de fonctions.

$$
\begin{aligned}
KmlExpr ::= & \; LiteralValue \\
| \; & \; V \mid C \\
| \; & \; O \text{ "(" } KmlExpr_{1,\ldots} \; KmlExpr_n \text{ ")"} \\
| \; & \; KmlExpr_1 \; O \; KmlExpr_2
\end{aligned}
$$

3.3 Assertions

Nous avons introduit des assertions sous forme de prédicats (pré/post-conditions, invariants, gardes, propriétés, ...). Voici la syntaxe simplifiée des prédicats :

```
Pred ::= Cond                                  /* condition */
       | Prop                                  /* proposition */
       | "not" "(" Pred ")"
       | Pred "or" Pred
       | Pred "and" Pred
       | Pred "implies" Pred
       | ("exists" | "forall" ) "|" Pred
```

Un **invariant** de composant est une propriété qui doit être vraie à tout instant et qui est vérifiée avant et après l'exécution de chaque service. Les invariants permettent donc de capturer le sens et les caractéristiques de validité de certaines propriétés des composants.

Une **pré-condition** de service est également un prédicat. Elle porte sur les arguments en entrée que le client (appelant) a pour devoir de respecter. Si la (pré)condition n'est pas respectée, le fournisseur (appelé) ne s'engage pas à exécuter correctement le service appelé. Les pré-conditions doivent être transparentes, c'est à dire qu'elles doivent être nécessaires et suffisantes pour que l'appelant puisse être servi et obtienne les garanties associées aux post-conditions.

Une **post-condition** est un prédicat qui garantit les sorties que le fournisseur (appelé) s'engage à respecter si le service s'exécute normalement. Le mot-clef `PRE` peut être utilisé dans les post-conditions ou lors du traitement des exceptions pour faire référence à l'état dans lequel était le composant juste avant l'exécution du service.

4 Un exemple simplifié de spécification

Nous illustrons le langage Kmelia par une partie de l'étude de cas CoCoME (Common Component Modelling Example). Ce *benchmark* a servi de base de comparaison de modèles à composants [26]. Le cas d'étude décrit un système de vente à distance sous forme d'une collection de composants (caisse, scanner, imprimante, lecteur de carte...) interconnectés et qui interagissent. Il inclut des fonctionalités directement liées aux achats du client (la lecture optique des codes de produits, le paiement par carte ou en espèces, etc.) et d'autres tâches administratives telles que la gestion du stock et la génération des rapports, etc. Nous en avons fait une modélisation très partielle pour cet article, en nous focalisant sur le processus de vente.

L'architecture du système est présenté dans la figure 1. Elle met en évidence trois composants : le poste du caissier, l'applicatif de vente et le consortium bancaire. Ces composants sont reliés par des services, qui est le mode de connexion en Kmelia.

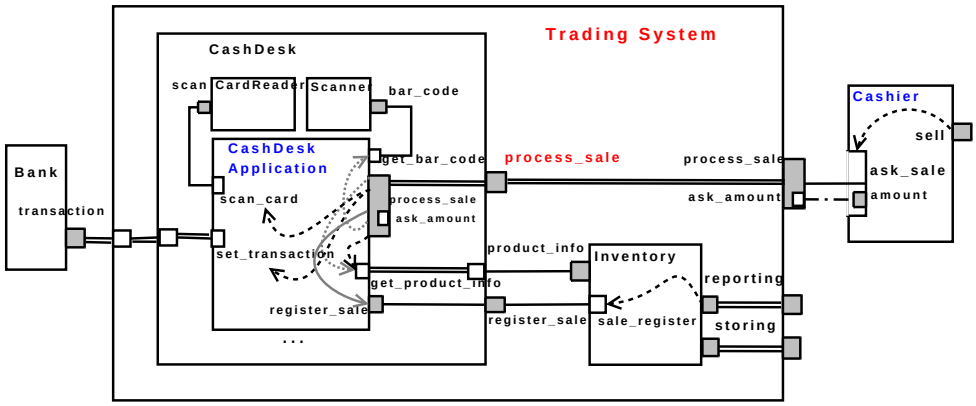


Figure 1. Assemblage abstrait et simplifié du cas CoCoME.

Le système de vente *Trading Sytem* est central à l'application. Il composé d'un composant de persistance *Inventory* et d'un poste de vente, la caisse *CashDesk*. Cette dernière offre un service de vente *process_sale* en se basant sur un composant applicatif *CashDeskApplication* et des périphériques d'entrée/sortie matérialisé par des composants (*Scanner*, *CardReader*, etc).

Dans la figure 1, les inclusions de composants dénotent la relation de composition entre composants et les traits doubles sur les services sont des *liens de promotion* (un service d'un composant est promu au niveau du composite qui le contient). L'interface des composants est mise en évidence sous la forme des services : les services offerts sont représentés par des boîtes grises, les services requis par des boîtes blanches.

Les traits simples entre services sont des *liens d'assemblage* qui associent des services offerts à des services requis (un service requis est « réalisé » par un service offert). Par exemple le service requis *ask_sale()* du composant *Cashier* est lié au service offert *process_sale()* du composant *CashDeskApplication*. Ce lien matérialise la satisfaction d'un besoin onctionnel mais représente aussi virtuellement un canal de communication entre les deux composants sur lequel sont véhiculés les interactions (message, données) entre les deux services et leurs dépendances.

Pour satisfaire le service requis *ask_sale()*, le service offert *process_sale* fait appel à d'autres services tels que *bar_code()* invoqué vers son service appelant mais il fait aussi appel à des services "externes" tels que *set_transaction*, pour enregistrer mes transaction ou *product_info* pour consulter la gestion de stock. Cette dépendance entre services est détaillée dans la spécification de l'interface du service. Illustrons ceci par l'exemple du service *process_sale* du listing 1 extrait de la spécification Kmelia du composant *CashDeskApplication*. Ce service décrit le processus de vente.

Listing 1. Kmelia specification CashDeskApplication

```
COMPONENT CashDeskApplication
INTERFACE
    provides : {process_sale , register_sale}
    requires : {get_product_info, set_transaction, scan_card}
TYPES
   PRODUCT :: struct {price:Integer; account:Integer; total:Integer};
   PAYMENTMODE :: enum {card, cash};
CONSTANTS
    null : Integer := -1;
    maxRef: Integer:=100
VARIABLES
    obs list_id : setOf Integer;
    obs state : enum {open, close}
   #...
INVARIANT
    @borned: size(list_id) <= maxRef
   #...
INITIALIZATION
    list_id := emptySet;
    state := close
   #...
SERVICES #--------- provided services ---------
provided process_sale(id : Integer) : Boolean
Interface
    calrequires : {get_bar_code, ask_amount}
    extrequires : {get_product_info, set_transaction, scan_card}
    intrequires : {register_sale}
Pre
   @openCD: ((state = open) and (id in list_id))
Variables
    prod_id, total, amount_var, rest : Integer;
    authorisation : Boolean;
    credit_info : String; prod_info : PRODUCT;
    payment_mode : PAYMENTMODE
Behavior
    Init i # initial state
    Final f # final state
{    i -- total := 0 --> e0,
    e0 -- __CALLER ? new_code() --> e1,
    e1 -- prod_id := _get_bar_code??get_bar_code() --> e2,
    e2 -- prod_info :=_get_product_info!!get_product_info(prod_id) --> e3,
    e3 -- {sum(prod_info, total) ; display(prod_info)} --> e4,
```

```
    e4 — __CALLER ? new_code() —> e1,
    e4 — __CALLER ? endSale()  —> e5,
    e5 — __CALLER ? payment(payment_mode) —> e6,
    e6 — [payment_mode = cash] display ("cash payment")—> e7,
    e6 — [payment_mode = card] display("card payment") —> e13,
    e7 — __CALLER!! ask_amount() —> e8,
    e8 — amount_var := __CALLER?? ask_amount(total) —> e9,
    e9 — [amount_var < total] __CALLER!! process_sale(false) —> f,
    e9 — [amount_var >= total] rest := amount_var - total  —> e10,
    e10 — __CALLER ! rest_amount(rest) —> e11,
    e11 — __SELF!! register_sale(compute_sale_string) —> e12,#()
    e12 — __CALLER!! process_sale(true) —> f,
    e13 — credit_info := _scan_card!! scan_card() —> e14,
    e14 — _set_transaction!! set_transaction(credit_info, total) —> e15,
    e15 — _set_transaction ? get_authorisation(authorisation)—> e16,
    e16 — [authorisation] _set_transaction ! debitAccount()—> e11,
    e17 — [not authorisation] __CALLER!! process_sale(false) —> f
}
Post    (state = open)
End
#————— other provided services ————
provided register_sale(info : String) : Void
   #...
End
#————— required services ————
required get_product_info(prod_id : Integer) : PRODUCT
End
required scan_card() : String
End
required set_transaction (credit_info : String; amount : Integer) : Boolean
End
required get_bar_code() : Integer
End
required ask_amount() : Integer
End
END_SERVICES
```

Un service offert est modélisé, d'un point de vue fonctionnel par sa signature (y compris les dépendances de services) et sa pre/postcondition. Par exemple, le service *process_sale* est invoqué en fournissant un identifiant de vendeur ; il n'est activable correctement (précondition) que si l'identifiant est référencé dans un ensemble d'identifiants acceptés et si le poste est ouvert. Le service est modélisé d'un point de vue dynamique par un système de transition qui indique les enchaînements d'actions de calcul et de communication (messages, services). Par exemple, le service *process_sale* interroge le scanner et le stock pour obtenir les références des produits, le caissier pour les différentes saisies, etc. Noter que cette représentation textuelle un peu lourde s'accompagne d'outils de saisie et de visulisation graphique dans l'outil.

Un service requis se modélise de manière similaire sur le point de vue fonctionnel à ceci près que les assertions sont définies sur un espace d'état virtuel (celui imaginé pour le fournisseur du service) [3]. Pour que l'interaction soit complète, on modélise le service qui invoque le service requis. Dans notre exemple, c'est le le service de vente *sell* qui joue ce rôle. Il modélise le comportement du caissier et sa description est donnée dans le listing 2. Noter que dans l'état *e4* la notation entre chevron indique que le sous-service *amount* est invocable dans cet état. Il pourra donc être invoqué par le service *process_sale*.

3. Voir [4] pour plus de détails.

Listing 2. Kmelia specification Cashier : sell

```
COMPONENT CASHIER
INTERFACE
    provides : {sell}
    requires : {ask_sale}
TYPES
   PAYMENTMODE :: enum {card, cash}
   # IDENTIFIERS : setOF Integer
VARIABLES
    identifier: Integer
   # ...
SERVICES #---------- provided services ----------
provided sell(id : Integer; mode : PAYMENTMODE)
Interface
    subprovides : {amount}
    extrequires : {ask_sale}
Variables # local to the service
    money : Integer;
    sale_success : Boolean
Behavior
    Init i
    Final f
{   i -- nop --> e0, //not detailed
    e0 -- _ask_sale!!ask_sale(id) --> e1,
    e1 -- _ask_sale ! new_code() --> e2,
    e2 -- _ask_sale ! endSale()  --> e3,
    e2 -- _ask_sale ! new_code() --> e2,
    e3 -- _ask_sale ! payment(mode) --> e4,
    #e4 <<amount()>>,       # subservice provided in state e4 only
    e4 -- [mode = cash]_ask_sale ? rest_amount(money) --> e4,
    e4 -- _ask_sale??ask_sale(sale_success) --> f
}
End
provided amount () : Integer
# answers the amount of cash that the user wants
Variables # local to the service
  a : Integer        # amount of cash wanted
Behavior
Init e0 # e0 is the initial state
Final e2 # e2 is a final state
{
  e0 -- {
       display("Please enter the amount of cash");
       a := readInt() # local variable
  } --> e1,
  e1 -- __CALLER!!amount(a) --> e2
  # send the amount of cash
}
End
#---------- required services ----------
required ask_sale(id : Integer) : Boolean
End
END_SERVICES
# end of Cashier
```

5 Vérification des propriétés impliquant des données et assertions

Les composants et leur assemblage peuvent être analysés sous diverses facettes. Dans un article précédent [8] nous avons traité l'interopérabilité statique dans les assemblages en quatre niveaux (signature, interface hiérarchique, assertions, interactions) sans toutefois détailler les assertions. Dans cet article nous

nous préoccupons uniquement des propriétés fonctionnelles et notamment de la correction fonctionnelle des services à travers la vérification des assertions.

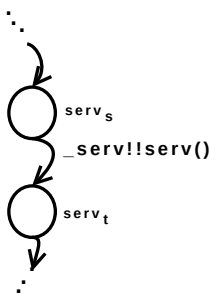
Proposition 1. *Les propriétés fonctionnelles exprimées par les assertions d'un service serv sont préservées si et seulement si :*
(i) elles sont vérifiées localement (à l'échelle du composant offrant serv) soit
1. *correction des pré/post avec l'automate y compris les appels de service,*
2. *correction des pré/post vis-à-vis de l'invariant du composant,*
3. *correction des enchaînements de services inclus ;*

(ii) elles sont vérifiées globalement au niveau de l'assemblage (architecture) :
1. *compatibilité des assertions sur des liens d'assemblage,*
2. *compatibilité des assertions sur des liens de promotion.*

Vérifications locales La vérification (i.2) se fait en utilisant des techniques classiques telles que celles utilisées dans Z ou B. La vérification (i.3) des protocoles est étudiée dans [6]. La vérification (i.1) est complexe car elle induit la construction de post-conditions à partir d'expressions Kmelia et d'automates. Nous abordons ici le problème des appels de services.

Nous rappelons que le comportement d'un service est décrit par un *eLTS*. Les *propriétés* d'un état s (notées $state\_properties(s)$) dans un *eLTS* de service sont les propriétés souhaitées (exprimées par le spécifieur, ou générées par un transformateur de prédicats) à cet état.

Considérons dans un *eLTS* une transition étiquetée par un appel de service $serv$; nous appelons l'état source de cette transition $serv_s$ et $serv_t$ son état de destination.



Notre hypothèse [4] pour la vérification des assertions du service $serv$ est de ne tenir compte que des variables qui sont présentes dans $state\_properties(serv_s)$ et/ou $state\_properties(serv_t)$. La vérification dans ce cas est définie par l'algorithme ($check_c$) ci-dessous :

$$check_c(serv) = \begin{cases} state\_properties(serv_s) \vdash P_{serv} \\ Q_{serv} \vdash state\_properties(serv_t) \end{cases}$$

Intuitivement, l'algorithme $check_c$ garantit que $state\_properties(serv_s)$ n'est pas contradictoire avec P_{serv}, et de la même manière que Q_{serv} ne contredit pas $state\_properties(serv_t)$.

4. Notons que cette hypothèse est relachée si les services utilisent un alphabet commun.

Vérification au niveau des assemblages Considérons un service *serv* avec les assertions P_{serv} et Q_{serv}. Un service requis (resp. offert) est noté $serv_r$ (resp. $serv_p$) .

Pour un lien d'assemblage, on vérifie les assertions de type contrat. L'algorithme de vérification d'un lien d'assemblage renvoie *vrai* lorsque (P_{serv_r} est plus forte que P_{serv_p} et Q_{serv_p} est plus forte que Q_{serv_r}. Nous formalisons cet algorithme ($check_a$) de la façon suivante

$$check_a(link(serv_r, serv_p)) \Rightarrow (P_{serv_r} \Rightarrow P_{serv_p}) \wedge (Q_{serv_p} \Rightarrow Q_{serv_r})$$

Dans un lien de promotion d'un service (offert ou requis) on véhicule toutes les propriétés du service promu *serv* au service promouvant (noté p_serv) et éventuellement on en ajoute d'autres. L'algorithme de vérification d'un lien de promotion de service requis renvoie *vrai* lorsque

$$check_p(promote(serv_r, p\_serv_r)) \Rightarrow (P_{serv_r} \Rightarrow P_{p\_serv_r}) \wedge (Q_{p\_serv_r} \Rightarrow Q_{serv_r})$$

L'algorithme de vérification d'un lien de promotion de service offert renvoie *vrai* lorsque

$$check_p(promote(serv_p, p\_serv_p)) \Rightarrow (P_{p\_serv_p} \Rightarrow P_{serv_p}) \wedge (Q_{serv_p} \Rightarrow Q_{p\_serv_p})$$

Notons que l'algorithme $check_p()$ peut être appelé récursivement pour vérifier les assertions des sous-services. Ces algorithmes peuvent être implantés avec B en s'inspirant des travaux sur la compatibilité d'interfaces décrites en B [21].

Illustration Illustrons la vérification (locale) d'appel de service sur l'exemple CoCoME. Nous nous intéressons au service `sell()` du composant `Cashier`. Lors de l'appel du service `ask_sale(id)` le service `sell()` envoie son identifiant `id` comme paramètre. Selon notre démarche de vérification, nous avons une obligation de preuve *op1* : $(id \in list\_id)$. Elle est issue de la pré-condition du service `process_sale()` et exprime que l'`id` doit appartenir à la liste des identifiants dans `CashDeskApplication`. En utilisant la procédure de vérification $check_c$, on vérifie non seulement la compatibilité des types mais on prouve également l'obligation de preuve *op1*. Cette obligation de preuve aurait permis de détecter une erreur si, par exemple, le service `ask-sale(id)` était appelé avec un identifiant inconnu.

6 Travaux connexes

La prise en compte des données dans les modèles à composants n'est pas nouvelles par contre il est plus original de combiner données, contrats, dynamique et communications dans un langage intégré. Kmelia le fait à un niveau qui permet la vérification *statique* de propriétés relatives aux aspects structurels, dynamiques et fonctionnels. En particulier le langage de données doit servir

non pas uniquement à la génération de code mais à la vérification des modèles abstraits.

Les modèles opérationnels tels que Corba, EJB ou .NET ne permettent pas de raisonner au niveau architectural qui nous intéresse ici. Il en est de même pour les modèles proches de la programmation tels que Java/A [10] ou ArchJava [1]. Certains modèles [11,14] proposent de repousser le langage de données au niveau de l'implantation : les types de données et les calculs associés sont alors définis, implantés et vérifiés par le compilateur. Ce choix pose problème pour les architectures logicielles car il est trop tardif, de plus il s'intègre mal avec les vérifications de structure, de contrat et de dynamique. D'autres modèles [18,27] prennent en compte des types de données et des contrats mais pas d'aspects dynamiques.

Les modèles avancés sur les aspects dynamiques [14,28,22,2,13] Dans Wright par exemple, la partie comportementale basée sur CSP est très détaillée (spécification et vérification) tandis que la partie donnée est mineure « *We will not carry out any specific formal proof using the developed model.* » [2] . Un modèle d'état et des opérations sont décrits dans un sous-ensemble de Z ; une opération correspond à un événement dans le modèle comportemental. Dans [25] la prise en compte des aspects données se fait sous forme de spécifications algébrique qui s'intègrent bien dans une vérification symbolique avec des systèmes de transition. Cependant le modèle ne supporte pas d'assertions.

Certaines approches se focalisent sur les contrats dans les communications. Par exemple, dans [15] l'idée est de définir des abstractions de communications (sortes de collaborations à la UML) puis de les réifier par des *medium* ou composants de communication. La partie contrat est décrite en OCL. C'est une approche complémentaire de la nôtre puisque dans Kmelia nous faisons abstraction de la mise en œuvre de la communication. Dans [16] l'idée est d'associer des contraintes (*may/must*) aux interactions définies dans les interfaces, et ainsi de définir des contrats comportementaux liant le client et le serveur. En Kmelia, la distinction entre contrainte du fournisseur et du demandeur se fait d'un point de vue méthodologique et non syntaxique. Par ailleurs, un service est atomique dans le modèle de Carrez (une opération avec son contrat) alors que dans Kmelia il est hiérarchique et son comportement dynamique (eLTS) doit respecter les assertions.

Fractal [20] propose différentes approches basées sur la séparation des préoccupations. L'aspect structurel est pris en compte dans Fractal ADL [19] ; les assertions sont traitées dans ConFract [18] et enfin la dynamique est étudiée dans Vercors [9] ou Fractal/SOFA [12].

Kmelia présente l'avantage de mettre en avant la notion de service, ce qui procure un pont relativement naturel avec les architectures à services.

7 Conclusion et perspectives

Nous avons présenté des enrichissements du langage Kmelia notamment sa partie données par des assertions de la forme pré/post. La vérification syntaxique

est opérationnelle dans l'outil COSTO. En fonction de cela nous avons défini une procédure de vérification des propriétés des services (propriétés exprimées sur les états de l'automate de comportement) et leur préservation au niveau des assemblages. Elle doit s'intégrer avec les vérifications de propriétés sur les aspects structurels et dynamiques.

A partir de cette procédure de vérification, nous avons les obligations à vérifier afin d'assurer la correction des assemblages de composants impliquant des services avec des interfaces dotés de pré et post-conditions. Dans [5], les vérifications étaient manuelles. Les expérimentations avec le langage B qui semblaient intéressantes de ce point de vue, ont été menées depuis, et les résultats sont présentés à CAL 2010. Nous avons vu mettre en œuvre un outil Kml2B qui étend la plateforme COSTO. Il permet de générer des spécifications de machines B ou Event-B en fonction des propriétés de cohérence à démontrer, que ce soit au niveau des composants ou des assemblages. Ces machines générées sont ensuite vérifiées en utilisant les plateformes de preuve dédiée à la méthode B, l'Atelier-B ou l'outil Rodin.

Pour compléter les propriétés prouvées sur les spécifications Kmelia dans le but d'obtenir des composants et assemblages corrects, nous travaillons sur la preuve de correction fonctionnelle des services. Ici, étant donné un service spécifié par ses pré/post-conditions, nous voulons prouver que le comportement (sous forme d'automate) donné au service est en adéquation avec les pré/post-conditions. Parmi les pistes que nous explorons, il y a le calcul de la précondition en partant de la post-condition et en remontant l'automate qui décrit le système ; nous explorons aussi une conversion en Event-B de l'automate de comportement puis une preuve de cohérence par rapport à l'invariant quand le déroulement du service arrive à l'état final.

Références

1. Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava : connecting software architecture to implementation. In *ICSE '02 : Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, New York, NY, USA, 2002. ACM.
2. Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, January 1997. Issued as CMU Technical Report CMU-CS-97-144.
3. Pascal André, Gilles Ardourel, and Christian Attiogbé. Spécification d'architectures logicielles en Kmelia : hiérarchie de connexion et composition. In *1ère Conférence Francophone sur les Architectures Logicielles*, pages 101–118. Hermès, Lavoisier, 2006.
4. Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix. Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies. In *6th International Workshop on Formal Aspects of Component Software(FACS 2009)*, LNCS, 2009. to appear.
5. Pascal André, Christian Attiogbé, and Messabihi Mohamed. Correction d'assemblages de composants impliquant des interfaces paramétrées. In *3e Conférence*

Francophone sur les Architectures Logicielles, volume RNTI-L-4 of *Revue des Nouvelles Technologies de l'Information*, pages 33–44. Cépaduès-Éditions, 2009.

6. Pascal André, Gilles Ardourel, and Christian Attiogbé. Defining Component Protocols with Service Composition : Illustration with the Kmelia Model. In *6th International Symposium on Software Composition, SC'07*, volume 4829 of *LNCS*. Springer, 2007.
7. Pascal André, Gilles Ardourel, and Christian Attiogbé. Composing Components with Shared Services in the Kmelia Model. In *7th International Symposium on Software Composition, SC'08*, volume 4954 of *LNCS*. Springer, 2008.
8. Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In *5th International Symposium on Software Composition, SC'06*, volume 4089 of *LNCS*. Springer, 2006.
9. Tomás Barros, Antonio Cansado, Eric Madelaine, and Marcela Rivera. Model-checking distributed components : The vercors platform. *Electron. Notes Theor. Comput. Sci.*, 182 :3–16, 2007.
10. Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electr. Notes Theor. Comput. Sci.*, 160 :75–96, 2006.
11. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12), 2006.
12. Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý, and Petr Tůma. *CoCoME in SOFA*, pages 1–2. Volume 5153 of Rausch et al. [26], 2008.
13. Tomas Bures, Petr Hnetynka, and Frantisek Plasil. Sofa 2.0 : Balancing advanced features in a hierarchical component model. In *SERA '06 : Fourth IC on Software Engineering Research, Management and Applications*, pages 40–48. IEEE Computer Society, 2006.
14. Carlos Canal, Lidia Fuentes, Ernesto Pimentel, Jos&#233 ; M. Troya, and Antonio Vallecillo. Adding Roles to CORBA Objects. *IEEE Trans. Softw. Eng.*, 29(3) :242–260, 2003.
15. Eric Cariou. *Contribution à un processus de réification d'abstractions de communication*. PhD thesis, Université de Rennes 1, July 2003.
16. Cyril Carrez, Alessandro Fantechi, and Elie Najm. Contrats comportementaux pour un assemblage sain de composants. In *Colloque Francophone sur l'Ingénierie des Protocoles (CFIP 2003)*, Paris, France, October 2003.
17. Paul C. Clements. A Survey of Architecture Description Languages. In *IWSSD '96 Proceedings*, page 16, Washington, DC, USA, 1996. IEEE Computer Society.
18. Philippe Collet and Roger Rousseau. ConFract : un système pour contractualiser des composants logiciels hiérarchiques. *L'Objet, LMO'05*, 11(1-2) :223–238, 2005.
19. T. Coupaye, V. Quéma, L. Seinturier, and J.-B. Stefani. *Intergiciel et Construction d'Applications Réparties*, chapter Le système de composants Fractal. InriAlpes, January 2007. sardes.inrialpes.fr/ecole/livre/pub/.
20. Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In Mario Südholt and Charles Consel, editors, *ECOOP Workshops*, volume 4379 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2006.

21. Arnaud Lanoix, Samuel Colin, and Jeanine Souquières. Développement formel par composants : assemblage et vérification à l'aide de B. *Technique et Science Informatiques (TSI)*, 26(8) :1007–1032, 2008. Numéro spécial AFADL07.
22. Jeff Magee, Jeff Kramer, and Dimitra Giannakopoulou. Behaviour analysis of software architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, pages 35–50, Deventer, The Netherlands, 1999. Kluwer, B.V.
23. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE*, 26(1) :70–93, january 2000.
24. M. Oussalah, T. Khammaci, and A. Smeda. *Les composants : définitions et concepts de base*, chapter 1, pages 1–18. Les systèmes à base de composants : principes et fondements. M. Oussalah et al. Eds, Editions Vuibert, 2005.
25. Sebastian Pavel, Jacques Noye, Pascal Poizat, and Jean-Claude Royer. Java Implementation of a Component Model with Explicit Symbolic Protocols. In *4th International Symposium on Software Composition, SC'05*, volume 3628 of *LNCS*. Springer, 2005.
26. Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and Frantisek Plasil, editors. *The Common Component Modeling Example : Comparing Software Component Models*, volume 5153 of *LNCS*. Springer, Heidelberg, 2008.
27. H. Schmidt. Trustworthy components-compositionality and prediction. *J. Syst. Softw.*, 65(3) :215–225, 2003.
28. D.M. Yellin and R.E. Strom. Protocol Specifications and Component Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

Session du groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

Reachability Analysis of Communicating Networks with Pushdowns

Alexander Heußner, Jérôme Leroux, Anca Muscholl, Grégoire Sutre

LaBRI, Université Bordeaux – France
ANR AVeriSS

The automatic verification of today's ubiquitous distributed applications demands for approaches that embrace different means of communication and synchronization. In the following, we will focus on models of *asynchronously* communicating *recursive* programs by introducing the class of *queueing concurrent processes* (QCP) that combines various local automaton models with asynchronous peer-to-peer communication over fifo channels that are reliable, point-to-point, and unbounded.

Although QCP have good expressive power, their reachability — which is the most fundamental question regarding safety verification — is undecidable in general. This is due to their subsumption of communicating finite-state automata, whose reachability is known to be undecidable [BZ83]. Further, adding pushdowns to a finite-state QCP yields an additional source of undecidability owing to pushdown automata synchronization. One of our main motivations is to separate these two sources of undecidability by considering behavioral restrictions that, at the one hand, render reachability decidable, and, on the other hand, conserve the simplicity and expressiveness of the model.

Our point of departure is the work of La Torre et al. [LMP08] that allowed to derive decidability results for directed forest architectures of *"well-queueing"* recursive QCP (RQCP). Well-queueing demands that a pushdown automaton can only dequeue/receive from a fifo queue if its stack is empty (but poses no restriction on enqueueing/sending). This corresponds to an event-based programming paradigm: tasks are executed by threads without interruption; when the local computation is finished (i.e., the stack is empty), the next task is started. Their work allowed to derive the decidability of reachability for well-qeueuing RQCP if the underlying network architecture is a *directed forest*. In the decidable case, their paper derived a doubly exponential upper bound by reduction to bounded-phase multi-stack pushdown automata (MSPDS) [LMP07]. Further, they also provided a second approach, inspired by recent work on reachability with bounded contexts [QR05], that allows to decide bounded-context reachability for well-queueing RQCP with a time bounded doubly exponentially in the number of contexts. Again, this is proven by reduction to bounded-phase MSPDS.

We extend the work of La Torre et al. in several directions. First, we add a dual notion to well-queueing: a pushdown process can enqueue (send) messages

only with empty stack (but can dequeue messages without restriction). This corresponds to an interrupt based programming paradigm where a thread can receive messages, e.g., tasks of a higher priority, while executing, but it can only send its result after the (recursive) computation is finished. *Oriented* architectures combine these two dual notions by fixing the behavior of the pushdown processes at each channel's two endpoints.

Second, we characterize the class of RQCP over oriented architectures that exhibits decidable reachability when restricting the runs to be *eager*. Informally, eagerness demands that the sending of a message is immediately followed by its reception, a notion closely connected to existentially 1-bounded communication [LM04]. Bounded communication for networks of finite automata is known to be decidable [GKM06], and, hence, allows us to investigate the influence of adding pushdowns to a finite-state model.

Main Results. An oriented architecture is called *confluent* if it contains a process that can communicate via the network with two distinct other processes that, with respect to this connection, can both use their pushdown stacks without restriction. We show that reachability of RQCP over eager runs is decidable for oriented architectures iff they are non-confluent. In the latter case, our constructions allows to derive EXPTIME-*completeness*.

Eagerness is a relatively strong requirement, hence, we show how it arises naturally by imposing a semantic restriction on the communication flow: the *mutex* restriction demands that in every reachable configuration there is no more than one non-empty channel per communication cycle. Thus, mutex can be seen as a generalization of the half-duplex restriction investigated in [CF05]. Obviously, polyforest architectures are mutex, and we can directly subsume La Torre et al.'s results. This idea can further be transferred to other classes of — possibly *infinite* — QCP, e.g., based on communicating Petri nets.

Our second main contribution is the extension of the bounded-context result to RQCP over oriented architectures. We provide a direct constructive proof that includes both well-queueing channels and their dual, as well as avoids [LMP08]'s reduction to MSPDS.

Practical Implications. Our approach allows to include both interrupt-driven and event-based threads that communicate under certain conditions.

The focus on unbounded, reliable, asynchronous fifo communication mirrors out interest to verify applications based on an underlying layer of TCP, e.g., client-server or peer-2-peer implementations based on the Berkeley Sockets API [Heu09].

Cyclic or mutual communication is the central property of most practical applications but is most often excluded in formal models in order to avoid undecidability (see the directed tree architectures of [LMP08]). The mutex property together with non-confluency allows us to model important cyclic architectures like (token) rings, or bidirectional communication in hierarchical overlay networks which are on the rise with the current focus on Grid computing (viz. Fig. 1).

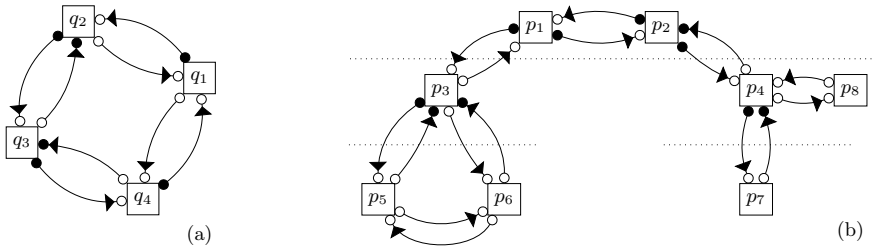


Fig. 1. *Non-confluent architectures: (a) a mutex ring (→ denotes the direction of the channel whose labelling ∘/• represents the restricted/unrestricted usage of the pushdown with respect to this channel), and (b) a hierarchical master-worker setting — tree-like architecture with bidirectional channels between master and workers (distribute tasks and collect results while in computation, send result to own master when computation is finished, i.e., stack empty) as well as channels whose both ends are restricted between workers of the same master.*

This abstract is based on [HLMS10]. A long version of this paper that includes all proofs that were omitted due to space limitations can be obtained at `http://hal.archives-ouvertes.fr/hal-00443529/`.

References

[BZ83] D. Brand, P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.

[CF05] G. Cécé, A. Finkel. Verification of programs with half-duplex communication. *Inf. Comput.*, 202(2):166–190, 2005.

[GKM06] B. Genest, D. Kuske, A. Muscholl. A Kleene theorem and model checking algorithms for existentially bounded communicating automata. *Inf. Comput.*, 204(6):920–956, 2006.

[Heu09] A. Heußner. Model extraction for sockets-based distributed programs. Technical report, LaBRI, October 2009.

[HLMS10] A. Heußner, J. Leroux, A. Muscholl, G. Sutre. Reachability analysis of communicating pushdown systems. in *Proceedings of FOSSACS '10*, 267–281. Springer, 2010.

[LMP07] S. La Torre, P. Madhusudan, G. Parlato. A robust class of context-sensitive languages. in *Proceedings of LICS '07*, 161–170. IEEE Computer Society, 2007.

[LMP08] S. La Torre, P. Madhusudan, G. Parlato. Context-bounded analysis of concurrent queue systems. in *Proceedings of TACAS '08*, 299–314. Springer, 2008.

[LM04] M. Lohrey, A. Muscholl. Bounded MSC communication. *Inf. Comput.*, 189(2):160–181, 2004.

[QR05] S. Qadeer, J. Rehof. Context-bounded model checking of concurrent software. in *Proceedings of TACAS '05*, 93–107. Springer, 2005.

Session de l'Action IDM

Ingénierie Dirigée par les Modèles

Meta-model Pruning

Sagar Sen, Naouel Moha, Benoit Baudry, and Jean-Marc Jézéquel

INRIA Rennes-Bretagne Atlantique, Campus universitaire de beaulieu, 35042 Rennes Cedex, France
{ssen,moha,bbaudry,jezequel}@irisa.fr

General-purpose modelling languages (GPMLs) such as the (UML)[1] are based on a a large number of classes and properties to model various aspects of a software system using the same language. UML is also extensible using the *profiles mechanism* [2] to provide modelling elements from specific domains such as services, aerospace systems, software radio, and data distribution. However, because of this increasing size of general purpose metamodels, most activities in a model-driven development process do not use the whole metamodel. In these cases it can be helpful to extract the subpart of the metamodel that is actually used. Our work presented at MODELS'09 [3], intrdoduces a metamodel pruning algorithm that automatically extracts this sub part. The algorithm also guarantees that the extracted metamodel preserves a supertype relationship with the initial metamodel. In the following we present several motivating scenarios and introduce the pruning algorithm.

1 Motivating scenarios

The motivation for us to develop a meta-model pruning algorithm comes from observations made by us and others in various phases of the MDE process.

Chain of Model Transformations A consequence of not defining the effective source meta-model of a model transformation is the non-compatibility/mis-match of outputs and inputs between transformations in chain. Consider a sequence of model transformations where output meta-model MM_o^a of model transformation MT_a is also the input meta-model MM_i^b for the next model transformation MT_b. However, we do not know if all models generated by MT_a can be processed by the model transformation MT_b as the concepts manipulated by the model transformations may be different. In [4], we identify this issue as one of the barriers to validate model transformations. Not identifying and dealing with this mismatch between the real input meta-model and real output meta-model can lead to serious software faults.

Testing Model Transformations When creating a model to test a model transformation you may need to use only a small number of concepts from the large declared source meta-model. In the context of automated testing, if you want to generate test models (such as using the tool Cartier [5]) then you would want to transform the smallest possible input meta-model to a formal language for constraint satisfaction. Transforming the entire meta-model to a formal language will lead to a enormous constraint satisfaction problem. These large constraint satisfaction problems are often intractable. Solving smaller constraint satisfaction problems obtained from a small set of concepts and subsequently with fewer variables is relatively feasible.

Software Process Modelling: Software process models contain several workflows. However, each workflow in a software process uses different sub-domains of a single shared meta-model such as the the UML. These workflows are often realized by different people and at different times. There are several software process methodologies that use the UML as the shared modelling language. The most popular of them is the Rational Unified Process (RUP) [6]. Figure **??**(a) shows the different workflows of RUP and the use of different subsets of UML for each workflow. Dedicated software processes such as ACCORD [7] use UML extended with domain-specific constructs to develop real-time systems.

2 A flexible metamodel pruning algorithm

Given a set of required classes and properties the rationale for designing the algorithm was to remove a maximum number of classes and properties facilitating us to scale a formal method to solve constraints from a relatively small input meta-model. For instance, we remove all properties which have a multiplicity

0..* and with a type not in the set of required class types. However, we also add some flexibility to the pruning algorithm. We provide options such as those that preserve properties (and their class type) in a required class even if they have a multiplicity 0..*. Whatever option is chosen, the resulting meta-model is a supertype of the large input meta-model [8], and identical meta-concept names are preserved. These properties ensure backward compatibility of the effective meta-model with respect to the large input meta-model.

Figure 1, displays an overview of the pruning algorithm. The inputs to the algorithm are: (1) A source meta-model $MM_s = MM_{large}$ (2) A set of required classes C_{req} (3) A set of required properties P_{req}, and (4) parameters to make the algorithm flexible for different pruning options.

The set of required classes C_{req} and properties P_{req} can be obtained from various sources as shown in Figure 1: (a) A static analysis of a model transformation can reveal which classes and properties are used by a transformation (b) The sets can be directly specified by the user (c) A model itself uses objects of different classes. These classes and their properties can be the sources for C_{req} and P_{req}.

The output of the algorithm is a pruned effective meta-model $MM_t = MM_{effective}$ that contains all classes in C_{req}, all properties in P_{req} and their associated dependencies. Some of the dependencies are mandatory such as all super classes of a class and some are optional such as properties with multiplicity 0..* and whose class type is not in C_{req}. A set of parameters allow us to control the inclusion of these optional properties or classes in order to give various effective meta-models for different applications. The output meta-model $MM_{effective}$ is a subset and a super-type of MM_s.

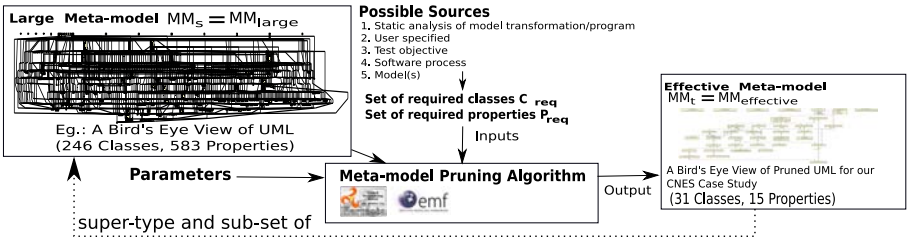


Fig. 1. The Meta-model Pruning Algorithm Overview

References

1. : UML 2.0 Specfication, http://www.omg.org/spec/UML/2.0/
2. Fuentes-Fernández, L., Vallecillo-Moreno, A.: An introduction to uml profiles. UPGRADE, European Journal for the Informatics Professional **5**(2) (April 2004) 5–13
3. Sen, S., Moha, N., Baudry, B., Jzquel, J.M.: Meta-model Pruning. In: ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, Colorado, USA (Oct 2009)
4. Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Communications of the ACM (2009)
5. Sen, S., Baudry, B., Mottu, J.M.: On combining mullti-formalism knowledge to select test models for model transformaion testing. In: ACM/IEEE International Conference on Software Testing, Lillehammer, Norway (April 2008)
6. Kruchten, P.: The Rational Unified Process: An Introduction. 3rd edn. Addison-Wesley Professional
7. Phan, T.H., Gerard, S., Terrier, F.: Real-time system modeling with accord/uml methodology: illustration through an automotive case study. Languages for system specification: Selected contributions on UML, systemC, system Verilog, mixed-signal systems, and property specification from FDL'03 (2004) 51–70
8. Steel, J., Jézéquel, J.M.: On model typing. Journal of Software and Systems Modeling (SoSyM) **6**(4) (December 2007) 401–414

Modeling Modeling

Pierre-Alain Muller[1], Frédéric Fondement[1], Benoît Baudry[2]

[1] Université de Haute-Alsace Mulhouse, France
{pierre-alain.muller, frederic.fondement}@uha.fr
[2] INRIA Rennes Bretagne Altantique, Rennes, France
benoit.baudry@irisa.fr

Many articles have already been written about modeling, offering definitions at various levels of abstraction, introducing conceptual frameworks or pragmatic tools, describing languages or environments, discussing practices and processes. It is amazing to observe in many calls for papers how modeling is now permeating all fields of software engineering. It looks like a lot of people are using models, as Monsieur Jourdain [6] was using prose, without knowing it.

While much has already been written on this topic, there is however neither precise description about what we do when we model, nor rigorous description of the relations among modeling artifacts. Therefore we propose to focus on the very heart of modeling, straight on the relation that we establish to represent something by something else, when we say that we model. Interestingly, the nature of these (some)things does not have to be defined for thinking about the relations between them. We will show how we can focus on the nature of relations, or on the patterns of relations that we may discover between these things.

In [7] we have proposed a contribution towards a theory of modeling. Here we summarize the definition of a canonical set of relations that can be used to ease and structure reasoning about modeling. We propose 5 representation relations that may be refined with nature (analytical/synthetical) and causality (correctness/validity).

Towards a model of modeling

We will use a very simple language to build this representation, based on "things" and "arrows" between them, such as the "objects" and "morphisms" found in Category Theory [2]. Things can be anything (this includes what other authors have called models and systems), and nothing has to be known about the internal structure of these things (which therefore do not have to be collections of "elements"). Conversely, arrows do not need to be functions between sets (thus arrows cannot be applied to "elements" but only composed with other arrows).

Let's start by modeling the fact that we have things which represent others things. As stated by Bran Selic [8], we first have to find a tradeoff between abstraction and understandability; therefore we will depart from the single *System class* view of Jean-Marie Favre [1], and distinguish between a *source* thing (that many authors call the model) and a *target* thing, although we understand that being a source thing or a target thing is relative to a given arrow, and does not imply anything about a given thing. This is represented in Figure 1, where the source is named X, the target Y, and the *RepresentationOf* relation μ.

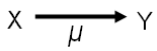


Figure 1: X is a representation of Y

We are using on purpose a very simple graphic concrete syntax for representing modeling relations. Our notation is based on arrows, and is intended to be easy to draw by hand (on blackboard and napkins).

Intention

Neither things nor representations of things are built in isolation. As said by Steinmüller, both exist for a given purpose, exhibit properties, are built for some given stakeholders. We can think about this as the *intention* of a thing. Intentional modeling [9] answers questions such as who and why, not what. The intention of a thing thus represents the reason why someone would be using that thing, in which context, and what are the expectations vs. that thing. It should be seen as a mixture of requirements, behavior, properties, and constraints, either satisfied or maintained by the thing.

As already said earlier, the "category theory kind" of thinking that we take in this paper does not require a description of the internals of the modeling artifacts (nor their intentions). Hence, it is enough to say that artifacts have an intention. The intentional flavor of models has also been used by Kuehne [5] in his description of metamodeling and by Gasevic et al. in their extension of Favre's megamodel [3]. The

consequences of intentional thinking applied to modeling can be understood and represented using Venn diagrams.

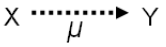
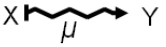
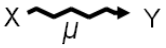
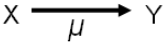
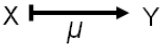
	Intention	Description	Notation
a)		X and Y have totally different intentions. This usually denotes a shift in viewpoints.	
b)		X and Y share some intention. X and Y can be partially represented by each other. The representation is both partial and extended.	
c)		X is a partial representation of Y. Everything which holds for X makes sense in the context of Y. Y can be partially represented by X.	
d)		X and Y share the same intention. They can represent each other. This usually denotes a shift in linguistic conformance.	
e)		X covers the intention of Y; X can represent Y, but X has additional properties. It is an extended representation.	

Table 1: Variations of the μ-relation, and graphical notation

Analytical vs. synthetical nature of representations

Several authors distinguish between analytical models and synthetical models (respectively descriptive and specification models in the sense of Seidewitz [8] and Favre [1]). An analytical representation relation states that the source expresses something about the target. A synthetical representation relation explains that the target is generated from the source.

Causality

Causality addresses the synchronization concern raised by Michael Jackson [4]; it expresses both *when* the μ-relation is established, and *how* (if ever) it is maintained over time. Causality is either continuous (the relation is always enforced) or discrete (the relation is enforced at some given points in time). Causality is also tightly coupled with the truth of Favre [1]; actually, causality is a concern about whether a representation is still meaningful when the truth has changed.

References

1. Favre, J.-M., *Foundations of Model (Driven) (Reverse) Engineering: Models - Episode I: Stories of The Fidus Papyrus and of The Solarus*. 2004, Dagstuhl Seminar 04101 on "Language Engineering for Model-Driven Software Development: Dagsthul, Germany.
2. Fokkinga, M.M., *A Gentle Introduction to Category Theory - The calculational approach*. 1994, University of Twente.
3. Gasevic, D., N. Kaviani, and M. Hatala. *On Metamodeling in Megamodels*. in Proceedings of *MODELS'07*. 2007: p. 91-105.
4. Jackson, M., *Some Basic Tenets of Description*. Software and Systems Modeling, 2002. **1**(1): p. 5-9.
5. Kuehne, T., *Matters of (Meta-) Modeling*. Software and Systems Modeling, 2006. **5**(4): p. 369-385.
6. Molière, *Le Bourgeois gentilhomme*. 1607.
7. Muller, P.-A., P.-A. Fondement, and B. Baudry. *Modeling Modeling*. in Proceedings of *MODELS'09*. 2009. Denver, CO, USA: p. 2-16.
8. Seidewitz, E., *What models means*. IEEE Software, 2003. **20**(5): p. 26-32.
9. Yu, E. and J. Mylopoulos. *Understanding "Why" in Software Process Modelling, Analysis, and Design*. in Proceedings of *ICSE'94*. 1994. Sorrento, Italy: p. 159-168.

Retours sur l'école de printemps Model-Driven Development for Distributed Realtime Embedded Systems

Jean-Philippe Babau (Université de Brest)
Sylvain Robert (CEA)

Cette année, du 20 au 24 avril, s'est tenue à Aussois la 4ème école internationale de printemps MDD4RES. Elle a été l'occasion de présenter des contributions académiques et industrielles aussi bien en terme d'avancées en ingénierie des modèles (modéles et langages, tests de transformations, analyse de performances, ...) que des standards adoptés (UML Marte, Autosar). Cet exposé donnera un aperçu de ce programme.

Session du groupe de travail LaMHA

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

OC4MC: Objective Caml for Multicore Architectures

Mathias Bourgoin, Benjamin Canou, Emmanuel Chailloux, Adrien Jonquet and Philippe Wang

Équipe APR - Département CALSCI
Laboratoire d'Informatique de Paris 6 (CNRS UMR 7606)
Université Pierre et Marie Curie (Paris 6)
4 place Jussieu, 75005 Paris, France
{Mathias.Bourgoin,Benjamin.Canou}@lip6.fr
{Emmanuel.Chailloux,Adrien.Jonquet,Philippe.Wang}@lip6.fr

Abstract. Objective Caml is well-known for its performance as a compiled programming language, notably thanks to its incremental generational automatic memory collection. However, for historical reasons, the latter was built for monoprocessor architectures. One consequence is the runtime library assumes there is effectively no more than one thread running at a time, which allows many optimisations for monocore architectures, e.g., concurrent data access is quite simplified: to be allowed to read or to write data in the (shared) heap, it has to lock a global mutex. The way it was built makes it not possible to take advantage of now widespread multicore CPU architectures. This paper presents our feedback on removing Objective Caml's garbage collector and designing a "Stop-The-World Stop&Copy" garbage collector to permit threads to take advantage of multicore architectures.
Key words: Objective Caml, Garbage Collector, Parallel Threads, Multicore

1 Introduction

History shows Caml dialects are generally designed for monoprocessor architectures. This is actual for Caml (the ancestor), Caml-Light (on the Zinc machine), Caml-Special-Light and Objective Caml (O'Caml) (1)[1]. For the last 25 years, the gain had been more important by optimizing the sequential runtime libraries or compiler schemes than by modifying them to target multiprocessors (2). One of the main success factors of O'Caml was its efficient implementation from Inria [9].

Currently, O'Caml's implementation of threads is only a way to express concurrent algorithms in the language since threads cannot actually be executed in parallel. But the recent rise of cheap multicore architectures – an average machine can run two or more threads in parallel – has created the need to use them. The O'Caml community would like concurrent programs to compute faster on

[1] We use the notation (n) to reference external links; see section Links at the end of the document for full URLs.

these architectures, as it is already the case for many other languages.

One way to achieve this goal is to add POSIX-like threads, which can actually run in parallel, to Objective Caml. Since the current runtime library has not ben designed with this model in mind, we need to provide a compatible alternative runtime library, in particular a new garbage collector, a new allocator and a new thread library.

The idea of modifying O'Caml's runtime library has already been used to certify safety-critical software development tools by the French company Esterel Technologies for its new SCADE SUITE 6™ qualifiable code generator (KCG) on O'Caml [11]. An O'Caml program such as KCG uses two kinds of library code: the O'Caml *standard library*, written mainly in O'Caml, and the *runtime library*, written in C and assembly language. Both are shipped with the O'Caml compiler and linked with the final executable. The difficulty of specifying and testing such low-level library code led to adapt and simplify it. The bulk of the modifications of the *runtime library* was to remove unessential features according to the coding standard of KCG. Most of the work consisted in simplifying the efficient but complex memory management subsystem. Esterel Technologies successfully replaced it by a plain Stop&Copy collector with a reasonable loss of performance [10].

One difficulty of this replacement was due to the tight coupling of the garbage collector with the O'Caml compiler (in-memory representation of values and entry points of the memory manager).

In this paper we propose to follow this way to exchange the garbage collection runtime library and thread library to be compliant with POSIX threads that allow to use parallel threads in a shared-memory concurrency model. One main constraint is to modify the least possible the O'Caml code generator and to focus modifications only for the runtime library. For that, we design a simple garbage collector with a unique copying, stop-the-world, compacting algorithm. In this case, the garbage collector is sequential and threads can be parallel with a synchronized mechanism when a garbage collector is called. This step allows to emphasize improved performances for real parallel programs in O'Caml. Once the garbage collector interface and thread library are defined using this assumption, it is possible to implement other garbage collector algorithms[8]

We have completely experimented this way, and the modifications of the Inria distribution are available as a patch called OC4MC (3).

The rest of this paper is organized as follows. Section 2 describes the main features of O'Caml's runtime library, mainly those which will change. Section 3 explains how to unplug the original garbage collector in a sequential world by refering to the Esterel Technologies experiment. Section 4 shows how to make the runtime library reentrant and interface its core with its thread library. Section 5 describes the synchronization mechanism of our garbage collector. Sec-

tion 6 details the implementation of our garbage collector algorithm. Section 7 presents some O'Caml benchmarks for multicore and comments results. Section 8 discusses related works while section 9 outlines our future work.

2 O'Caml's runtime library

O'Caml's high performance is partially due to its runtime library, which is written in C, plus a per-architecture assembly file that allows O'Caml calls and C calls to live together. Its original garbage collector allows a very fast allocation.

2.1 Garbage Collection

O'Caml has a two-generation garbage collector, derived from [3]. To allocate a value in the young generation heap, there are two possibilities: whether there is enough space, in which case a pointer is decremented by the size of the allocation, or there is not enough space, in which case the Stop&Copy garbage collector is triggered. The Stop&Copy part of O'Caml's garbage collection consists of copying the useful values of the young heap to the old heap. The latter is cleaned with an incremental Mark&Sweep&Compact algorithm, which consists of marking the values to sweep the useless ones, and sometimes compact the heap to take back the empty wholes left after value sweeping.

Thence O'Caml provides a particularly efficient allocation, with a Stop&Copy part that is fast because the young heap is small. We also have the possibility to program acceptable user interfaces as the old generation is cleaned incrementally so that we should never wait too long for the garbage collection.

2.2 Foreign function interface

For each supported architecture, there is an assembly interface to allow C function calls and O'Caml function calls to live together. Those files also contain fast memory allocation code and exception mechanism code. One should not ignore those files when modifying the runtime library design.

2.3 Thread library

O'Caml supports concurrency through:

- A POSIX threads-like low-level shared memory model, including mutexes and conditions.
- A higher level model based on Concurrent ML [12], implemented over the low-level thread library.

We focus on the low-level thread library, in particular its implementation and interaction with the runtime library.

2.4 Thread library implementation

or how threads are scheduled to run sequentially

POSIX model threads, the simple case Current thread library is useful to write concurrent programs. It provides a set of functions which match POSIX thread library functions for the C language. But will not allow threads to run simultaneously. Indeed, they share a mutex that prevents parallel memory allocations. Then to start the Stop&Copy garbage collection, there is only a single thread to stop. As functional programs tend to allocate a lot (of small values), it is generally reliable to schedule the threads on allocation, which is the mechanism that is used in O'Caml. However, scheduling at each and every allocation may cost too much, so a tick-counting thread is launched to measure the time the current thread has been running.

Blocking operation case Operations such as I/O operations (e.g. listen on a socket) or locking operations (e.g. `Mutex.lock`) may block a thread for a long time. During this, the thread cannot access the heap anyway as it is blocked, so it should be safe to allow other threads to run. A mechanism allows to declare such operations ("enter/leave blocking section") so that when they occur, the scheduler will detach the thread during the time of its blocking operation and allow another thread to run. At the end of a blocking operation, the thread is attached back to the scheduler.

Non allocating threads A thread that never allocates memory, without ever invoking a blocking operation, may prevent the scheduler to trigger, and may block the whole program. It is assumed that such programs do not occur in practice, and if it ever might occur then the programmer should use `Thread.yield`.

3 Replacing O'Caml's garbage collector: The Esterel Technologies experiment

Civil avionics software certification authorities assess standard software engineering rules for safety-critical software development. Such software dysfunction may have lethal consequences to their users (flight commands, railway traffic lights, etc). The DO-178B standard defines all the constraints ruling the aircraft software development. Code development as it is recognized by certification authorities follows the traditional V-Model dear to the software engineering industry. Traceability during each step of the development process is mandatory.

For Scade 6, Esterel Technologies decided to use O'Caml [10], which is very well suited for writing compilers, but quite outstanding in a very conservative domain such as civil avionics, even though DO-178B encourage the use of the best language for a given project. Classical language in this field is C, or subsets of C++ or Ada. Taking a new path meant demonstrating the compatibility

between DO-178B and O'Caml, by showing the process was under control (e.g. generated code is as expected, runtime library is predictable). To do so, Esterel Technologies decided not to use the object layer nor experimental features of O'Caml. While the standard library was welcome as fully documented and unit tested, the runtime library was not usable as such. Indeed, to make it more understandable, it was partially rewritten, in particular the garbage collection. The incremental two-generation garbage collector was much too complex to certify, and so was replaced by a simple one-generation Stop&Copy garbage collector. This allowed to dramatically decrease the number of lines of code and to make it easy to document (125 lines of C instead of 1200, for the garbage collector).

This experiment showed it was feasible to replace some relatively big parts of the runtime library. We supposed giving parallel-capable concurrent threads to O'Caml was feasible as well. Section 4 details this issue.

How to replace the garbage collector Basically, to replace O'Caml's garbage collector, the process consists of the following steps

1. regrouping the garbage collector global variables (essentially heap pointers),
2. providing get and set functions over them and use them,
3. and replacing the allocation functions and the collection code.

However, this works only if the new garbage collector still prevents simultaneous threads. Indeed, some global variables used in the runtime library makes parallel threads unsafe, as they may use the same temporary variable simultaneously. We address this issue in the next sections.

4 Runtime library reentrance and parallel threads

In the current runtime library source code, threading primitives are already separated from core functionality. However, the core assumes that threading primitives allow only one thread to run at a time. In fact, this source code separation has only been made to provide several implementations of threads (over POSIX, Win32, etc.), but does not permit to change the threading model. In this section, we exhibit the main problems preventing the runtime reentrance and how we solved them.

Execution context A first problem is the program context being stored in global variables in the core module. For this to work in presence of threads, current threading module implementations use

- a global lock preventing threads to run in parallel,
- and a context save/restore mechanism in per-thread context objects when switching between threads.

Our solution for threads to access their execution contexts in parallel is to leave the threading module handle the context. Global variables accesses in the core module are replaced by calls to context accessors from the threading module.

Global variables Along with the execution context, other global variables exist. For these, we have to distinguish between three kinds of use:

1. Temporary (i.e. that does not need to be saved on thread switch) thread local data is stored in global variables: we used either the same solution as the one for the execution context or more lightweight solution like adding function parameters
2. Shared data: we use a global lock mechanism provided by the threading module
3. Performance-critical shared data: memory management structures for which we had to implement a new parallel-compliant memory manager.

Memory management structures There are three main memory management structures.

1. Heaps which depend on the garbage collection algorithm we will use and will be described later
2. Global roots: we used global locks
3. Local roots which are local to threads and contain pointers to O'Caml stack segment boundaries and to O'Caml values in C stack segments. The local roots are stored in the per-thread structure and as we described earlier can be updated concurrently (foreign function interface is untouched)However, the garbage collector must have access to all roots (globals and locals) which can't be writable by other threads during a collection. We had to implement a synchronisation mechanism to be sure, no thread is able to update its roots during a collection. This mechanism will be detailed in the following section.

Interface between the core and threads To sum up, the core and threading modules are now interfaced as follows:

- The threading module defines a way to register thread-local data. When the core accesses such data, it must pass through the associated accessors. In our implementation, these accessors perform direct accesses to per-thread structures. A sequential implementation (like OCaml's current one) could simply use global variables.
- For global data, the threading module defines a global lock mechanism. In our parallel implementation, we use a mutex lock.
- The mechanism is generic, however, for performance reasons, we provide a way to redefine manually some critical functions, possibly in assembly.

5 A garbage collector for parallel threads

5.1 Stopping the World

To run a collection, the garbage collector has to know the exact state of all memory management structures (including the local roots). The heap has to

be inaccessible for every other thread. Before a collection, the collector has to stop every running thread, namely: stop the world. Moreover, this mechanism has to stop the threads while their local roots are updated and exact. One of our main constraints is to keep the O'Caml compiler unchanged. However, due to the current compiler, the only moment where the local roots of a thread are exacts is at an allocation. Thus, there is no other choice than to stop the threads during an allocation. Besides, this is the way the current sequential implementation behaves as the collector can only be triggered after a failed allocation while every thread is stopped (every thread being stopped during an allocation by the scheduler).

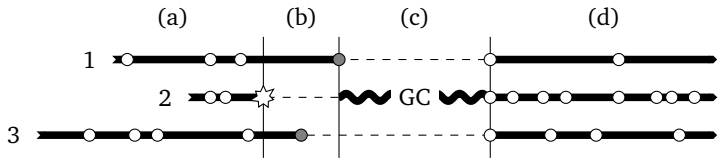


Fig. 1. Stop & Copy

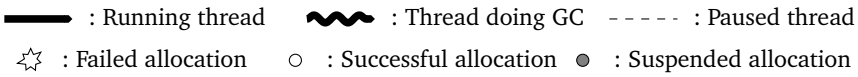


Fig. 5.1 describes the stop the world mechanism implemented, with three threads.

(a) Each thread may allocate until thread 2 fails an allocation, which means the need of a collection.
(b) Then, every other running thread will stop on the next allocation (ensuring the correctness of its local roots).
(c) Every thread stopped, the garbage collector runs.
(d) Each thread resumes its pre-collection behaviour starting by the allocation which made it stop.

As in the original implementation we use the enter/leave blocking section mechanism (described section 2.4) to allow blocking operation without preventing the stop of the world. Stopping the world allows the implementation of a sequential garbage collector.

5.2 Interface between thread library and garbage collector

To use various garbage collectors with different thread library implementations, we had to define clearly the interactions between the two.

As the core runtime library, the garbage collector uses accessors provided by the thread library. We also added primitives to iterate over threads so that the garbage collector does not have to know how thread-local data are stored.

On its side, the garbage collector defines primitives that has to be used by the thread library implementation to ensure that the system is in a correct state. For example, the garbage collector has to be notified when a thread has been paused. Indeed, a garbage collector must not wait for paused threads when stopping the world, and has to prevent them to be resumed before the collection has ended.

6 Our garbage collector

For a language as O'Caml, it is very important to have a very low cost allocation mechanism for small objects. O'Caml's current allocator dedicates a small (young) heap to small objects. This heap is a contiguous memory segment with a cursor indicating the end of the used zone, as shown in Fig. 2.

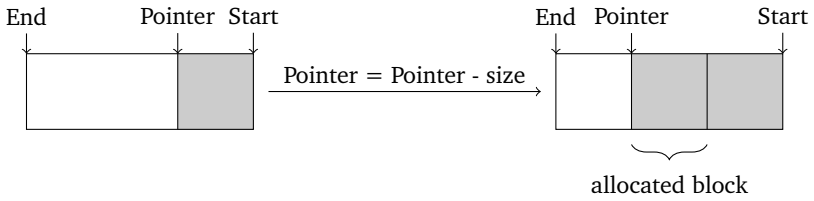


Fig. 2. Allocation in OCaml's small heap

Used space Free space

Thus, allocating is simply decreasing the cursor, testing if it has not crossed the limit, and returning the new cursor as the pointer to the newly allocated block. Of course, in presence of concurrent access to the heap variables, this is not possible. Moreover, adding a lock mechanism around the allocation would be too costly.

We will now describe our (fairly simple) memory management solution, which provides fast allocation for small objects, but allows several threads to allocate at the same time.

We first describe the structure of our heap, then we describe the garbage collection mechanism. We first describe the simple version, called *full* collection, then a more complex variation adding *partial* collection cycles.

6.1 Heap structures

We shall first present the structure of our heap. A graphical representation is shown in Fig. 3.

Local heaps Each thread has a small heap using the same cursor mechanism as O'Caml's small heap. These small heaps are called pages, in a page table.

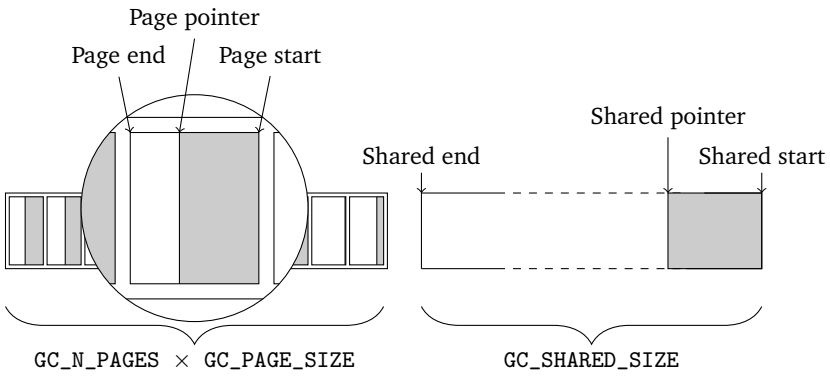


Fig. 3. Heap structures

Used space Free space

Hence, threads can allocate small objects in their own pages simultaneously safely. When a thread has filled its page, it takes a new page in the table, in mutual exclusion with others.

Shared heap For bigger objects, a big shared heap is used. Each allocation requires a lock. Its size varies on demand at every full collection, as we explain below.

6.2 Full collection

This first form of garbage collection cycle is called *full* since it operates on the whole heap. It is not the default algorithm but can be enabled by setting the compilation option `GC_ENABLE_PARTIAL` to `0`.

Algorithm When all pages are taken and a thread fails to allocate into one of its pages, or an allocation into the shared heap fails, then a collection is triggered.

We use a Stop&Copy algorithm to copy all living values from the pages and the shared heap into a new shared heap (Figure 4).

Every living heap-allocated value is copied into a new shared heap. The set of pages is then cleared and each threads gets a new empty page. In the end, we have a set of empty pages, and a new shared heap containing all (and only) living heap-allocated values.

Heap size The number of pages can be set with the `GC_PAGE_SIZE` environment variable, and their size can be set with the `GC_N_PAGES` environment variable. The `GC_SHARED_HEAP` environment variable defines the size of the shared heap.

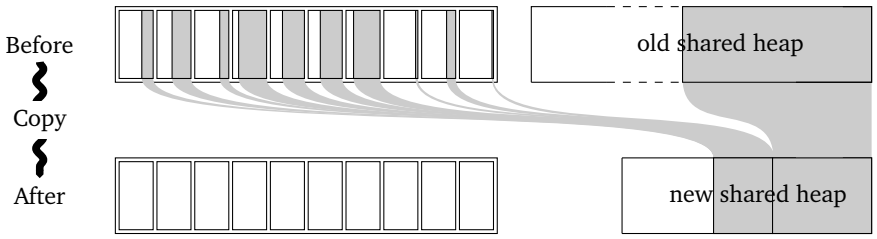


Fig. 4. Full garbage collection cycle

While running, the amount of memory needed by the program may increase. We thus added a mechanism to automatically grow and shrink the shared heap. For this, the new shared heap allocated during a full collection is set to the sum of all used space (in pages and shared heap).

With this solution, if the shared heap is full and entirely alive and the pages are empty, the new shared heap ends up being already full. This new size is thus weighted by some constant $k > 1$ (we chose 1.5 in practice).

$$size(new\ shared\ heap) = k \times (used\ space(old\ shared\ heap) + used\ space(page\ set))$$

If the weighted new size is not sufficient to contain the failed allocation (i.e. if the programmer tried to allocated more than $k - 1$ times the size of the old heap) the new size is maximized accordingly.

These precautions are sufficient to prevent triggering the GC in an infinite loop, but if the heap shrinks too much, it might be triggered too often. To solve this, we also added a minimal, configurable size to the heap.

6.3 Partial collection

This second form of collection is used complimentarily to the full collection. It is triggered when an allocation in a page fails and there is no more page left, but there is enough space in the shared heap store local values.

The performance of the two versions will be discussed after the results are presented.

Again, a Stop&Copy algorithm is used to flush alive values from the entire page table into the shared heap. The pages are then reinitialized and reassigned. This algorithm is graphically represented by Fig. 5.

Backward pointers In a Stop&Copy algorithm, the values considered alive are the ones that can be reached from the roots by following pointers in the from-space. When a pointer leads out of the from-space, it is not followed. A problem arises when the programmer affects a value in its page to a mutable cell in the shared heap, creating a *backward pointer*, and then forgets this value. In this case, the value is indeed alive: there exists a path leading to it from the roots, but this path is not entirely in the from-space. To prevent this kind of values

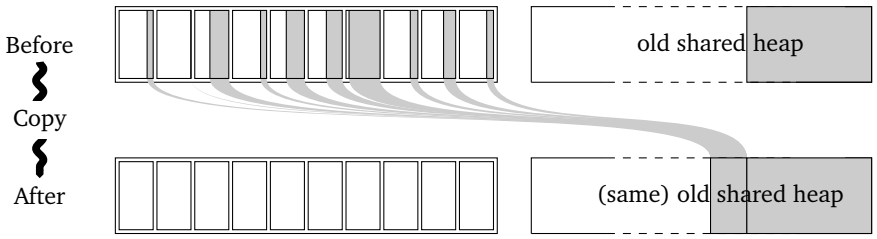


Fig. 5. Partial garbage collection cycle

from being freed on a local collection, we add them explicitely to the roots on such an affectation.

We use linked lists of memory chunks (instead of singleton pointers for efficiency) containing backward pointers to alive values. Every thread has its own list so we don't introduce mutexes. On a partial collection, we consider them as part of the root set. They are of course discarded just after the cycle since the values have been copied to the shared heap.

6.4 Optimisations and limitations

For performance reasons, we had to use macros (or inline functions) and link statically the three parts described above. This does not hinder the clear separation we have shown at source level, but means that the garbage collector cannot be changed at run or load time. Moreover, to achieve good performance, even if a generic version calling the C function of the garbage collector can be used, writing an assembly version of the allocator using the garbage collector structures was necessary to obtain best performance.

7 Benchmarks

Our performance benchmarks were made on Mac Pro running Ubuntu Linux 9.10 for x86-64bit architecture. The following gives some details about the hardware:

– The Mac Pro hosts two quad-core CPUs (Intel Xeon 2.8 GHz, without hyperthreading) which makes a total of 8 cores. The memory speed is 800 MHz and its capacity is 2×2 GB (4 GB in dual channel), and the bus speed is double (1.6 GHz). The number of cores determines the number of possible parallel threads. The memory speed is mandatory since with multicores, it easily becomes too low because it is shared between cores.

Our benchmarks were made with the first form of garbage collection (without partial collection) as described in 6.2.

Since there was very little hope of taking advantage of multicore by programing threads in O'Caml, there are very few existing benchmarks. Thus we distinguish two types of programs:

– the classic Caml benchmarks, such as Knuth-Bendix (KB), which are computed several times within threads,
– some programs written for the occasion, such as the sieve of Eratosthenes which is written in two very different paradigms, which are written with concurrency (and possibly parallelism) in mind.

Then, some programs will use a number of concurrent threads lesser or proportional to the avaiable number of cores, and other programs use a great number of concurrent threads.

7.1 Sequential programs

O'Caml programs are mainly sequential. We compared our implementation with O'Caml for sequential programs. Those tests were only testing our allocator and collector as the runtime is mainly the original O'Caml one and as the thread library we modified is unused in sequential programs.

Our benchmarks consist of the following programs:

– **KB**: the Knuth-Bendix completion algorithm. This is a fully functional program using exceptions intensively to compute over terms
– **Nucleic**: floating-point calculations involving trees [7].
– **FFT**: Fast Fourrier Transform, computing floats.
– **QuickSort**: classic (in-place) quick sort algorithm for arrays.

	O'Caml	**OC4MC**			
# threads	1TH	1TH	2TH	4TH	7TH
KB	4.90s	9.47s	14.17s	20.16s	22.36s
NUCLEIC	1.20s	3.98s	5.29s	4.99s	4.31s
FFT	2.21s	4.10s	3.13s	3.10s	3.04s
QUICKSORT	9.93s	9.71s	5.01s	2.65s	1.81s

Fig. 6. Benchmarks for sequential programs

Those tests allowed us to confirm that our allocator and garbage collector were less efficient than the original O'Caml ones. As shown in fig. 6, most programs are slower with OC4MC than with O'Caml with the exception of programs with a high computation over allocation ratio.

7.2 Parallel programs

We wrote the following specific test programs:

– **sieve**: the sieve of Eratosthenes. It starts with a big allocation of a Boolean matrix. Each cell represents an integer. Then each thread removes all multiples of uncomputed integers. We ran the tests with integers between 2 and 300 000.

– **matmult**: a simple matrix multiplication. The main loop is parallelized each thread computing its own lines of the final matrix. Matrix multiplication naive algorithm has a $O(n^3)$ complexity which means that the ratio of computation over allocation is very high. For our benchmarks we multiplied 1000×1000 matrixes.
– **life**: the classical game of life: a cellular automaton.It's an imperative object oriented program. It generates a universe (a board) where initialy three cells are alive. Each thread manages a section of this universe updating its cells at each step of the program. At the end of a step the main thread awaits for the other threads to end their calculation before allocating a new updated board and discard the old board. For each updated cell, dead or alive, a new object is allocated. For our benchmarks we limited the universe to a 200×200 board.
– **pi**: a π computation. For each point in a square, pi tests if the point is included inside the incircle of the square. The number of points inside this circle divided by the total number of points inside the square gives $\pi/4$. The square is divided equally into subsquares, each thread computes a subsquare.
– **sieve in CML-style**: n successive integers are passed through non-prime-number filtering channels. The first filter removes all multiples of 2. For each filter, when the first passing number is found (which is a prime number), a new filter initiates to filter its multiples. At the end, each created filter corresponds to a created thread and to a prime number. We ran the tests with integers between 2 and 9000, which creates about 1100 parallel threads.

	O'Caml	**OC4MC**				
# threads	1TH	1TH	2TH	4TH	7TH	14TH
SIEVE	68.68s	68.67s	38.08s	20.01s	11.13s	8.67s
speedup	**1**	**1**	**1.80**	**3.43**	**6.16**	**7.92**
MATMULT	16.49s	17.64s	8.70s	4.55s	2.70s	2.40s
speedup	**1.06**	**1**	**2.02**	**3.87**	**6.53**	**7.35**
LIFE	13.15s	16.80s	12.29s	10.21s	10.25s	9.97s
speedup	**1.28**	**1**	**1.36**	**1.65**	**1.64**	**1.69**
PI	22.88s	22.79s	11.38s	5.69s	3.26s	2.88s
speedup	**0.99**	**1**	**2.4**	**4.01**	**7.0**	**7.91**

Fig. 7. Benchmarks for little number of threads

	O'Caml	**OC4MC**
SIEVE CML-STYLE	89s	59s

Fig. 8. Benchmarks for great number of threads

7.3 Comparison with O'Caml

For most sequential programs, our alternative runtime library will provide predictable yet acceptable loss of performance, because for sequential programs, our garbage collector is particularly naive comparing to O'Caml's. It is also with no surprise that loss of performance is higher with programs that intensively allocate short-life small data. In this case, multithreading may as much allow a gain of performance as a loss of performance, depending on how memory is used: multithreading may increase or decrease cache miss.

However, as Fig. 7 shows, OC4MC may also provide speedup. Indeed, with *sieve* and *matmult* show that the speedup can be close to the number of cores, while *life* shows that the speedup is actual even though limited, probably due to memory speed limitation as it is a program that allocate a lot of small data.

Fig.8 shows that thread intensive programs may gain speedup thanks to switching-cost limitation, even with heavy structures. Plus, we may gain performance with existing programs written without parallelism but concurrency in mind. However, we shall not forget that shared-memory concurrent programming is hard, and adding parallelism for better performance does not ease the task as computation may actually run in parallel which makes debugging even harder.

Eventually, with a relatively simple garbage collector (which offers space for optimizations and improvements), OC4MC brings and shows interesting performance compared to O'Caml when run on multicore architectures, by allowing threads to allocate memory (and compute) in parallel.

In [6], Jon Harrop compared different versions of the matrix multiplication program using higher level abstractions such as the "parallel for" loop. He also tried to use a higher order implementation of the program which decreased scalability. This has been corrected in the latest versions of OC4MC. This shows OC4MC allows the use of functionnal abstractions to allow high level parallelism without decreasing its performances.

7.4 Impact of partial collections

	OC4MC				
# threads	1TH	2TH	4TH	7TH	14TH
SIEVE	68.67s	38.08s	20.01s	11.13	8.67s
SIEVE PARTIAL	68.67s	34.39s	17.4s	10.0s	8.63
LIFE	16.80s	12.29s	10.21s	10.25	9.97s
LIFE PARTIAL	13.65s	8.57s	7.18s	7.34s	7.30

	OC4MC
SIEVE CML-STYLE	59s
SIEVE CML-STYLE PARTIAL	59s

Fig. 9. Benchmarks with partial collection

The perfect case to show better performance when enabling partial collections is quite clear: a program allocating big data when starting, then allocating a lot of short-life data in every thread.

- The drawback of having a faster collection cycle operating over less data is straightforward: it has to be triggered more often. With a Stop The World technique, this means a greater delay for synchronization. If every thread allocates frequently, this delay can remain short.
- When a program allocates big data once and for all, having partial collections can importantly reduce collection times.
- If the data allocated are short-lived, then the amount of space copied to the shared heap is minimal and full collections are triggered less often.

Of course, one could also find worst cases, that is why we provide the option to use both algorithms.

In practice, on our example set, it seems that enabling partial collections is most of the time a good idea. An example is shown in Fig. 9.

8 Related works

There have been many attempts to give multicore support for typed functional languages, whether by adapting runtime libraries or by giving language extensions for parallel programming, or both.

[13] presents runtime support for multicores in parallel Haskell. Parallelism is explicited by using a "`par`" combinator, allowing the computation of its first parameter by a task ("`spark`") manager at anytime while the evaluation continues. This runtime uses a parallel garbage collector and different optimization techniques are presented in the paper.

Manticore [4] is a SML-based functional language providing two new features: CML's thread mechanism (first-class synchronous communication), and parallel structures (e.g. arrays) for data parallelism. Its two-generation garbage collector works with a per-thread heap without backward pointers and a shared heap. Each thread may collect on its own local heap. If collection fails, then a global garbage collector is triggered to copy from local heaps to shared heap.

Other approaches have introduced basic parallel computation mechanisms. CoThreads (4) for O'Caml introduces communication between processes which may run on different processors, with the same interface as O'Caml thread library. There is currently an attempt to give multicore support for Mlton/SML (5), by implementing low-level threads with additional higher level abstractions. F# (which combines the functional and imperative core of Caml and the object-oriented model of C#, for .NET) (6) provides an interface to .NET CLR's thread system. Two other levels of abstraction complete this low level layer for concurrent programming: message-passing and asynchronous workflow.

In the same vein, OC4MC's low level threading mechanism allows to build higher level abstractions for efficient and expressive parallel programming. For instance, the Event module (O'Caml's module to offer CML features) is built upon that layer. Following this, we can hope to use our work to allow existing parallel programming systems built upon O'Caml to take advantage of multicore architectures. We can cite CamlP3l [2] (skeleton programming: map, pipe, ...),

Objective Caml-Flight [1] (data parallelism), and BSML [5] (data parallelism and cost model).

9 Future works and conclusion

The experiment was successful since we effectively produced an adaptation of O'Caml for current multicore architectures showing promising performance results.

However, the strict guidelines we chose to follow, while founded, made this project take a lot of time and restrained our possibilities. For instance, compiler modifications have been kept lightweight for we hope they could be included as options in the standard distribution.

With less limitations, OC4MC would probably lead to better results. For instance, we have in mind some more intrusive modifications to the compiler:

- Use of local heaps without backward pointers by exporting at runtime the non-mutability property of constructed types, and thus have local garbage collections without the need to stop everyone.
- Insert checkpoints in the code so that the stop-the-world mechanism could be more responsive and work even for non allocating threads
- Or dump more compiling information in the executable about the state of the roots during execution to have less constraints for stopping the world.

As a result, OC4MC provides a runtime-level experiment platform to develop new threading models or garbage collectors. It also brings the possibility to design language-level concurrency abstractions over its parallel shared memory low-level thread library.

Acknowledgements

We are thankfull to Jane Street Capital for choosing OC4MC as an Ocaml Summer Project 2008 including the associated financial participation.
We also thank Jon Harrop who tested an early version of OC4MC and gave us his support during all the project.

Links

(1) The Caml Language
http://caml.inria.fr
(2) A brief history of Caml
http://www.pps.jussieu.fr/%7Ecousinea/Caml/caml%5Fhistory.html
(3) OC4MC distribution
http://www.ortsa.com:8480/ocmc/web/
(4) CoThreads
http://www.pps.jussieu.fr/~li/software/

(5) Multicore MLton
http://www.cs.cmu.edu/~spoons/parallel/
(6) F#
http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/

References

1. Emmanuel Chailloux and Christian Foisy. A portable implementation for Objective Caml flight. *Parallel Processing Letters*, 13(3):425–436, 08 2003. `http://www-apr.lip6.fr/~chaillou/Public/Dev/nocf/`.
2. Marco Danelutto, Roberto DiCosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the OCamlP3L experiment. In *Proceedings ACM workshop on ML and its applications*. Cornell University, 1998.
3. Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–123, New York, NY, USA, 1993. ACM.
4. Matthew Fluet, Mike Rainey, John Reppy, Adam Shaw, and Yingqi Xiao. Manticore: A heterogeneous parallel language. In *Proceedings of the Workshop on Declarative Aspects of Multicore Programming (DAMP 2007)*, pages 37–44, January 2007.
5. G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, aug 2002. `http://frederic.loulergue.eu/research/bsmllib/bsml-0.4beta.html`.
6. J. Harrop. Ocaml4multicore (oc4mc). *The OCaml Journal*, October 2009. `http://www.ffconsultancy.com/products/ocaml_journal/index.html`.
7. P. H. Hartel and M. Feeley *et al*. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, Jul 1996.
8. Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000. `http://www.cs.kent.ac.uk/people/staff/rej/gc.html`.
9. Xavier Leroy. The Objective Caml system release 3.10 : Documentation and user's manual. Technical report, Inria, 2008.
10. Bruno Pagano, Olivier Andrieu, Benjamin Canou, Emmanuel Chailloux, Jean-Louis Colaço, Thomas Moniot, and Philippe Wang. Certified development tools implementation in Objective Caml. In *Practical Aspects of Declarative Languages (PADL 08)*, pages 2–17, January 2008.
11. Bruno Pagano, Olivier Andrieu, Thomas Moniot, Benjamin Canou, Emmanuel Chailloux, Philippe Wang, Pascal Manoury, and Jean-Louis Colaço. Experience Report: Using Objective Caml to develop safety-critical embedded tool in a certification framework. In *International Conference of Functional Programming (ICFP 09)*, September 2009.
12. John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, Cambridge, England, 1999.

13. Simon Simon Marlow, Simon Peyton Jones, and Satnam Singh. Runtime Support for Multicore Haskell. In *International Conference of Functional Programming (ICFP 09)*, September 2009.

New Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons

Frédéric Gava and Ilias Garnier

LACL, University of Paris-Est
gava@univ-paris12.fr

LIST laboratory, CEA Saclay, Essonne, France
ilias.garnier@cea.fr

1 Introduction

This paper appear in the workshop APDCM, part of IPDPS'2009. In the original paper, a semantics study were done. For lack of space, we do not present it.

Since the paper "Go To Statement Considered Harmful", structured sequential programming is the norm. It is absolutely not the case for parallel programming. In this way, they, less or more, managed the communications with the usual problems of (un)buffered or (un)blocking sending, which are source of deadlocks and non-determism.

BSP is a parallel model which offers a high degree of abstraction. It is a "stop-the-world" model. BSML is an extension of ML to code this kind of algorithms using a a small set of primitives which are currently implemented as a parallel library for the ML programming language Objective Caml.

One of the primitive, called superposition, is dedicated to the parallel composition of BSML expressions (notably for divide-and-conquer algorithms) without any need of subset synchronisation (which are not BSP friendly). It is based on sequentially interleaved threads of BSP computations, called *super-threads*. Informally, it is equivalent to pairing in BSML.

Currently, the super-threads are implemented over the system threads. That leads to efficiency problems with OCcaml. This restriction is unnecessary and to overcome this limitation, we present another implementation which uses a global continuation-passing-style (CPS) transformation of BSML programs.

2 New implementation of the superposition

The CPS transformation of the super-threads is based of the fact that each super-threads work until its terminated or doing BSP communication. Them, it schedules the execution of it's sub-threads until they are terminated.

The full transformation of a program to CPS considerably impedes performance. This overhead is usually alleviated using transformation-time reductions to eliminate the so-called "administrative redexes" on the programs. However, that does not suffice. Aiming at high-performance computing, we can not afford to transform unnecessary expressions. Observing how some very limited parts of

the program need continuations, it seems natural to try to convert only the required expressions. We thus need a partial CPS transformation: the expressions to be transformed are those susceptible to reduce a superposition expression. Since we must cope with higher-order functions, the partial CPS transformation is guided by a flow analysis which yields a straightforward flow inference algorithm whose purpose is to decide if an expression is susceptible to reduce a superposition and thus CPS-converting or not.

The full implementation is available at `http://lacl.univ-paris12.fr/gava/cps-super-bsml-comp.tar.gz`. Currently our implementation works on a large subset of OCaml without objects, labels and functors.

Our partial CPS transformation needs simple types. Thus, we need to monomorphise the whole program. After type inference, the syntax tree is annotated with either ground types or type schemes, which are introduced only at **let** bindings. Each of these bindings is possibly instantiated with different types. Monomorphisation is the act of duplicating these bindings for each instantiation type (duplicating polymorphically typed functions for each needed domain type). We must also specialise type declarations to take into account impure functions: we scan the whole program, registering each type used inside algebraic data constructors or records and instantiating declarations accordingly. We must then perform a topological sort to take into account the fact that a polymorphic type may be used with a type declared after. In fine, to maximize the efficiency of the generated code, we use a similar process for flows: instead of duplicating functions based on types, we duplicate them based on flows.

3 Application to algorithmic skeletons

Anyone can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic patterns of computation (farm, pipe, *etc.*). Skeletal programming proposes that such patterns be abstracted and provided as a programmer's toolkit with specifications which transcend architectural variations but implementations which recognise them to enhance performance. The core principle of skeletal programming is conceptually straightforward. Its simplicity is a strength.

Having skeletons in BSML would have the advantage of the BSP pattern of communications (collective ones) and the expressivity of the skeleton approach. For our purpose and to have interesting benchmarks, we take for model the implementation of the OCamlP3l skeletons language (P3L's set of skeletons for ocaml) and base them on our parallel superposition primitive. For lack of space we do not present the ocamlp3l's set of skeletons.

The combination of P3L's skeletons generates a process network. This network takes in input a stream of data. Then each datum is transformed by the network independently of other data and finally the output is another stream of data of the same arity. In this way, they can be composed. Thus, if we suppose that the stream contains n data, the execution of the network will be composed n times using the superposition.

Each skeleton is implemented by dynamically created tasks which are distributed accross the processor using a round-robin fashion. Then, once all the tasks created, their execution are *superposed* using the superposition. For each execution, the input processor of the network send a data to the processor that have been dynamically designated to execute the sub-network. The parallelism arises from data being distributed over all superposed tasks.

Our example is a parallel PDE solver which works on a set of subdomains (taken from ocamlp3l's article). On each subdomain it applies a fast Poisson solver written in C. All the tests were run on the new LACL cluster composed of 20 Intel Pentium dual core E2180 2Ghz with 2GBytes of RAM interconnected with a Gigabyte Ethernet network. The MPI implementation of BSML were used (ocamlp3l is suck to TCP/IP).

As might be expected, OCamlP3l is faster than our naive implementation but not much. Barriers slow down the whole program but bulk-sending accelerates the communications: in the P3L run there exists a bottleneck due to the fact that sub-domains are centralised and therefore the amount of communication treated by one process may cause an important overhead. In BSML, the data are each time completely distributed, which reduces this overhead but causes a loss of time in the distribution of the data.

We did not benchmark the old implementation against the newer, because the older could not handle the number of concurrent threads. Our aim was not to beat OCamlP3l, whose implementation is far more complicated than ours but have both BSML and OCamlP3l.

4 Conclusion

In this paper we have defined a new implementation of a multi-threading primitive (called parallel superposition) for a high-level BSP and data-parallel language. This implementation uses a global CPS transformation which have been optimised using a flow analysis. Different optimisations such as defunctorization and monomorphi(*flow*)sation have also been added for performance issues. Our implementation relies on semantics investigations, allowing us to better trust it. Furthermore, it works on an important subset of OCaml.

The presented techniques are not novel, except our CPS transformation and the monoflowisation. Moreover, we find that the combination of all our transformations on a large subset of OCaml is quite new, if not on the pure theoretical front, at least as a tool (we think that it could be adapted to handle other constructs such as call/cc). Also, we are not aware of any implementation of P3L skeletons using the pure BSP paradigm: all implementations that we know of are implemented over pre-existing low level libraries.

The ease of use of this new implementation of the superposition will be experimented by developing less naive implementations of the OCamlP3l's skeletons as using a smarter heuristic for load balancing computations which will depend of the BSP's architecture parameters.

A Scalable Parallel Algorithm for Join Queries Evaluation on Heterogeneous Distributed Systems

Mohamad Al Hajj Hassan and Mostafa Bamha

LIFO, Université d'Orléans
B.P. 6759, 45067 Orléans Cedex 2, France
Email: {alhassan,bamha}@univ-orleans.fr

Abstract. Owing to the fast development of network technologies, executing parallel programs on distributed systems that connect heterogeneous machines became feasible but we still face some challenges: Workload imbalance in such environment may not only be due to uneven load distribution among machines as in parallel systems but also due to distribution that is not adequate with the characteristics of each machine. In this paper, we present a scalable parallel join algorithm called DFA-Join (Dynamic Frequency Adaptive Join) for heterogeneous distributed architectures based on an efficient dynamic data distribution and task allocation which makes it insensitive to data skew and ensures perfect balancing properties during all stages of join computation. The performance of this algorithm is analyzed using the scalable and portable BSP (Bulk Synchronous Parallel) cost model. We show that DFA-Join algorithm guarantees optimal complexity and near linear speed-up while reducing communication and disk input/output costs to a minimum. These results are confirmed by a series of tests.

1 A brief overview of "DFA-Join": An efficient Approach for Evaluating Join Queries on Heterogeneous Distributed Systems

Join is an expensive and frequently used operation in relational databases. The *join* of two relations R and S on attribute A of R and attribute B of S (A and B of the same domain) is the relation, written $R \bowtie S$, obtained by concatenating the pairs of tuples from R and S for which $R.A = S.B$. The *semi-join* of S by R is the relation $S \ltimes R$ composed of the tuples of S which occur in the join of R and S. Semi-join reduces the size of input relations and
$R \bowtie S = R \bowtie (S \ltimes R) = (R \ltimes S) \bowtie (S \ltimes R)$.

Join operation is one of the most widely used operations in relational database systems, but it is also a heavily time consuming operation. For this reason it was a prime target for parallelization. In PDBMS[1], relations are generally partitioned among processors by *horizontal fragmentation* with the values of a chosen attribute. Three methods are used [6]: hash partitioning, block-partitioning (also called range partitioning) and cyclic (also called round-robin) partitioning. Relation indexes are also partitioned and partitioning is expected to be balanced. Fragmentation is physical for a *Shared Nothing*[2] (SN) machine and logical for a *Shared Disk* (SD) machine[16].

Parallel join usually proceeds in two phases: a redistribution phase (generally based on join attribute hashing) and then a sequential join of local fragments. Many parallel join algorithms have been proposed. The principal ones are: *Sort-merge join*, *Simple-hash join*, *Grace-hash join* and *Hybrid-hash join* [17]. All of them (called hashing algorithms) are based on hashing functions which redistribute relations such that all the tuples having the same attribute value are forwarded to the same node. Local joins are then computed and their union is the output relation. Their major drawbacks is to be vulnerable to both *Attribute Value Skew* (which causes the imbalance of the output of the redistribution phase) and *Join Product Skew* (imbalance of the output of local joins) [18, 15, 13]. The former affects immediate performance and the latter affects the efficiency of output or pipelined operations in the case of multi-join queries.

[1] PDBMS : Parallel Database Management Systems.
[2] Shared Nothing machine : a distributed architecture where each processor has its memory and own disks.

Research has shown that join is parallelizable with near-linear speed-up on parallel systems such as Shared Nothing machines but only under ideal balancing conditions : data skew may have disastrous effects on the performance of parallel algorithms on such architectures [2, 4, 18, 7, 14].

Several parallel algorithms were presented to handle data skew while treating join queries on parallel database systems [20, 5, 10, 7, 12]. However, these algorithms are not efficient for the following reasons:

- they can't be scalable (and thus can't guarantee linear-speedup) because their routing decisions are generally performed by a coordinator processor while the other processors are idle,
- they can't solve the load imbalance problem as they base their routing decisions on incomplete or statistical information, and generally induce high communication and disk input/output costs during data redistribution,
- they can't solve data skew problem because data redistribution is generally based on pure hashing data into buckets and data hashing is known to be inefficient in the presence of skewed data [7],
- the communication cost in these algorithms is almost very high because all tuples of both relations are redistributed even if they do not participate in the join result and usually the size of relations to be joined is very large.

Optimal skew-handling parallel algorithms for evaluating join on homogeneous Shared Nothing (SN) machines were presented in [4, 2, 1]. These algorithms override the drawbacks of the algorithms stated earlier because the creation of communication templates, used for data redistribution among processors, is based on distributed histograms that hold complete data-distribution information. This step is jointly performed in parallel by all processors, *each one not necessarily computing the list of its own messages, so as to balance the overall process*. This makes the algorithms scalable because we do not have idle processors waiting for a coordinator node. In addition, the communication cost is reduced to a minimum because only tuples participating in the join result are redistributed. This allows us to reduce disk's input/output operations whenever exchanged data doesn't fit in memory. These algorithms also avoid the problem of Join Product Skew (JPS) while balancing the load of different processors even in the presence of Attribute Value Skew (AVS). We have proved, using the BSP cost model [19], that distributed histogram management has a negligible cost when compared to the provided efficiency gains in reducing communication costs, avoiding load imbalance between processors, and massively reducing the number of the join buckets (and thus join computation time) owing to the fact that the generated join buckets contains only tuples participating effectively to the join result. We have also proved using the same model that these algorithms have a near-linear speed-up performance on homogeneous SN architectures. This was also assured by a series of the experimental results.

Today with the rapid development of network technologies, query processing on distributed systems that may connect heterogeneous pools of machines is gaining the interest of database and scientific computing communities due to their processing power and memory capacity. But the heterogeneity of these systems introduced new challenges which were not present in parallel systems such as Shared Nothing machines because in order to benefit effectively from the power of distributed systems no processor must be idle while other processors are overloaded [11]. Hence, the actual characteristics of resources such as CPU power, Input/Output speed, connection speed, available memory, etc. must be taken into consideration while assigning jobs to the nodes. It is known that in parallel systems where machines are usually homogeneous, workload imbalance may be due to uneven load distribution, but in heterogeneous distributed systems it may be due to load distribution which is not proportional to the actual capabilities of each machine [8]. Moreover, in multi-user systems the capacity of a machine may highly vary from one instant to an another.

Algorithms used for join processing on a homogeneous parallel systems may not be effective on heterogeneous systems even if they efficiently handle the problem of data skew because they use a static strategy in load allocation. Hence, in order to obtain an acceptable performance on such systems an adaptive or a dynamic load distribution strategy, that takes in consideration the power of each machine and reacts with the system state, must be used [11].

Recently, Zhang et al. [21] proposed three adaptive hash-based algorithms for evaluating join operations on such environments where a non-static redistribution strategy is used. In these algorithms, they addressed the problem of buffer overflow while building the hash tables by dynamically allocating additional machines. The algorithms start by creating the hash table of the build relation on a specific number of nodes. During this phase, if the memory of a node is used up, additional nodes will be employed in order to maintain the buckets in memory. The experimental results showed that the performance of the algorithms significantly degrades in the presence of skewed data. This is because they use hash functions for assigning join attribute values to the computing nodes. However, it's known that hash functions are not efficient in the presence of skewed data. In addition, the communication cost is high because all tuples of both relations are redistributed even those which will not participate in the final join result.

In this paper, we propose and present the performance results of a dynamic parallel algorithm called *DFA-join* (Dynamic Frequency Adaptive join) to evaluate join operations on heterogeneous distributed systems (a detailled description of DFA-Join is presented in [9]). In this algorithm, we balance the load between processors in a manner that no processor in the system may be idle while other processors are overloaded even in the presence of highly skewed data. To this end, we use a two-step (static and dynamic) load assignment approach. In the first static step each processor receives a number of buckets whose total join size is proportional to its actual capacity. Then, during the join phase, overloaded processors can forward some of the non-treated buckets in their local buffers to underloaded processors. This combination of static and dynamic approach allows us to efficiently handle the effect of AVS on heterogeneous systems in a manner to get total processing time approximately the same on all processors. We can also control the join product skew using a threshold value, where an idle processor does not accept additional load if its local join result size is greater then the given threshold.
To ensure the extensibility of the algorithm, processors are partitioned into disjoint sets. Each set (group) of processors has a designed local coordinator node which is responsible of balancing the load between processors in the group. If a set of processors finishes its assigned tasks before the other sets, then it may request them the transfer of some of their load. In this algorithm, we use distributed histograms[3] to find tuples that participate in the result of the join operation and only these tuples are redistributed, and thus the communication cost is decreased to minimum. This also reduces disk's input/output whenever the exchanged data doesn't fit in memory and reduces task's reallocation during join computation. It is proved in [3, 4], that histogram management has a negligible cost when compared to the gain it provides in reducing the communication cost and balancing load between processors. The performance of our algorithm is analyzed using the BSP (Bulk-Synchronous Programming) cost model [19] which shows that our algorithm guarantees optimal performance even for highly skewed data on heterogeneous distributed multi-processors architectures. These results were confirmed by a series of tests [9].

References

1. M. Bamha. An optimal and skew-insensitive join and multi-join algorithm for ditributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *Lecture Notes in Computer Science*, pages 616–625. Springer-Verlag, 2005.
2. M. Bamha and G. Hains. A skew-insensitive algorithm for join and multi-join operation on shared nothing machines. In *the 11th International Conference on Database and Expert Systems Applications DEXA'2000*, volume 1873 of *Lecture Notes in Computer Science*, London, United Kingdom, 2000. Springer-Verlag.
3. M. Bamha and G. Hains. An efficient equi-semi-join algorithm for distributed architectures. In *Proceedings of the 5th International Conference on Computational Science (ICCS'2005). 22-25 May, Atlanta, USA*, volume 3515 of *Lecture Notes in Computer Science*, pages 755–763. Springer-Verlag, 2005.

[3] Histograms are implemented as balanced trees (B^+-tree): A data structure that maintains an ordered set of data to allow efficient search and insert operations.

4. M. Bamha and G. Hains. A frequency adaptive join algorithm for Shared Nothing machines. *Journal of Parallel and Distributed Computing Practices (PDCP), Volume 3, Number 3, pages 333-345*, September 1999. Appears also in Progress in Computer Research, F. Columbus Ed. Vol. II, Nova Science Publishers, 2001.
5. Soon M. Chung and Arindam Chatterjee. An adaptive parallel distributive join algorithm on a cluster of workstations. *J. Supercomput.*, 21(1):5–35, 2002.
6. D. J. DeWitt and J. Gray. Parallel database systems : The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, June 1992.
7. D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri. Practical Skew Handling in Parallel Joins. In *Proceedings of the 18th VLDB Conference*, pages 27–40, Vancouver, British Columbia, Canada, 1992.
8. Anastasios Gounaris. Resource aware query processing on the grid. Thesis report, University of Manchester, Faculty of Engineering and Physical Sciences, 2005.
9. M. Al Hajj Hassan and M. Bamha. An efficient parallel algorithm for evaluating join queries on heterogeneous distributed systems. In *Proceedings of the 16th Annual International Conference on High Performance Computing (HiPC'09)*, Kochi (Cochin), India, 2009. IEEE Computer Society Press.
10. K. A. Hua and C. Lee. Handling data skew in multiprocessor database computers using partition tuning. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proc. of the 17th International Conference on Very Large Data Bases*, pages 525–535, Barcelona, Catalonia, Spain, 1991. Morgan Kaufmann.
11. H. Karatza and R. Hilzer. Load sharing in heterogeneous distributed systems. In *Proceedings of the 2002 Winter Simulation Conference*, pages 489–496, 2002.
12. M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for skew in the super database computer (SDC). In Dennis McLeod, Ron Sacks-Davis, and Hans Schek, editors, *Very Large Data Bases: 16th International Conference on Very Large Data Bases, August 13–16, 1990, Brisbane, Australia*, pages 210–221, Los Altos, CA 94022, USA, 1990. Morgan Kaufmann Publishers.
13. H. Lu, B.-C. Ooi, and K.-L. Tan. *Query Processing in Parallel Relational Database Systems*. IEEE Computer Society Press, Los Alamos, California, 1994.
14. A. N. Mourad, R. J. T. Morris, A. Swami, and H. C. Young. Limits of parallelism in hash join algorithms. *Performance evaluation*, 20(1/3):301–316, May 1994.
15. V. Poosala and Y. E. Ioannidis. Estimation of query-result distribution and its application in parallel-join load balancing. *In: Proc. 22th Int. Conference on Very Large Database Systems, VLDB'96, pp. 448-459, Bombay, India*, September 1996.
16. E. Rahm and T. Stöhr. Analysis of parallel scan processing in shared disk database systems. *Proc. of EURO-PAR'95 Conference, Stockholm, Springer-Verlag, LNCS*, pages 485–500, August 1995.
17. D. Schneider and D. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data, Portland, Oregon*, pages 110–121, New York, NY 10036, USA, 1989. ACM Press.
18. M. Seetha and P. S. Yu. Effectiveness of parallel joins. *IEEE, Transactions on Knowledge and Data Enginneerings*, 2(4):410–424, December 1990.
19. L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
20. Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. Handling data skew in parallel joins in shared-nothing systems. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1043–1052, New York, NY, USA, 2008. ACM.
21. Xi Zhang, Tahsin Kurc, Tony Pan, Umit Catalyurek, Sivaramakrishnan Narayanan, Pete Wyckoff, and Joel Saltz. Strategies for using additional resources in parallel hash-based join algorithms. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 4–13, Washington, DC, USA, 2004. IEEE Computer Society.

Session du groupe de travail LTP

Langages, Types et Preuves

Types Abstraits et Types Existentiels Ouverts
(Résumé étendu)

Benoît Montagu Didier Rémy

INRIA Paris-Rocquencourt

Introduction L'abstraction de types est un ingrédient essentiel des systèmes de modules à la ML. Elle s'exprime habituellement par le sous-typage d'une composante de type concrète vers une composante de type abstraite, c'est-à-dire par l'oubli explicite d'une définition de type. Un type abstrait, privé de définition, est alors identifié par son chemin d'accès.

La notion de chemin forme historiquement le fondement des systèmes de modules [4,2,3,7]. Pourtant, d'autres approches sont possibles [5,9,1,8], où les types abstraits sont exprimés par des types existentiels. Abstraire une composante de type, revient alors à créer un paquet existentiel dont le témoin est la définition que l'on souhaite cacher. Ainsi, Système F a un pouvoir d'expression suffisant pour écrire tous les programmes valides des modules de ML.

Toutefois, programmer directement dans Système F s'avère déplaisant : les constructions pour manipuler les types existentiels contraignent fortement la structure du programme, alors que cette contrainte disparaît en utilisant les modules de ML. Nous proposons $\mathsf{F}^{\curlyvee}$ (lire *F-zip*), une variante de Système F, où une programmation modulaire avec existentiels redevient possible.

Types existentiels ouverts Afin de rendre l'utilisation des types existentiels plus modulaire, nous décomposons les anciennes primitives en parties plus élémentaires. La limite de pages nous contraint à ne pas donner leur règles de typage.

On peut découper la primitive unpack ainsi : d'abord on définit avec la primitive ν une portée pour la variable de type α, c'est-à-dire pour le type abstrait ; puis on ouvre le paquet existentiel dans une définition locale en nommant sont témoin α avec la primitive open ; enfin on utilise le paquet existentiel ouvert.

$$\mathsf{unpack}\ M\ \mathsf{as}\ \langle\alpha, x\rangle\ \mathsf{in}\ M' \triangleq \nu\alpha.\,(\mathsf{let}\ x = \mathsf{open}\ \langle\alpha\rangle\ M\ \mathsf{in}\ M')$$

Le découpage de pack se présente ainsi : on crée d'abord un type existentiel de témoin inconnu β avec la primitive $\exists$; puis on définit son témoin τ à l'aide de la primitive Σ ; enfin on utilise cette définition pour cacher le témoin dans le type de retour à l'aide d'une coercion, qui remplace les occurrences de τ par α.

$$\mathsf{pack}\ \langle\tau, M\rangle\ \mathsf{as}\ \exists\alpha.\,\tau' \triangleq \exists\beta.\,\Sigma\,\langle\beta\rangle\,(\alpha = \tau)\,(M : \tau') \quad \text{if } \alpha \notin ftv(M)$$

Il convient d'ajouter certaines restrictions, sans quoi il serait possible de cacher plusieurs témoins différents sous le même nom et casserait la sûreté. Nous traitons alors les noms des témoins de manière *linéaire*, en découpant les contextes de typage d'une manière similaire à celle de la logique linéaire.

Notre langage est équipé d'une sémantique à réduction en appel par valeur dont les règles principales sont la β-réduction et l'*extrusion* des Σ. Cette dernière est imposée par l'invariant de linéarité.

Propriétés Nous prouvons que notre système de types est sûr (via préservation et progrès). De plus, nous donnons deux encodages depuis et vers Système F et démontrons qu'ils préservent typage et sémantique. Ainsi F et $\mathsf{F}^{\curlyvee}$ ont les mêmes types habités et font les mêmes calculs.

Extensions Nous proposons pour $\mathsf{F}^{\curlyvee}$ quelques extensions naturelles : valeurs récursives, types récursifs, équations mutuellement récursives entre types abstraits.

Travaux futurs Ce travail n'est qu'une première étape vers une programmation modulaire avec Système F. Le langage $\mathsf{F}^{\curlyvee}$ ne permet *que* de lever la contrainte de structure des programmes faisant usage des existentiels. D'autres aspects sont nécessaires pour rendre les programmes plus concis : d'abord l'utilisation de sortes singletons semble appropriée pour modéliser les définitions de types et fournit une théorie équationnelle puissante ; ensuite, une forme d'inférence partielle pour les existentiels et pour les universels est aussi indispensable. L'étude des propriétés de $\mathsf{F}^{\curlyvee}$ mérite d'être poursuivie : l'extension des résultats de parametricité et de préservation de l'abstraction de Système F vers $\mathsf{F}^{\curlyvee}$ est considérée.

Nota Bene Pour une version longue, le lecteur est invité à consulter [6].

Références

1. Derek Dreyer. Recursive type generativity. *Journal of Functional Programming*, pages 433–471, 2007.
2. Xavier Leroy. A syntactic theory of type generativity and sharing. *Journal of Functional Programming*, 6(5) :667–698, 1996.
3. Mark Lillibridge. *Translucent Sums : A Foundation for Higher-Order Module Systems*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997.
4. David B. MacQueen. Using dependent types to express modular structure. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 277–286, New York, NY, USA, 1986. ACM.
5. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Trans. Program. Lang. Syst.*, 10(3) :470–502, 1988.
6. Benoît Montagu and Didier Rémy. Modeling abstract types in modules with open existential types. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, January 2009.
7. Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proceedings of European Conference on Object-Oriented Programming*, pages 201–224, 2003.
8. Andreas Rossberg, Claudio Russo, and Derek Dreyer. F-ing modules. In *Proceedings of Types in Language Design and Implementation (TLDI)*, Madrid, Spain, 2010.
9. Claudio V. Russo. Types for modules. *Electronic Notes in Theoretical Computer Science*, 60, January 2003.

Lucy-n : une extension n-synchrone de Lustre *

Louis Mandel Florence Plateau Marc Pouzet

LRI, Université Paris-Sud 11
INRIA Saclay
{mandel,plateau,pouzet}@lri.fr

Nous nous intéressons aux modèles et aux langages pour la programmation d'applications de traitement de flux ayant des contraintes de temps-réel comme les applications multimédias. Dans le cas d'une application vidéo par exemple, il faut traiter des flots infinis de pixels sur lesquels sont appliquées des opérations successives, ou *filtres*. Il y a des contraintes de temps-réel car la fréquence d'affichage des pixels ne peut être diminuée. Il y a également des contraintes de performance car des milliards d'opérations doivent être effectuées par seconde. Enfin, il y a des contraintes de sûreté : on veut garantir que le comportement de l'application est déterministe, sans blocage et que la mémoire nécessaire à l'exécution est bornée.

Afin de respecter ces contraintes, les ingénieurs du domaine sont habitués à modéliser leurs systèmes par des réseaux de processus de Kahn [7]. Dans ce modèle, des nœuds de calcul s'exécutent de manière concurrente et communiquent par des canaux munis de *mémoires tampon* (ou *buffers*) infinies de type *First In First Out*. La lecture est bloquante si le buffer est vide, et comme les buffers sont de taille infinie, l'écriture est non bloquante. On ne peut pas tester l'absence de valeur dans un buffer.

L'intérêt ce modèle de programmation est d'être concurrent tout en préservant le déterminisme. De plus, il peut être vu comme un modèle mathématique où chaque nœud correspond à une fonction sur les suites et le réseau correspond à la composition de ces fonctions, qui est aussi une fonction de suites. On sait ainsi raisonner de manière formelle sur les systèmes décrits dans ce formalisme. Néanmoins, l'inconvénient de ce modèle est qu'il ne donne pas de garanties sur l'exécution en temps-réel, l'absence de blocage et le caractère borné de la mémoire nécessaire à la communication.

Il est possible d'implémenter des réseaux de Kahn sans prendre le risque d'écritures dans des buffers pleins en mettant en place un mécanisme de rétroaction (ou *back pressure*). Ce mécanisme rend l'écriture bloquante. Ainsi, il permet d'éviter les débordements de capacité des buffers, mais il crée des blocages artificiels dans le cas où les tailles des buffers ont été sous-estimées. Ce problème peut être traité en augmentant les tailles durant l'exécution en cas de blocage dû à un manque de place [12]. Cependant, on aimerait avoir des garanties statiques sur l'exécution. Pour cela, il faut savoir calculer des tailles suffisantes pour les buffers si elles existent, et rejeter les réseaux qui nécessitent des buffers de taille infinie.

Le problème de savoir si un réseau de Kahn quelconque peut être exécuté en mémoire bornée est indécidable [2]. Pour avoir des garanties sur le caractère borné des buffers nécessaires, il faut donc se placer dans un sous-ensemble des réseaux de Kahn sur lesquels des informations supplémentaires sont disponibles. C'est l'approche choisie par le modèle des *Synchronous Dataflow* [8, 11] qui se restreignent à des réseaux de Kahn dans lesquels le nombre de valeurs produites et consommées par chaque nœud à chaque activation est connu statiquement. Cela permet de calculer un ordonnancement statique périodique et une taille suffisante pour les buffers pour cet ordonnancement.

L'approche consistant à se placer dans un sous-ensemble des réseaux de Kahn est aussi l'approche suivie par le modèle de programmation synchrone [1]. Dans ce modèle, on ne considère que les réseaux qui peuvent être exécutés sans buffers. Pour cela, chaque canal est associé à une *horloge* qui définit les instants où une valeur est présente sur le canal.

L'intérêt des langages synchrones est de permettre de spécifier formellement une application temps-réel tout en étant compilables vers du code exécutable. Ils expriment la concurrence des différents trai-

*Nous présentons ici un résumé étendu de notre article publié aux vingt et unièmes Journées Francophones des Langages Applicatifs [10].

tements, et sont donc bien adaptés à une compilation permettant d'exécuter parallèlement des filtres. Ils offrent des garanties fortes, comme le déterminisme, l'absence de blocage et un besoin en mémoire borné. La contrepartie des garanties fournies malgré une grande expressivité des rythmes est un manque de souplesse dans la composition des flots. La composition sans buffer n'est acceptée que si le calcul d'horloges sait vérifier l'égalité des horloges des flots que l'on compose.

La vérification des horloges en Lustre [3] et Lucid Synchrone [13] est un problème de typage et peut être décrite par un système de types [4, 6]. Chaque expression est associée à un type d'horloges et le compilateur vérifie que le programme respecte des contraintes sur ces types. Considérons par exemple la règle suivante :

$$\frac{H \vdash e_1 : ck \quad H \vdash e_2 : ck}{H \vdash e_1 + e_2 : ck}$$

Elle indique qu'une addition de deux flots e_1 et e_2 est correcte à condition que e_1 et e_2 aient le même type d'horloges ck. Dans ce cas, le type d'horloges du résultat $e_1 + e_2$ est aussi ck. Ainsi, pour typer un programme, les questions qui se posent concernent toutes l'égalité de types. L'idée du modèle n-synchrone [5] est d'assouplir cette contrainte d'égalité afin de permettre la composition de flots non nécessairement synchrones dès lors que leurs horloges sont proches l'une de l'autre. Si par insertion d'un buffer borné, un flot x dont l'horloge est de type ck peut être consommé un peu plus tard sur une horloge de type ck', on dira que ck est un sous-type de ck'. Cette relation sera notée $ck \ \mathbf{<:} \ ck'$. On étend alors le langage avec une construction `buffer` qui indique les points où il faut appliquer la règle de sous-typage :

$$\frac{H \vdash e : ck \quad ck \ \mathbf{<:} \ ck'}{H \vdash \texttt{buffer}\ e : ck'}$$

Le langage Lucy-n que nous proposons, est un langage synchrone flot de données semblable à Lustre, étendu avec l'opérateur de bufferisation. Par ailleurs, les expressions d'horloges (ce) peuvent être définies avec tout langage d'horloges s'évaluant vers des séquences booléennes infinies et muni des opérations suivantes :

- un test d'égalité des horloges, pour vérifier que les communications réalisées sans buffer sont synchrones ;
- un test d'*adaptabilité* des horloges, pour vérifier que les communications réalisées à travers un buffer sont n-synchrones ;
- une opération permettant de calculer une taille suffisante pour chaque buffer.

En Lustre, les expressions d'horloges peuvent être définies par n'importe quelle expression booléenne du langage. Ces expressions pouvant être arbitrairement compliquées, le test d'égalité des horloges se limite à de l'égalité syntaxique. Dans ce cadre, on peut difficilement imaginer un test d'adaptabilité et un calcul des tailles de buffers. On doit donc utiliser un langage d'horloges plus simple pour pouvoir utiliser des communications par buffers.

Dans l'article [10], nous présentons Lucy-n en limitant les expressions d'horloges à des mots binaires ultimement périodiques. Nous définissons ensuite le système de types qui vérifie que les types d'horloges sont égaux lorsque les communications se font sans buffers et collecte des contraintes de sous-typage lorsque les communications se font avec des buffers. Puis, nous montrons comment résoudre les contraintes de sous-typage en utilisant les techniques d'abstraction présentées dans [9]. Enfin, nous terminons l'article par un exemple d'application vidéo programmé en Lucy-n.

Références

[1] A. Benveniste, P. Caspi, S.A. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), January 2003.

[2] J. Buck. *Scheduling Dynamic Dataflow Graphs with Bounded Memory Using the Token Flow Model.* PhD thesis, EECS Department, University of California, Berkeley, 1993.

[3] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. Lustre: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 178–188. ACM, 1987.

[4] Paul Caspi and Marc Pouzet. Synchronous Kahn Networks. In *ACM SIGPLAN International Conference on Functional Programming*, May 1996.

[5] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. *N*-Synchronous Kahn Networks: a Relaxed Model of Synchrony for Real-Time Systems. In *ACM International Conference on Principles of Programming Languages*, January 2006.

[6] Jean-Louis Colaço and Marc Pouzet. Clocks as First Class Abstract Types. In *Third International Conference on Embedded Software (EMSOFT'03)*, October 2003.

[7] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, August 1974.

[8] E. Lee and D. Messerschmitt. Synchronous dataflow. *IEEE Trans. Comput.*, 75(9), 1987.

[9] L. Mandel and F. Plateau. Abstraction d'horloges dans les systèmes synchrones flot de données. In *Vingtièmes Journées Francophones des Langages Applicatifs*, February 2009.

[10] L. Mandel, F. Plateau, and M. Pouzet. Lucy-n : une extension n-synchrone de Lustre. In *Vingt-et-unièmes Journées Francophones des Langages Applicatifs*, January 2010.

[11] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cycle-static dataflow. In *Proceedings of the 29th Asilomar Conference on Signals, Systems and Computers*, page 204, Washington, DC, USA, 1995. IEEE Computer Society.

[12] Thomas Martyn Parks. *Bounded scheduling of process networks*. PhD thesis, EECS Department, University of California, Berkeley, Berkeley, CA, USA, 1995.

[13] Marc Pouzet. *Lucid Synchrone, version 3. Tutorial and reference manual*. Université Paris-Sud, LRI, April 2006. Distribution available at: www.lri.fr/~pouzet/lucid-synchrone.

A3PAT, an Approach for Certified Automated Termination Proofs (extended abstract)*

Évelyne Contejean[1], Pierre Courtieu[2], Julien Forest[3], Andrei Paskevich[1], Olivier Pons[2], and Xavier Urbain[1,3]

[1] LRI, Univ Paris-Sud, CNRS, Orsay, F-91405, France
[2] Cédric, CNAM, Paris, F-75141, France
[3] Cédric, ENSIIE, Évry, F-91025, France

1 Introduction

Verification of programs and specifications may involve a significant amount of formal methods to guarantee required properties for complex or critical systems. In the context of proving, users rely on proof assistants with which they interact, step by step, until the proof is totally and mechanically verified. However, formal proof may be costly, and as proof assistants often lack automation, there is an increasing need to use fully automated tools providing powerful (and intricate) decision procedures. Although some proof assistants can check the soundness of a proof, the results of automated provers are often taken *as is*, even if the provers may be subject to bugs. Since application fields include possibly critical sectors such as security, code verification, cryptographic protocols, etc., reliance on verification tools is a crucial issue.

Automated provers and proof assistants do not mix easily. One of the strengths of skeptical proof assistants like Coq [6] or Isabelle/HOL [5] is a *highly reliable* procedure that checks the soundness of proofs. In particular, they need to check mechanically the proof of *each* property used. Certified-programming environments based on these proof assistants find this an additional guarantee. However, this means that the proof assistant has to check a property proven by an external procedure before accepting it. Therefore, such a procedure must return a *proof trace* that is verifiable by the assistant.

The A3PAT project (`http://a3pat.ensiie.fr`) aims to bridge the gap between proof assistants yielding formal guarantees of reliability, and highly automated tools one has to trust. We want to provide both enhanced automation for proof assistants (by proof delegation), and formal certificates for automatically generated proofs (by mechanical checking of proof traces).

In the framework of first-order *term rewriting* techniques, which has proven to be most useful in programming, specification and proof automation, we focus on the proofs of *termination*: the fundamental property of a program any execution of which yields a result.

This work presents contributions of different natures. We describe a methodology for the important challenge of automatically generating proof traces for first-order term rewriting techniques, and in particular for termination proofs. Then we propose two improvements regarding termination/certification techniques: a full formalisation and an extension of an efficient termination criterion, and an alternate approach for dealing with graphs.

We claim that demanding computations, *if their results can be checked easily*, should not be handled by a proof assistant alone, but rather delegated to satellite tools. The solution we propose in this work is based on a termination engine and trace generator (C*i*ME3), and a very general formal library on rewriting (Coccinelle). The trace generator can act as a compiler for proofs traces (submitted as `xml` files) from other termination engines. To this goal, a common proof format is defined in collaboration with the CeTA/IsaFoR [7] team (`http://cl-informatik.uibk.ac.at/software/cpf/`). The trace generator outputs Coq scripts that refer to the formal library Coccinelle. The Coq proof assistant can then check the termination proofs, using definitions and theorems in Coccinelle.

* Work partially supported by the French ANR (ANR-05-BLAN-0146)

An interesting benefit of abstract formalisations is that keeping them as general as possible may lead to relaxing premises of theorems. A first example of this is our extended notion of matrix interpretations [3]. In the present work, we give the first full formalisation in Coq of the powerful *subterm criterion* [4] for termination of rewriting systems[4], and we propose an extension thereof [1]. Our extended subterm criterion is as follows. First of all, we assign to every symbol f a integer $p(f)$. Then we define a projection operation π over non-variable terms as:

$$\begin{array}{ll} \pi(f(t_1,\ldots,t_n)) = t_{p(f)} & \text{if } 1 \leq p(f) \leq n \\ \pi(f(t_1,\ldots,t_n)) = f(t_1,\ldots,t_n) & \text{otherwise} \end{array}$$

Theorem 1. *Let R be a term rewriting system and $D = D' \cup \langle u_0, v_0 \rangle$ be a set of dependency pairs of R. If there exists a projection π as above such that:*

- *$\pi(v_0)$ is a proper subterm of v_0 ;*
- *$\pi(u_0)(\to_R \cup \rhd)^+ \pi(v_0)$;*
- *for every pair $\langle u, v \rangle$ of D' that is strongly connected to $\langle u_0, v_0 \rangle$, $\pi(u)(\to_R \cup \rhd)^\star \pi(v)$;*

then, $\mathsf{SN}(\twoheadrightarrow_{D',R}|_{\mathsf{min}(R)})$ implies $\mathsf{SN}(\twoheadrightarrow_{D,R}|_{\mathsf{min}(R)})$.

This theorem is stronger than the original subterm criterion [4] on two accounts: firstly, we allow a term to be projected to itself, and secondly, we consider the relation $\to_R \cup \rhd$ instead of $\rhd$. The extended subterm criterion is implemented in C*i*ME, and Theorem 1 is completely formalised in Coccinelle. Certified proofs of termination involving this criterion can be obtained with full automation using our approach.

Another contribution of the present work addresses graph analysis. In [2], we allowed certification of (termination) proofs based on a graph analysis, and could manage efficiently graphs containing thousands of arcs. Here, we propose a new formalisation for graphs that, in contrast to [2], relies thoroughly on generic theorems in Coccinelle (and not to *ad-hoc* proofs produced by the previous approach). This new formalisation allows proofs to be more local, while keeping an implicit representation of graphs[4].

All tools and libraries developed in the project are freely available from the A3PAT webpage: `http://a3pat.ensiie.fr`.

References

1. É. Contejean, P. Courtieu, J. Forest, A. Paskevich, O. Pons, and X. Urbain. A3PAT, an Approach for Certified Automated Termination Proofs. In *Proceedings of PEPM'10, ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 63–72. ACM, 2010.
2. P. Courtieu, J. Forest, and X. Urbain. Certifying a Termination Criterion Based on Graphs, Without Graphs. In C. Muñoz and O. Ait Mohamed, editors, *21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'08)*, volume 5170 of *Lecture Notes in Computer Science*, pages 183–198, Montréal, Canada, Aug. 2008. Springer-Verlag.
3. P. Courtieu, G. Gbedo, and O. Pons. Improved matrix interpretations. In J. van Leeuwen et al., editor, *Proceedings of SOFSEM2010, International Conference on Current Trends in Theory and Practice of Computer Science*, Lecture Notes in Computer Science. Springer, 2010. To appear.
4. N. Hirokawa and A. Middeldorp. Tyrolean Termination Tool: Techniques and Features. *Information and Computation*, 205(4):474–511, 2007.
5. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
6. The Coq Development Team. *The Coq Proof Assistant Documentation – Version V8.2*, June 2008.
7. R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTa. In T. Nipkow and C. Urban, editors, *22st International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468, Munich, Germany, Aug. 2009. Springer-Verlag.

[4] Another formalisation by the IsaFoR team was obtained independently and roughly at the same time for Isabelle/HOL.

Session du groupe de travail MFDL

Méthodes Formelles dans le Développement du Logiciel

Application du Model Checking aux commandes de vol : l'expérience Airbus

Thomas Bochot[1,2], Pierre Virelizier[1], Hélène Waeselynck[3], Virginie Wiels[2]

[1] AIRBUS France, 316 route de Bayonne, 31060 Toulouse Cedex 03, France
[2] ONERA/DTIM, 2 avenue Edouard Belin, 31055 Toulouse, France
[3] LAAS-CNRS, Université de Toulouse, 7 avenue du Colonel Roche, F-31077 Toulouse, France
Email : prénom.nom@airbus.com, @laas.fr, @onera.fr

Résumé

Cet article présente des études réalisées par Airbus sur l'utilisation du model checking pour la vérification de systèmes critiques, les leçons que l'on peut en tirer et les moyens pour étendre l'utilisation industrielle de la vérification formelle au niveau modèle.

1. Introduction

Le système de commande de vol (CDV) est un des systèmes les plus critiques à bord d'un avion et comme les autres systèmes embarqués, sa taille et sa complexité ne cessent d'augmenter au fil des programmes. La vérification et la validation (V&V) d'un tel système sont essentielles, mais difficiles. Les méthodes classiques de V&V sont la simulation et le test, mais les méthodes formelles constituent maintenant une solution crédible en complément des méthodes classiques. Cet article présente les expériences effectuées par Airbus sur l'application du model checking aux CDV, les leçons que l'on peut en tirer, et les perspectives qui en découlent.

La section 2 décrit brièvement le processus de développement du système de CDV d'Airbus ; la section 3 synthétise les résultats de trois études R&D passées ; la section 4 présente une expérimentation menée avec succès sur la fonction Ground Spoiler (une partie des CDV) ; la section 5 discute des leçons tirées de cette étude ; la section 6 conclut et propose des pistes pour des travaux futurs.

2. Processus de développement Airbus

La figure 1 synthétise le cycle de développement des CDV au sein d'Airbus. Ce cycle est découpé en trois niveaux : avion, système et équipement. Les systèmes les plus critiques comme les systèmes de CDV sont conçus en utilisant le langage formel SCADE [1]. À partir des modèles SCADE, un générateur de code produit automatiquement la majeure partie du code embarqué. Certaines vérifications pour le code généré peuvent donc être effectuées au niveau de la conception SCADE. Les principales activités de V&V réalisées par Airbus sont les suivantes :

- **Tests de modèle:** le système considéré est validé dans un environnement simulé. Ces tests sont effectués sur des simulateurs de bureau fournissant notamment un ensemble de commandes pour représenter les actions du pilote. Les testeurs définissent des scénarios de test à partir d'exigences fonctionnelles textuelles détaillées. Les scénarios sont exécutés sur le simulateur ; les testeurs décident ensuite si le test est correct ou non.
- **Simulations de niveau avion:** plusieurs systèmes sont validés dans un environnement simulé.
- **Vérification formelle sur le code:** Airbus utilise des outils basés sur l'interprétation abstraite pour vérifier des propriétés non fonctionnelles sur les programmes (telles que l'absence d'erreur à l'exécution) et utilise des outils basés sur la preuve pour vérifier des propriétés fonctionnelles sur les programmes écrits manuellement. Ces utilisations des méthodes formelles ont été couronnées de succès [2] et des extensions sont en cours d'étude. Elles ne sont pas présentées dans cet article ;

nous nous plaçons au niveau système et considérons l'utilisation d'un autre type de technique : le model checking.

- **Tests d'intégration** du logiciel : à l'issue desquels le logiciel est installé dans les calculateurs.
- **Essais labo:** premiers tests sur les équipements réels, ciblant un système unique ou l'intégration de plusieurs systèmes.
- **Essais au sol et essais en vol :** sur des avions prototypes équipés d'instruments dédiés aux essais.

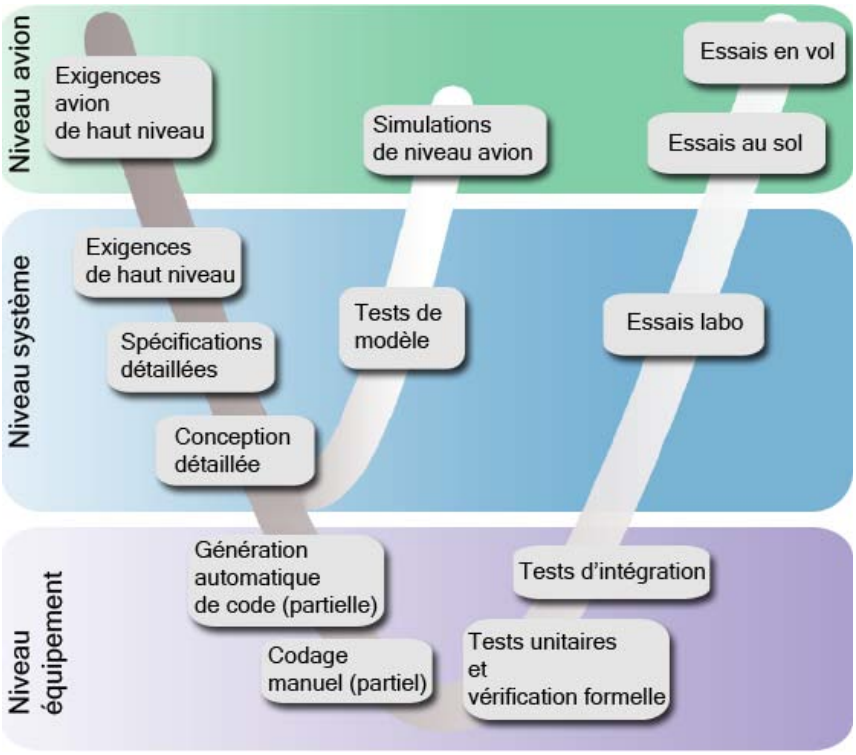


Figure 1: Processus de développement Airbus

Cet article se concentre sur la vérification au niveau système, actuellement réalisée par des tests sur les modèles de conception. Comme la conception est réalisée avec le langage formel SCADE, nous avons étudié la possibilité d'utiliser des analyses formelles pour une partie de la vérification. Il faut noter que la modélisation SCADE est une conception détaillée à partir de laquelle le code embarqué est généré. Les modèles cibles sont donc plus complexes que des modèles qui seraient définis spécifiquement pour des besoins de vérification formelle comme dans [3].

Techniquement, un modèle SCADE est composé de planches [1]. Chaque planche contient un petit nombre de noeuds. Un noeud est une fonction unitaire qui calcule des valeurs de sortie à partir de valeurs d'entrée.

3. Études précédentes

Cela fait maintenant plusieurs années qu'Airbus mène des expérimentations sur la vérification formelle des CDV. Trois études sont mentionnées ici, parmi lesquelles une seule (la première) a fait l'objet d'une publication passée. Dans chacune de ces études, l'objectif est de vérifier que la conception SCADE est conforme à ses spécifications, en utilisant la méthode de l'observateur synchrone [4].

3.1 Première étude

Cette première étude [5], conduite en collaboration avec l'ONERA, avait pour but d'évaluer la faisabilité d'une approche formelle pour vérifier la conception détaillée du calculateur secondaire de commande de vol de l'A340 500-600. Les exigences que l'on cherchait à vérifier étaient des exigences haut niveau. Les outils utilisés durant les expérimentations étaient NP-TOOLS [6] et Lesar [7] :

- NP-TOOLS (de Prover Technology AB) est un outil de vérification pour les circuits combinatoires basé sur la méthode de Stålmarck.
- Lesar est un outil de « model checking » académique qui analyse les états atteignables.

Les résultats expérimentaux ont fait ressortir des limitations techniques liées aux calculs numériques. Cependant, les résultats obtenus sur les aspects booléens de la modélisation étaient encourageants. À tel point qu'à la suite de cette étude, Esterel Technologies et Prover Technology ont décidé d'intégrer une fonctionnalité de model checking (Prover Plug-in) à l'environnement de conception SCADE. L'outil obtenu se nomme SCADE Design Verifier.

3.2 Seconde étude

Cette seconde étude avait pour objectif d'évaluer les capacités de l'outil Scade Design Verifier à vérifier différents types de propriétés couramment rencontrées dans les CDV. Le calculateur secondaire des commandes de vol de l'A340 500-600 a fait l'objet de douze expérimentations. Le tableau 1 synthétise ces expérimentations qui permettent de couvrir différentes facettes des systèmes de commande de vol :

- Multi processeurs : les CDV Airbus utilisent des architectures multiprocesseurs asynchrones, tolérantes aux fautes.
- Dans un processeur donné, différentes parties logicielles sont exécutées à différents cycles temporels (selon un séquencement prédéterminé).
- Calculs numériques : certaines parties des modèles contiennent des calculs numériques complexes sur des entiers ou des flottants (par exemple, modélisation d'un filtre du second ordre).

Tableau 1. Résultats de la seconde étude

#	Multi processeurs	Multi horloges	Calculs numériques	Terminaison de l'analyse
1	✓			✓
2	✓	✓		✓
3				✓
4			✓	
5		✓		✓
6	✓	✓	✓	
7	✓			✓
8				✓
9				✓
10	✓	✓		
11			✓	✓
12			✓	

Lorsqu'aucune des trois colonnes du milieu n'est cochée, cela signifie que l'expérimentation porte sur une fonction booléenne réalisée sur un unique processeur, et mettant en jeu une unique horloge. La colonne de droite indique si l'outil de model checking a été capable de terminer son analyse (que la propriété ait été jugée valide ou invalide).

Les expérimentations sur les aspects multiprocesseurs ont fait ressortir des problèmes de modélisation. Deux phénomènes ont dû être modélisés : les délais de communication et l'asynchronisme des horloges entre les deux processeurs communicants. Ces deux

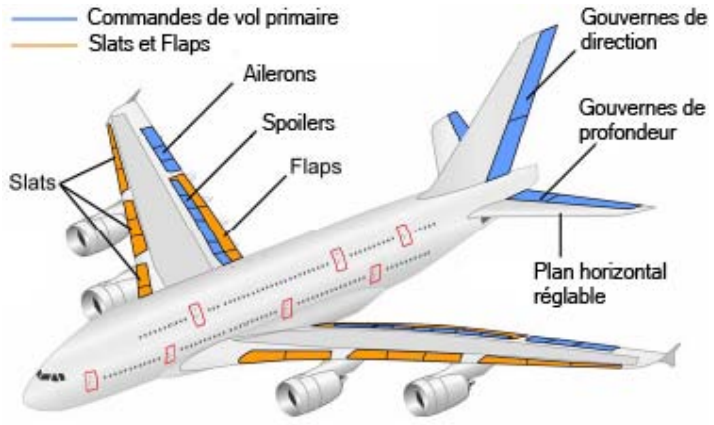


Figure 2: Surfaces de contrôle de l'A380

phénomènes augmentent la complexité du modèle de manière significative. Les expérimentations 1, 2, 7 ont réussi car elles utilisaient un modèle simplifié. L'expérimentation 10, en revanche, a été un échec : on observe une explosion combinatoire due à une prise en compte plus fine des phénomènes asynchrones mentionnés plus haut. Il existe donc un compromis entre précision du modèle et faisabilité de la vérification. Il est important de remarquer que SCADE n'est pas dédié à la modélisation de comportements asynchrones.

Une étape de modélisation manuelle est également nécessaire lorsque l'on traite un ensemble de planches SCADE impliquant plusieurs horloges. Cette étape, qui s'avère fastidieuse à mettre en œuvre, a été automatisée. La vérification de propriétés de sûreté n'a pas posé de problème particulier. Par exemple, l'expérimentation 5 a pu aboutir à une conclusion pour l'analyse d'un modèle composé de 100 planches SCADE utilisant quatre horloges différentes.

Concernant les problèmes numériques, l'expérimentation 11 a abouti à une conclusion en analysant un problème contenant des nombres à virgule flottante. Ce résultat est plutôt satisfaisant, comparé aux médiocres performances constatées lors de la première étude. Il reflète les améliorations apportées à l'outil de model checking entre ces deux études, pour mieux prendre en compte les traitements numériques. Cependant, l'analyse d'équations non linéaires reste hors de portée, menant les expérimentations 4, 6 et 12 à l'échec.

La conclusion de cette étude est que le model checking peut être utilisé sur beaucoup de fonctions d'un système de CDV, pourvu qu'elles n'impliquent pas de calculs numériques trop complexes ni de communication inter processeurs.

3.3 Troisième étude

Suite aux résultats de l'étude précédente, Airbus a décidé d'étudier une utilisation massive du model checking pour vérifier des exigences système de bas niveau, telles que décrites dans les spécifications détaillées. Pour évaluer l'approche, il a été décidé de comparer le « retour sur investissement » entre une approche de vérification par test et une approche de vérification par model checking. Un total de 135 expérimentations ont eu lieu portant sur trois fonctions du système de commande de vol primaire de l'A400M.

L'étude conclut que le taux de détection d'erreur du model checking n'est pas plus élevé que celui du test et que, de plus, son temps de mise en œuvre est deux à trois fois plus élevé. Une raison de ce faible taux de détection d'erreur peut provenir du fait que les exigences vérifiées sont de bas niveau. Les propriétés qui en découlent sont très souvent proches et en quelque sorte similaires à la spécification détaillée. Le pouvoir de détection d'erreur semble décroître à mesure que les propriétés sont syntaxiquement proches du modèle. Donc, il en ressort que, dans le processus de développement Airbus, le model checking n'est pas une approche efficace pour vérifier les exigences de bas niveau, dans la conception SCADE.

4. Application du Model checking à la fonction ground spoiler

Nous allons maintenant parler d'une étude plus récente qui traite de la fonction ground spoiler du programme A380. Dans cette section, nous présentons brièvement les expérimentations effectuées, une présentation plus détaillée étant exclue pour des raisons de confidentialité. Les résultats obtenus seront commentés ultérieurement, en section 5.

La fonction ground spoiler a pour but de garder l'avion au sol à l'atterrissage ou en cas d'arrêt d'urgence lors de la phase de décollage (*rejected take-off*). Les ailerons et les spoilers (fig. 2) sont utilisés pour casser la portance de l'aile, ce qui plaque l'avion au sol et augmente l'efficacité du freinage. De plus, le braquage de ces surfaces provoque un effet d'aerofreinage. La fonction est réalisée de telle manière qu'une activation intempestive soit « extrêmement improbable » lorsque l'avion est en vol.

Le cas d'étude est un exemple dans lequel une analyse formelle pourrait compléter utilement la validation actuelle de la conception SCADE. La propriété que l'on souhaite vérifier est une propriété haut niveau de sécurité qui stipule que la fonction ground spoiler ne doit pas s'activer lorsque l'avion est en vol. Nous disposons de deux modélisations SCADE de la fonction : une version erronée et une corrigée. La première contient une faute de conception n'ayant pas été détectée lors des tests de niveau modèle. La faute a été révélée lors des essais labo. Notre but est de démontrer que cette faute aurait pu être trouvée plus tôt par une approche de model-checking, ce qui aurait été moins coûteux. Nous voulons aussi démontrer que la version corrigée satisfait l'exigence de sécurité.

La fonction ground spoiler est une bonne candidate à l'application du model checking. Elle est en grande partie basée sur du raisonnement booléen. Les calculs numériques y sont peu nombreux et peu complexes (additions et comparaisons de valeurs réelles). Cependant, la présence de compteurs temporels peut être une source de problème, du fait du nombre de cycles que l'outil a besoin de considérer pour réaliser l'analyse.

Compte tenu des études passées (§3), le contexte de la vérification formelle est délibérément réduit. Nous allons nous focaliser sur un petit nombre de planches SCADE extraites de la conception complète. Nous exclurons les aspects multiprocesseurs et focaliserons l'analyse sur le fonctionnement nominal de la fonction : pas de reset et pas de panne de capteur.

L'analyse de la version erronée a démarré en incluant une seule planche SCADE, celle qui calcule l'ordre final envoyé aux surfaces de contrôle. L'analyse a nécessité plusieurs étapes de vérification :

- Un certain nombre d'itérations initiales ont été nécessaires pour obtenir un observateur adapté au problème de vérification. Nous avons expérimenté différentes formulations de la propriété de sûreté et des hypothèses restreignant le contexte de vérification, avant d'en retenir une.
- Une fois la formalisation établie, nous obtenons un contre-exemple qui n'est pas simple à interpréter. La violation de la propriété a été causée par l'absence de reset d'une bascule. La condition de reset (RCond) est maintenue à faux. Pour comprendre si ce comportement est possible, nous avons dû étendre l'analyse pour inclure la planche Scade produisant RCond. Nous avons donc modifié l'observateur en conséquence. Le modèle analysé est maintenant composé de deux planches SCADE.
- L'analyse formelle du modèle étendu a confirmé le contre-exemple précédemment obtenu.
- Nous avons décidé de poursuivre notre recherche d'erreur. Comme nous ne voulions pas modifier les planches SCADE, nous avons modifié l'expression de la propriété de telle manière à exclure le contre-exemple précédent. L'analyse formelle renvoie un nouveau contre-exemple avec RCond maintenu à vrai.

Les contre-exemples correspondent à des scénarios opérationnels différents. Dans les deux cas, un problème de reset de bascule se produit. Mais, chacun de ces problèmes résulte d'une trajectoire spécifique (inhabituelle) de l'avion combinée à des actions pilote. Le second scénario est celui qui avait été trouvé tardivement par les essais labo. Le premier est une manifestation alternative de la même faute de conception.

La version corrigée de la fonction ground spoiler emploie une logique interne différente qui n'utilise plus RCond. Pour vérifier cette logique, nous avons réutilisé le même observateur que celui

obtenu à la fin de l'étape 1, avec de légères adaptations pour prendre en compte le changement d'interface de la fonction. Les premières tentatives de vérification ont été des échecs (l'outil de model checking ne parvenait pas à terminer son calcul). Après l'introduction d'hypothèses sur des variables intermédiaires, l'analyse formelle a pu se conclure avec succès. Des vérifications dédiées nous ont ensuite permis de démontrer la validité des hypothèses précédemment introduites, complétant ainsi l'analyse formelle.

5. Les enseignements de cette étude

Après cette présentation factuelle, nous revenons sur ce qui, à notre avis, constitue les principaux enseignements de l'étude.

5.1. Efficacité de la vérification formelle

Bien que la fonction Ground Spoiler utilise peu de calculs numériques, elle est suffisamment complexe pour poser problème à l'outil de vérification, notamment à cause de la présence de compteurs temporels. Il a fallu 48 heures pour réaliser l'analyse de la version correcte (sur un Pentium 1.7-GHz avec 256 Mb de RAM). En ce qui concerne la version incorrecte, la production de contre-exemples a pris de quelques minutes à quelques heures, selon la longueur des contre-exemples et la stratégie d'exploration choisie (SCADE propose deux stratégies). Les contre-exemples produits avaient une longueur comprise entre 50 et 160 cycles.

D'un point de vue de la détection d'erreur, la vérification formelle a été très efficace. Rappelons que les tests du modèle n'avaient pas permis de révéler le problème. Ceci est dû au fait que les aspects très dynamiques ne sont pas la préoccupation principale de ces premiers tests : les trajectoires avion inhabituelles, telles que celles des scénarios qui violent la propriété, ne sont pas considérées à cette étape de validation. La dynamique du vol est traitée de façon approfondie par les tests labo (en utilisant le simulateur de vol) et bien sûr par les essais en vol. Cependant, plus les fautes de conception sont révélées tôt, moins leur correction coûte cher. Notre étude a montré que la vérification formelle peut offrir une solution pratique pour trouver de telles fautes lors de la conception détaillée. L'étude a également fourni de nombreuses informations sur le comportement erroné, en permettant d'identifier deux scénarios de violation de la propriété (correspondant à deux trajectoires d'avion différentes).

5.2. Expression des propriétés

Passer d'exigences en langage naturel à des énoncés formels de propriétés est connu comme un problème difficile. Dans notre cas, le problème est exacerbé par le fait que les modèles à vérifier sont exprimés à un niveau unitaire détaillé, tandis que les exigences informelles sont relatives à un niveau avion plus abstrait. Par conséquent, il n'est pas facile d'exprimer ces exigences en utilisant les variables concrètes des planches SCADE concernées.

Ce problème avait déjà été identifié dans la première étude mentionnée en section 3.1. Dans cette étude, un exemple de difficulté était l'expression formelle de la notion de manche « actif ». Informellement, le manche pilote ou copilote est actif s'il peut avoir un effet sur les surfaces de l'avion. Formaliser cette notion en termes d'entrées/sorties des planches SCADE s'était avéré délicat et plusieurs formulations alternatives avaient été étudiées. Dans l'étude Ground Spoiler, une difficulté a été de formaliser la notion d'avion « en vol ».

Un écueil serait de paraphraser le modèle SCADE (voir les conclusions de la troisième étude, section 3.3). Par exemple, la fonction Ground Spoiler calcule une variable intermédiaire pour savoir si l'avion est au sol ou pas. Si le même calcul est utilisé dans l'observateur, le modèle satisfera la propriété de façon triviale. Il a donc fallu trouver une formalisation indépendante.

Il est intéressant de noter que ce problème d'expression de propriétés est beaucoup moins aigu pour l'analyse des résultats de tests. Pour les tests sur le modèle, comme pour les tests labo, l'environnement de simulation fournit une référence externe pour déterminer quelle est la configuration « réelle » de l'avion (par opposition avec la configuration « perçue » par les fonctions

logicielles). En l'occurrence, déterminer si l'avion est en vol ne présente pas de difficulté lorsque l'on a accès aux données du simulateur.

5.3. Détermination des hypothèses

Déterminer un ensemble d'hypothèses qui permettent de restreindre le domaine de la vérification est un problème récurrent quelle que soit la méthode de vérification choisie.

Deux types d'hypothèses existent dans notre étude :

- Des hypothèses pour exclure des comportements irréalistes,
- Des hypothèses pour simplifier le problème de vérification et permettre l'analyse automatique.

En ce qui concerne la première catégorie, il n'est pas facile en pratique de différencier les comportements réalistes et irréalistes. Il y a une difficulté inhérente due à la prise en compte de la dynamique du vol. Alors que des modèles dynamiques complexes sont facilement intégrés dans les environnements de test, leur prise en compte par la vérification formelle n'est pas concevable : l'analyse automatique ne serait pas possible. De plus, des problèmes de performance nous obligent à nous concentrer sur un (petit) sous-ensemble de planches SCADE, ce qui signifie que les entrées considérées ne sont pas forcément les entrées du système. Par conséquent, nous ne prétendons pas caractériser le domaine des entrées valides de la fonction considérée. Les hypothèses permettent seulement d'éliminer des comportements grossièrement irréalistes (par exemple, nous ajoutons des domaines de valeur pour les données numériques ou nous exprimons des relations évidentes entre certaines entrées).

Au contraire des hypothèses précédentes, les hypothèses de simplification nous amènent à exclure délibérément certains comportements réalistes. Par exemple, nous avons mentionné que l'analyse ne considère qu'une seule unité de calcul dans des conditions nominales de fonctionnement : nous ne prenons pas en compte les erreurs de capteurs ou les resets. Un autre exemple concerne les trains d'atterrissage. Nous supposons que les roues sont soit toutes arrêtées soit toutes en rotation à la même vitesse. En réalité, les roues peuvent bien sûr avoir des comportements différents, mais nous avons jugé que ces considérations alourdiraient inutilement l'analyse.

Pour résumer, la vérification est faite sous des hypothèses qui concernent un sous-ensemble des comportements possibles sans pour autant exclure tous les comportements irréalistes. Un ensemble d'hypothèses donné peut toujours être remis en question ; un avantage de la formalisation est précisément de rendre ces hypothèses explicites. En règle générale, nous avons évité de mettre des hypothèses sur les actions du pilote : la satisfaction des exigences ne dépend donc pas des procédures opérationnelles de l'équipage.

5.4. Interprétation des contre-exemples

Si la vérification formelle se termine avec succès, on a démontré que la fonction respecte bien la propriété sous les hypothèses définies. Mais si un contre-exemple est trouvé, une analyse supplémentaire est nécessaire. Rappelons que la vérification est effectuée au niveau unitaire : les configurations d'entrée du contre-exemple doivent donc être interprétées au niveau système. De plus, les hypothèses de vérification ne sont pas suffisantes pour exclure tous les comportements irréalistes. Il faut donc déterminer si le contre-exemple correspond à un scénario avion réel, ce qui nécessite un effort significatif.

La situation est différente pour les tests. Toutes les catégories de test, y compris les tests de modèle, reposent sur des scénarios de test au niveau système qui ont une signification physique claire (par exemple un scénario d'atterrissage). L'analyse des résultats de test est donc facilitée. Par exemple, pour analyser les résultats des tests sur le modèle, les testeurs utilisent des chronogrammes qui visualisent les traces de test. L'interprétation de l'évolution des variables est guidée par la compréhension du scénario considéré.

Les chronogrammes ne s'avèrent pas aussi utiles pour analyser les contre-exemples. L'évolution de certaines variables semble incompréhensible d'un point de vue physique (par exemple, l'outil de vérification peut donner des valeurs arbitraires aux variables qui ne contribuent pas

directement à la violation de la propriété). Pourtant, malgré quelques détails irréalistes, le contre-exemple peut indiquer un problème réel dans la fonction considérée.

Comme les chronogrammes n'étaient pas très utiles, une autre approche a été adoptée. Elle a consisté à :

1. identifier les parties du modèle activées par le contre-exemple au cours du temps ;
2. essayer de trouver un scénario de niveau système qui reproduit le schéma d'activation observé.

La deuxième étape est la plus difficile et requiert de l'expertise dans le domaine avionique. La première étape, par contre, pourrait être automatisée à condition de préciser la notion de schéma d'activation.

5.5. Approche itérative de vérification

Du fait des problèmes évoqués ci-dessus (expression des propriétés et des hypothèses, interprétation des contre-exemples), l'analyse formelle de la conception SCADE est loin de constituer une approche "presse-bouton". Cela ressemble plutôt à une approche itérative, comme décrit sur la figure 3.

Des itérations sont nécessaires pour explorer des formulations alternatives du problème de vérification (aussi bien de la propriété que des hypothèses). Les étapes de vérification associées fournissent des retours utiles pour améliorer cette formulation. L'échec de l'analyse suggère éventuellement le besoin d'hypothèses supplémentaires. Les contre-exemples irréalistes mettent en cause la pertinence des hypothèses ou de la propriété exprimée. De façon générale, la figure 3 montre bien l'importance du jugement humain dans l'analyse des contre-exemples. Il faut remarquer que dans certains cas, la recherche d'une interprétation système du contre-exemple peut soulever des questions sur d'autres parties du modèle SCADE. L'intégration de ces parties dans le modèle analysé peut nécessiter la formulation de nouvelles hypothèses ou propriétés.

Étant donné un problème de vérification, le premier contre-exemple trouvé par l'outil n'est pas forcément le plus intéressant. Il peut représenter un comportement irréaliste que l'on n'a pas pu exclure par des hypothèses simples. L'analyse doit alors se poursuivre pour savoir si d'autres contre-exemples peuvent être trouvés, qui pourraient révéler une violation de la propriété. Ou bien, dans le cas où un problème réel est révélé, il peut être utile d'identifier différents scénarios dans lesquels ce problème se manifeste. Lors de l'étude, nous avons rencontré ces différents cas justifiant la recherche de contre-exemples supplémentaires. Malheureusement, SCADE Design Verifier (comme la plupart des model checkers) ne fournit pas plusieurs contre-exemples. Répéter la vérification produira

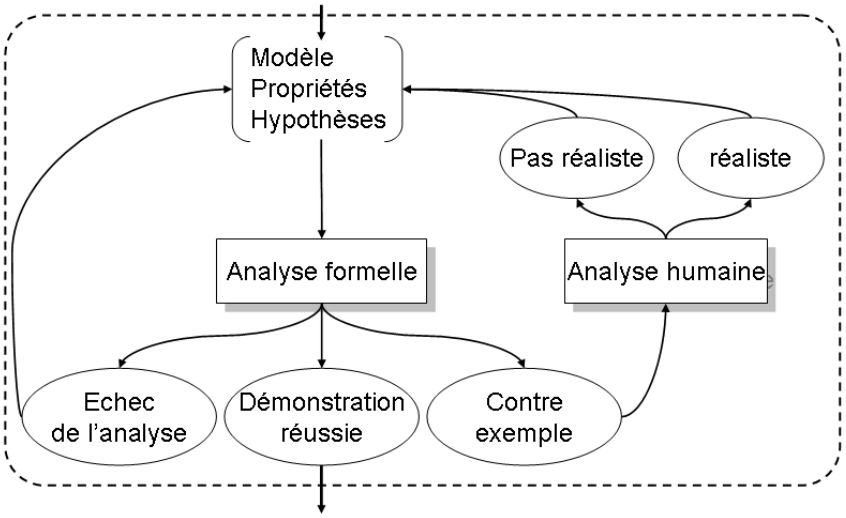


Figure 3: Méthodologie de l'analyse

toujours le même contre-exemple. Pour pallier ce problème, nous avons dû modifier manuellement l'observateur pour exclure le contre-exemple déjà trouvé. De cette façon, nous avons pu explorer des scénarios alternatifs de violation de la propriété. Nous avons finalement trouvé la faute révélée lors des essais labo et démontré qu'il existait plusieurs scénarios de violation associés à cette faute.

6. Conclusion

Les expérimentations décrites dans cet article ont pour cadre le processus de développement Airbus pour les systèmes de commande de vol. Ce cadre particularise la mise en œuvre du model-checking de la façon suivante :

- La formalisation intervient lors de la phase de conception système. Les modèles analysés sont donc très détaillés, et de niveau concret.
- Les fonctions étudiées s'exécutent dans un environnement multiprocesseurs et multihorloges (y compris au sein d'un même processeur).
- Certaines fonctions contiennent essentiellement de la logique booléenne, d'autres font appel à des calculs numériques qui peuvent être complexes.
- Les propriétés vérifiées peuvent être du même niveau d'abstraction que les modèles (spécifications détaillées) ou de niveau plus élevé (exigences de sécurité).

L'étude de la fonction Ground Spoiler fait suite à une série d'expérimentations qui avaient déjà permis de cerner le champ d'application du model-checking dans ce cadre spécifique. La fonction et la propriété ciblées rentraient a priori dans le champ d'application identifié. Outre la confirmation de la faisabilité de l'analyse formelle, l'objectif était de se confronter à un problème réel, qui a été révélé lors des essais labo.

Les résultats obtenus montrent que le model checking peut être très efficace pour trouver des fautes subtiles qui sont autrement révélées plus tard dans le cycle de développement. Nous pensons donc qu'une utilisation industrielle du model checking est possible et intéressante pour Airbus au niveau de la conception système. Pour le moment, il semble que l'utilisation la plus efficace soit une utilisation « poids plume » (*lightweight*) [10], où l'analyse formelle est appliquée à un petit nombre de fonctions critiques et se concentre sur un sous-ensemble de comportements possibles. Le model checking n'est pas encore prêt pour une utilisation plus étendue (telle que celle mise en oeuvre au niveau du code [2]).

Au-delà des retours sur l'applicabilité et l'efficacité du model-checking dans le cadre Airbus, les études menées permettent de pointer sur des besoins industriels en termes d'outillage.

Même dans le cas d'une utilisation poids plume, on peut rencontrer des problèmes de performances. Pour la fonction Ground Spoiler les premières tentatives de preuve ont échoué et la tentative finale a pris 48 heures environ. Il serait intéressant que l'outil soit capable de proposer des métriques pour indiquer la difficulté de la preuve pour un modèle et une propriété donnée. De telles métriques permettraient de guider la décision de tenter ou non une vérification formelle, et dans l'affirmative, de planifier l'attribution de ressources de calcul en adéquation avec la difficulté estimée.

Une autre conclusion importante de l'étude Ground Spoiler est que le model checking ne se résume à appuyer sur un bouton pour lancer une analyse automatique. Il s'agit plutôt d'une approche itérative, comme décrite sur la figure 3. De plus, le processus présenté en figure 3 va lui-même être répété plusieurs fois : même si la preuve se termine avec succès, l'utilisateur peut vouloir aller plus loin dans l'analyse en supprimant une hypothèse, en modifiant la propriété ou en élargissant le modèle analysé.

Les outils de vérification ne supportent pas ce processus itératif de façon suffisante. Il existe clairement un besoin de mieux outiller la gestion de configuration de la vérification, pour pouvoir par exemple archiver les expériences successives et leurs résultats, extraire un historique à partir d'une archive, faciliter la comparaison des configurations considérées. Un autre problème est l'impossibilité de générer plusieurs contre-exemples. La modification manuelle de l'observateur pour éviter le contre-exemple précédent n'est pas une solution satisfaisante. Nos perspectives sont de chercher à définir une approche automatisée. Un problème intéressant est de forcer l'outil à produire des contre-exemples qui illustrent des scénarios de violation *différents*. Lors de l'interprétation d'un contre-exemple, une première étape consistait à chercher à caractériser la violation observée, en identifiant les parties du modèle activées au cours du temps. Nous allons étudier l'intérêt d'une telle analyse structurelle non seulement pour faciliter la compréhension d'un contre-exemple, mais aussi pour guider le model-checker vers l'exploration de comportements différents.

7. Références

[1] www.esterel-technologies.com/products/scade-suite
[2] S. Duprat, J. Souyris, D. Favre-Félix, "Formal verification workbench for Airbus avionics software," in Embedded Real-Time Software (ERTS), 2006.
[3] S. Miller, A. Tribble, M. Whalen, M. Heimdahl, "Proving the shalls – early validation of requirements through formal methods," Int. Journal of Software Tools for Technology Transfer, 8(4), pp. 303-319, August 2006.
[4] H. Halbwachs, F. Lagnier, and P. Raymond. "Synchronous observers and the verification of reactive systems," in Third Int. Conf. on Algebraic Methodology and software Technology (AMAST'93), Workshops in computing, Springer Verlag. Twente, June 1993.
[5] O. Laurent, P. Michel and V. Wiels, "Using formal verification techniques to reduce simulation and test effort," In Formal Methods for increasing software productivity, FME2001. Springer, 2001.
[6] M. Ljung, "Formal modelling and automatic verification of lustre programs using NP-Tools," Master thesis, Prover Technology AB and dept. of Teleinformatics, KTH, Stockholm, 1999.
[7] C. Ratel, « Définition et réalisation d'un outil de vérification formelle de programmes Lustre : le système LESAR », Ph.D dissertation, INPG, July 1992.
[8] M. Sheeran and G. Stalmarck, "A tutorial on Stalmarck's proof procedure for propositional logic," in Formal Methods in Computer-Aided Design, LNCS 1522, pp. 82-100. Springer Verlag, 1998.
[9] D. Brière and P. Traverse, "AIRBUS A320/A330/A340 electrical flight controls – A family of fault-tolerant systems," in Fault-tolerant computing (FTCS-23), 1993.
[10] H. Saiedian (Ed.), "An invitation to formal methods," IEEE Computer, 29(4), pp. 16-30, April 1996.

Définition d'une sémantique Event-B pour les patrons de raffinement de buts KAOS

Abderrahman Matoussi, Frédéric Gervais et Régine Laleau

LACL, Université Paris-Est Créteil Val de Marne (UPEC)

{abderrahman.matoussi,frederic.gervais,laleau}@u-pec.fr

Résumé

Tandis que les spécifications répondent à la question « que fait le système ? », les buts résolvent quant à eux les questions « qui ? quand ? comment ? » [5]. Ainsi, les buts jouent un rôle très important dans l'ingénierie des besoins et par conséquent dans le développement logiciel. Néanmoins, le processus de développement associé aux méthodes formelles marginalise la phase d'analyse des besoins. En effet, ce processus ne commence qu'à partir de la phase de spécification. Par conséquent, le fossé entre ces deux phases s'élargit et la réconciliation paraît de plus en plus difficile. L'objectif de cet article est d'inclure la phase d'analyse des besoins et plus précisément la méthodologie KAOS dans le développement logiciel associé aux méthodes formelles tout en restant au même niveau d'abstraction. Pour cela, nous proposons une approche constructive dans laquelle les modèles Event-B sont construits d'une manière incrémentale à partir des modèles de buts KAOS en se basant sur les patrons de raffinement de buts. Nous justifions le choix d'Event-B par : (i) la possibilité d'utiliser la méthode durant tout le processus de développement logiciel ; (ii) la maturité des outils qui supportent la notation Event-B.

Mots-clé. Ingénierie des besoins, KAOS, Event-B, patrons de raffinement de buts KAOS.

1 Introduction

L'emploi des méthodes formelles pour la spécification des systèmes complexes est de plus en plus croissant. Ces méthodes ont montré leur capacité de produire de tels systèmes pour des grands projets industriels tel que la ligne 14 du métro parisien [6] ou la navette Roissy Val [7] qui ont utilisé la méthode B [1]. Dans la plupart des méthodes formelles, un modèle

mathématique initial peut être raffiné en plusieurs étapes jusqu'à ce que le raffinement final contienne assez de détails pour une implémentation. La plupart du temps, ce modèle initial est tiré d'une description informelle obtenue dans la phase d'analyse des besoins. Par conséquent, le maillon le plus faible dans la chaîne de développement logicielle est le fossé entre les besoins textuels ou semi-formels et la spécification formelle initiale. En effet, les méthodes formelles et d'analyse des besoins sont deux techniques complémentaires pour le développement des systèmes complexes. Malgré cette complémentarité, peu de travaux ont essayé de rapprocher ces deux techniques. Par conséquent, le fossé entre la phase d'analyse des besoins et la phase de spécification est en train de s'élargir et la réconciliation paraît de plus en plus difficile.

Parmi les méthodes d'analyse des besoins intéressantes, l'ingénierie des exigences orientée buts proposée par la méthode KAOS [4] permet la structuration des buts d'un système et la représentation des frontières du système. Quelques travaux [10, 11, 15, 16] ont essayé d'assurer le passage entre des spécifications informelles (exprimées en KAOS) vers un modèle formel. Néanmoins, cette réconciliation demeure verticale puisque ces travaux génèrent directement des spécifications formelles à partir de la dernière étape de la méthodologie KAOS. En effet, ces travaux ne tiennent pas compte des premiers modèles KAOS tels que le modèle de buts au moment de la spécification formelle. Pour remédier à ce problème, notre objectif est d'inclure la phase d'analyse des besoins dans le développement logiciel associé aux méthodes formelles en exprimant le modèle de buts KAOS avec Event-B [3]. Pour cela, nous présentons : i) une sémantique Event-B pour un sous-ensemble des buts KAOS appelés les buts « à réaliser » et ii) une formalisation en Event-B des patrons de raffinement de buts KAOS. Ce papier est une suite de nos articles précédents [12, 13] qui proposent une formalisation Event-B du modèle de buts KAOS.

La suite du papier est organisée comme suit. La section 2 présente quelques préliminaires sur la méthodologie KAOS et la méthode Event-B. La section 3 donne un aperçu global de notre approche. Les sections 4, 5 et 6 s'intéressent à la formalisation Event-B des patrons de raffinement de buts (respectivement) *par jalons*, *ET* et *OU*. La partie discussion et les travaux liés sont présentés dans les sections 7 et 8. La section 9 conclut le papier en présentant aussi les perspectives.

2 Préliminaires

Dans cette section, nous présentons brièvement KAOS et Event-B.

2.1 La méthodologie KAOS

KAOS (Knowledge Acquisition in autOmated Specification) est une méthodologie pour l'ingénierie des exigences qui résulte des travaux de recherche menés à l'Université de Louvain [4]. Cette méthodologie permet aux analystes de construire des modèles des exigences et de déduire des documents à partir de ces modèles. Pour cela, KAOS s'appuie sur un raisonnement orienté par les buts. Un but KAOS définit un objectif que le système et son environnement doivent réaliser grâce à la coopération de différents agents (matériel, logiciel ou humains). KAOS a distingué les buts des propriétés du domaine qui sont des déclarations descriptives sur l'environnement telles que des lois physiques ou des normes organisationnelles. Le modèle des exigences KAOS se compose de cinq sous-modèles fortement liés par des règles de cohérence :

- Le modèle central est *le modèle de buts* qui décrit les buts du système et son environnement. Ce modèle est organisé dans une hiérarchie obtenue grâce au raffinement de buts de plus haut niveau (les buts stratégiques) vers des buts de bas niveau (les besoins).
- *Le modèle objets* : il permet de décrire le vocabulaire du domaine. Il est représenté par un diagramme de classes UML.
- *Le modèle des responsabilités* : il permet d'assigner les besoins aux différents agents. Ces agents appartiennent au système à construire (agents internes) ou à son environnement (agents externes).
- *Le modèle des opérations* : il décrit les caractéristiques des opérations du système et leurs liens avec les modèles de buts, objets et des responsabilités.
- *Le modèle des comportements* : il résume tous les comportements que les agents doivent accomplir pour satisfaire les besoins. Ces comportements sont exprimés sous la forme d'opérations exécutées par les agents responsables.

L'importance du modèle de buts est due au rôle central joué par les buts dans le processus d'ingénierie des exigences. Par exemple, il est possible de dériver certains modèles KAOS (le modèle des opérations, le modèle objets...) à partir du modèle de buts d'une façon systématique. De plus, le modèle de buts permet de supporter très tôt différentes formes d'analyse des

exigences telles que l'analyse de risques, l'analyse de conflits, ou l'évaluation d'options alternatives [4].

Les buts dans KAOS [4] peuvent être raffinés selon des raffinements ET ou OU. Un raffinement ET implique que la conjonction des sous-buts est une condition suffisante pour réaliser le but parent. Un raffinement OU associe un but à des sous-buts alternatifs pour lesquels l'accomplissement du but de plus haut niveau exige l'accomplissement d'au moins un de ses sous-buts. KAOS offre un ensemble de patrons de raffinement [5] qui décomposent les buts. Ces patrons ne peuvent être utilisés que dans le contexte de différentes tactiques tel que *le raffinement par jalons* qui consiste à identifier les sous-buts comme des étapes successives dans le temps permettant de satisfaire le but de plus haut niveau. Ces patrons sont groupés selon le comportement de différents types de buts. Ces derniers peuvent être « à réaliser » (un jour), « à assurer » (toujours), « à éviter un jour » ou « à éviter toujours ». Un principe de la méthodologie KAOS est d'arrêter le raffinement lorsqu'il est possible d'assigner chaque besoin à un agent particulier. Le lecteur peut se référer à [4] pour une description plus détaillée de ces notions.

KAOS fournit aussi une couche facultative de spécification formelle de buts grâce à la logique temporelle linéaire (LTL). Cette étape de formalisation [4, 5] permet de vérifier par exemple que les raffinements de buts sont corrects et complets en utilisant essentiellement des patrons formels de raffinement. Même si une telle vérification est importante pour détecter par exemple des sous-buts manquants dans des exigences incomplètes, l'utilisation de ce type de logique ne peut pas combler le fossé entre les exigences et les autres phases de développement puisque les concepteurs seront obligés d'utiliser une autre méthode formelle plus concrète pour développer leurs systèmes. Par conséquent, il est difficile de valider les spécifications formelles par rapport aux exigences même si ces dernières ont été exprimées avec LTL.

2.2 Event-B

La méthode B classique [1] est une méthode formelle de modélisation et de construction de logiciels. Elle s'appuie sur les concepts mathématiques de la théorie des ensembles. Event-B [3] est une méthode formelle adaptée à la modélisation des systèmes réactifs par raffinement. Une spécification Event-B est décomposée en deux parties : (i) *le contexte* qui contient la partie statique du modèle telle que les ensembles énumérés, les constantes et les axiomes ; (ii) *la machine* qui contient la partie dynamique telle que les variables et les événements. L'approche Event-B permet non seulement de modéliser le système, mais aussi son environnement à travers la notion

d'observation des événements. Chaque événement est décrit sous la forme d'action gardée et il est susceptible d'être déclenché quand sa garde devient vraie. Par conséquent, l'état du système change en fonction de l'action associée à l'événement. La validation d'un modèle Event-B est définie essentiellement par un invariant que chaque état dans le système doit satisfaire. Ainsi, on doit montrer que chaque événement dans le système préserve cet invariant. Cette vérification est réalisée au travers d'un certain nombre d'obligations de preuve. En Event-B, le raffinement est un processus qui transforme une spécification abstraite et non-déterministe en un système concret et déterministe qui préserve la fonctionnalité de la spécification originale. Pendant le raffinement, les événements abstraits sont raffinés pour pouvoir prendre en compte les nouvelles variables. De nouveaux événements peuvent aussi être observés. Ces événements raffinent l'événement abstrait qui ne fait rien (skip). Les obligations de preuves sont générées à chaque étape de raffinement afin d'assurer la cohérence du raffinement. Event-B fournit actuellement un logiciel libre sous la forme d'une plate-forme (dérivée de « Eclipse ») de spécification et de preuve, appelé Rodin [8]. Les avantages de l'utilisation de l'outil Rodin sont entre autres : (i) la possibilité de demander à tout moment la validation de fragments de modèles par les preuves ; les preuves devenant ainsi un outil d'aide à la modélisation ; (ii) la construction incrémentale de modèles.

3 Un aperçu général de l'approche

3.1 Motivation

En conception de systèmes, la première phase est celle d'analyse des besoins. Elle est suivie par la phase de spécification, puis celle de développement. La méthode Event-B a montré qu'elle était très pertinente pour les deux dernières phases. L'approche proposée dans cet article a pour objectif d'introduire dans la méthode Event-B la phase d'analyse des exigences. Il pourra être possible alors de prouver le modèle de besoins et d'établir des liens formels entre ce modèle et la spécification d'un système. Comme nous l'avons souligné auparavant, nous avons choisi KAOS comme une méthode orientée but d'ingénierie des exigences parce que dans KAOS l'accent est mis sur le raisonnement semi-formel et formel de buts comportementaux pour la dérivation de raffinements de but, des opérationnalisations de but, etc, tandis que dans i* [23] par exemple (l'autre méthode orientée but), l'accent est plus sur le raisonnement qualitatif des buts *soft*. Nous justifions le choix d'Event-B par sa grande similarité et complémentarité avec la méthodologie

KAOS : (i) Event-B et KAOS sont basés sur la notion d'observation qui permet de modéliser à la fois le système et son environnement ; (ii) la notion de raffinement (l'approche constructive) caractérise à la fois KAOS et Event-B. Ainsi, ces points offrent la possibilité d'utiliser Event-B dans les premières phases du développement de systèmes complexes. Cela peut aider les concepteurs dans la construction d'une modélisation formelle du système au complet.

Puisque les buts jouent un rôle important dans l'ingénierie des exigences en fournissant un pont qui lie les besoins des clients à la spécification de système [25], l'approche proposée consiste à dériver une représentation Event-B directement à partir du modèle de buts KAOS. Par conséquent, nous montrons qu'il est possible d'exprimer des modèles de buts KAOS avec des méthodes formelles (comme Event-B) en restant au même niveau d'abstraction. Cependant, il n'est pas possible de vérifier que les deux modèles sont équivalents puisque les modèles KAOS sont seulement semi-formels. En fait, l'expression Event-B du modèle de buts KAOS permet de lui donner une sémantique précise de même que les traductions existantes des spécifications UML dans des spécifications B donnent une sémantique formelle aux diagrammes de classe ou aux diagrammes de transition [21, 22].

3.2 Les buts « Achieve »

Pour atteindre notre objectif, nous formalisons avec Event-B les patrons de raffinement de buts qui aident les analystes à produire une hiérarchie de buts KAOS. Dans ce papier, nous nous concentrons sur le type de buts le plus fréquemment utilisé : les buts « Achieve » (à réaliser). Les autres types de buts seront étudiés dans de futurs travaux. Un but « Achieve » prescrit des comportements attendus où une certaine condition cible doit être tôt ou tard établie une fois qu'une autre condition est vérifiée dans l'état actuel du système (cet état actuel est arbitraire). Un but « Achieve » dans KAOS est dénoté comme suit : *Achieve[**ConditionCible** From **ConditionActuelle**]*. La description temporelle et informelle de cette notation est la suivante, sachant que le préfixe ***ConditionActuelle*** est optionnel (autrement dit, il peut être vrai) :

*[Si **ConditionActuelle** alors] tôt ou tard **ConditionCible**.*

3.3 Formalisation des buts « Achieve »

Si nous faisons référence aux concepts de garde et de post-condition qui existent dans Event-B, on peut considérer un but KAOS comme une

post-condition du système, puisqu'un but signifie qu'une propriété doit être établie. Ainsi, notre formalisation repose sur le fait de représenter chaque but KAOS comme un événement Event-B et la propriété comme la post-condition de cet événement. Pour cela, nous utilisons la relation de raffinement Event-B et des obligations de preuve supplémentaires (faites sur mesure) afin de dériver tous les sous-buts du système au moyen des événements Event-B. Pour mieux illustrer cette idée, on considère qu'un but « Achieve » G est raffiné en deux sous-buts G_1 et G_2 comme le montre la Figure 1.

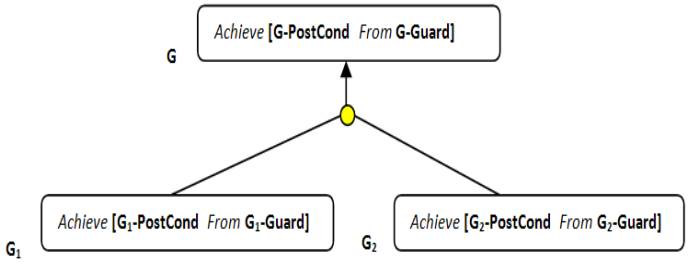


Fig. 1 – Example d'un modèle de buts KAOS

Chaque niveau i ($i \in [0..n]$) dans la hiérarchie de buts KAOS est représenté comme un modèle Event-B M_{i-1} qui raffine le modèle Event-B M_{i-1} lié au niveau $i-1$. De plus, nous représentons chaque but comme un événement Event-B où : (i) la condition actuelle de but est considéré comme la garde de l'événement ; (ii) La partie **then** traduit la condition cible de ce but (voir la Figure 2).

On peut se demander comment la contrainte temporelle (le mot-clé « tôt ou tard ») peut être exprimée dans notre formalisation Event-B, puisqu'il n'y a pas de notion de temps dans ce langage. En fait, J.R Abrial [2] explique que la dimension temporelle ne doit pas être explicitement considérée et recommande d'utiliser les événements eux-mêmes pour exprimer cette dimension. Ainsi, dans Event-B, chaque événement observable est si petit (on dit qu'il est « atomique ») qu'on peut considérer que son exécution ne prend aucun temps, et par conséquent, seul un événement peut avoir lieu dans une unité de temps. Quand l'unité est grande, le temps correspondant est très abstrait et quand l'unité est petite, le temps correspondant est très concret. Autrement dit, le temps est étiré en se déplaçant d'une abstraction à son raffinement. Cet étirement révèle certains « détails de temps », quelques nouveaux événements. Il est donc clair que cette interprétation se

MACHINE Abstract M_0	MACHINE First M_1 REFINES Abstract M_0
EvG $\triangleq$ **when** *G-Guard* **then** *G-PostCond* **end**	**EvG1** $\triangleq$ **when** G_1-*Guard* **then** G_1-*PostCond* **end** **EvG2** $\triangleq$ **when** G_2-*Guard* **then** G_2-*PostCond* **end**
(a) Modèle Abstrait M_0	(b) Modèle Raffiné M_1

FIG. 2 – La représentation Event-B du modèle de buts KAOS

colle bien avec la définition d'un but « Achieve ». Cependant, si des délais sont associés à un tel but, nous serons obligés d'introduire de nouvelles variables afin de modéliser le temps. Ce point sera considéré à l'avenir quand des buts non-fonctionnels seront formalisés.

Dans les sections suivantes, nous proposons une sémantique Event-B pour chaque patron de raffinement de buts KAOS. Basé sur l'ensemble classique des règles d'inférence d'Event-B [3], nous avons identifié les obligations de preuve systématiques pour chaque patron de raffinement de buts. Pour cela, le plan suivant sera utilisé pour décrire chaque patron de raffinement :

1. **Description du patron KAOS :** Nous donnons une courte définition informelle du patron de raffinement de buts KAOS.
2. **Sémantique formelle du patron :** Nous proposons de donner une sémantique Event-B au patron de raffinement de buts KAOS comme suit :
   (a) **Définition formelle :** Nous présentons la sémantique Event-B donnée à chaque patron en construisant quelques modèles mathématiques ensemblistes basés sur la sémantique des traces du développement Event-B [3].
   (b) **Identification des obligations de preuve :** Nous présentons juste quelques arguments informels définissant exactement ce que nous devons prouver pour chaque patron de raffinement de buts KAOS. Une argumentation formelle des obligations de preuve identifiées est détaillée dans [14].

4 L'expression du patron de raffinement de buts « par jalons » en Event-B

4.1 Description du patron KAOS

Ce patron de raffinement [4] raffine un but « Achieve » en introduisant des états de jalon intermédiaires G_1..., G_n pour atteindre un état satisfaisant la condition cible (dénoté par *G-PostCond*) à partir d'un état satisfaisant la condition actuelle (dénoté par *G-Guard*) comme le montre la Figure 3 (avec juste deux sous-buts).

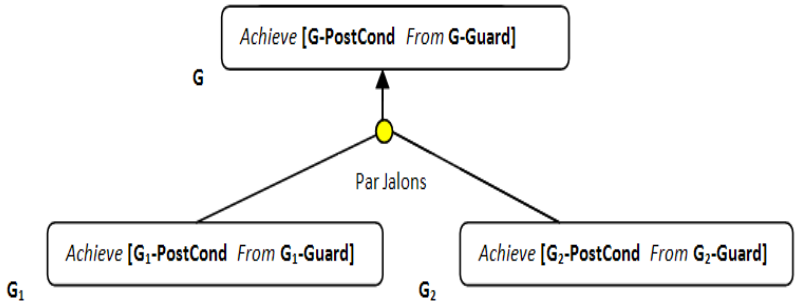


Fig. 3 – Le patron de raffinement de buts « par jalons »

Le premier sous-but G_1 est un but « Achieve » avec comme condition du jalon sa condition cible. Ce but exprime que la condition du jalon (dénoté par G_1-*PostCond*) doit être tôt ou tard établie si la condition actuelle spécifique G_1-*Guard* (qui peut être plus large que la condition actuelle *G-Guard* du but parent) est vérifiée dans l'état actuel. Le deuxième sous-but est un but « Achieve » aussi, il exprime que tôt ou tard la condition cible spécifique G_2-*PostCond* (qui peut être plus large que la condition cible *G-PostCond* du but parent) doit être établie si la condition du jalon spécifique G_2-*Guard* (dérivé à partir G_1-*PostCond*) est vérifiée dans l'état actuel.

4.2 Sémantique formelle du patron

Puisque la satisfaction de tous les sous-buts KAOS (selon un ordre bien déterminé) implique la satisfaction du but parent, l'événement abstrait **EvG** est raffiné par *la séquence* de tous les nouveaux événements (**EvG1**, **EvG2**).

4.2.1 Définition formelle

Nous proposons une extension syntaxique des règles de preuve de raffinement Event-B afin de supporter qu'un événement abstrait est raffiné par la séquence de nouveaux événements comme suit :

(**EvG1** ; **EvG2**) Refines **EvG**

4.2.2 Identification des obligations de preuve

Nous allons donner des règles systématiques définissant exactement ce que nous devons prouver pour ce patron pour s'assurer que la séquence d'événements concrets **EvG1 ; EvG2** raffine son abstraction. En fait, nous devons prouver trois différents lemmes :

- *La contrainte d'ordre* exprime la caractéristique « par jalon » entre les événements Event-B.

$$G_1\text{-}PostCond \Rightarrow G_2\text{-}Guard \qquad \textbf{(PO1)}$$

- *Le renforcement de la garde* assure que la garde concrète est plus forte que la garde abstraite. Autrement dit, il n'est pas possible d'exécuter la version concrète tandis que la version abstraite ne l'est pas. Le terme « plus fort » signifie que la garde concrète implique la garde abstraite.

$$G_1\text{-}Guard \Rightarrow G\text{-}Guard \qquad \textbf{(PO2)}$$

- *Le raffinement correct* assure que la séquence des événements concrets transforme les variables concrètes d'une manière qui ne contredit pas l'événement abstrait.

$$G_2\text{-}PostCond \Rightarrow G\text{-}PostCond \qquad \textbf{(PO3)}$$

5 L'expression du patron de raffinement de buts « ET » en Event-B

5.1 Description du patron KAOS

Ce patron de raffinement [4] raffine un but « Achieve » en deux (ou plus) sous-buts si la conjonction de ces sous-buts est suffisante pour établir

la satisfaction du but parent G comme le montre la Figure 3 (avec juste deux sous-buts).

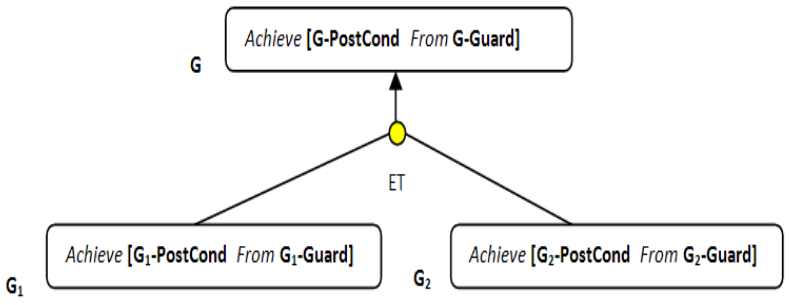


Fig. 4 – Le patron de raffinement de buts « ET »

Restriction : Dans KAOS rien n'est dit sur l'ordre d'exécution des différents sous-buts. La formalisation en Event-B révèle que cela peut constituer un sérieux problème quand ces sous-buts manipulent des variables partagées. Pour mieux expliquer ce problème, on suppose par exemple qu'une variable partagée x est modifiée par deux sous-buts (G_1 et G_2). G_1 augmente x de 6 et G_2 multiplie x par 4. Ainsi, il est facile de constater que l'ordre d'exécution des deux sous-buts est important. Pour éviter ce problème, une condition suffisante est de forcer un raffinement ET à manipuler seulement *un ensemble de variables disjoint*. Cette solution très contraignante est proche d'autres travaux comme [1] avec le concept de comportement parallèle. S'il y a un raffinement ET avec *des variables partagées*, nous proposons de transformer cette forme de raffinement ET en un raffinement « par jalons » et ensuite explicitement spécifier l'ordre des modifications sur ces variables partagées.

5.2 Sémantique formelle du patron

De manière analogue au raffinement par jalons, la satisfaction de tous les sous-buts KAOS implique la satisfaction du but parent. Cependant, l'exécution des nouveaux événements ne doit pas nécessairement suivre un ordre spécifique. De là, notre idée (inspiré des algèbres de processus [24]) est que ces événements (**EvG1**, **EvG2**) sont exécutés dans un ordre arbitraire : **EVG1 ;EVG2** ou **EVG2 ;EVG1**. Cela correspond à la sémantique de l'opérateur *d'entrelacement* dans les algèbres de processus.

5.2.1 Définition formelle

Nous proposons une extension syntaxique des règles de preuve de raffinement Event-B afin de supporter qu'un événement abstrait est raffiné par l'entrelacement des nouveaux événements comme suit :

(**EvG1** ||| **EvG2**) Refines **EvG**

5.2.2 Identification des obligations de preuve

Nous allons donner des règles systématiques définissant exactement ce que nous devons prouver pour ce patron pour s'assurer que l'entrelacement des événements concrets **EvG1 ; EvG2** raffine son abstraction. Pour cela, nous devons prouver trois différents lemmes :

– *Le renforcement de la garde* assure que la garde concrète est plus forte que la garde abstraite. La garde concrète de **EvG1** ||| **EvG2** peut être l'un ou l'autre G_1-*Guard* (si nous exécutons **EvG1** d'abord) ou G_2-*Guard* (si nous exécutons **EvG2** d'abord). Par conséquent, nous devons prouver que :

$$G_1\text{-}Guard \Rightarrow G\text{-}Guard \quad \textbf{(PO1)}$$
$$G_2\text{-}Guard \Rightarrow G\text{-}Guard \quad \textbf{(PO2)}$$

– *Le raffinement correct* assure que l'entrelacement des événements concrets transforme les variables concrètes d'une manière qui ne contredit pas l'événement abstrait.

$$(G_1\text{-}PostCond \wedge G_2\text{-}PostCond) \Rightarrow G\text{-}PostCond \quad \textbf{(PO3)}$$

6 L'expression du patron de raffinement de buts « OU » en Event-B

6.1 Description du patron KAOS

Ce patron de raffinement [4] raffine un but « Achieve » en deux (ou plus) sous-buts si seulement l'un ou l'autre (pas les deux) de ses sous-buts sont réalisés comme le montre la Figure 3.

Dans chaque sous-but, une condition cible spécifique doit être atteinte pour atteindre la condition cible *G-PostCond* du but parent. Il faut noter

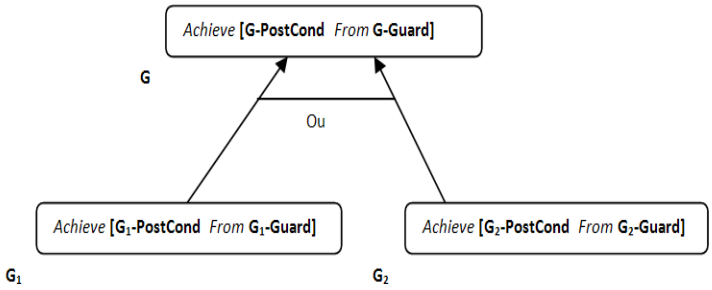


Fig. 5 – Le patron de raffinement de buts « OU »

que ce patron de raffinement introduit une certaine forme de non-déterminisme bornée qui sera résolu plus tard dans la phase d'implémentation.

6.2 Sémantique formelle du patron

6.2.1 Définition formelle

Puisque la satisfaction d'exactement un sous-but KAOS implique la satisfaction du but parent, nous proposons de raffiner l'événement abstrait **EvG** comme suit :

$$(\textbf{EvG1}\ XOR\ \textbf{EvG2})\ \text{Refines}\ \textbf{EvG}$$

Ce raffinement OU-exclusif peut être considéré comme un raffinement OU-inclusif avec une caractéristique supplémentaire d'exclusivité comme suit :

$$(\textbf{EvG1}\ XOR\ \textbf{EvG2})\ \text{Refines}\ \textbf{EvG} = \begin{cases} (\textbf{EvG1}\ OR\ \textbf{EvG2})\ \text{Refines}\ \textbf{EvG} \\ \text{Caractéristique d'exclusivité} \end{cases}$$

6.2.2 Identification des obligations de preuve

Nous allons donner des règles systématiques définissant exactement ce que nous devons prouver pour ce patron pour pour s'assurer que chaque événement concret (**EvG1** ou **EvG2**) raffine son abstraction :

– *Le renforcement de la garde* assure que la garde concrète (G_1-*Guard* or G_2-*Guard*) est plus forte que la garde abstraite de l'événement **EvG**.

$$G_1\text{-}Guard \Rightarrow G\text{-}Guard \quad \textbf{(PO1)}$$
$$G_2\text{-}Guard \Rightarrow G\text{-}Guard \quad \textbf{(PO2)}$$

– *Le raffinement correct* assure que chaque événement concret (**EvG1** or **EvG2**) transforme les variables concrètes d'une manière qui ne contredit pas l'événement abstrait **EvG**.

$$G_1\text{-}PostCond \Rightarrow G\text{-}PostCond \quad \textbf{(PO3)}$$
$$G_2\text{-}PostCond \Rightarrow G\text{-}PostCond \quad \textbf{(PO4)}$$

– *La caractéristique d'exclusivité* assure que seulement un événement (**EvG1** ou **EvG2**) peut être exécuté et pas tous les deux.

$$G_1\text{-}PostCond \Rightarrow \neg G_2\text{-}Guard \quad \textbf{(PO5)}$$
$$G_2\text{-}PostCond \Rightarrow \neg G_1\text{-}Guard \quad \textbf{(PO6)}$$

La plupart de ces obligations de preuve pourraient être prouvées automatiquement par la version actuelle de Rodin [8]. En fait, cette sémantique de raffinement Event-B est très proche de celle proposée par Rodin. Néanmoins, de nouvelles contraintes doivent être ajoutées pour exprimer la caractéristique d'exclusivité.

7 Discussion

On peut se demander si la formalisation des conditions cibles KAOS comme des post-conditions Event-B est adéquate, puisque l'exécution des événements Event-B n'est pas obligatoire. Dans la sémantique Event-B, tous les événements dont la garde est vraie peuvent être exécutés et il y a nécessairement un événement qui sera exécuté. Le choix se fait de manière non-déterministe. Pour assurer que les contraintes impliquées par les raffinements de buts seront respectées par la spécification en Event-B, nous partons du principe suivant. Le but de plus haut niveau est exprimé sous la forme d'un événement abstrait qui est le seul à pouvoir se déclencher. Par conséquent, la sémantique d'Event-B assure que le but sera atteint tôt ou tard. Pour les sous-buts dérivés à partir du but de plus haut niveau, les obligations de preuve permettent d'assurer que les contraintes ET, OU et jalons seront respectées. En revanche, pour vérifier que d'autres événements ne peuvent pas s'entrelacer avec les buts spécifiés en Event-B, les obligations de preuve à générer sont beaucoup trop longues à exprimer et, de plus, elles sont complexes à prouver avec les techniques classiques de preuve. C'est pour-

quoi nous avons l'intention d'utiliser des techniques de « model-checking » pour prouver la cohérence globale du modèle de buts et pour vérifier des propriétés de type logique temporelle. De là, notre idée est de mélanger des obligations de preuve locales (générées par le prouveur d'Event-B) avec des preuves produites par le « model-checking ». Pour cela, un outil comme ProB [18] pourrait être un outil très intéressant.

L'intérêt de cette approche proposée est que nous pouvons prouver des propriétés de cohérence locales sur le modèle de buts KAOS en prouvant les obligations de preuve produites par le raffinement Event-B. En effet, si nous pouvons prouver par exemple les obligations de preuve du raffinement d'un événement **EvG** par **EvG1** ou **EvG2** cela signifie que la décomposition OU de but G est correct. Autrement, cela signifie qu'une ou plusieurs expressions des événements (**EvG**, **EvG1**, **EvG2**) ne sont pas correctes ou que certains sous-buts manquent ou que le patron de raffinement de buts est mal appliqué. Cependant, nous pouvons jamais assurer que l'expression des événements Event-B correspond exactement à l'expression du but lié puisque ce dernier est informel. Cela signifie que les preuves Event-B de la contrepartie formelle du modèle de buts KAOS ne sont pas suffisantes pour valider la conformité de la spécification vis-à-vis des exigences originales. Pour cela, nous pouvons utiliser une technique d'animation pour valider la spécification formelle dérivée et par conséquent sa contrepartie semi-formelle (le modèle de buts) vis-à-vis les exigences initiales des clients. Cette étape d'animation indique non seulement qu'il y a des déviations par rapport aux exigences initiales, mais aide aussi à la détection de quelques erreurs de spécification. Le lecteur peut se référer à [20] pour une description plus détaillée de ce point. Pour cela, l'outil ProB [18] peut être aussi très utile.

8 Travaux similaires

Notre approche proposée vise à exprimer des modèles KAOS avec Event-B en restant au même niveau d'abstraction. Cela va faciliter la transition entre la phase d'analyse des besoins et la phase de spécification. Dans ce qui suit, nous présentons les différents travaux qui se situent dans le même contexte que le notre.

Les auteurs de [16] présentent une approche qui définit le lien entre le modèle des exigences KAOS et la phase de spécification. Ce lien consiste à dériver des machines B abstraites à partir des différents modèles KAOS. Pour cela, les auteurs associent une machine B à chaque agent KAOS puisque les agents sont les entités actives qui permettent de réaliser les opérations.

Ainsi, toutes les opérations KAOS pour un agent donné sont représentées en B comme des opérations pour la machine correspondante. De plus, tous les buts « Maintain » (à assurer toujours) qui sont sous la responsabilité d'un agent sont vus comme des invariants pour la machine correspondante à cet agent.

En utilisant aussi la méthode B classique, [11] transforme le modèle des exigences KAOS liés seulement aux aspects de la sécurité (des buts « Maintain ») vers un seul modèle équivalent en B. Ce modèle B abstrait est ensuite raffiné en utilisant le principe de raffinement B qui produit des spécifications conformes aux exigences initiales. Pour cela, [11] associe une machine B à chaque objet KAOS concerné par les buts opérationnels. De cette façon, la relation en KAOS entre les différents objets est traduite en B en utilisant les clauses de liaison entre les machines tel que INCLUDES, USES ou SEES. La particularité de [11] est qu'il introduit « le concept d'accomplissement de buts » qui est représenté en B par des valeurs de retour des opérations B modélisant seulement les buts « Maintain ». Ainsi, chaque opération B relative à un but opérationnel rend une valeur indiquant si le but « Maintain » relié à cette opération a été réalisé ou non. Cependant, « le concept d'accomplissement de buts » ne peut pas être appliqué pour les autres types de buts tels que les buts « Achieve ».

[10] propose l'outil GOPCSD (Goal-oriented Process Control System Design) qui est une adaptation de la méthode KAOS. Cet outil construit dans un premier temps le modèle de buts (différent du modèle de buts KAOS) en utilisant plusieurs entités (agents, buts...). Puis, elle prouve la cohérence et la complétude du modèle construit. Enfin, l'outil GOPCSD transforme ce modèle de buts en une spécification B équivalente qui peut être ensuite raffinée et traduite en code exécutable. Dans le même esprit, [15] propose un générateur automatique pour transformer un modèle des exigences KAOS dans des spécifications VDM++. Ce générateur connecte les opérations KAOS à celles dans VDM++ et les entités KAOS aux objets ou types VDM++ .

Récemment, [19] a présenté une approche basée sur la vérification constructive qui consiste à faire des liens entre les exigences (exprimées comme des formules de logique temporelle linéaire) et une spécification de système (exprimée comme une machine Event-B étendue avec la notion d'obligations [17]). Les exigences initiales sont incluses comme des assertions de vérification qui peuvent être vérifiées par des outils comme ProB [18], montrant que la spécification proposée respecte en effet ces exigences.

Néanmoins, la réconciliation présentée par tous ces travaux reste partielle parce qu'ils ne considèrent pas toutes les parties du modèle de buts KAOS,

mais seulement les buts opérationnels (les buts feuilles). Par conséquent, le modèle formel n'inclut aucune information sur les buts non-opérationnels et surtout sur le type de raffinement de buts. Dans ce papier, nous avons exploré comment faire face à ce problème en utilisant une nouvelle approche qui exprime tout le modèle de buts KAOS avec une méthode formelle (Event-B) en restant au même niveau d'abstraction. Notre approche peut être ainsi considérée comme complémentaire aux approches existantes. En outre, ce que nous présentons peut être très utile en pratique pour (i) vérifier systématiquement que toutes les exigences KAOS sont représentées dans le modèle Event-B ; (ii) vérifier systématiquement que chaque élément dans le modèle Event-B a un correspondant dans KAOS.

9 Conclusion et perspectives

Cet article présente une nouvelle approche constructive et dirigée par les buts qui montre qu'il est possible de remonter les méthodes formelles jusqu'à la phase d'analyse des besoins. Nous avons montré que l'extension de KAOS avec une étape de formalité fournit une meilleure précision et par conséquent une forme d'analyse plus riche. La contribution majeure de notre approche est qu'elle établit la première brique vers la construction du pont entre le monde non-formel et celui du formel aussi étroit et concis que possible. Cette brique équilibre le compromis entre la complexité des méthodes formelles (Event-B) et l'expressivité d'approches semi-formelles (KAOS). La suite du travail va consister d'une part à étudier les autres concepts du modèle de buts KAOS tels que les propriétés du domaine et les buts non-fonctionnels, d'autre part à appliquer l'approche sur des cas d'études plus complexes. De plus, il serait intéressant d'établir la correspondance entre cette représentation Event-B obtenue à partir de modèle de buts KAOS et les phases postérieures de développement. Nous planifions aussi d'outiller notre approche en développant un logiciel libre construit à partir de logiciels libres disponibles : l'outil Topcased [9] (pour la partie expression des besoins) et l'outil Rodin [8] (pour la partie spécification formelle).

Références

[1] J.R. Abrial. The B-Book : Assigning programs to meanings. Cambridge University Press, 1996.

[2] J.R. Abrial. Extending B without changing it (for developing distributed systems). The First Conference on the B-Method, pages 169–190, IRIN, Nantes, France, 1996.

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

[4] A. van Lamsweerde. Requirements Engineering : From System Goals to UML Models to Software Specifications. Wiley, 2009.

[5] R. Darimont and A. van Lamsweerde. Formal Refinement Patterns for Goal-Driven Requirements Elaboration. In *SIGSOFT '96*, pages 179–190, San Francisco, California, USA, October 1996. ACM SIGSOFT.

[6] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. METEOR : A successful application of B in a large project. In *FM '99 : Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems*, Volume I pages 369–387, 1999. Springer.

[7] F. Badeau and A. Amelot. Using B as a high level programming language in an industrial project : Roissy val. In *Proceedings of the 4th International Conference of B and Z Users (ZB'05)*, pages 334–354, Guildford, UK, 2005. Springer.

[8] RODIN - Rigorous Open Development Environment for Complex Systems. *http ://rodin.cs.ncl.ac.uk/*

[9] TOPCASED - a Toolkit in OPen source for Critical Aeronautic SystEms Design. *http ://www.topcased.org/*

[10] I. El-Madah and T. Maibaum. Goal-Oriented Requirements Analysis for Process Control Systems Design. In *MEMOCODE 2003*, pages 45–46, Mont Saint-Michel, France, June 2003. IEEE Computer Society.

[11] R. Hassan and S. Bohner and S. El-Kassas and M. Eltoweissy. Goal-Oriented, B-Based Formal Derivation of Security Design Specifications from Security Requirements. In *ARES 2008*, pages 1443–1450, Barcelona, Spain, March 2008. IEEE Computer Society.

[12] A. Matoussi and F. Gervais and R. Laleau. A First Attempt to Express KAOS Refinement Patterns with Event B. In *ABZ 2008*, pages 338, London, UK, September 2008. Springer.

[13] A. Matoussi and F. Gervais and R. Laleau. De KAOS vers Event-B : Approche dirigée par les buts. In *Actes de la conférence AFADL'2009*, pages 71–86, Toulouse, France, Janvier 2009.

[14] A. Matoussi and F. Gervais and R. Laleau. An Event-B formalization of KAOS goal refinement patterns. In *Technical Report TR-LACL-*

2010-1, LACL, University of Paris-Est (Paris 12), *http ://lacl.univ-paris12.fr/Rapports/TR/TR-LACL-2010-1.pdf*, 2010

[15] H. Nakagawa and K. Taguchi and S. Honiden. Formal Specification Generator for KAOS : Model Transformation Approach to Generate Formal Specifications from KAOS Requirements Models. In *ASE 2007*, pages 531–532, Atlanta, Georgia, USA, November 2007. ACM.

[16] C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *REMO2V'2006*, Luxembourg, June 2006.

[17] J. Bicarregui and A. Arenas and B. Aziz and P. Massonet and C. Ponsard. Towards Modelling Obligations in Event-B. In *ABZ 2008*, pages 181–194, London, UK, September 2008. Springer.

[18] M. Leuschel and M.J. Butler. ProB : A Model Checker for B. In *K. Araki, S. Gnesi, D. Mandrioli (eds), FME 2003 : Formal Methods, LNCS 2805*, pages 855–874, 2003. Springer.

[19] B. Aziz and A. Arenas and J. Bicarregui and C. Ponsard and P. Massonet. From Goal-Oriented Requirements to Event-B Specifications. In *In : First Nasa Formal Method Symposium (NFM 2009)* , Moffett Field, California , USA, April 2009.

[20] A. Mashkoor and A. Matoussi. Towards Validation of Requirements Models. In *2nd International Conference on Abstract State Machines (ASM), Alloy, B and Z (ABZ'10)*, Orford, Canada, 2010. To appear.

[21] A. Mammar and R. Laleau. A formal approach based on UML and B for the specification and development of database applications. In *Automated Software Engineering 13(4)*, pages 497-528, 2006.

[22] C. Snook and M. Butler. UML-B : formal modelling and design aided by UML. In *ACM Transactions on Software Engineering and Methodology, 15(1)*, pp. 92-122, 2006.

[23] E. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *RE'97*, pages 226-235, 1997. IEEE Computer Society.

[24] D. Sangiorgi. Locality and interleaving semantics in calculi for mobile processes. In *Theor. Comput. Sci.*, Volume 155, pages 39–83, 1996. Elsevier Science Publishers Ltd.

[25] W.N. Robinson and S. Pawlowski. Surfacing Root Requirements Interactions from Inquiry Cycle Requirements Documents. In *The Third IEEE International Conference on Requirements Engineering (ICRE'98)*, pages 82–89, Colorado Springs, CO, USA, 1998. IEEE Computer Society Press.

Session du groupe de travail MTV2

Méthodes de Tests pour la Validation et la Vérification

Exploration aléatoire de modèles

Johan Oudinet

LRI (Univ. Paris-Sud – CNRS)
Bât 490
91405 Orsay Cedex
johan.oudinet@lri.fr

RÉSUMÉ. *Cet article présente des méthodes d'explorations probabilistes qui garantissent une bonne couverture des chemins du modèle quelle que soit sa topologie. Ces méthodes sont basées sur des techniques de comptage et de tirage uniforme de structures combinatoires qui dans leurs versions de bases sont très coûteuses en espace mémoire. Ce papier présente des améliorations qui permettent d'explorer uniformément des modèles plus gros en tirant des chemins plus longs. Une étude de la complexité binaire, accompagnée de résultats expérimentaux de chacune des variantes possibles permet de décider quelle variante utiliser en fonction de la taille du modèle, des longueurs de chemins désirées et des ressources disponibles.*

ABSTRACT. *This article presents optimizations of a randomized method that generates paths while ensuring a good coverage of the model, regardless its topology. The optimizations aim at diminishing the required memory, thus allowing the generation of longer paths. Pure random exploration generally leads to a bad coverage of the model. Methods, based on counting and uniform drawing in combinatorial structures, can ensure a good coverage of paths. Due to memory consumption, such methods can neither explore very large models nor generate very long paths. In this paper, we leverage the limitation of path lengths by using new algorithms with better space complexity. Experimental results show significant improvement over previous randomized approaches. This work opens new perspectives to efficiently explore models for simulation, random testing and model-checking purposes.*

MOTS-CLÉS : *génération aléatoire uniforme, exploration de modèles, arithmétique flottante.*

KEYWORDS: *uniform random generation, model exploration, floating point arithmetic.*

1. Introduction

Cet article présente des méthodes d'explorations probabilistes de modèles finis non probabilistes. Il existe plusieurs critères de couverture (états, transitions, MC/DC, chemins) en fonction des propriétés qu'on cherche à vérifier. Ici, nous nous intéressons à la couverture uniforme de chemins.

Le problème de l'explosion combinatoire est un phénomène connu de toute personne qui étudie des modèles, quelque soit leur nature. Les systèmes considérés deviennent de plus en plus complexes et leurs modèles croissent de façon exponentielle. Par exemple, la composition de modèles conduit généralement à des modèles de taille exponentielle. Ou bien, le simple fait de considérer les types de données, même en les bornant, produit là encore des modèles de très grandes tailles.

L'exploration aléatoire d'un modèle est une approche classique en simulation, mais est aussi utilisée pour le test (Dwyer *et al.*, 2007) et le model-checking (Grosu *et al.*, 2005; Hérault *et al.*, 2004).

Pour explorer aléatoirement un modèle représenté par un graphe, l'approche la plus naturelle est l'utilisation de marches aléatoires. Pour effectuer une marche aléatoire sur un graphe $\mathcal{G}$, il suffit de connaître tous les états du système ainsi que leurs successeurs, et d'être capable d'attribuer à chacun de ces successeurs une probabilité telle que la somme des probabilités de tous les successeurs d'un sommet soit égale à 1. Dans le cas d'une marche aléatoire *isotrope*, tous les successeurs sont équiprobables. L'algorithme 1 permet de générer un chemin d'une longueur inférieure ou égale à n.

Algorithme 1 Marche aléatoire isotrope sur un graphe

Entrées: Un graphe $\mathcal{G}$, un sommet initial s_0 et une longueur n
Sortie: Un chemin σ tel que $|\sigma| \leq n$

```
i ← 0
s ← s_0
tant que i ≠ n et succ(s) ≠ ∅ faire
    Choisir un sommet s' uniformément parmi les successeurs de s (succ(s))
    σ ← σ ∪ t avec t la transition s → s'
    i ← i + 1
    s ← s'
fin tant que
retourne σ
```

N'ayant besoin d'aucune autre connaissance que l'état courant et la liste de ses successeurs, une marche aléatoire isotrope semble être le candidat idéal pour explorer de très grands modèles. Malheureusement, la distribution de probabilité sur les chemins induite est difficile à déterminer car elle dépend de la topologie du graphe. Il peut arriver qu'une marche aléatoire isotrope soit totalement inefficace, comme pour l'exemple suivant.

Soit le graphe de la figure 1, l'espérance du nombre N de marches aléatoires isotropes à effectuer avant d'obtenir n chemins distincts (de longueur n) est de :

$$\begin{aligned} E(N) &= E(N_1) + E(N_2) + \cdots + E(N_n) \\ &= \frac{1}{p_1} + \frac{1}{p_2} + \cdots + \frac{1}{p_n} \\ &= 1 + 2 + \cdots + 2^{n-1} \\ &= 2^n - 1 \end{aligned} \qquad [1]$$

où $E(N_i)$ (resp. p_i) désigne l'espérance (resp. la probabilité) d'obtenir un nouveau chemin après avoir effectué $i-1$ marches aléatoires isotropes distinctes. En d'autres termes, l'équation [1] montre qu'en utilisant des marches aléatoires isotropes, il faut en moyenne un temps exponentiel pour couvrir les chemins du graphe de la figure 1.

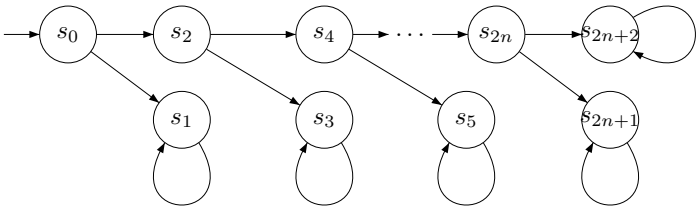


Figure 1. *Un exemple de graphe pathologique pour la marche aléatoire isotrope*

Ce temps exponentiel est dû au fait qu'à chaque sommet la marche aléatoire doit faire un choix entre soit un sommet qui mène à un seul chemin, soit un sommet d'où partent un nombre exponentiel de chemins. Or dans le cas d'une marche isotrope, ces deux sommets ont la même probabilité d'être tirés, favorisant ainsi la probabilité d'apparition d'un chemin au détriment d'un très grand nombre de chemins. Si on était capable de connaître le nombre de chemins partant de chacun des sommets au moment de choisir le sommet suivant, alors on pourrait guider la marche aléatoire afin de rééquilibrer la probabilité d'apparition de tous les chemins et d'obtenir, au mieux, une distribution uniforme sur les chemins.

Cet article présente des méthodes d'explorations probabilistes qui garantissent une bonne couverture des chemins du modèle quelle que soit sa topologie. Ces méthodes sont basées sur des techniques de comptage et de tirage uniforme de structures combinatoires qui dans leurs versions de bases sont très coûteuses en espace mémoire. Nous avons déjà développé des techniques de tirage modulaire qui se basent sur le tirage de chemins uniforme dans chaque composant (Gaudel *et al.*, 2008). L'objectif de cet article est d'améliorer le tirage dans chaque composant, dont la taille est supposée tenir en mémoire, afin de pouvoir tirer des chemins encore plus longs et donc d'être capable de tirer des chemins plus longs aussi avec les techniques de tirage modulaire.

La section 2 explique une méthode utilisée par Denise *et al.* (Denise *et al.*, 2004; Gouraud *et al.*, 2001) pour faire du tirage uniforme de chemins de longueur inférieure

ou égale à n, qui se base sur une méthode récursive pour compter les chemins (Flajolet *et al.*, 1994). Cet article présente, dans les sections 3 et 4, deux optimisations qui permettent de tirer des chemins encore plus longs et plus rapidement. Les résultats des premières expériences menées avec ces nouvelles méthodes sont présentés dans la section 5.

2. Tirage uniforme de chemins

Il est nécessaire d'être en mesure de compter les chemins partant d'un sommet afin d'obtenir un tirage uniforme sur ces chemins. Dans cette section, je résumerai les travaux de Denise *et al.* concernant le tirage uniforme de chemins de longueur inférieure ou égale à n. Mais avant cela, nous avons besoin d'un minimum de formalisme.

Les modèles peuvent être représentés de différentes manières selon la sémantique que l'on veut leur donner et le type d'analyse que l'on désire effectuer. Pour la suite de cet article, nous utiliserons la notion d'automate.

Définition 1. *Un automate fini $\mathcal{A}$ est une structure :*

$$\mathcal{A} = \langle \mathcal{X}, \mathcal{S}, s_0, \mathcal{F}, \mathcal{T} \rangle$$

où $\mathcal{X}$ est un alphabet d'étiquettes, $\mathcal{S}$ un ensemble fini d'états, s_0 l'état initial, $\mathcal{F} \subseteq \mathcal{S}$ un ensemble d'états finaux, et $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{X} \times \mathcal{S}$ un ensemble de transitions.

Définition 2. *Un chemin σ de longueur n dans un automate $\mathcal{A}$ est une séquence de transitions :*

$$\sigma = t_1 t_2 \cdots t_n$$

telle que : $\forall i, t_i = s_{i-1} \times a_i \times s_i$ et $s_n \in F$.

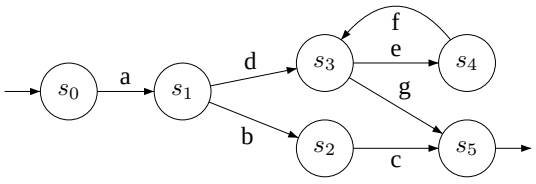


Figure 2. *Un exemple d'automate*

La figure 2 donne un exemple d'automate avec un seul état final (s_5). Le chemin $\sigma = s_0, a, s_1, b, s_2, c, s_5$ est un des deux chemins de longueur 3 de cet automate. Il y a en tout 6 chemins de longueur ≤ 9.

Soit un automate $\mathcal{A}$ et un entier n, $\mathcal{P}_n(\mathcal{A})$ (resp. $\mathcal{P}_{\leq n}(\mathcal{A})$) désigne l'ensemble des chemins de longueur n (resp. inférieure ou égale à n) dans $\mathcal{A}$. Lorsqu'il n'y a pas d'ambiguïté sur $\mathcal{A}$, on utilisera la notation réduite $\mathcal{P}_n$ (resp. $\mathcal{P}_{\leq n}$). On peut noter que la taille de ces ensembles est généralement exponentielle par rapport à n.

La problématique est le tirage uniforme des chemins de $\mathcal{P}_{\leq n}$. Mais avant cela, nous allons nous intéresser au tirage uniforme de chemins de longueur exactement n, c'est-à-dire aux chemins qui sont dans $\mathcal{P}_n$. Nous verrons ensuite qu'une petite modification dans l'automate permet d'engendrer des chemins de longueurs $\leq n$, sans changer la méthode.

Supposons qu'à l'étape en cours, la marche aléatoire est sur l'état s, qui a k successeurs : $s_1, s_2, \ldots, s_k$, et qu'il reste m étapes avant d'obtenir un chemin de longueur n. Pour garantir l'uniformité des chemins de $\mathcal{P}_n$, la marche aléatoire doit choisir le successeur s_i (pour $1 \leq i \leq k$) avec la probabilité suivante :

$$P(s_i) = \frac{l_{s_i}(m-1)}{l_s(m)} \qquad [2]$$

avec $l_s(m)$ qui désigne le nombre de chemins de longueur m qui vont de s à un état appartenant à $\mathcal{F}$.

Pour obtenir un tirage uniforme il est nécessaire de calculer $l_s(m)$ pour tout $s \in \mathcal{S}$ et tout m, $0 \leq m \leq n$. Ce calcul peut être fait avec la relation de récurrence suivante :

$$\begin{cases} l_s(0) = 1 & \text{si } s \in \mathcal{F} \\ l_s(0) = 0 & \text{si } s \notin \mathcal{F} \\ l_s(i) = \sum\limits_{s \to s'} l_{s'}(i-1) & \forall i > 0 \end{cases} \qquad [3]$$

Ainsi, après un prétraitement qui consiste à stocker au préalable les valeurs $l_s(i)$ pour tout s et pour $0 \leq i \leq n$ dans un tableau à deux dimensions - appelé la table de comptage - on peut tirer uniformément des chemins de longueur n.

En ce qui concerne le prétraitement, le calcul de la table de comptage nécessite $\mathcal{O}(n \times d \times S)$ additions, où n est la longueur des chemins à tirer, S le nombre d'états de l'automate et d son degré sortant maximal. Chaque opération est effectuée en utilisant une arithmétique entière. Or, les deux opérandes, qui représentent des nombres de chemins, peuvent avoir une taille importante (en $\mathcal{O}(n)$) car la place nécessaire pour représenter un nombre est d'un ordre de grandeur logarithmique par rapport à sa valeur et le nombre de chemins de longueur n peut être exponentiel par rapport à n. Ainsi, le coût de chaque opération arithmétique est au pire en $\mathcal{O}(n)$ (Knuth, 1997) et la complexité du prétraitement en nombre d'opérations binaires est en $\mathcal{O}(n^2 \times d \times S)$. Concernant la complexité en mémoire du prétraitement, le stockage de la table de comptage requiert $\mathcal{O}(n \times S)$ grands nombres. Chaque nombre pouvant avoir une taille en $\mathcal{O}(n)$, l'occupation mémoire est en $\mathcal{O}(n^2 \times S)$.

En ce qui concerne le tirage, l'obtention d'un chemin de longueur n demande $\mathcal{O}(n \times d)$ opérations arithmétiques. Or, comme précédemment, chaque opération arithmétique étant en $\mathcal{O}(n)$, la complexité binaire du tirage d'un chemin de longueur n est en $\mathcal{O}(n^2 \times d)$. Et l'occupation mémoire nécessaire est celle de la table, $\mathcal{O}(n^2 \times S)$, plus celle d'un chemin de longueur n, $\mathcal{O}(n)$, soit une complexité mémoire en $\mathcal{O}(n^2 \times S)$ pour le tirage d'un chemin de longueur n.

Mais qu'en est-il des chemins de longueur $\leq n$? Pour les obtenir, il suffit d'ajouter un état s'_0 qui devient le nouvel état initial de $\mathcal{A}$, une (τ-)transition qui va de s'_0 à s_0, et une (τ-)boucle sur s'_0. Il est évident qu'un chemin de longueur $n+1$ qui commence par $k+1$ (τ-)transitions dans ce nouvel automate correspond à un chemin de longueur $n-k$ dans l'automate précédent. On vérifie aussi aisément que tout ancien chemin de longueur inférieure ou égale à n est maintenant représenté par un chemin de longueur exactement $n+1$.

Les limites de cette méthode sont : l'espace mémoire nécessaire pour la table de comptage, qui peut être très important dès lors qu'on s'intéresse à tirer des chemins longs dans de très gros modèles, et dans une moindre mesure, le temps de génération qui peut devenir quadratique en n si les opérations sont faites avec une arithmétique entière. Les deux sections suivantes décrivent, respectivement, deux variantes de cette méthode qui peuvent être combinées pour dépasser ces limites.

3. Génération uniforme sans table

Si on regarde plus en détails la création et l'utilisation de la table de comptage dans la méthode de tirage décrite à la section précédente, on remarque les points suivants :

– La table est remplie, avec la relation de récurrence [3], des lignes 0 à n.

– Puis, à chaque génération d'un chemin de longueur n, la table est parcourue dans le sens inverse, à savoir de la ligne n à la ligne 0. Or, à chaque étape de la marche aléatoire, seules deux lignes (les lignes $i-1$ et i) sont utiles.

Par exemple, en considérant la table de comptage du tableau 1, la marche aléatoire qui permet de générer le chemin de longueur 5 n'aura besoin que des lignes 3 et 4 lorsqu'elle sera sur l'état s_1 pour choisir avec probabilité 1 l'état s_3.

	sommets					
n	s_0	s_1	s_2	s_3	s_4	s_5
0	0	0	0	0	0	1
1	0	0	1	1	0	0
2	0	2	0	0	1	0
3	2	0	0	1	0	0
4	0	1	0	0	0	0
5	1	0	0	0	0	0

Tableau 1. *Table de comptage associée au graphe de la figure 2, pour* $n = 5$

Ainsi, si on était capable de calculer, pour tout état s, $l_s(i-1)$ à partir des $l_s(i)$, il ne serait alors plus nécessaire de garder en mémoire toute la table de comptage, mais seulement deux lignes, que l'on mettrait à jour à chaque étape de la marche aléatoire. La méthode devient :

Prétraitement : Calculer la dernière ligne (n) de la table avec l'équation [3]. Seule la dernière ligne est conservée en mémoire.

Génération de chemins : Le principe reste identique, sauf qu'à chaque étape de la marche aléatoire, on calcule la ligne précédente de la table de comptage.

Plus précisément, pour être en mesure de calculer la ligne précédente lors du tirage, nous allons formuler le problème sous une forme matricielle. Soit $A \in \mathbb{N}^{S\times S}$ la matrice de transitions de $\mathcal{A}$, et le vecteur $F \in \mathbb{N}^S$ défini ainsi :

$$F[i] = \begin{cases} 1 & \text{si } s_i \in \mathcal{F} \\ 0 & \text{sinon.} \end{cases} \quad [4]$$

Le vecteur $L_n = A^n F$ représente, pour chaque état, le nombre de chemins de longueur n qui finissent dans un état final. La problématique de l'inversion du système de récurrences défini par l'équation [3] est alors équivalente au problème d'algèbre linéaire suivant : Trouver un entier n_0 et une matrice $B \in \mathbb{Q}^{S\times S}$ tels que

$$\forall i \geq n_0, \quad B.A^{i+1} = A^i. \quad [5]$$

L'ajout de n_0, l'entier minimum à partir duquel la relation est valide, est obligatoire car il peut exister un rang où il est impossible de retrouver la ligne précédente. Par exemple, si le modèle ne comporte aucun chemin de longueur supérieure ou égale à 7, alors A^7 est la matrice nulle et il est impossible de trouver A^6 à partir de A^7. La solution à ce problème d'algèbre est composée de deux étapes :

1) Trouver le plus petit i tel que le rang de A^{i+1} soit égal au rang de A^i, ce i est en fait n_0 [1].

2) Comme le rang de A^{n_0} est égal au rang de A^{n_0+1}, en notant f l'application linéaire associée à A et f_{n_0} celle associée à A^{n_0}, on a que la restriction de f à l'image de f_{n_0} est un isomorphisme (c.à-d., que la matrice C correspondante est inversible). Il suffit alors de calculer cette matrice C, de l'inverser et de revenir dans l'espace d'origine pour obtenir la matrice B recherchée.

L'obtention de A^{n_0} et de son rang demande des multiplications successives de matrices creuses d'entiers. Chaque entier est borné par $\mathcal{O}(d^{n_0})$ où d est le degré sortant maximal de $\mathcal{A}$; leur taille est donc en $\mathcal{O}(n_0)$. La complexité binaire de la première étape est en $\mathcal{O}(S^3 \times n_0^2)$ (ou en $\mathcal{O}(S^2 \times n_0^2)$ si la matrice est creuse). L'opération coûteuse de la deuxième étape est l'inversion de la matrice C qui est de taille $r \times r$ (r étant le rang de A^{n_0}), ce qui donne une complexité binaire en $\mathcal{O}(r^3)$ (ou en $\mathcal{O}(r^2)$ si C est une matrice creuse).

Ainsi, l'inversion d'un système de récurrences est polynomiale en la taille du système, S. Mais, comme pour le prétraitement quand on utilise la table de comptage,

1. Comme le rang de A^{i+1} est toujours positif ou nul et inférieur ou égal à celui de A^i, on a que n_0 existe toujours et qu'il vaut au plus S. On aurait pu choisir directement $n_0 = S$ mais la taille de l'automate étant grande, il est préférable d'essayer de trouver un n_0 beaucoup plus petit.

cette inversion n'a besoin d'être effectuée qu'une seule fois quelque soit le nombre de chemins tirés ensuite. Cependant, le coût de génération d'un chemin de longueur n a augmenté car le calcul de la ligne précédente de la table de comptage en utilisant les relations de récurrences définies par B nécessite $S \times d$ multiplications d'un rationnel par un grand entier. La complexité binaire d'une multiplication d'un entier de taille n par un rationnel de taille constante est en $\mathcal{O}(n)$, la complexité binaire du tirage passe en $\mathcal{O}(n^2 \times d \times S)$. Mais il faut prendre en compte le gain en taille mémoire qui est très important quand on veut tirer des chemins très longs.

4. Calcul flottant

Une variante possible (Denise *et al.*, 1999) est d'éviter le coût prohibitif de chaque opération en remplaçant l'arithmétique entière par une arithmétique flottante.

L'utilisation de nombres à virgule flottante assure un temps de calcul pour chaque opération en $\mathcal{O}(b)$ où b est la taille de la mantisse, choisie bien inférieure à n ; le coût de chaque opération devient indépendant de n et est considéré comme une constante. Les calculs sont certes approchés, mais avec l'emploi de bibliothèques telles que MPFR (Fousse *et al.*, 2007) , on a la garantie d'obtenir le réel - exprimable avec la précision choisie - le plus proche de la valeur exacte. Alain Denise et Paul Zimmermann donnent une borne théorique de la déviation maximum qui peut arriver en utilisant une arithmétique flottante certifiée. En interprétant leur théorème 4.1 avec nos notations, on a que la probabilité $\tilde{p}_n$ d'un chemin de longueur n dévie au plus de $\mathcal{O}(n \times d \times 2^{-b})$ par rapport à la probabilité uniforme p_n. Mais en pratique, que ce soit dans les expériences réalisées par Alain Denise et Paul Zimmermann ou dans celles de la section 5, l'erreur constatée est souvent beaucoup plus faible.

L'utilisation d'une arithmétique flottante permet de réduire – d'un facteur n – les complexités binaires en temps et en espace des variantes qui utilisaient une précision infinie. Le tableau 2 récapitule pour d fixé, ainsi que b et n_0, les complexités (en nombre d'opérations binaires et en mémoire) obtenues en fonction de la variante utilisée : avec table ou sans table, avec calcul exact ou flottant.

5. Expériences

Dans cette section, je présente les résultats de la comparaison, en temps et en mémoire, des quatre variantes possibles. Ainsi qu'une étude de la déviation par rapport à l'uniformité pour les deux variantes qui utilisent le calcul flottant.

5.1. *Implémentation et méthodologie*

Le calcul du système de récurrences inverses, nécessaire pour la génération de chemins sans table de comptage, est réalisé par le processus suivant : Tout d'abord,

un script SAGE (un environnement libre de calcul formel (Stein, 2007)) est créé automatiquement à partir du graphe du modèle : il commence par construire la matrice de transitions (en utilisant une représentation de matrice creuse), puis il calcule l'indice minimum à partir duquel l'inversion est possible et enfin il calcule le système de récurrences inverses à l'aide de la fonction `solve_left` de SAGE .

Toutes les expériences ont été réalisées sur un serveur *Debian 64 bits*, composé d'un processeur Intel Xeon 3GHz avec 16Gio de mémoire vive. Les graphes, au format `GraphViz` après conversion, proviennent tous de la base de modèles VLTS (Very Large Transition Systems (Garavel *et al.*, 2003)). Ces modèles correspondent à des systèmes industriels réels et leur matrice de transition est creuse (la valeur de d est petite par rapport à S). Leur nom est de la forme *vasy_X_Y*, où *X* est le nombre d'états divisé par 1000, et *Y* est le nombre de transitions divisé par 1000. Toutes les mesures ont été réalisées 10 fois et nous indiquons la valeur moyenne.

Les quatre variantes de la méthode de génération uniforme de chemins sont accessibles via une interface commune écrite en *C++* (tous les codes sources sont disponibles sur ma page Internet). Cette interface générique permet de passer aisément d'une variante à l'autre. J'ai utilisé plusieurs outils et bibliothèques que je mentionne ici : la commande `bcg_io` de la boîte à outils CADP (Garavel *et al.*, 2001) pour convertir les modèles de VLTS au format `GraphViz`; plusieurs bibliothèques *C++* de BOOST, BGL (Boost Graph Library (Siek *et al.*, 2000)) pour la manipulation de graphes, RANDOM (Maurer *et al.*, 2000) pour la génération de nombres aléatoires ; et les bibliothèques GMP (Granlund, 2007) et MPFR (Fousse *et al.*, 2007) pour le calcul exact et le calcul flottant, respectivement.

		Prétraitement		Tirage	
Méthode	Calcul	Temps	Espace	Temps	Espace
table	exact	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2 \times S)$
sans table	exact	$\mathcal{O}(n^2 \times S + S^2)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(n^2 \times S)$	$\mathcal{O}(n \times S)$
table	flottant	$\mathcal{O}(n \times S)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(n)$	$\mathcal{O}(n \times S)$
sans table	flottant	$\mathcal{O}(n \times S + S^2)$	$\mathcal{O}(S)$	$\mathcal{O}(n \times S)$	$\mathcal{O}(S)$

Tableau 2. *Récapitulatif des complexités en nombre d'opérations binaires et en espace mémoire en fonction de la variante utilisée. n désigne la longueur maximale des chemins tirés et S le nombre d'états dans le système. Dans un souci de clarté, nous avons considéré comme des constantes les valeurs suivantes : le degré maximal d de l'automate, la valeur n_0 à partir de laquelle le système d'inversion est correct, et la taille de mantisse b choisie pour les calculs flottants*

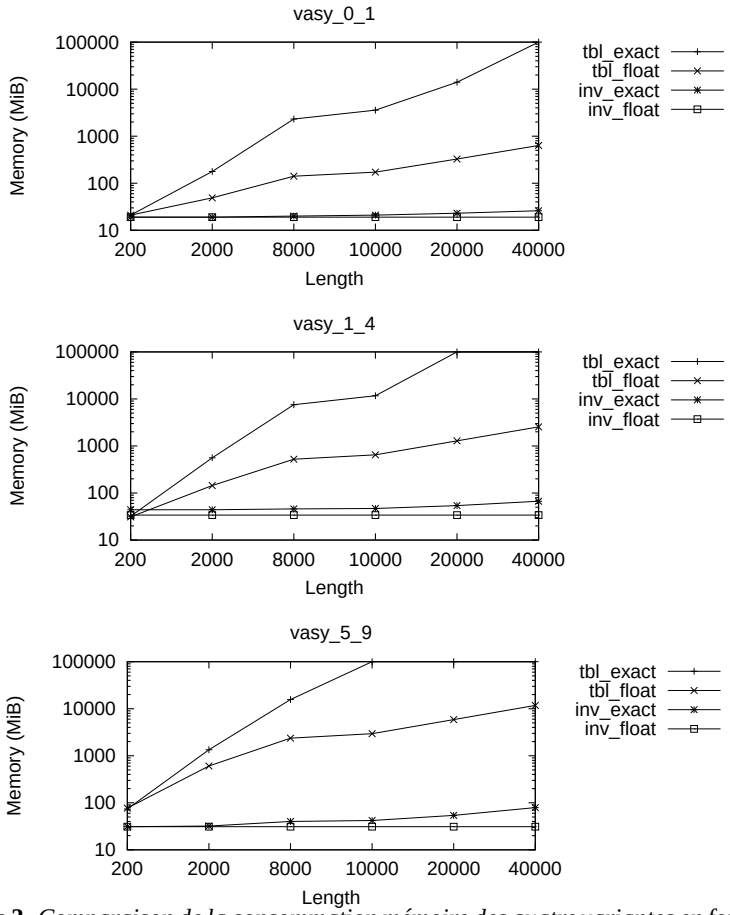


Figure 3. *Comparaison de la consommation mémoire des quatre variantes en fonction de la longueur des chemins, sur trois modèles différents*

5.2. *Consommation mémoire*

La figure 3 montre la consommation mémoire nécessaire pour chacune des quatre variantes (*tbl_exact* pour la version d'origine, à savoir le calcul de la table de comptage avec une précision infinie ; *tbl_float* pour la version aussi avec table de comptage mais où tous les calculs sont faits avec une précision de 64 bits ; et enfin *inv_exact* et *inv_float* pour les versions qui ne gardent pas la table de comptage en mémoire et où les calculs sont respectivement exacts et avec une précision de 64 bits). Pour chacun des 3 modèles, les quatre variantes ont été exécutées pour tirer des chemins dont la longueur varie entre 200 et 40000. La valeur de 100000Mio est fictive et signifie que

la méthode demandait plus de mémoire que les 16Gio disponibles. Ainsi il est impossible de tirer des chemins de longueur 10000 ou plus dans le modèle vasy_5_9 avec la méthode d'origine.

À chaque fois, la version en calcul flottant nécessite moins de mémoire que son homologue en calcul exact. En ce qui concerne les versions qui ne gardent pas en mémoire la table de comptage, elles consomment toujours beaucoup moins de mémoire que les variantes avec table de comptage, à part pour de très petites longueurs où le stockage de la matrice B, permettant d'inverser les récurrences, peut être plus coûteux que la table de comptage elle-même.

5.3. *Temps d'exécution*

La figure 4 récapitule le temps mis par chacune des quatre variantes pour tirer 100 chemins dans les 3 même modèles que précédemment. Les temps de tirage étant plus longs pour les variantes sans table de comptage, un seul chemin a été généré avec ces variantes et le temps de génération a été multiplié par 100 pour obtenir le temps de tirage de 100 chemins. De plus, la valeur fictive de 10 millions de secondes signifie que l'expérience a duré plus de 24h.

Les méthodes sans table de comptage permettent de tirer des chemins très longs en utilisant peu de mémoire. En revanche, lorsqu'il est possible de garder la table de comptage en mémoire, il est préférable d'utiliser les variantes avec table qui permettent de tirer des chemins beaucoup plus rapidement.

Les méthodes en calcul flottant sont plus rapides et consomment moins de mémoire que leurs homologues en calcul exact. On pourrait penser qu'il est toujours préférable d'utiliser le calcul flottant, mais il faut faire attention à la déviation possible par rapport à l'uniformité qui pourrait dans certains cas être problématique. C'est l'étude de la section suivante.

5.4. *Déviation par rapport à l'uniformité*

En reprenant la notation $l_s(i)$, définie à la section 2 pour le nombre de chemins de longueur i partant du sommet s calculé à l'aide de la table de comptage avec calculs exacts, et en nommant $\widetilde{l_s}(i)$ la même valeur obtenue en calcul flottant, on peut mesurer la déviation maximale, en mesurant l'erreur relative de ces deux valeurs par la formule suivante :

$$Err = \max_{\substack{s \in S \\ 0 < i \leq n}} \frac{|l_s(i) - \widetilde{l}_s(i)|}{l_s(i)} \qquad [6]$$

La figure 5 indique la valeur de Err, obtenue par la formule [6], pour les deux méthodes utilisant le calcul flottant (préfixe *tbl* pour la version avec table de comptage et

préfixe *inv* pour celle sans table de comptage) sur les 3 modèles (vasy_0_1, vasy_1_4, vasy_5_9).

À chaque fois, l'algorithme avec table de comptage est stable numériquement avec une déviation maximale inférieure à 10^{-17}. Par contre, la version sans table est instable numériquement. Cette instabilité numérique s'explique par la présence de valeurs négatives dans la matrice B qui peuvent provoquer un phénomène d'annulation entre deux nombres proches ayant pour conséquence d'aggraver les erreurs d'arrondis (Enge *et al.*, 2009).

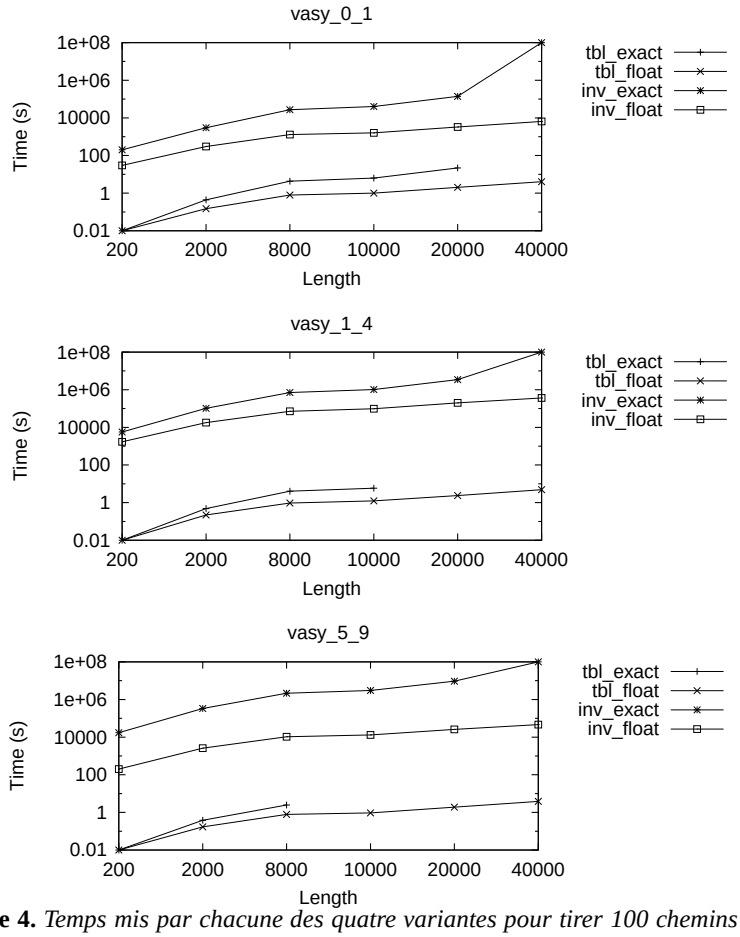


Figure 4. *Temps mis par chacune des quatre variantes pour tirer 100 chemins dans chacun des 3 modèles, en fonction de la longueur des chemins*

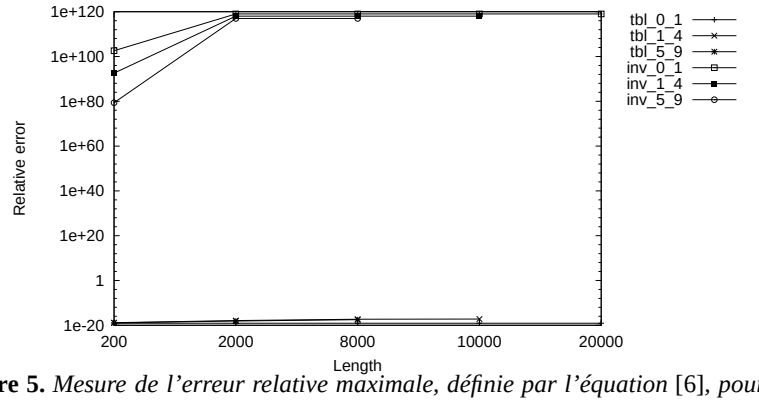


Figure 5. *Mesure de l'erreur relative maximale, définie par l'équation* [6]*, pour les deux variantes qui utilisent le calcul flottant (*inv *et* tbl*) sur les 3 modèles*

6. Conclusions et perspectives

Nous avons développé dans cet article des méthodes qui permettent l'exploration efficace de très grands modèles. Elles réalisent une exploration *aléatoire* tout en garantissant une bonne couverture des chemins du modèle quelle que soit sa topologie, ce qui n'est pas le cas avec une marche aléatoire isotrope. Notons que ces méthodes pourraient être étendues à la couverture des états et des transitions selon les principes donnés dans (Gaudel *et al.*, 2008; Gouraud *et al.*, 2001).

Une limite que l'on pouvait reprocher à la première version de cette méthode était la nécessité de la table de comptage. Cette table impose d'avoir un espace mémoire proportionnel au produit du nombre d'états de l'automate (S) par la longueur des chemins tirés (n), soit en $\mathcal{O}(n^2 \times S)$; ce qui est considérable dès lors qu'on s'intéresse à tirer de longs chemins dans de très grands modèles (Gaudel *et al.*, 2008). Mais, avec les améliorations introduites aux sections 3 et 4, l'espace mémoire nécessaire est désormais en $\mathcal{O}(S)$. Ce qui est excellent car on rappelle que dans le cas d'exploration modulaire, S désigne la taille du composant et non du système global.

De plus, les expériences réalisées dans la section 5 montrent que les variantes avec calcul flottant utilisent moins de mémoire et sont plus rapides qu'en calcul exact, au détriment de l'exactitude de l'uniformité sur les chemins (même si en pratique aucune différence n'a été constatée pour la variante avec table). La version sans table est utile si les ressources disponibles ne permettent pas de stocker la table de comptage mais le tirage est plus lent. Ces méthodes offrent des alternatives intéressantes à l'utilisation de marches aléatoires isotropes pour explorer des modèles, que ce soit à des fins de simulation, de test, ou de model-checking.

Le choix de la borne n sur la longueur des chemins peut sembler difficile à définir, mais en fonction du critère de couverture désiré, il est souvent facile à déterminer. Par exemple, si le critère est de considérer les chemins qui passent au plus une fois

dans chaque boucle, n est égal à la longueur du plus long chemin élémentaire. Dans le domaine du model-checking, cette borne dépend de la propriété à vérifier, mais c'est souvent le diamètre qui est utilisé. Néanmoins, pour les cas où on ne saurait connaître cette borne ou qu'elle soit trop grande, on peut tout à fait choisir arbitrairement un entier n, souvent à partir du nombre d'états du système. Le choix de cet entier présente des analogies avec le choix d'une fonction objectif, comme c'est le cas dans toutes les métaheuristiques qui sont les alternatives actuelles aux marches aléatoires isotropes. Une nouvelle perspective qui permettrait de s'affranchir du choix de cette borne serait d'utiliser le générateur de Boltzmann (Flajolet *et al.*, 2007) dont le paramètre permet de régler la longueur moyenne des chemins obtenus, au lieu d'avoir une longueur fixe.

Remerciements

Je remercie Matthias Krieger pour son aide précieuse qui a permis de trouver une solution au problème d'algèbre linéaire défini à l'équation [5], ainsi qu'Alain Denise et Marie-Claude Gaudel pour leurs conseils avisés.

7. Bibliographie

Denise A., Gaudel M.-C., Gouraud S.-D., « A Generic Method for Statistical Testing », *ISSRE*, p. 25-34, 2004.

Denise A., Zimmermann P., « Uniform Random Generation of Decomposable Structures Using Floating-Point Arithmetic », *TCS*, vol. 218, p. 233-248, 1999.

Dwyer M., Elbaum S., Person S., Purandare R., « Parallel Randomized State-Space Search », *ICSE*, p. 3-12, 2007.

Enge A., Lefèvre V., Théveny P., Zimmermann P., « Ecole d'été CNC'2 (Calcul Numérique Certifié) », 2009, http://www.loria.fr/~zimmerma/cnc2.html.

Flajolet P., Fusy É., Pivoteau C., « Boltzmann Sampling of Unlabelled Structures », *ANALCO*, vol. 126, SIAM Press, p. 201-211, 2007.

Flajolet P., Zimmermann P., Cutsem B. V., « A Calculus for the Random Generation of Labelled Combinatorial Structures », *TCS*, vol. 132, p. 1-35, 1994.

Fousse L., Hanrot G., Lefèvre V., Pélissier P., Zimmermann P., « MPFR : A Multiple-Precision Binary Floating-Point Library with Correct Rounding », *ACM Transactions on Mathematical Software*, vol. 33, n° 2, p. 13 :1-13 :15, June, 2007.

Garavel H., Descoubes N., « Very Large Transition Systems », , http://tinyurl.com/yuroxx, July, 2003.

Garavel H., Lang F., Mateescu R., An overview of CADP 2001, Technical Report n° 254, INRIA, 2001.

Gaudel M.-C., Denise A., Gouraud S.-D., Lassaigne R., Oudinet J., Peyronnet S., « Coverage-biased random exploration of large models », *MBT*, 2008.

Gouraud S.-D., Denise A., Gaudel M.-C., Marre B., « A New Way of Automating Statistical Testing Methods », *ASE*, p. 5-12, 2001.

Granlund T., « GNU MP : The GNU Multiple Precision Arithmetic Library, version 4.2.4 », , http://gmplib.org/manual/, 2007.

Grosu R., Smolka S., « Monte Carlo Model Checking », *TACAS*, p. 271-286, 2005.

Hérault T., Lassaigne R., Magniette F., Peyronnet S., « Approximate Probabilistic Model Checking », *VMCAI*, p. 73-84, 2004.

Knuth D. E., *Seminumerical Algorithms*, vol. 2 of *The Art of Computer Programming*, third edn, Addison-Wesley, Boston, MA, USA, 1997.

Maurer J., Abrahams D., Dawes B., Rivera R., « Boost Random Number Library », , http://www.boost.org/libs/random/, June, 2000.

Siek J., Lee L.-Q., Lumsdaine A., « Boost Random Number Library », , http://www.boost.org/libs/graph/, June, 2000.

Stein W., *SAGE Mathematics Software*, The SAGE Group. 2007, http://www.sagemath.org.

Modélisation et Détection Formelles de Vulnérabilités Logicielles par le Test Passif*

Amel Mammar[1], Ana Cavalli[1], Willy Jimenez[1]
Edgardo Montes de Oca[2]
Shanai Ardi[3], David Byers[3], and Nahid Shahmehri[3]

[1]Institut Telecom SudParis, CNRS/SAMOVAR
9 rue Charles Fourier, 91011 Evry, France
{amel.mammar,ana.cavalli,willy.jimenez}@it-sudparis.eu
[2]Montimage, 39 rue Bobillot Paris 75013, France
edgardo.montesdeoca@montimage.com
[2]Linköpings universitet, SE-58183 Linköping, Sweden
{shaar,davby,nahsh}@ida.liu.se

Abstract. L'utilisation de modélisations formelles est devenue une partie intégrante du processus de développement de logiciels sûrs. En effet, une bonne modélisation du système à développer permet d'améliorer la qualité des logiciels en détectant, par exemple, certaines vulnérabilités avant même leurs déploiements. Dans cette optique, ce papier propose une nouvelle méthode de modélisation de vulnérabilités ainsi qu'un langage formel pour l'expression précise sans ambiguïté des causes et événements pouvant les produire. La définition d'un tel langage formel permet également la détection automatique des vulnérabilités par des outils de test. Plus précisément, nous illustrons l'utilisation de l'outil de test passif *TestInv*, développé au sein de notre équipe, pour la détection automatique de vulnérabilités exprimées dans le langage formel ainsi défini. Notre approche a l'avantage de produire un nombre beaucoup plus réduit de faux positifs tout en maintenant à jour la base de connaissances de l'outil *TestInv*. L'approche proposée est illustrée à travers l'exemple de vulnérabilité CVE-2005-3192 représentant un "buffer overflow" dans un programme C.

Keywords: Vulnérabilité, modélisation, spécification formelle, détection automatique.

1 Introduction

L'impact important voire catastrophique de vulnérabilités -failles de sécurité- dans un système a amené les utilisateurs à multiplier leurs efforts pour la définition de méthodes et le développement d'outils pour la détection et l'élimination de ce type d'erreurs de programmation dès les phases préliminaires de conception de tout logiciel. Actuellement, il existe un nombre important de techniques et d'outils qui contribuent à l'amélioration de la qualité du logiciel. Comme exemples de ces techniques et outils relativement utilisés dans l'industrie, on peut citer les méthodes formelles, les techniques de vérification et de validation et également les analyseurs statiques et dynamiques de codes [S. 04, KMC06]. Nos travaux de recherche portent plus particulièrement sur les techniques de modélisation et de détection formelles de vulnérabilités. Dans ce domaine, les approches existantes sont peu nombreuses et ne se basent pas toujours sur une modélisation formelle précise des vulnérabilités qu'elles traitent [Cov08, For08, Klo08]. De plus, les outils de détection sous-jacents produisent un nombre conséquent de faux positifs et de faux négatifs. Notons également qu'il est assez difficile pour un utilisateur de savoir quelles vulnérabilités sont détectées par chaque outil vu que ces derniers sont très peu documentés.

Pour répondre au besoin de modélisation de vulnérabilités logicielles, une méthode graphique a été développée à l'Université de Linköping [ABS06, BASD06, BS07]. Cette méthode consiste à

* The research leading to these results has received funding from the European CommunityŠs Seventh Framework Program (FP 7/2007-203) under the grant agreement number 215995 (http://www.shields-project.eu/)

identifier les différents événements et conditions (causes) qui peuvent potentiellement produire une vulnérabilité donnée. Ces causes sont ensuite représentées sous forme de graphe orienté appelé *graphe de causes de la vulnérabilité* (VCG pour *vulnerability cause graph*). Les VCGs ont l'avantage de donner une vue à la fois intuitive et synthétique des différents scénarios qui produisent la vulnérabilité. Ils facilitent également la communication entre les différents acteurs participant au développement du logiciel. En dépit de ces avantages indéniables, l'aspect semi-formel des VCG ne permet pas l'utilisation d'outil pour la détection automatique de présence de vulnérabilité dans un code. Pour pallier à cet inconvénient, le présent article propose un langage formel, appelé *Condition de Détection de Vulnérabilité* (VDC pour *Vulnerability Detection Condition*), qui permet d'associer une description formelle précise et non ambigüe à chaque VCG en traduisant ses différentes causes en prédicats logiques sur lesquels il devient possible de raisonner. L'obtention d'une telle description formelle facilite le développement d'outils pour la détection automatique de vulnérabilités. Le formalisme de VDC est inspiré des travaux antérieurs de Télécom & Management SudParis et Montimage sur le test passif des protocoles [BCNZ05, CMML08]. La technique de test passif s'est avérée très efficace pour la détection de fautes; une application de cette approche pour le protocole WAP est présentée dans [BCNZ05]. Dans cet article, nous montrons l'utilisation de l'outil *TestInv*, supportant cette technique, pour la détection de vulnérabilités.

En résumé, les contributions principales de ce travail sont les suivantes :

- Un formalisme graphique VCG et un langage formel VDC pour la modélisation des vulnérabilités (Sections 2 et 3).
- Une approche de génération de l'expression formelle VDC d'une vulnérabilité à partir de sa modélisation graphique VCG (Section 3).
- Une méthode d'utilisation des VDCs pour la détection automatique de vulnérabilités (Section 4).
- L'outil de test passif *TestInv* et son application à un programme C contenant la vulnérabilité " buffer overflow ". Cette vulnérabilité a été préalablement modélisée par un VCG pour lequel un VDC a été généré (Section 4).

2 Modélisation des vulnérabilités

La modélisation des vulnérabilités [BASD06] désigne la méthode structurée de description des causes des vulnérabilités connues et récurrentes. Cette méthode consiste à analyser la vulnérabilité en question afin de déterminer les conditions et les événements pouvant produire la vulnérabilité considérée. Ces conditions et événements sont ensuite structurés sous forme de graphe appelé *graphe de causes de la vulnérabilité* (VCG pour *vulnerability cause graph*). La méthode de modélisation de vulnérabilités est similaire à celle d'analyse des causes racines dans le processus d'identification des causes à différents niveaux comme l'implémentation, la conception, les besoins, etc. [ABS06].

Un VCG est un graphe acyclique orienté composé d'un nœud *exit* représentant la vulnérabilité à modéliser et d'un certain nombre de nœuds causes dénotant des conditions ou événements pouvant produire la vulnérabilité considérée. Il existe trois types de nœuds causes: simples représentant des conditions ou événements simples, composites représentant une combinaison de conditions ou d'événements modélisés par un VCG, conjonctions représentant l'effet conjoint de deux ou plusieurs nœuds causes. Les arcs d'un VCG modélisent la succession des causes menant à la vulnérabilité.

Le VCG modélisant la vulnérabilité CVE-2005-3192 (buffer overflow dans le viewver xpdf) est représenté par la Figure 1. Les nœuds “Utilisation de buffers non adaptatifs" et “Utilisation non sûre de malloc” sont des exemples de nœud simple et composé respectivement.

3 Description formelle des causes

Les VCG sont non seulement utilisés pour la description des scénarios à risque menant à l'apparition de vulnérabilités, ils peuvent être exploités pour détecter les causes responsables de ces vulnérabilités. Afin qu'un outil de test puisse détecter automatiquement la présence de ces causes, celles-ci

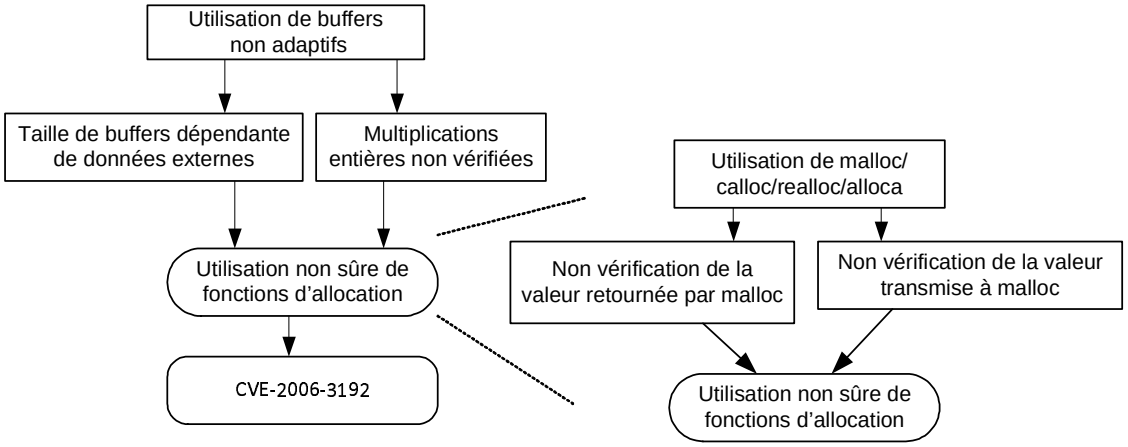


Fig. 1. Graphe de causes de la vulnérabilité CVE-2005-3192 avec nœuds simples et composites

doivent être formellement définies. Notons qu'une cause représente n'importe quel fait incluant des conditions ou des événements qui peuvent être interprétés ou pas par un outil automatique (ex. utilisation d'une fonction particulière, qualité des documents, etc.). Par conséquent, il n'est pas toujours possible d'associer une sémantique formelle aux différentes causes d'une vulnérabilité donnée. Par conséquent, les outils ne seront capables de détecter qu'un sous ensemble des causes responsables des vulnérabilités. Dans cette section, nous décrivons un formalisme permettant de décrire sans ambiguïté les causes apparaissant dans un VCG représentant une vulnérabilité.

3.1 Condition de Détection de Vulnérabilité

Un langage formel, appelé *invariants*, a été déjà défini par Bayse et. al [BCNZ05]. Basé sur les expressions régulières, ce langage permet d'exprimer des propriétés comportementales sur des systèmes communicants du domaine des télécommunications et des protocoles. Il s'agit, par exemple, de spécifier que l'occurrence d'un événement particulier e_1 est toujours précédée par l'apparition d'un autre événement e_2. Dans cet article, nous proposons d'étendre ce langage d'invariants afin de pouvoir exprimer formellement les causes d'un VCG. En effet, on devrait être capable d'exprimer des propriétés telle que:"*un espace mémoire alloué dynamiquement ne doit pas être utilisé (lu ou écrit) sans avoir vérifié préalablement le succès de l'opération d'allocation*". Pour cela, nous proposons dans ce papier une adaptation du langage d'invariants que nous désignons par *Condition de détection de Vulnérabilité* (VDC pour *Vulnerability Detection Condition*). L'idée clé du concept de VDC est de détecter l'utilisation d'une action (fonction) dangereuse sous certaines conditions particulières. Par exemple, il est dangereux d'utiliser un espace mémoire non alloué. La définition formelle du concept de VDC est comme suit.

Définition 1 *(Condition de Détection de Vulnérabilité). Soient Act un ensemble de noms d'actions, Var un ensemble de variables, et P un ensemble de prédicats sur (Var* $\cup$ *Act). Une condition de détection de vulnérabilité Vdc est définie formellement par la grammaire BNF suivante(les longs crochets désignent des éléments optionnels):*

$$Vdc ::= a/P(Var, Act)|a[/P(Var, Act)]; P'(Var, Act)$$

avec a représentant une action, $P(Var, Act)$ *et* $P'(Var, Act)$ *désignent des prédicats sur les variables Var et les actions Act. Le VDC* $a/P(Var, Act)$ *signifie que l'action a est exécutée sous des conditions spécifiques représentées par le prédicat* $P(Var, Act)$*. Similairement, le VDC* $a[/P(Var, Act)]; P'(Var, Act))$ *représente l'exécution de a, sous les conditions* $P(Var, Act)$*, suivie d'une instruction dont l'exécution satisfait* $P'(Var; Act)$*. Bien évidement, si l'action a n'est suivie d'aucune autre action, le prédicat* $P'(Var, Act)$ *est considéré comme vérifié.*

Des conditions de détection de vulnérabilités plus complexes peuvent être définies inductivement en utilisant les différents connecteurs logiques.

Définition 2 *(Conditions de Détection des Vulnérabilités: cas général). Si* Vdc_1 *et* Vdc_2 *sont deux conditions de détection des vulnérabilités, alors* $(Vdc_1 \vee Vdc_2)$ *et* $(Vdc_1 \wedge Vdc_2)$ *sont également des conditions de détection des vulnérabilités.*

3.2 Quelques exemples de VDCs

La condition de détection de vulnérabilité Vdc_1 suivante peut être utilisée pour détecter l'accès à une zone mémoire non allouée ou libérée. Le prédicat $Assign(x, y)$ dénote l'assignation d'une valeur y à une variable mémoire x, $IsNotAllocated$ vérifie si la mémoire x est non allouée:

$$Vdc_1 = Assign(x, y)/IsNot\_Allocated(x)$$

Dans les langages de programmation comme C/C++, certaines fonctions peuvent entraîner des vulnérabilités si ces dernières sont appliquées hors limites de ses arguments. L'utilisation de variables entachées (tainted en anglais) comme argument d'une fonction d'allocation mémoire est un exemple bien connu d'une telle vulnérabilité exprimée par la condition de détection de vulnérabilité Vdc_2 ci-dessous. Une variable est dite entachée si sa valeur est issue d'une source non sûre. Une telle valeur peut être obtenue à partir d'une lecture dans un fichier, d'une entrée de l'utilisateur, etc.

$$Vdc_2 = memoryAllocation(S)/tainted(S)$$

Une bonne pratique de la programmation consiste à vérifier la valeur retournée par une fonction d'allocation mémoire même si cette dernière n'est pas utilisée dans le programme. La condition de détection de vulnérabilité Vdc_3 permet de détecter l'absence d'une telle instruction de vérification:

$$Vdc_3 = (u := memoryAllocation(S)); notChecked(u, null)$$

3.3 Génération formelle de VDC à partir de VCG

En utilisant le concept de conditions de détection de vulnérabilité (VDC) défini dans la section précédente, il est possible d'associer une description formelle pour un VCG représentant une vulnérabilité. La traduction d'un VCG en un VDC est réalisée par les étapes suivantes:

1. Identification des causes quantitatives. Les causes d'un VCG sont classifiées en deux catégories:

   Causes quantitatives. Elles peuvent être détectées et vérifiées suivant un processus formel sans intervention humaine. L'utilisation d'une fonction particulière d'un langage, comme l'allocation mémoire, est un exemple de cette catégorie.

   Causes qualitatives. Elles ne peuvent pas être évaluées ou vérifiées sans intervention humaine. L'évaluation de ce type de causes est complètement subjective et peut différer d'une personne à une autre. L'appréciation de la clarté de rédaction d'un document est un exemple de cette catégorie car elle dépend du niveau d'expertise du relecteur.
   Comme seules les causes quantitatives peuvent être détectées automatiquement, la traduction d'un VCG en VDC concerne uniquement ce type de causes.
2. Identification des scénarios. Dans le VCG, chaque chemin ayant pour cible le nœud *exit* représente un scénario potentiellement dangereux.
3. Identification de l'action maîtresse et des autres actions de chaque scénario. Les causes quantitatives peuvent être classées en actions et conditions.

   Une action désigne un point particulier dans le programme où une tâche ou une instruction modifiant la valeur d'un objet donné est exécutée. Comme exemples d'actions, nous pouvons citer l'affectation de variables, la copie d'une zone mémoire, l'ouverture d'un fichier, etc.

Une condition désigne un état particulier du programme défini par la valeur et le statut de chaque variable. Pour un buffer par exemple, nous pouvons déterminer s'il est alloué ou pas.
Chaque scénario doit contenir une action maître *Act_Master* qui produit la vulnérabilité. Tous les autres nœuds de ce scénario représentent des conditions notées $\{C_1, \ldots, C_n\}$ sous lesquelles l'action *Act_Master* est exécutée.
Parmi ces conditions, une condition particulière C_k, appelée *condition omise*, peut exister. Cette condition doit être satisfaite par l'éventuelle action suivant l'action *Act_Master*.

4. Définition formelle des prédicats: l'utilisateur doit exprimer chaque condition par un prédicat formel.
5. En se basant sur les définitions 1 et 2, expression de la condition de détection de vulnérabilité associée à chaque scénario. Soient $\{P_1, \ldots, P_k, \ldots, P_n\}$ les prédicats modélisant les conditions $\{C_1, \ldots, C_k, \ldots, C_n\}$. La condition de détection de vulnérabilité exprimant formellement ce scénario dangereux est définie par:

$$Act\_Master/(P_1 \wedge \ldots \wedge P_{k-1} \wedge P_{k+1} \ldots \wedge P_n); P_k$$

6. Définition du VDC global représentant le VCG comme la disjonction des VDCs associés aux différents scénarios (Vdc_i désigne le VDC associé à chaque scénario i):

$$Vdc_1 \vee \ldots \vee Vdc_n$$

Considérons par exemple la vulnérabilité CVE-2005-3192 de débordement mémoire associée au viewer PDF xpdf. Cette vulnérabilité est due à la copie d'une donnée utilisateur dans une zone mémoire (buffer) allouée dynamiquement sans avoir vérifié au préalable la taille de la donnée par rapport à celle de la zone. En effet, un utilisateur malveillant pourrait exploiter cette faille en exécutant un code particulier dans l'espace de l'application. Ainsi, l'attaquant peut avoir accès à la machine et aux données critiques qu'elle peut contenir. Le code le plus vulnérable a été trouvé dans la fonction textttStreamPredictor:: StreamPredictor utilisée dans le fichier textttStream.c.

Comme indiqué par le VCG associé à cette vulnérabilité (Voir Figure 2), la donnée entière fournie par l'utilisateur n'est pas vérifiée quand cette dernière est utilisée par la fonction *StreamPredictor* pour le calcul de la taille d'un buffer. Les différentes causes du VCG sont numérotées afin de pouvoir les référencer plus facilement dans la suite du papier.

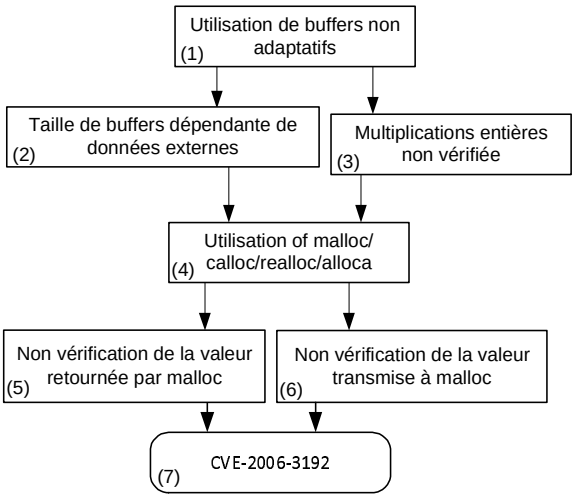


Fig. 2. Graphe de causes de la vulnérabilité du viewer PDF xpdf

D'après la Figure 2, quatre scénarios sont possibles pour produire la vulnérabilité considérée:

$$(1,2,4,5,7),(1,2,4,6,7)$$
$$(1,3,4,5,7),(1,3,4,6,7)$$

Nous devons donc définir le VDC associé à chacun de ces scénarios. La méthode de définition des Vdcs étant similaire pour tous ces scénarios, seul le calcul du Vdc du premier scénario est présenté dans ce papier. Pour le scénario $(1,2,4,5,7)$, l'action maîtresse qui peut conduire à une vulnérabilité est l'utilisation d'une fonction d'allocation mémoire (nœud 4) sous les conditions suivantes:

L'usage de buffers non adaptatifs : le programme utilise des buffers dont les tailles sont fixées à l'exécution ou la compilation. Ce type de buffers ne peut contenir qu'une quantité limitée de données. Par conséquent, toute tentative d'écriture au delà de cette capacité conduirait à un débordement mémoire. Contrairement à ce type de buffers, les buffers adaptatifs ont la possibilité d'adapter leur taille pour contenir la donnée assignée. Pour chaque buffer non adaptatif, le prédicat suivant est vérifié:

$$nonAdaptiveBuffer(B)$$

La taille du buffer est dépendante de données externes: la taille d'un buffer alloué dynamiquement est calculée, entre autres, à partir de données utilisateur pour permettre à ce dernier de manipuler sa taille. Si une taille importante de ce buffer peut engendrer un déni de service, une taille trop petite peut conduire également à un débordement mémoire. Pour chaque buffer B dont la valeur est obtenue d'une source non sécurisée, le prédicat suivant est vérifié:

$$tainted(B)$$

Utilisation de malloc/calloc/realloc: le code considéré utilise des fonctions de gestion de mémoire similaires à celles du langage C telles que `malloc`, `calloc` ou `realloc` pour l'allocation de mémoire. Pour la fonction d'allocation mémoire f, le prédicat suivant est vérifié:

$$memoryAllocation(f)$$

La valeur retournée par la fonction malloc n'est pas vérifiée: le programme ne contient pas de mécanisme pour traiter de manière sûre le défaut de mémoire (i.e. traiter les valeurs nulles retournées par la fonction `malloc`). Exécuter des programmes sujets à des débordements mémoires non contrôlés peut générer des comportements imprévisibles qui peuvent être exploités par des utilisateurs malveillants. Cette cause est détectée quand la valeur u retournée par une fonction d'allocation n'est pas vérifiée pour savoir si elle est nulle ou pas. Par conséquent, pour chaque valeur u retournée par une fonction d'allocation mémoire, le formule suivante est définie:

$$notChecked(u, null)$$

Enfin, le VDC modélisant le scénario $(1,2,4,5,7)$ est défini par:

$$u := f(B)/\left(\begin{array}{c} memoryAllocation(f) \\ \wedge \\ nonAdaptiveBuffer(B) \\ \wedge \\ tainted(B) \end{array}\right); notChecked(u, null)$$

Le VDC ci-dessus exprime une potentielle vulnérabilité quand:

1. une fonction donnée d'allocation mémoire f est utilisée avec un buffer non adaptatif B dont la valeur est produite d'une source non sûre,
2. la valeur retournée par la fonction f n'est pas vérifiée pour s'assurer qu'elle n'est pas nulle.

4 Détection des vulnérabilités par les VDCs

Dans cette section, nous illustrons notre approche de détection de vulnérabilité basée sur le test passif et la notion de VDC définie précédemment. Nous présentons également une application de cette approche pour montrer comment il est possible de détecter l'occurrence de la vulnérabilité CVE-2005-3192 par l'outil de test *TestInv* développé par Montimage en collaboration avec l'équipe de recherche de Télécom & Management SudParis [CMML08]. *TestInv* est aujourd'hui capable d'analyser des traces exécutables produites durant l'exécution d'un programme et de faire le lien entre les traces et le code source du programme étudié.

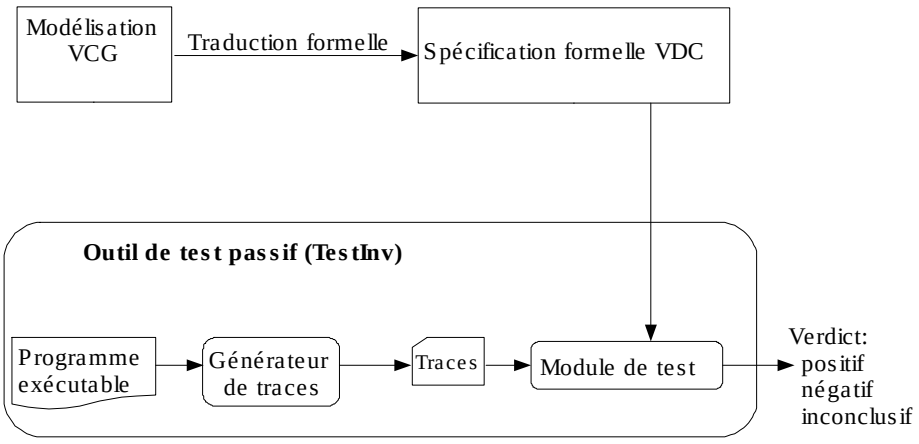


Fig. 3. Test passif pour la détection de vulnérabilités

4.1 Test passif pour la détection de vulnérabilités

Le test passif a pour objectif de détecter des erreurs dans un système en observant uniquement ses entrées/sorties (ou d'autres caractéristiques observables) sans perturber son fonctionnement normal. Cette technique de test consiste à collecter des traces d'exécution du système (une partie des instructions exécutées) et détecter les éventuelles erreurs ou déviations de ces traces vis-à-vis du modèle formel du système. Le modèle formel peut être une spécification du système [ACC+04, LNS+97, MA01], des propriétés formelles que ce dernier doit satisfaire, ou des formules de VDCs comme c'est le cas dans le présent papier. La Figure 3 montre le processus de test passif pour la détection de vulnérabilités. Comme outil de test passif, *TestInv* accepte comme entrée des formules VDCs afin de détecter la présence des vulnérabilités considérées dans des traces d'exécution d'un système. La détection de vulnérabilités procède comme suit:

1. *Modélisation de la vulnérabilité:* la vulnérabilité à détecter est représentée par un VCG.
2. *Traduction du VCG en VDC:* cette étape est réalisée par un expert ayant la connaissance nécessaire du langage cible mais également du code assembleur généré par le compilateur utilisé. Cet expert doit pouvoir identifier les mots clés et les concepts qu'une trace d'exécution peut contenir.
3. *Extraction des traces d'exécution:* en utilisant l'outil *TestInv*, des traces d'exécution contenant toutes les informations nécessaires au test des formules VDCs sont analysées. L'outil permet de déduire différentes informations sur les variables du programme: le caractère sûr ou pas de la source d'une donnée, c'est-à-dire le caractère non teinté ou teinté de la donnée, le fait qu'une variable soit initialisée ou pas, contrôlée ou pas, etc. Comme indiqué dans la Figure 3, l'outil prend en charge l'exécution du programme, produit des traces pour chaque instruction (desassemblage) et le module de test analyse ces traces, garde les informations sur les variables et utilise les VDCs pour déterminer s'il y a une vulnérabilité.

4. *Vérification de la présence de vulnérabilité:* finalement, la détection de la présence éventuelle de la vulnérabilité est opérée par l'outil *TestInv* qui produit un diagnostic avec le numéro de ligne dans le code (C/C++). Pour pouvoir déterminer la présence d'une vulnerabilité, il est nécessaire de compiler le programme avec des informations pour le deboggage. Si l'information n'est pas disponible, alors l'outil indiquera seulement l'addresse de l'instruction dans l'exécutable. En fonction de la présence ou pas de la vulnérabilité, le résultat est soit positif, soit négatif. Il peut être également inconclusif si l'outil ne connaît pas toutes les informations nécessaires à l'évaluation des VDCs.

Il est important de noter que l'étape 1 (resp. étape 2), bien que manuelle, n'est réalisée qu'une seule fois à chaque nouvelle découverte de vulnérabilité (resp. cause). De plus, les étapes 3 et 4 peuvent être semi-automatisées si des informations nécessaires à l'évaluation des VDCs, telle que l'adaptativité ou non d'un buffer, ne peuvent être obtenues que par l'utilisateur.

4.2 Étude de cas

Dans cette section, nous illustrons l'application de notre approche sur un petit programme C contenant une vulnérabilité de débordement mémoire similaire à celle du CVE-2005-3192. Pour détecter la vulnérabilité, notre outil doit être capable de déterminer:

- si une fonction d'allocation mémoire est utilisée grâce à une liste prédéfinie de fonctions pour chaque langage de programmation,
- les différentes variables destinées à recevoir les valeurs de retour de ces fonctions,
- si une variable donnée est teintée ou pas en utilisant le générateur de traces,
- si la taille d'une variable donnée a été vérifiée ou pas[1].

C code	TestInv
size_t bytes_read;	
char tmp[7];	
bytes_read = read(fd, tmp, 6);	*tmp* et *bytes_read* sont teintés
tmp[bytes_read]='\0';	utilisation d'une variable teinté comme position pour changer un buffer sans la vérifier avant
int i=atoi(tmp);	i est teintée
char *buff;	
buff=malloc(i);	malloc dépend de la valeur de i sans la vérifier avant
strcpy(buff,"x");	dépend de la valeur de i et la valeur retournée par malloc n'est pas vérifiée

Table 1. Type d'analyses faites par l'outil *TestInv*

Le tableau 1 représente les types d'analyse faits par l'outil sur une sélection de lignes de code d'un exemple de programme contenant la vulnérabilité considérée. Notons que si la taille de i est vérifiée avant l'appel à la fonction *malloc* et que la valeur retournée est également vérifiée, alors nous n'aurons aucune vulnérabilité. La vulnérabilité n'est signalée que si l'ensemble des conditions nécessaires sont vérifiées: i est teintée, la taille de i n'est pas vérifiée, et *buff* n'est pas vérifié avant utilisation.

La Figure 4 montre les traces de l'exemple et les diagnostics produits par l'outil. Les endroits où l'outil détecte les variables teintés sont signalés par des commentaires en italique. Comme TestInv vérifie toutes les conditions avant de signaler la présence d'une vulnérabilité, le nombre de faux positifs est nettement réduit comparé à d'autres outils existants qui pourraient être utilisés pour

[1] Cette vérification est utile pour la condition modélisant le nœud 3 qui appartient à d'autres scénarios non détaillés dans ce papier.

détecter ce type de vulnérabilités comme Valgrind avec le plug-ins memcheck [NS07] (qui détecte si un buffer est mal utilisé) et flayer[2] (qui détecte si une variable est taintée)

5 Conclusions et perspectives

L'utilisation des outils automatiques pour le test des aspects sécurité est fortement recommandée par les meilleures démarches de développement de produits logiciels. Dans cet article, nous proposons une spécialisation du concept de VCG afin de pouvoir détecter automatiquement, par les outils de test, les vulnérabilités qu'ils modélisent. Pour cela, nous définissons le langage formel VDC qui permet d'associer une sémantique formelle aux VCG et aux causes qu'ils incluent. Une telle modélisation formelle des VCG est utilisée par les outils de test pour une détection automatique de la présence des vulnérabilités considérées. Nous avons également montré comment on pouvait exploiter l'outil de test passif *TestInv* pour l'analyse de la trace d'exécution d'un programme en vue de détection de vulnérabilités potentielles. Notons que le taux de faux positifs est inversement proportionnel au degré de précision des VCG. Comme les modèles de vulnérabilités sont indépendants de l'outil de test, il devient possible à n'importe quel utilisateur de mettre à jour les modèles de vulnérabilités déjà existants en ajoutant de nouveaux modèles. Ceci est un avantage indéniable par rapport aux outils commerciaux existants où des modèles de vulnérabilités ne peuvent être ajoutés ou mis à jour que par leurs propriétaires.

Afin de montrer la faisabilité de notre approche, nous travaillons actuellement sur l'application de notre méthode pour la modélisation de différents types de vulnérabilités logicielles. Nous envisageons également d'apporter quelques améliorations à l'outil de test Passif *TestInv* afin qu'il soit capable de déduire encore plus d'informations sur les variables du code à analyser.

References

[ABS06] S. Ardi, D. Byers, and N. Shahmehri. Towards a structured unified process for software security. In *Procedeengs of the ICSE 2006 Workshop on Software Engineering for Secure Software (SESS06)*, Shanghai, China, 2006.

[ACC+04] B. Alcalde, A. R. Cavalli, D. Chen, Davy Khuu, and David Lee. Network protocol system passive testing for fault management: A backward checking approach. In *FORTE*, pages 150–166, 2004.

[BASD06] D. Byers, S. Ardi, N. Shahmehri, and C. Duma. Modeling software vulnerabilities with vulnerability cause graphs. In *Procedeengs of the International Conference on Software Maintenance*, Philadelphia, PA, USA, 2006.

[BCNZ05] E. Bayse, A. Cavalli, M. Núnez, and F. Zaïdi. A passive testing approach based on invariants: application to the wap. *Computer Networks and ISDN Systems*, 48(2):247–266, 2005.

[BS07] D. Byers and N. Shahmehri. Design of a process for software security. In *Proceedings of the Second International Conference on Availability, Reliability and Security, ARES2007*, Vienna, Austria, 2007.

[CMML08] A. R. Cavalli, E. Montes de Oca, W. Mallouli, and M. Lallali. Two complementary tools for the formal testing of distributed systems with time constraints. In *The 12th IEEE International Symposium on Distributed Simulation and RealTime Applications*, Vancouver, Canada, October 2008. IEEE Computer Society.

[Cov08] Coverity. Prevent, 2008. http://www.coverity.com/ (accessed September).

[For08] Fortify Software. Fortify SCA, 2008. http://www.fortifysoftware.com/products/sca (accessed September).

[Klo08] Klocwork. K7, 2008. http://www.klocwork.com(accessed September).

[KMC06] Chunguang Kuang, Qing Miao, and Hua Chen. Analysis of software vulnerability. In *ISP'06: Proceedings of the 5th WSEAS International Conference on Information Security and Privacy*, pages 218–223, Stevens Point, Wisconsin, USA, 2006. World Scientific and Engineering Academy and Society (WSEAS).

[2] http://www.code.google.com/p/flayer

```
...
bytes_read = read(fd, tmp, 6);
 80483e5:       mov    DWORD PTR [esp+0x8],0x6
 80483ed:       lea    eax,[ebp-0x1b]                  <-- va être teintée par l'outil car la valeur
                                                           depend d'un fichier extern
 80483f0:       mov    DWORD PTR [esp+0x4],eax
 80483f4:       mov    eax,DWORD PTR [ebp-0x14]
 80483f7:       mov    DWORD PTR [esp],eax
 80483fa:       call   804831c <read@plt>
 80483ff:       mov    DWORD PTR [ebp-0x10],eax  <-- va être teintée par l'outil car la valeur de
                                                           retour depend d'un fichier extern
/home/ed/test.c:12
tmp[bytes_read]='\0';
 8048402:       mov    eax,DWORD PTR [ebp-0x10]
 8048405:       mov    BYTE PTR [ebp+eax-0x1b],0x0  <-- utilisation d'une variable teintée
                                                           comme position pour changer un buffer
FOUND VULNERABILITY: use_of_tainted_value_to_change_buffer : line /home/ed/test.c:12
/home/ed/test.c:13
int i=atoi(tmp);
 804840a:       lea    eax,[ebp-0x1b]
 804840d:       mov    DWORD PTR [esp],eax
 8048410:       call   804832c <atoi@plt>
 8048415:       mov    DWORD PTR [ebp-0xc],eax    <-- va être teintée par l'outil car la valeur
                                                           depend d'une variable teinté
/home/ed/test.c:15
char *buff;
buff=malloc(i);
 8048418:       mov    eax,DWORD PTR [ebp-0xc]
 804841b:       mov    DWORD PTR [esp],eax
 804841e:       call   804833c <malloc@plt>
 8048423:       mov    DWORD PTR [ebp-0x8],eax
FOUND VULNERABILITY: use_of_tainted_value_to_determine_buffer_size : line /home/ed/test.c:15
/home/ed/test.c:16
strcpy(buff,"x");
 8048426:       mov    eax,DWORD PTR [ebp-0x8]
 8048429:       mov    WORD PTR [eax],0x78
FOUND VULNERABILITY: did_not_check_return_value : line /home/ed/test.c:16
...
Program terminated normally (Exit status: 0x0008)
Error disassembling next instruction (address: 0xB7DC74C0)
TOTAL NUMBER OF 3 VULNERABILITIES FOUND
```

Fig. 4. Exemple de traces et des vulnérabilités détectées par *TestInv*

[LNS+97] D. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *ICNP '97: Proceedings of the 1997 International Conference on Network Protocols (ICNP '97)*, page 113, Washington, DC, USA, 1997. IEEE Computer Society.

[MA01] R. E. Miller and K. A. Arisha. Fault identification in networks by passive testing. In *Advanced Simulation Technologies Conference (ASTC)*, pages 277–284. IEEE Computer Society, 2001.

[NS07] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[S. 04] S. Redwine and N. Davis. Processes to produce secure software. 2004. Task Force on Security Across the Software Development Lifecycle, Appendix A.

Combining Frama-C and PathCrawler for C Program Debugging

Omar Chebaro[1,2], Nikolai Kosmatov[1], Alain Giorgetti[2,3], and Jacques Julliand[2]

[1] CEA, LIST, Laboratoire Sûreté des Logiciels, PC 94, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr
[2] LIFC, University of Franche-Comté, 25030 Besançon Cedex France
firstname.lastname@lifc.univ-fcomte.fr
[3] INRIA Nancy - Grand Est, CASSIS project, 54600 Villers-lès-Nancy France

Extended Abstract

Software validation remains a crucial part in software development process. Software testing accounts for about 50% of the total cost of software development. Automated software validation is aimed at reducing this cost. The increasing demand has motivated much research on automated software validation. Two major techniques have improved in recent years, dynamic and static analysis. Traditionally, they were viewed as separate domains.

Static analysis examines program code and reasons over all possible behaviors that might arise at run time. It is often necessary to use approximations. Static analysis is conservative and sound: the results may be weaker than desirable, but they are guaranteed to generalize to all executions. Dynamic analysis operates by executing a program and observing this execution. Dynamic analysis is efficient and precise because no approximation or abstraction needs to be done: the analysis can examine the actual, exact run-time behavior of the program for the corresponding test case. It can be as fast as program execution.

The pros and cons of the two techniques are apparent. If dynamic analysis detects an error then the error is real. However, it cannot in general prove the absence of errors. On the other hand, if static analysis reports a potential error, it may be a false alarm. However, if it does not find any error (of a particular kind) in the overapproximation of program behaviors then the analyzed program clearly cannot contain such errors.

Recently, there has been much interest in combining dynamic and static methods for program verification. Static and dynamic analyses can enhance each other by providing valuable information that would otherwise be unavailable. This paper reports on an ongoing project that aims to provide a new combination of static analysis and structural testing of C programs. We implement our method using two existing tools: Frama-C, a framework for static analysis of C programs, and PathCrawler, a structural test generation tool.

Frama-C [1] is being developed in collaboration between CEA LIST and the ProVal project of INRIA Saclay. Its software architecture is plug-in-oriented and allows fine-grained collaboration of analysis techniques. Static analyzers are implemented as plug-ins and can collaborate with one another to examine a C program. Let us introduce the value analysis plug-in based on abstract interpretation. This plug-in computes and stores supersets of possible value ranges of variables at each statement of the program. Among

other applications, these over-approximated sets can be used to exclude the possibility of a run-time error. The value analysis is correct: it emits an alarm for an operation whenever it cannot guarantee the absence of run-time errors for this operation. The value analysis memorizes abstract states at each statement and provides an interface for other plug-ins to extract these states.

Developed at CEA LIST, PathCrawler [2] is a test generation tool for C functions respecting *the all-paths criterion,* which requires to cover all feasible program paths, or *the k-path criterion,* which restricts the generation to the paths with at most k consecutive iterations of each loop. The PathCrawler generation method is similar to the so-called *concolic,* or *dynamic symbolic execution.* The user provides the C source code of the function under test. The generator's main loop is rather simple: given a partial program path π, the main idea is to symbolically execute it using constraints. A solution of the resulting constraint solving problem will provide a test case exercising a path starting with π. Then concrete execution of the test case allows to obtain the complete path. The partial paths are explored in a depth-first search.

We present an original combination of static analysis and structural test generation for validation of C programs, in particular, for detection of run-time errors. The main idea is to call first a static analysis tool (Frama-C) in order to generate alarms for the statements for which the absence of run-time errors is not ensured by static analysis. Second, these alarms are transferred to a structural test generation tool (PathCrawler) where they guide test generation trying to confirm alarms by activating bugs on some test cases. Our ongoing implementation of this method, called SANTE, assembles two heterogeneous tools using quite different technologies (such as abstract interpretation and constraint logic programming).

We evaluate our method by several experiments on real-life C programs, and compare the results with other methods. Static analysis alone will in general just generate alarms (some of which may be false alarms), whereas our method allows to confirm some alarms as real bugs and provides a test case activating each bug. This is done automatically, avoiding time-consuming alarm analysis by the validation engineer, at least for confirmed alarms, the task which requires significant expertise, experience and deep knowledge of source code. Stand-alone test generation, when it is not guided by generated alarms for some statements, does not detect as many bugs as our combined method. When guided by the exhaustive list of alarms for all potentially threatening statements (not filtered by static analysis), test generation usually has to examine more infeasible paths and takes more time than our combined method (or even times/spaces out). In all cases, our method outperforms the use of each technique independently.

References

1. Frama-C: A framework for static analysis of C programs (2007-2010) http://frama-c.cea.fr/.
2. Botella, B., Delahaye, M., Hong-Tuan-Ha, S., Kosmatov, N., Mouy, P., Roger, M., Williams, N.: Automating structural testing of C programs: Experience with PathCrawler. In: AST'09, Vancouver, Canada (May 2009)

Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Une architecture de composants répartis adaptables

An Phung-Khac, Jean-Marie Gilliot et Maria-Teresa Segarra

Département Informatique, TELECOM Bretagne
Technopôle Brest-Iroise, CS83818
F-29238 Brest Cedex 3

{an.phungkhac,jm.gilliot,mt.segarra}@telecom-bretagne.eu

RÉSUMÉ. *Plusieurs travaux récents proposent des solutions ou des frameworks dédiés au développement d'applications adaptables qui peuvent ainsi dynamiquement changer leur comportement pendant l'exécution afin de s'adapter au contexte d'exécution courant. Cependant, avec ces approches, les tâches à la charge des développeurs sont encore complexes. Ces tâches incluent la définition des variantes et la spécification des actions d'adaptation, qui dans le contexte des systèmes répartis, incluent des contraintes liées aux parties distribuées. Dans cet article, nous présentons une approche de développement d'applications réparties adaptables permettant la génération correcte des variantes d'une application et des actions d'adaptation à exécuter en vue de faciliter la tâche des développeurs.*

ABSTRACT. *Adaptive applications modify their behavior according to the current execution context by changing from a consistent variant to another one. The variants include implementations, configurations, parameters, architectures, etc. In the context of distributed, component-base applications, specifying consistent variants and transitions between them is critical to ensure the correctness of the collaborations after adaptation. In this paper, we present an approach that helps developpers build correct architectural variants and transitions between them for such class of applications.*

MOTS-CLÉS : *adaptation architecturale, composants répartis, approche IDM.*

KEYWORDS: *architecture adaptation, distributed components, MDE approach.*

.

1. Introduction

Les applications informatiques sont omniprésentes dans notre environnement. Celles-ci doivent désormais pouvoir fonctionner dans des contextes très variables, même si ceci requiert la modification de leurs structures internes sans même s'interrompre. Le contexte comprend les dispositifs, les réseaux, le comportement de l'environnement et de l'utilisateur. On parle d'adaptation dynamique.

Plusieurs travaux récents proposent des solutions ou des frameworks dédiés au développement de telles applications adaptables qui peuvent dynamiquement changer leur comportement pendant l'exécution afin de s'adapter au contexte d'exécution courant. Cependant, avec ces approches, les tâches de développement d'une application répartie adaptable à la charge des développeurs sont encore complexes. Cette complexité est dûe notamment aux deux tâches suivantes :

– *Spécification des variantes de l'application* : Lors d'une session d'adaptation, l'application doit changer d'une variante cohérente à une autre. Dans le cas des applications réparties, une telle variante se compose de parties distribuées qui collaborent pour offrir des fonctions particulières. La spécification de telles variantes peut alors s'avérer une tâche difficile.

– *Spécification des transitions entre variantes* : les transitions d'adaptation entre variantes sont aussi complexes. Chaque transition comprend des actions de changement d'architecture, de configuration et/ou de paramètre. En plus, lorsque l'application gère des données, il est important que celles-ci soient transférées à la nouvelle variante. Ce transfert est d'autant plus complexe que les applications sont distribuées, les actions devant alors être réparties.

Dans cet article, afin de faciliter ces tâches, nous présentons une approche de développement d'applications réparties adaptables grâce à laquelle les variantes architecturales d'une application et des actions d'adaptation à réaliser sont correctement et automatiquement générées.

Dans notre approche, une application répartie est d'abord spécifiée au niveau abstrait comme l'interconnexion d'un ensemble de composants clients et d'un composant de communication (le composant de communication est aussi appelé *médium*). Puis, par un processus de raffinement, cette spécification abstraite est transformée en plusieurs variantes d'implantation. Celles-ci sont ensuite regroupées dans un nouveau médium que nous appelons *médium adaptable* et des contrôleurs d'adaptation y sont greffés qui sont en charge d'effectuer les changements d'une variante à une autre. Le médium adaptable ayant une structure clairement définie et connue, la planification des actions d'adaptation peut être automatisée.

L'article est organisé de la manière suivante. La section 2 introduit la notion de composant de communication ou médium sur laquelle repose notre proposition. Nous

utilisons cette architecture dans notre approche de développement de médiums adaptables qui sera montré dans la section 3. Ensuite, la section 4 discute des travaux connexes. Enfin, la section 5 conclut l'article.

2. Médium

2.1. *Définition*

La notion de médium a été proposée au sein de notre équipe dans (Cariou *et al.*, 2002). Un composant de communication (ou *médium*) est un composant logiciel dans le sens où il est défini par une interface de services offerts et requis. La particularité de ce composant réside dans le fait qu'il est dédié à la communication et donc qu'il a par nature une mise en œuvre répartie. Un composant de communication n'est pas une unité de déploiement. C'est une unité architecturale logique construite pour être distribuée. Une application est le résultat de l'interconnexion d'un ensemble de composants clients et de composants de communication. Cette particularité des composants de communication présente un atout majeur dans le sens où elle permet de séparer dans une application la définition de l'aspect fonctionnel (dans les composants clients) de la définition de l'aspect communication (dans les composants de communication) (Cariou *et al.*, 2002).

2.2. *Exemple : médium de réservation*

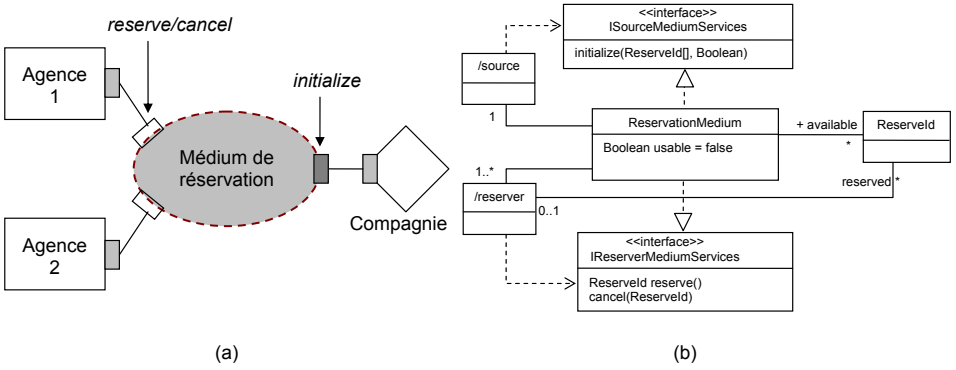


Figure 1. *Spécification abstraite du médium de réservation*

À titre d'illustration, nous reprenons et simplifions l'exemple publié dans (Cariou *et al.*, 2002) sur une compagnie aérienne possédant un siège et des agences localisées sur des sites distants. Le problème consiste à attribuer un identificateur de réservation

ou numéro de siège d'un avion à une personne souhaitant effectuer une réservation pour un vol donné. L'ensemble des identificateurs de réservation disponibles (*available* dans la figure 1b) est initialisé par le siège de la compagnie. Les identificateurs de réservation peuvent être gérés par un composant de communication qui offre par exemple, un service *setReserveIdSet* pour initialiser les identificateurs, un service *reserve* pour effectuer une réservation et un service *cancel* pour annuler une réservation. On peut construire une application de réservation en interconnectant ce composant de communication avec les composants représentant le siège et les agences de la compagnie aérienne comme illustré sur la figure 1a. Dans ce cas, pour effectuer (respectivement annuler) une réservation, chaque agence se connecte au composant et utilise le service *reserve* (respectivement *cancel*). De même, pour initialiser les identificateurs, le siège de la compagnie se connecte au composant et utilise le service *setReserveIdSet*.

2.3. *Répartition, déploiement*

Au niveau abstrait, un composant de communication est représenté par une seule entité logique. Cette entité logique n'est pas une unité de déploiement. Lors du déploiement, il est éclaté en plusieurs éléments appelés gestionnaires de rôle (ou *Managers*). Chacun des gestionnaires de rôle est associé à un composant client et est déployé avec lui sur le même site (figure 2). Chaque gestionnaire de rôle implante les services offerts par le composant de communication au composant client auquel il est associé et communique avec les autres gestionnaires de rôle pour assurer le service de communication. Le composant de communication est alors défini par l'agrégation logique des gestionnaires de rôle.

Les gestionnaires de rôle gèrent aussi les données du médium. Plusieurs stratégies ou algorithmes différents (Chord (Stoica *et al.*, 2001), Pastry (Rowstron *et al.*, 2001), ...) peuvent être utilisés pour implanter la gestion des données du composant de communication par les gestionnaires de rôle. Chaque stratégie aboutit à une implantation spécifique (centralisée, distribuée, répartie, mixte, etc.) du composant de communication. Plusieurs variantes d'implantation réparties peuvent donc exister pour l'architecture de déploiement d'un composant de communication. Dans (Kaboré *et al.*, 2007), un processus de raffinement de médiums, de sa spécification abstraite aux variantes d'implantation de l'architecture (plus de 10 variantes) a été présenté. Ce processus se compose de plusieurs étapes correspondant à des préoccupations séparées au début du processus. A chaque étape, des choix de conception sont introduits. De cette façon, le processus de raffinement assure la cohérence fonctionnelle des parties distribuées de chaque variante architecturale du médium.

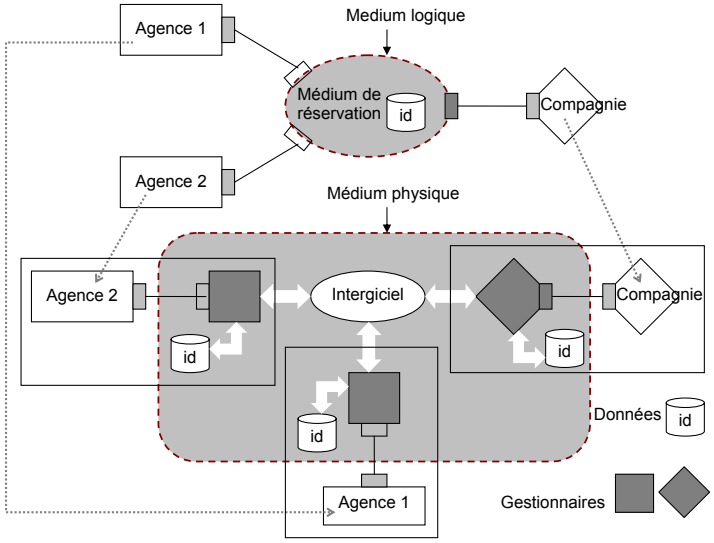


Figure 2. *Apperçu de l'architecture de déploiement du médium de réservation*

3. Médium adaptable

3.1. *Approche proposée*

Nous avons utilisé la notion de médium ((Cariou *et al.*, 2002)) et le processus de raffinement de médium ((Kaboré *et al.*, 2007, Kaboré *et al.*, 2008)) pour construire des applications réparties adaptables. La figure 3 montre le principe de notre approche :

– *Spécification abstraite :* Afin de développer une application répartie adaptable, nous utilisons le médium pour spécifier l'application. Elle est donc spécifiée comme l'interconnexion d'un médium et des composants fonctionnels.

– *Génération des variantes architecturales :* A partir de la spécification abstraite, nous utilisons le processus de raffinement pour transformer cette spécification abstraite en des spécifications d'implantation correspondant aux choix de conception introduits. Les choix de conception sont identifiés grâce à des préoccupations ((Kaboré *et al.*, 2007)).

– *Spécification des transferts des données entre les variantes :* Le processus de raffinement se compose de plusieurs étapes qui raffinent pas à pas la spécification abstraite au niveau d'implantation en plusieurs variantes. Pendant ce processus, nous spécifions aussi les actions de transfert des données entre ces variantes en identifiant des fonctions à utiliser pour récupérer et distribuer ces données.

– *Composition :* Les variantes architecturales sont composées dans une architecture de médium adaptable appelée *adapt-medium*. Cette architecture est constituée de deux parties : fonctionnelle et adaptative. La partie fonctionnelle contient les variantes architecturales dont une est activée pendant l'exécution. La partie adaptative correspond à un contrôleur d'adaptation qui utilise la spécification des transitions pour générer et exécuter des plans d'adaptation afin de changer la variante activée et transférer des données.

L'application répartie adaptable est donc constituée du médium adaptable et les composants clients. Elle peut dynamiquement changer la variante de médium lorsque le contexte change.

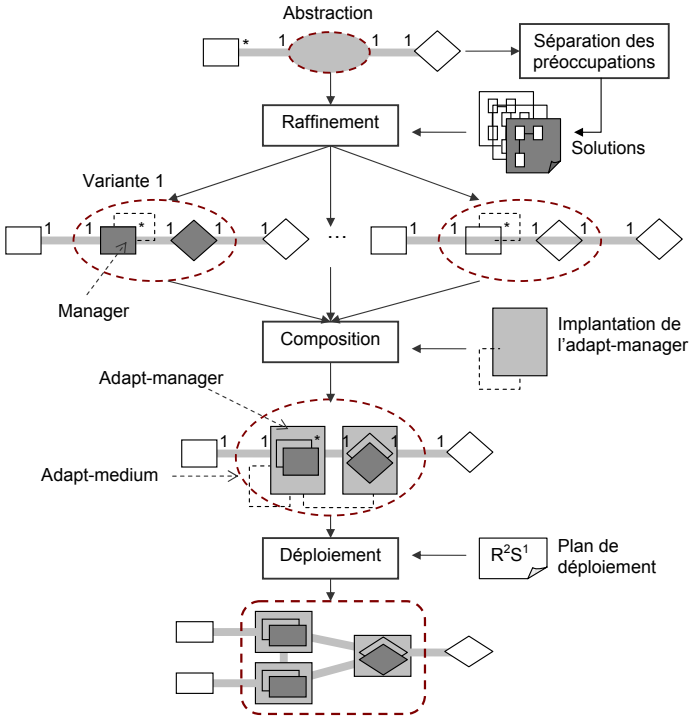


Figure 3. *Notre approche de développement d'applications réparties adaptables*

Dans les sections suivantes, nous précisons comment la génération des variantes, la spécification des transitions et la composition sont réalisées sur un exemple de médium adaptable de réservation. Ce médium peut être utilisé dans des applications de réservation qui peuvent dynamiquement changer de variante de médium afin de s'adapter au contexte.

3.2. *Génération des variantes architecturales*

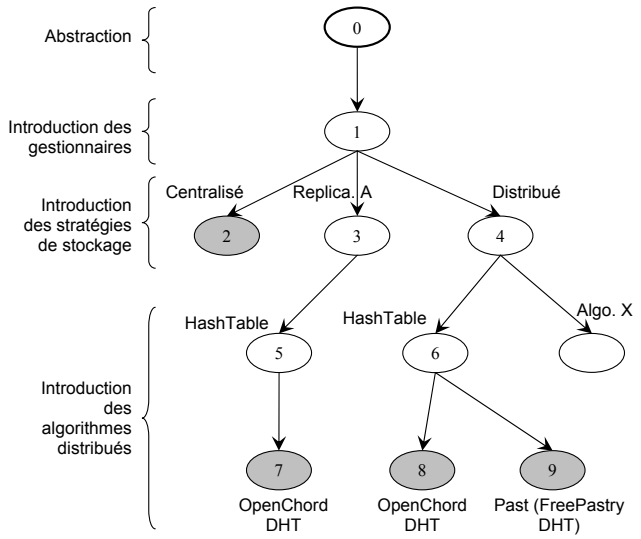


Figure 4. *Processus de raffinement du médium de réservation*

Le processus de raffinement se compose de plusieurs étapes qui transforment pas à pas la spécification abstraite en plusieurs spécifications d'implantation en introduisant des choix de conception. La figure 4 montre le processus de raffinement du médium de réservation. Comme le montre la figure, la spécification abstraite (le nœud 0) est raffinée en quatre variantes d'implantations (les nœuds 2, 7, 8, 9).

Le nœud 1 correspond à la spécification du médium où les gestionnaires de rôle sont introduits. Comme le montre la figure 5, un gestionnaire est associé à un composant client, les gestionnaires offrent alors les services du médium comme les portes de communication des composants clients.

Dans cette étape, les données (l'ensemble des identificateurs disponibles *available*) sont encore gérées par une partie abstraite du médium (la classe *ReservationMedium*). Le problème majeur dans la définition d'une implantation répartie d'un médium porte sur la gestion de la répartition de ces données. Pour faciliter cette définition, nous décomposons la gestion de la répartition de l'ensemble *available* en deux préoccupations successives :

– Stratégie de stockage de données : Les identificateurs disponibles peuvent être stockés de manière centralisée chez un gestionnaire *ReserverManager* (nœud 2), répliquée sur un ensemble des gestionnaires *ReserverManager* (nœud 3), ou distribué sur les gestionnaires *ReserverManager* (nœud 4).

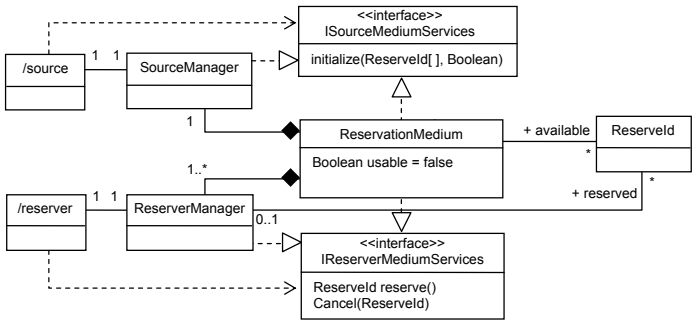


Figure 5. *Introduction des gestionnaires de rôle*

– Algorithme réparti : Les identificateurs disponibles sont partagés entre les gestionnaires *ReserverManager*. Lorsqu'un gestionnaire ne gère pas l'identificateur requis, il a besoin d'un algorithme pour le trouver. Les algorithmes peuvent être classifiés par le type de données d'indexation utilisé. Par exemple lorsque les identificateurs sont distribués ou répliqués, OpenChord ((Bamberg University, Distributed System Group, 2007)) qui implante l'algorithme Chord ((Stoica *et al.*, 2001)) peut être utilisé.

La figure 6 indique la spécification correspondant au nœud 6. Les gestionnaires de données distribuées *SourceDistributedDataManager* et *ReserverDistributedDataManager* sont introduits. Ils implantent la stratégie distribuée de stockage de données. L'ensemble des identificateurs disponibles est maintenant distribué sur les gestionnaires de rôle *ReserverManager* comme des ensembles *localAvailable*. Pour un gestionnaire de rôle *ReserverManager*, un gestionnaires de données *ReserverDistributedDataManager* est le proxy de l'ensemble *available*.

Dans cette étape, le type de données *HashTable* a été choisi pour l'indexation des identificateurs disponibles. Ces données *hashTable* sont encore gérées par la partie abstraite *ReservationMedium* du médium.

L'étape suivante du raffinement permet de générer les nœuds 8 et 9. Dans cette étape, les objets des algorithmes sont introduits, la partie abstraite du médium disparaît, les spécifications sont complètement au niveau d'implantation.

3.3. *Spécification du transfert de données entre les variantes*

La génération automatique des plans d'adaptation pour un médium adaptable requiert l'identification des actions d'adaptation sur l'architecture du médium et des actions pour le transfert de données entre variantes. Les permières peuvent être obtenues à partir des étapes du processus de raffinement décrit dans le paragraphe précédent. Pour les deuxièmes, nous identifions les actions élémentaires de transfert de données

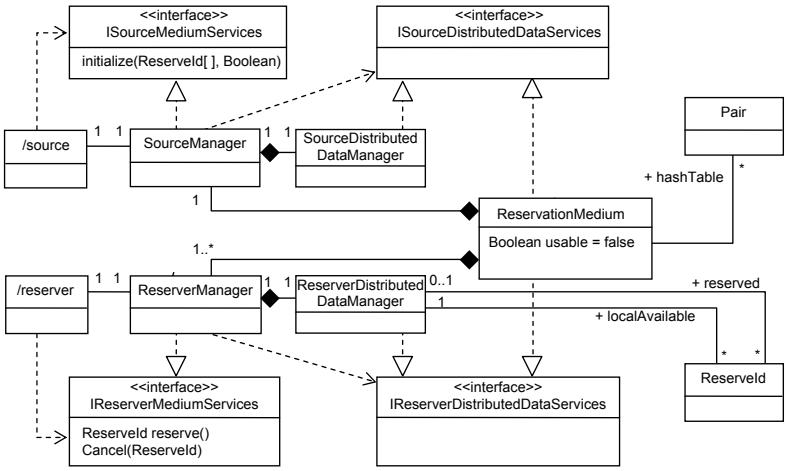


Figure 6. *Introduction de stratégies de stockage de données*

entre deux nœuds connexes dans l'arbre de raffinement. Les actions comprennent les services du médium à utiliser pour transférer ces données. Par exemple, pour la chaîne des choix de conception de la spécification abstraite (nœud 0) à la variante correspondant au noeud 8 (appelée V8), les actions élémentaires identifiées sont les suivantes :

```
(0) -> (1):
  // Il n'y a pas de donnée à transféfer
(1) -> (4):
  // distribution de l'<available>
  for ID in (1).available
  {
    (4).SourceManager.insert(ID)
    // le service <insert> est implanté dans <SourceManager>
  }
(4) -> (1):
  // restaurer l'<available>
  (1).available = null
  for m in {all managers}
  {
    (1).available.add(m.localAvailable)
  }
(4) -> (6):
  // Il n'y a pas de donnée à transréfer
(4) -> (8):
  // Construire le table de hachage pour <OpenChord>
  for m in {all managers}
```

```
  {
    for ID in m.localAvailable
    {
      m.ChordObject.add(ID,m.name)
    }
  }
(8) -> (4):
  // Il n'y a pas de donnée à restaurer
```

Pour la variante correspondant au nœud 2 (appelée V2), les actions élémentaires identifiées sont les suivantes :

```
(0) -> (1):
  // Il n'y a pas de donnée à transférer
(1) -> (2):
  (2).serverNode.available = (1).available
(2) -> (1):
  (1).available = (2).serverNode.available
```

En utilisant ces actions élémentaires, nous pouvons générer les actions de transfert des données entre ces deux variantes. Par exemple, pour transférer les données de la variante V2 (centralisée) à la variante V8 (distribué, OpenChord), le chemin est (2)->(1)->(4)->(8) et les actions nécessaires de transfert sont :

```
variante V2 à variante V8 : {
  // (2) -> (1) :
  (1).available = (2).serverNode.available
  // (1) -> (4):
  for ID in (1).available
  {
    (4).SourceManager.insert(ID)
  }
  // (4) -> (8):
  for m in {all managers}
  {
    for ID in m.localAvailable
    {
      m.ChordObject.add(ID,m.name)
    }
  }
}
```

Ces actions élémentaires sont composées avec les variantes architecturales dans un médium adaptable où les actions de transfert de données sont générées dynamiquement.

3.4. *Composition*

Les variantes architecturales et les actions élémentaires de transfert de données sont intégrées dans un médium adaptable appelé *adapt-medium*. Ce médium adaptable est une agrégation logique de gestionnaires de rôle adaptables (*adapt-managers*), un par composant client. Nous avons implanté les *adapt-managers* en utilisant le canevas d'adaptation DYNACO (Buisson *et al.,* 2005) qui separe les méchanismes d'adaptation en un décideur, un planificateur, et un exécuteur.

Comme le montre la figure 7, chaque gestionnaire de rôle adaptable se compose d'un composite (*CompositeManager*) qui encapsule les variantes possibles pour le gestionnaire, d'un contrôleur d'adaptation (*AdaptationController*) et d'un composant *MediumLogic*. Le *CompositeManager* contient les variantes du gestionnaire de rôle associé à un même composant client. Par exemple, la figure 8 montre l'architecture du gestionnaire de réservation pour les variantes V2 et V8. Ces variantes seront intégrées dans le *adapt-manager* associé au client de réservation. En ce qui concerne le *AdaptationController*, il est constitué de trois composants : le *decider* décide du moment de lancer une session d'adaptation et la variante à adopter, le *planner* utilise les actions élémentaires pour générer le plan d'adaptation à exécuter (transformation architecturale et transfert de données) et le *executor* qui réalise effectivement les actions. Le composant *MediumLogic* gère l'architecture globale des variantes du médium adaptable.

Le processus de raffinement ainsi que la composition pour obtenir un médium adaptable peuvent être automatisés par des transformations de modèles ((Kaboré *et al.,* 2007, Phung-Khac *et al.,* 2008)).

4. Travaux connexes

Plusieurs travaux récents proposent des solutions ou des frameworks dédiés au développement d'applications adaptatives qui peuvent dynamiquement changer leur comportement afin de s'adapter au contexte courant. Mais, à notre connaissance, il n'existe par de solutions permettant la planification automatique des adaptations distribuées.

Dans le contexte des applications adaptables basées sur des composants, (Buisson *et al.,* 2005, Chefrour, 2005, Kon *et al.,* 2005) ont proposé des plateformes spécialisables qui offrent des mécanismes d'adaptation/reconfiguration. Ces travaux se concentrent sur la structure des méchanismes d'adaptation des applications. Avec ces approches les tâches de planification de l'adaptation sont prises en charge par les développeurs lors du développement.

Dans (Ayed *et al.,* 2007, Geihs *et al.,* 2006), les auteurs ont proposé des approches de composition d'applications adaptatives. Cependant, ces approches ne sont pas applicables pour les systèmes ayant des besoins d'adaptation distribuée.

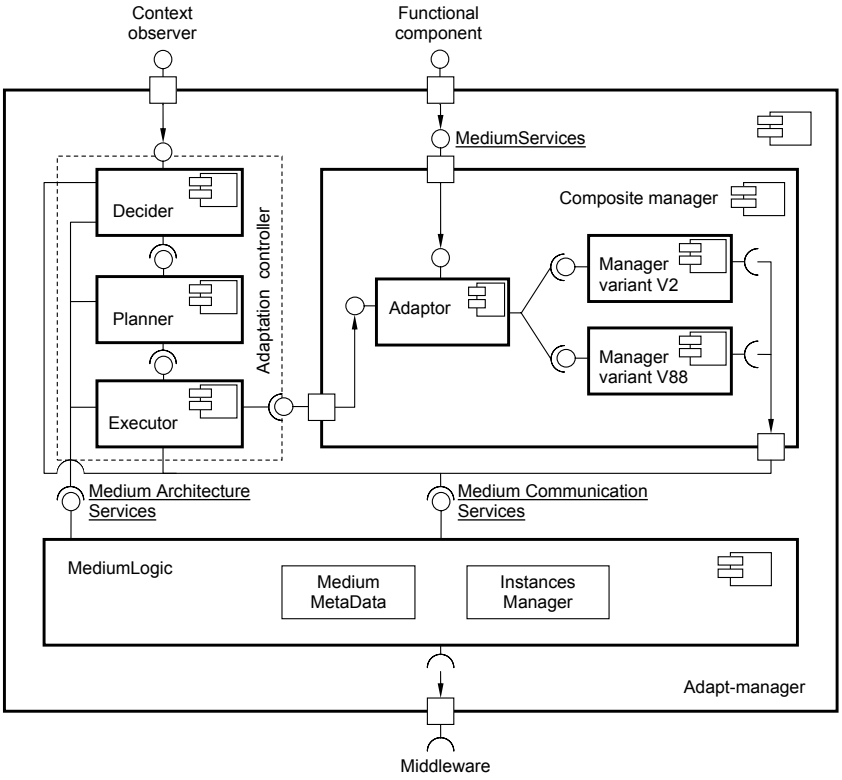


Figure 7. *Architecture du composant* adapt-manager

Dans le domaine de la robotique, certains travaux proposent des solutions pour la planification automatique des actions d'adaptation. Par exemple, (Sykes *et al.*, 2008) a proposé un modèle en trois couches où les plans d'adaptation sont générés à partir des modèles de but exprimés en logique temporelle. Cependant, les plans n'impliquent pas des applications distribuées.

Dans (Bencomo *et al.*, 2008), les auteurs proposent une approche de modélisation de variantes architecturales des applications basées sur les composants. Mais ces variantes ne sont pas distribuées.

5. Conclusion

Nous avons présenté notre approche de développement d'applications réparties adaptables qui peut simplifier les tâches de construction des variantes des applications et de planification des transitions lors d'une adaptation.

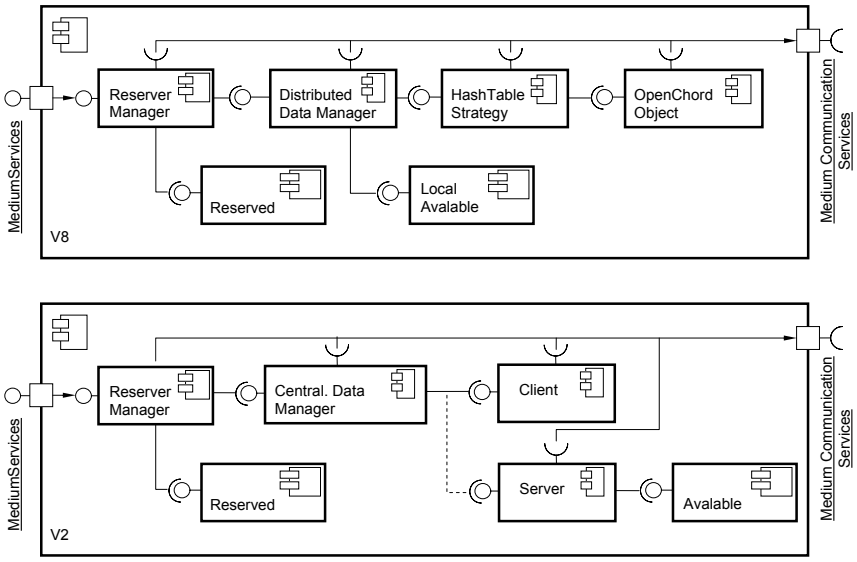


Figure 8. *Modèles de composants des gestionnaires des variantes V2 et V8*

Dans notre approche, une application distribuée est d'abord spécifiée au niveau abstrait comme l'interconnexion d'un ensemble de composants client et d'un composant de communication ou médium. Puis, par un processus de raffinement réifié par des transformations de modèles, cette spécification est transformée dans plusieurs variantes d'implantation. Chacune des variantes est constituée d'un gestionnaire de rôle par composant client, chaque type de composant client ayant un type de gestionnaire de rôle. Pour chacun de ces types, nous regroupons les variantes d'implantation qui ont été obtenues par le processus de raffinement. Enfin, nous introduisons un contrôleur d'adaptation par gestionnaire de rôle qui s'occupe de gérer le changement dynamique de la variante courante. L'application résultant de l'interconnexion des composants clients et de notre médium adaptatif peut ainsi changer l'architecture de communication afin de satisfaire les besoins exprimés.

Notre médium adaptatif comportant les différentes variantes et les plans de transfert de données, la session d'adaptation peut être menée en trois étapes : 1) démarrage de la nouvelle variante, 2) transfert des données de l'ancienne variante à la nouvelle et 3) changement de variante active.

Le fait de nous appuyer sur un processus basé sur des modèles nous permet effectivement de réaliser de manière automatique la coordination des changements de variantes et les plans de transfert des données. Le concepteur de telles applications est ainsi déchargé d'un des aspects les plus complexes de la construction et peut alors se concentrer sur la spécification des variantes souhaitées.

Enfin, nos travaux se sont focalisés sur l'automatisation de la planification et exécution d'adaptations distribuées. Notre hypothèse de travail a été, donc, l'existence d'une fonction de décision (Buisson *et al.*, 2005) qui déclenche les adaptations lorsque ceci est nécessaire. L'ajout d'une telle fonction dans notre approche requiert d'enrichir le processus de raffinement avec des informations sur le(s) contexte(s) d'exécution pertinent(s) pour chaque variante d'implantation et d'intégrer dans le médium adaptatif une fonctionnalité de décision exploitant cette information.

6. Bibliographie

Ayed D., Berbers Y., « Dynamic Adaptation of CORBA Component-Based Applications », *Proceedings of the 2007 ACM symposium on Applied Computing (SAC'07)*, ACM Press, p. 580-585, 2007.

Bamberg University, Distributed System Group, « OpenChord », http ://www.uni-bamberg.de/projects/openchord, 2007.

Bencomo N., Blair G., Flores C., Sawyer P., « Reflective Component-based Technologies to Support Dynamic Variability », *2nd Workshop on Variability Modelling of Software-intensive Systems (VaMoS08)*, Germany, 2008.

Buisson J., André F., Pazat J.-L., « A framework for dynamic adaptation of parallel components », *ParCo 2005*, 2005.

Cariou E., Beugnard A., Jézéquel J.-M., « An Architecture and a Process for Implementing Distributed Collaborations », *Proceedings of the 6th IEEE International Enterprise Distributed Object (EDOC 2002)*, IEEE Computer Society, Lausanne, Switzerland, p. 132-143, September, 2002.

Chefrour D., « Developing component-based adaptive applications in mobile environments », *SAC '05 : Proceedings of the 2005 ACM symposium on Applied computing*, ACM Press, New York, NY, USA, p. 1146-1150, 2005.

Geihs K., Khan M. U., Reichle R., Solberg A., Hallsteinsen S., Merral S., « Modeling of Component-Based Adaptive Distributed Applications », *Proceedings of the 2006 ACM symposium on Applied Computing (SAC'06)*, ACM Press, p. 718-722, 2006.

Kaboré E., Beugnard A., « Automatisation d'un processus », *RSTI - L'Objet*, vol. 13/1007, p. 105-135, 2007.

Kaboré E., Beugnard A., « Implementing a Data Distribution Variant with a Metamodel, Some Models and a Transformation », *Proceeding of the 8th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'08)*, Lecture Notes in Computer Science, Springer-Verlag, Oslo, Norway, June, 2008. to appear.

Kon F., Marques J. R., Yamane T., Campbell R. H., Mickunas M. D., « Design, implementation, and performance of an automatic configuration service for distributed component systems : Research Articles », *Softw. Pract. Exper.*, vol. 35, n° 7, p. 667-703, 2005.

Phung-Khac A., Beugnard A., Gilliot J.-M., Segarra M.-T., « Model-Driven Development of Component-based Adaptive Distributed Applications », *Proceeding of the 23rd ACM Symposium on Applied Computing (SAC'2008), track on Dependable and Adaptive Distributed Systems (DADS)*, ACM Press, Fortaleza, Ceará, Brazil, March, 2008.

Rowstron A., Drusche P., « Pastry : Scalable, distributed object location and routing for large-scale peer-to-peer systems. », *IFIP/ACM Middleware*, novembre, 2001.

Stoica I., Morris R., Karger D., Kaashoek M. F., Balakrishnan H., « Chord : A Scalable Peer-to-Peer Lookup Service for Internet Applications », *ACM SIGCOMM Conference*, San Diego, 2001.

Sykes D., Heaven W., Magee J., Kramer J., « From goals to components : a combined approach to self-management », *SEAMS '08 : Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, ACM, p. 1-8, 2008.

OCL contracts for the verification of model transformations

Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam

Université de Pau et des Pays de l'Adour / Liuppa
B.P. 1155, 64013 Pau Cedex France
{Eric.Cariou, Nicolas.Belloir, Franck.Barbier}@univ-pau.fr,
Nidal.Djemam@etud.univ-pau.fr

Abstract. A model-driven engineering process relies on a set of transformations which are usually sequentially executed, starting from an abstract level to produce code or a detailed implementation specification. These transformations may be entirely automated or may require manual intervention by designers. In this paper, we propose a method to verify that a transformation result is correct with respect to the transformation specification. This method both includes automated transformations and manual interventions. For that, we focus on transformation contracts written in OCL. This leads to making the proposed method independent of modeling and transformation tools. These contracts are partially or fully generated through a dedicated tool.

1 Introduction

A software development process based on model-driven engineering relies on a set of transformations. They are usually executed in series to start from an abstract level of modeling and to arrive at the final code or to a detailed implementation specification. These transformations can be entirely automated or require the manual intervention of designers. Indeed, even if the main goal of model-driven engineering is to automate a complete software process, designers must often intervene on models and make choices about particular actions to be carried out.

It is important to be able to guarantee that the realized transformations are valid, *i.e.* they respect the transformation specification. This is an even more important issue when designers deal manually with models. In this case, models can be extensively modified without particular constraints and it should be made sure that modifications occur within the required scope.

In [6,5], we established conceptual bases of specifying transformations through model transformation contracts written in OCL (Object Constraint Language [11]). A couple of models verify a transformation if they respects the associated contract. In this paper, we validate in practice this approach by providing a method and its implementation in the context of endogenous transformations (i.e., metamodels remain constant when transforming). Our method allows the contract verification to be carried out starting from two models – one representing the

source model and the other the target model of the transformation – generated or obtained as output of any tool, including when designers process models manually. The choice of OCL as language expression contract is justified by the fact that OCL is usable within several technological spaces and is a standard relatively well known and accepted by model designers.

Compared to our previous work, we also deeply discuss key issues, such as the OCL contract expression context problem and mappings between elements of the source and the target models. We show that contract writing can be greatly facilitate thanks to automatically generated mapping functions through adequate tools. Finally, we point out the limitation of the single context for expressing OCL constraints, which leads to problems when manipulating simultanously several models like in the case of model transformations.

Section 2 recalls the concept of model transformation contract. Section 3 gives an example of a model transformation and of its contract. Section 4 presents the method of definition and verification of a contract and applies it to our example. It also presents dedicated tools that help in writing and verifying contracts in the context of the Eclipse/EMF platform[1]. Lastly, before the conclusion, we discuss related works and the choice of OCL as contract expression language.

2 Model transformation contracts

Programming and designing by contracts [7,2] consist in specifying what does a software component, a program or a model, in order to know how to properly use it. Design by contracts also allows the assessment of what has been computed with respect to the expressed contracts. In terms of computing operations, being operations at code level or operations in models, contracts are embodied in pre and post-conditions. A pre-condition defines the state of a system to be respected (if not, the system is in abnormal state) so that operations can be called. Post-conditions establish the state of a system to respect after calls. One can also specify invariants that have to be permanently respected. In contrast with pre and post-conditions, invariants are not expressed in relation with operations inputs and outputs.

In the research context about model transformations, we proposed in [6,5] to apply these principles to specific operations: those of model transformations. We thus want to express contracts on transformations themselves. These contracts describe expected model transformation behaviors. Formally, constraints on the state of a model before the transformation (source model) are described. Similar constraints on the state of the model after the transformation (target model) are required. Post-conditions guarantee that a target model is a valid result of a transformation with respect to a source model. Pre-conditions ensure that a source model can effectively be transformed.

A transformation contract is defined by three distinct sets of constraints:

Constraints on the source model : constraints to be respected by a model to be able to be transformed

[1] `http://www.eclipse.org/emf`

Constraints on the target model : general constraints (independently of the source model) to be respected by a model for being a valid result of the transformation

Constraints on element evolution : contraints to be respected on the evolution of elements between the source model and the target model, in order to ensure that the target model is the correct transformation result according to the source model contents

3 Model transformation example: interface addition

As an illustration, we study in this article a transformation that consists in a refinement wich adds interfaces to a UML class diagram. To simplify the writing of the associated contract, we provide a simplified UML meta-model because the original meta-model is relatively complex in terms of element number and of navigation possibilities between these elements. This simplified meta-model is exposed and discussed in the next section.

3.1 Meta-model of class diagrams

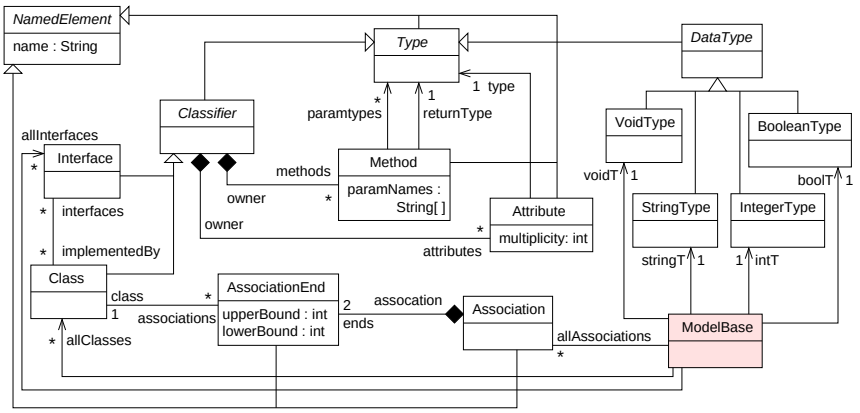


Fig. 1. Meta-model of simplified class diagrams

Figure 1 is the meta-model of simplified class diagrams. One finds there the same basic concepts as in a UML class diagram: class, interface, type, method, attribute, association as well as four primitive types (Void, Boolean, Integer and String). The concepts which are not present, by simplification, are for example visibilities or specializations. The `ModelBase` element (Fig. 1 on the right hand side, bottom) has a special role: it represents, as its name shows, the base of the

model. It does not define a concept of the domain but is a help in navigating on the model by giving direct access to the whole set of the main model elements. This kind of element is not mandatory but it can be useful in model manipulation with some tools, particulary in the context of the Eclipse/EMF platform we used.

Some OCL constraints, the well-formedness rules, have to be added to supplement the class diagram of the meta-model, for instance to check that an interface does not have an attribute.

3.2 Example of refinement: addition of interfaces

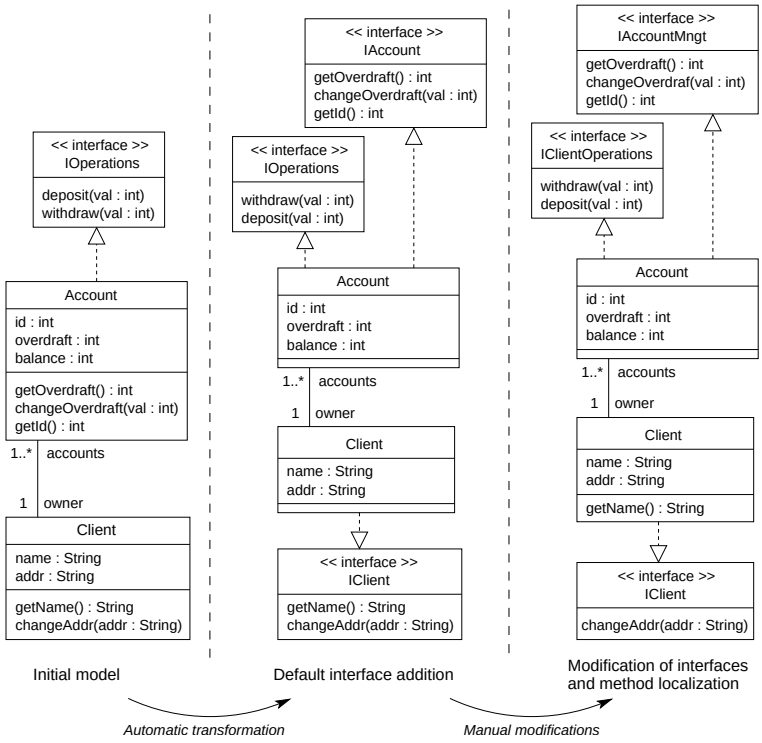


Fig. 2. Refinement example: addition and modification of interfaces

The principle of refinement used in our example is to add one or more interfaces to each class and to move a part or all the methods implemented by this class to its factorized interfaces. This refinement is carried out in two steps:

1. Automatic addition of an interface by default: for each class of the diagram, we create (except if it already existed) an interface named `IclassName`.

All the methods directly supported by the class are then moved to this interface. This operation is entirely automated and is realized by a model transformation tool.
2. Possible revised layout of methods and interfaces: the addition of an interface by default is not systematically appropriate. Designers may then manually modify for each class the list of its interfaces (renaming, suppression, addition) as well as the method localizations (moving of a method towards another interface or the initial class).

Finally, after these two steps, we obtain a target model which is the transformation of the source model and corresponds to an interface addition refinement. The main constraint to satisfy for having the refinement correct is that each class always implements the same list of methods, directly or through its interfaces. In a detailed way, the transformation contract associated with this refinement is the following:

- Constraints on the source model: none, any class diagram can be refined;
- Constraints on the target model: each class implements at least one interface (even if it does not contain methods);
- Constraints on element evolution from the source model towards the target model: all the classes are maintained and each implements, directly or indirectly through its interfaces, the same method list as before the transformation.

Figure 2 shows an example of such a refinement, with its two steps. We can notice that the designer has renamed the two interfaces of the `Account` class and that the `getName` method has been straightforwardly replaced in the `Client` class. We observe that the refinement is correct since all the classes are preserved and each class always implements, directly or indirectly, the same list of methods. Finally, each class implements at least one interface.

4 Transformation verification by contract

In this section, we present in details a method for contract definition and verification. This method is the most generic as possible. It must thus be applicable to two models, one supposed the source model and the other the target model of the transformation or the refinement, defined or obtained from a large variety of tools, including when the designer intervene manually on models.

4.1 Contract expression context for checking element evolution

In [6], we defined two techniques to write a transformation contract, and more precisely the constraints on the evolution of the elements between the source model and the target model. These constraints are special because they require at the same time to refer to the source model and to the target model. OCL constraints being expressed in a single context, it is then necessary to be able to manipulate both models from a single context.

Specification of the transformation operation To carry out this problem, the first technique consists in attaching the transformation operation to an element of the meta-model and to specify it in OCL: the model before the transformation is the source model and once the transformation has been executed, the model modified by the operation is the target model. Within the post-condition, we can refer to elements of the target model and of the source model via the OCL `@pre` operator. Thus, the pre-condition allows the contract's part relating to the source model to be written and the post-condition contains the two other parts of the contract (on the target model and on the element evolution). For our interface addition example, we could have a contract of the following form:

---

context ModelBase::addInterfaces()
pre: -- *constraints on source model*
post:
-- *constraints on target model and*
-- *constraints on element evolution between source and target models*
allClasses -> size() = allClasses@pre -> size() **and** ...

Here, the transformation operation is called `addInterfaces` and is attached to the `ModelBase` meta-element. The post-condition of the operation checks particularly that the number of classes after the transformation is the same as before. We can see that one can manipulate together elements before the transformation (from the source model) and elements after the transformation (from the target model).

This method is conceptually relevant because it associates a contract with a transformation operation as one can associate a contract in OCL with any operation of a class (via pre and post-conditions). However it is not adapted to our problem, for two reasons. Firstly, this contract is checkable only when the transformation is executed. This implies that we need a tool which at the same time makes it possible to execute a transformation and to check the associated constraints. This strongly reduces the set of usable tools. The second reason is the consequence of the first: it is necessary to execute a transformation to validate its contract, which requires an entirely automatic execution. If the designer modifies by hand the generated model, it is thus impossible to validate the contract.

Concatenation of source and target models When defining the contract, to avoid the problems we described above, we use the second technique we defined. Its general principle is to concatenate the source model and the target model in a more global model. Next, we define in OCL a set of invariants associated with the global model, and thus at the same time with elements of the source model and the target model. This model concatenation is a "trick" required to overpass the OCL limitation of invariants being expressed for a single context and as a consequence, being checked on a single model.

In a previous step of this current work, we presented in [4] a concatenation method where the designer had to explicitely add on metamodels elements for realizing model concatenation. He had also to make by hand this model concatenation or to develop a dedicated program to do it for each kind of metamodel. This was not acceptable because, as far as possible, the model designer must not be compeled by this need of model concatenation for checking the transformation contracts.

In the context of the Eclipse/EMF platform, we have then developped a tool that automatically realizes the concatenation for any meta-model. The tool takes as input the meta-model, the source and the target model. In a first step, it adds to the meta-model a `ModelReference` class which contains a string attribute called `modelName`. All classes of the meta-model are modified to specialize this new class. In a second step, the tool adds all elements of the source model and all elements of the target model into a third global model conforming to the modified meta-model. During this step, each element is tagged through the `modelName` attribute with the "source" or "target" string value, depending on the model it belongs. As output, our tool returns the modified meta-model and the global model containing all elements of both source and target models with indication of their origin[2].

4.2 Definition of the contract

As shown, the contract is composed of three parts: the constraints the source model must respect, the constraints the target model must respect and the constraints on the evolution of the elements between the source model and the target model. The first two parts are easy to express, we have to define invariants associated with the meta-model, like this is done in a common way. For our example of interface addition, the source model is any kind of class diagram without particular restriction; no constraint on the source model has to be defined. For the target model, each class of the diagram has to implement at least one interface. This is expressed in the following way:

context Class **inv** hasAnInterface:
self.interfaces -> notEmpty()

A standard OCL evaluator can then be used to check that this constraint is validated by the target model.

The third part concerning the evolution constraints between the two models is more complex to establish, because it is associated with the global model which concatenates the source model and the target model within one single model. It

[2] We work here on transformations implying a single source model and a single target model. The concatenation principles we present are directly generalizable for managing more than one source or target model. One has just to give an unique name for each model to tag its elements. In a more general way, our complete method is directly generalizable to manage more than one target or source model.

is then necessary to determine explicitly references on the source model and on the target model and their elements. This is easy to do as all elements of the global model, including our `ModelBase` instances, are tagged in order to indicate if they belong to the source or the target model. Moreover, we need to express that an element on a model is mapping another element on the other model and to get this mapped element to be able to express evolution constraints between these elements. This can be achieved by a set of OCL utility functions.

Mapping between elements of source and target models In our refinement example, the main goal of the contract, concerning the evolution of the elements between the source and the target models, consists in checking that each class of the target model implements, directly or via its interfaces, the same methods as its equivalent class on the source model. To validate that, it is then necessary to be able to determine which is its equivalent class. In a more general way, it is necessary to be able to determine that a given element on a model has or has not an element of the same type which maps it on the other model, and to be able to get this mapping element if it exists. If we take the example of Figure 2, the class `Account` on the target model (the final model) maps the class `Account` on the source model (the initial model). This mapping is checked by comparing the names of the classes and by checking that their attributes (here `id`, `overdraft` and `balance`) are the same. It is thus necessary to also check the mapping of the attributes.

The mapping of an element of a model to an element of the same type on another model can be of three kinds:

Full mapping an element maps to an identical element of the other model, in the meaning that all its attributes and all its references have also a mapping[3] in the other model.
For example, in our meta-model, the meta-element defining an attribute has a name, a reference on a type and a reference on its owner. Two attribute instances will be in full mapping if they have the same name, if their types are mapped and if their owners are also mapped.

Partial mapping an element has a mapping element in the other model in the meaning that an under-set of its attributes and references are also mapped[3] in the other model.
In our example, a class is in partial mapping with a class of the other model. Indeed, if the name must be the same one and if the list of attributes must be mapped for the two classes, we must not on the other hand check that their methods or their implemented interfaces are mapped. These elements belong to the model parts that are modified during the refinement application and their validity is checked separately.

[3] Mappings are thus checked in a transitive way (the mapping of the classes implies the mapping of the attributes of these classes which implies in its turn the mapping of the types of each attribute, etc.) but it is not necessary to remain in the same mode of mapping (full everywhere or partial everywhere). For example, the mapping of the classes could be full but the mapping of its attributes will be partial.

No mapping it is not necessary to check that an element has a mapping in the other model.
In our example, it should not be checked that an interface is in mapping with an interface of the other model. Indeed, during refinement, the interface list may change according to the designer choices (interface renaming, creation or removing, modification of the method list).

These mappings between elements are implemented in OCL by a series of utility functions (via the `def` operator). Depending on the complexity of the meta-model and the number of required mappings between the elements, the number of these functions can be important and tiresome to write by hand. This problem is easily avoidable through a dedicated tool able to automatically generate these functions for a given meta-model. Indeed, these functions are always based on the same structure, by checking the mapping of attributes and reference of an element in a transitive way. For a reference or attribute list, the mapping of the list elements is checked one by one. We have implemented a tool applying these principles. It analyzes any meta-model and proposes to the designer to choose, for each meta-model element type, the kind of desired mapping (for a partial mapping, the required attributes and references will be selected). The tool then generates all OCL mapping functions that the contract designer will complete or slighty modify for defining the specific part of the contract. Figure 3 is a screenshot of our tool in the context of mapping choices for the class element.

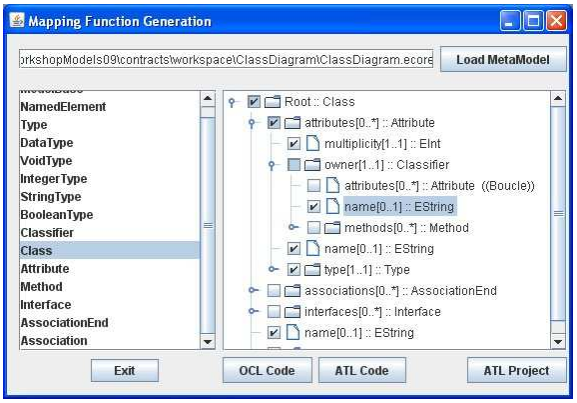


Fig. 3. Mapping function generation tool: selection for class mapping

When defining these functions, it is necessary to pay attention to an important detail: to avoid mapping cycles. For example, an attribute has a field `owner` of `Classifier` type referencing the owner of the attribute. When the mapping

of two classes is checked, the mapping of their attributes is transitively checked; but if for an attribute we check the mapping of the `owner` field, we loop back in the checking of the mapping of the initial classes. The mapping of this field must then not be checked. If we need to have an in-depth mapping, i.e. applying transitively to a great number of elements, this problem becomes all the more important and complex to manage. Our tool informs the designer if such a cycle is present for the current mapping choices.

Constraints on element evolution for the interface addition example The third part of the contract, concerning the evolution of the elements between the source model and the target model is expressed on Figure 4 by the invariant of lines 10 and 11. It checks the contents of the source model and of the target models through the global model. `targetModel` and `sourceModel` corresponds, respectively to the model base of the source and the target model. They are easily defined through OCL in the following way, thanks to the `modelName` attribute added to each model element during the concatenation:

```
context ModelBase def: sourceModel : ModelBase =
                ModelBase.allInstances() -> any (modelName = 'source')
context ModelBase def: targetModel : ModelBase =
                ModelBase.allInstances() -> any (modelName = 'target')
```

The invariant consists in calling the function `sameClasses` (1) which, for two model bases, begins to check that the number of classes is the same (2). Then, for each class of the first model (3), it gets its set of methods by concatenating the methods of the class and of all its interfaces (4). It checks via the function `hasMappingClass` that the current class has a mapping in the other model (5) and if this is not the case the contract is not validated (9). It gets via the function `getMappedClass` the mapped class on the other model (6) and its complete set of methods (7). Then it is checked that the two classes have the same sets of methods via the call of `sameMethodSet` (8).

Due to lack of place, we will not detail all the mapping functions used by this contract. Those which relate to the classes are nevertheless defined on Figure 5. The main function is `classMapping` which applies a partial mapping between two classes: comparison of the name and of the attribute set (via the call of `sameAttributes`). The mapping functions for attributes will have a similar form.

The function `sameMethodSet` (Figure 6) compares two sets of methods. This is done by checking initially that the sets have the same number of elements. Then, it is checked, via the mapping function for method (`methodMapping`), that each element of the first set has a mapping element in the second one. Contrary to constraints of Figure 4 that have to be written explicitely by the contract designer, mapping functions of Figure 5 and Figure 6 are fully generated by our tool we described above.

We can notice that this contract not only checks that method sets are conserved for each class, but it also ensures that some elements are not modified

```
1  context ModelBase def: sameClasses(mb : ModelBase) : Boolean =
2      self.allClasses -> size() = mb.allClasses -> size() and
3      self.allClasses -> forAll( c |
4          let myMethods : Set(Method) = c.interfaces -> collect(i | i.methods)
                                          -> union(c.methods) -> flatten() in
5          if c.hasMappingClass(mb)
           then
6              let eqClass : Class = c.getMappedClass(mb) in
7              let eqClassMethods : Set(Method) = eqClass.interfaces -> collect(i|
                       i.methods) -> union(eqClass.methods) -> flatten() in
8              c.sameMethodSet(myMethods, eqClassMethods)
           else
9              false
           endif)

10 context ModelBase inv checkInterfaceContract:
11 targetModel.sameClasses(sourceModel)
```

Fig. 4. Evolution constraints between the source and target models

```
context Class def: classMapping(cl : Class) : Boolean =
    self.name = cl.name and
    self.sameAttributes(cl)

context Class def: hasMappingClass(mb : ModelBase) : Boolean =
    mb.allClasses -> exists( cl | self.classMapping(cl))

context Class def: getMappedClass(mb : ModelBase) : Class =
    mb.allClasses -> any ( cl | self.classMapping(cl))
```

Fig. 5. Mapping functions for a class

```
context Classifier def: sameMethodSet(
                        mets1 : Set(Method), mets2 : Set(Method)) : Boolean =
    mets1 -> size() = mets2 -> size() and
    mets1 -> forAll ( m1 |
        mets2 -> exists ( m2 | m1.methodMapping(m2)) )
```

Fig. 6. Mapping function for a method set

during the transformation. Indeed, the contract is validated if all classes are conserved with the same name and with the same attribute list. Il also verifies that all methods are conserved without modification of their signatures (through the calls of `methodMapping` in `sameMethodSet`). This ensures a well defined scope for the manual modification of the model.

All these functions and invariants forming the contract are written in standard OCL. It is thus checkable by any OCL evaluator implementing the standard and able to read Ecore models. From a practical point of view, we have chosen the ATL[4] tool to validate the contract on a global model. Although being basically a tool of model transformation, ATL fully implements the OCL standard and can easily be used to verify OCL constraints through a model transformation, as explained in [3].

Interest of explicit mapping functions Intuitively, one can consider that the problem of element mappings between the source and the target models is a key issue when it deals with expressing that such element (or such set of elements) corresponds in the other model to such other element (or such other set of elements) that can be very different in contents and structure. Indeed, this leads to complex mapping functions to define. These kinds of mappings are required in exogenous transformations but can also be defined for endogenous transformations. For instance, the OCL helper "sameClasses" of Figure 4 can be considered as a complex mapping function.

Our generated mapping functions only check that an element of a given type has an equivalent element of the same type on the other model. These mappings are then more simple to define. Nevertheless, we argue that they are key points in contract definition, and that they offer two major interests.

Firstly, it is important for an element to be able to get precisely its equivalent element on the other model, in order to apply evolution constraints to them. If we want to avoid mapping errors, we often need to express in-depth mapping, by transitively checking mappings of element attributes and references and so on. As an example, let consider our class diagram meta-model and the way it defines associations between classes. As we can have on a class diagram two associations with the same name, it is not enough to simply compare association names. We need to transitively compare their two association ends (name and bounds) and also compare the associated classes (at least their names) for these association ends.

Secondly, for almost all endogenous transformations and particulary refinements, we need to check that some model elements are not changed during the transformation. This is for the most important if we let the designer manually modify the model. For our example, we must check that name and attributes of classes and all associations are not modified during the interface addition refinement application. A contract dedicated to verify the unmodified parts of a model is only composed of mapping functions that can be entirely automatically generated. As a consequence, a contract validating unmodification on any

[4] Atlas Transformation Language: `http://www.eclipse.org/m2m/atl`

kind of model during an endogenous transformation can be fully automatically generated thanks to our mapping function generation tool.

5 OCL choice and related works

We have chosen to use standard OCL as contract expression language. We argue that this choice is relevant for several reasons. First, OCL is by nature a constraint expression language and a contract is a set of constraints. Next, our concern is to be able to define a contract, as far as possible, independently of modeling platforms and tools. In this scope, OCL is a good candidate since it is a standard usable on varied kinds of models, like UML, MOF or for the Eclipse/EMF platform. We can thus check OCL constraints on models generated by a wide range of transformation engines. Finally, OCL is a relatively well-known language. It is indeed difficult to define a precise meta-model or any model without the obligation to add assertions. For simplicity and commodity, these are often written in OCL. OCL is also used as a query language and is sometimes enhanced in transformation languages like ATL or QVT-compliant [12] languages like SmartQVT[5]. The last reason of choosing OCL is that it is a formal constraint language relatively well accepted by "lambda" designers which usually do not want to use formal techniques or languages. [13] argues also in this direction by writing that "(formal languages) are usually hardly accepted by software engineers".

Some other works have also chosen to use standard OCL to specify either model transformations or refinements, or even transformation contracts similar to ours. For example, [13] formally defines model refinements in OCL. [14] defines in OCL transformation contracts for ensuring model consistency and also use OCL for expressing code refactoring. [1,9] also propose to use model transformation contracts written in OCL to specify a transformation test oracle.

In almost all these approaches, as in ours, it is required to define mappings between elements of the source and the target models. But these mappings are always defined in an *ad hoc* way for the considered context, and sometimes, only in an implicit way. In contrast, we proposed a general method and a tool to explicitly define and generate mappings between elements in the context of endogenous transformations. The other difference is that most of these approaches are less general than ours on the way of obtaining and manipulating the models. Moreover, they can be dedicated to particular software environments and specific purpose. For example, the refinement specification in [13] is done by using the UML dependency relation stereotyped by `<<refine>>` which implies the definition of a UML diagram in which elements are explicitly bound by means of this dependency. However, no method or tool is proposed for, starting from two models (obtained in an unspecified way), automatically defining a model in conformity with this representation.

On another hand, model transformation verification is a large field where, of course, a lot of different techniques and tools can be used instead of OCL.

[5] `http://smartqvt.elibel.tm.fr`

These techniques can be based on formal specification, such as graph transformation and specification like in [10,15]. This presents the main drawback to be hardly accessible to the lambda designer. Avoiding this problem, recently dedicated MDE platforms or tools have introduced specification languages that can be used to verify transformations. We can cite for instance Kermeta[6], Epsilon[7] or openArchitectureWare[8]. Their languages are similar to OCL from the point of view of their goal, syntax (even if generally more simple), semantics and expressivity but with the possibility of manipulating several models simultanously. They are then good candidates in replacing OCL for expressing contracts but they oblige in learning and using a new language. So, from our point of view, the best solution will be simply to extend OCL to allow multiple contexts for OCL expressions, as for instance proposed by [8]. We say this is a need in order to improve the usability of OCL and should be included in the next OCL standard specification. Other secondary improvements, based on features available in the specification languages we just cited, would also be interesting, such as syntax simplification or outputs on constraint evaluation results. Finally, if remaining in the context of using standards, QVT Relation [12] would be a good candidate to express transformation contracts instead of OCL.

6 Conclusion and perspectives

We presented a method to specify and verify a model transformation contract concerning any kind of endogenous transformation. The general idea is to consider a couple of models, one being the source and the other the target of a transformation and to check that this couple strictly respects the transformation contract expressed as sets of invariants. The model transformation can be realized automatically and/or via a manual intervention of a designer. The contract is written in OCL to be as far as possible independent of platforms and tools.

We showed that the intuitive way of specifying a transformation contract through a couple of pre and post-conditions is not usable from a practical point of view, because it does not allow manual intervention on models during the transformation. We also showed the need and interest of detailing one-to-one mapping functions between elements. Indeed, they help in defining and structuring a contract. Moreover, they allow the direct and complete definition of a contract verifying unmodified parts of a model during its transformation. In the context of the Eclipse/EMF platform, we developped a tool that generates these mapping functions for any meta-model. For other technical spaces (UML, MOF, ...) similar tools could be defined to help in contract definition.

We pointed out the main problem of OCL, the single expression context, which leads to the need for concatenating two models within a more global model (this concatenation is realized automatically by our tools). Within model

[6] `htpp://www.kermeta.org`
[7] `htpp://www.eclipse.org/gmt/epsilon/`
[8] `htpp://www.openarchitectureware.org`

transformations, *i.e.* when manipulating several models at the same time, this becomes a very strong limitation and the next OCL specification should include a multi-context feature.

A forthcoming issue of our approach is to generalize it with exogenous transformations, *i.e.* with different source and target meta-models. The main difficulty is here too to go on the main limitation of OCL: constraints must only be written for a single meta-model context. A possible solution to this problem is for example to make the concatenation of the two meta-models in a more global third one in order to get a single meta-model as context for expressing OCL constraints, as suggested in [1].

References

1. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. Le Traon. Model Transformation Testing Challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing.*, July 2006.
2. A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making Components Contract Aware. *IEEE Computer*, 32(7):38–45, 1999.
3. J. Bézivin and F. Jouault. Using ATL for Checking Models. In *GraMoT 2005 workshop*, volume 152 of *ENTCS*, 2005.
4. E. Cariou, N. Belloir, and F. Barbier. Contrats de transformations pour la validation de raffinement de modèles *(in french)*. In 5$^{\text{èmes}}$ *journées sur l'Ingénierie Dirigée par les Modèles (IDM 09)*, 2009.
5. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. Model Tranformation Contracts and their Definition in UML and OCL. Technical Report 2004-08, LIFL, April 2004.
6. E. Cariou, R. Marvie, L. Seinturier, and L. Duchien. OCL for the Specification of Model Transformation Contracts. Workshop OCL and Model Driven Engineering, UML 2004, 2004.
7. B. Meyer. Applying "Design by Contract". *IEEE Computer (Special Issue on Inheritance & Classification)*, 25(10):40–52, 1992.
8. T. Milan, L. Sabatier, P. Bazex, and C. Percebois. NEPTUNE II, une plate-forme pour la vérification et la transformation de modèles. *Génie Logiciel*, (85), 2008.
9. J.-M. Mottu, B. Baudry, and Y. Le Traon. Reusable MDA Components: A Testing-for-Trust Approach. In *MoDELS 2006*, volume 4199 of *LNCS*. Springer Verlag.
10. A. Narayanan and G. Karsai. Towards Verifying Model Transformations. *ENTCS, Elsevier Science Publishers B. V.*, 211, 2008.
11. OMG. Object Constraint Language (OCL) Specification, version 2.0, 2006. `http://www.omg.org/spec/OCL/2.0/`.
12. OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, version 1.0, 2008. `http://www.omg.org/spec/QVT/1.0/`.
13. C. Pons and D. Garcia. An OCL-Based Technique for Specifying and Verifying Refinement-Oriented Transformations in MDE. In *MoDELS 2006*, volume 4199 of *LNCS*. Springer Verlag, 2006.
14. P. Van Gorp. *Model-Driven Development of Model Transformations*. PhD thesis, University of Antwerp, Dept. of Mathematics and Computer Science, 2008.
15. D. Varro. Towards Formal Verification Of Model Transformations. In *PhD Student Workshop of FMOODS 2002*, 2002.

Session du groupe de travail Transformations

A pragmatic structural approach for the specification and verification of model transformations (position paper)*

Marc Pantel
IRIT/Université de Toulouse
Marc.Pantel@enseeiht.fr

Nassima Izerrouken
Continental
IRIT/Université de Toulouse
Nassima.Izerrouken@enseeiht.fr

Adrien Champion
IRIT/Université de Toulouse
Adrien.Champion@enseeiht.fr

Jean-Charles Dalbin
Airbus SAS
Jean-Charles.Dalbin@airbus.com

Frédéric Pothon
ACG-Solutions
Frederic.Pothon@acg-solutions.fr

Abstract

Qualification (i.e. verification) of automated model transformation is mandatory for their adoption in the development of certified safety critical systems. Current industrially available qualified toolsets relies on classical validation and verification activities such as document and software proofreading, and the various kind of testing activities relying mainly on human oracles. These activities are error prone and far from being exhaustive thus leaving potential holes in the product V & V schemes. State of the art in academic software engineering advocates the use of formal methods and semantics based proof of soundness. However, this brings a significant gap between the current know how of common software engineers from the industry, and the mandatory skills for semantics based verification. This contribution first presents the activities conducted in the ITEA GeneAuto projects related to the use of proof assistants for the development of an automated code generator, and then propose a pragmatic structural approach for the specification and verification of transformations in order to introduce a separation of concerns between the various partners. It relies on common technologies such as MOF and OCL on the industry side, and on formal methods and semantics based verification on the academic side.

*This work was partly funded by the French DGCIS through the ITEA and ITEA2 Eureka framework, This draft position paper was written specifically for the «*journées IDM/GDR GPL Transformation*» workshop

1 Introduction

Model driven engineering now plays a key role in the development of safety critical systems. Domain Specific Modeling Languages provide the most appropriate expressiveness to the system designers as well as early validation through model animation, and early verification through static analysis and model checking. Automated model transformations then allow to combine the various validated and verified models and produce parts of other intermediate models and even of the final product thus reducing low level and error prone systematic development activities.

Qualification of these automated model transformations is mandatory to be able to rely on the V & V activities conducted at model level and reduce the costly, often incomplete and sometime inefficient verification of the final product. Qualification requires that the development of these transformations fulfills the same constraints as the product whose development they are used for. Thus, the validation and verification of the transformation itself is a key point.

Industrial partners such as Airbus SAS or Esterel Technologies have designed development and verification processes for the automated model transformations that are currently in use for software inside safety critical systems such as the A380 plane. Currently, in most cases, these processes rely on precise natural language specifications, on proofreading of the various development artifacts and on all kind of tests. The semantics of the languages are only used implicitly in the writing of the specifications and tests. Formal methods are mostly not used, either at specification or verification level. The cost of these processes application is rising regularly as the transformations are more complex and widely used. It seems that we are reaching the limit of what can be done relying on pure structural approaches without explicitly handling of the semantics through formal methods.

This contribution first relates the experiments conducted in the ITEA GeneAuto project regarding the introduction of formal methods for the specification and verification of automated model transformation, inside the common industrial process that led to fruitful exchanges with the French aeronautics certification authorities. Then it summarizes the main difficulties encountered in the collaboration between industrial and academics partners and proposes a structural approach in order to bridge the gap between the various partners and ease the separation of concerns between structural and semantics technologies.

2 The GeneAuto project

GeneAuto is a three year (2006-2008) project funded by ITEA [42, 41]. Its purpose was the development of a qualifiable automated code generator from a subset of Simulink and Stateflow to MISRA ANSI C. GeneAuto involved academic partners INRIA, IRIT and TUT, service societies KRATES and ALYOTECH, safety critical system designers Airbus SAS, EADS Astrium, Barco, Continental Automotive, Israel Aircraft Institute and Thales Alenia Space. GeneAuto fulfilled several purposes:

- Define a common subset of Simulink/Stateflow for the development of safety critical real-time embedded control and command systems for aeronautic, automotive and space applications;
- Define a common set of requirements for an automated code generator (ACG) from this modeling language to imperative sequential programming languages (such as the MISRA subset of ANSI C, or ADA);
- Define a development and verification process for this ACG taking into account the qualification constraints of the various application domains (DO178B, ECSS, ISO 26262, ISO 61508) and exchange with certification authorities to check the conformance of these proposals;
- Experiment formal specification and verification technologies for ACG [19, 18, 17]. Several approaches from the state of the art [12] were evaluated: Proof assistant [27, 23, 3], Proven development [39, 25][1], Translation validation based on model checking [31, 45], abstract interpretation [33] or ad-hoc technologies [4, 43] and Automated test generation with automated oracles [26, 5, 21, 38]. The key points that led to the choice of proof assistants was that, on the one hand, it minimized the risk of residual errors that could not be detected by the V & V scheme; and on the other hand Leroy et al [23, 3] has shown through the CompCert[2] project that it could be applied to real size industrial toolsets;
- Develop an open source toolset[3] satisfying these requirements.

The final qualification of the toolset was out of the scope of GeneAuto as it usually depends on the specific application that the ACG is used for.

We thus designed a development and verification process for both classical (with implicit handling of the semantics) and formal method (with explicit handling of the semantics) based approaches that will be described briefly in the following sections.

3 Classical development and verification process

This process for the development and verification of qualified model transformations using classical technologies relies on the following steps:

- User requirements writing: The tool users express their needs related to the use of the tool. These user requirements are usually specified using natural languages, and are validated and verified by proofreading. They define the input and output languages, the transformation preconditions and postconditions. For example, in GeneAuto, the input language was a subset of Simulink/Stateflow with

[1] http://www-users.cs.york.ac.uk/susan/bib/ss/decco/index.htm
[2] http://compcert.inria.fr
[3] http://www.geneauto.org

project specific design guidelines; the output language was ANSI C language respecting the MISRA constraints. The semantics of the language is an implicit requirement that is usually not written explicitly. They take into account the usage context for the tool. The tool users must provide deployment and functional test scenarios and expected results that must ensure the conformance of the tool with respect to the user requirements.

- Tool requirements writing: The tool developers refines the user requirements and proposes a technical solution to the tool user requirements. One key point is the toolset architecture that split the tool as a sequence of simpler almost autonomous sub-tools, called elementary tools, that are easier to specify, implement and verify. These tool requirements are usually also expressed using natural languages or with semi-formal graphical ones, and are validated and verified by proofreading. They provide all the details needed for implementing the toolset and must be accepted by the tool users. For example, in GeneAuto, regarding the Simulink data flow models, it expressed how the blocks were sequenced, and what code patterns were generated for each Simulink blocks from the library. The tool developers must provide functional, integration and unit test scenarios and expected results that must ensure the conformance of the tool with respect to the tool requirements. All the tool requirements are explicitly linked to the user requirements that they are refined from. If some additional requirements are expressed (called derived requirements) independently of the user requirements, they must be clearly motivated as they cannot be verified using user provided tests.

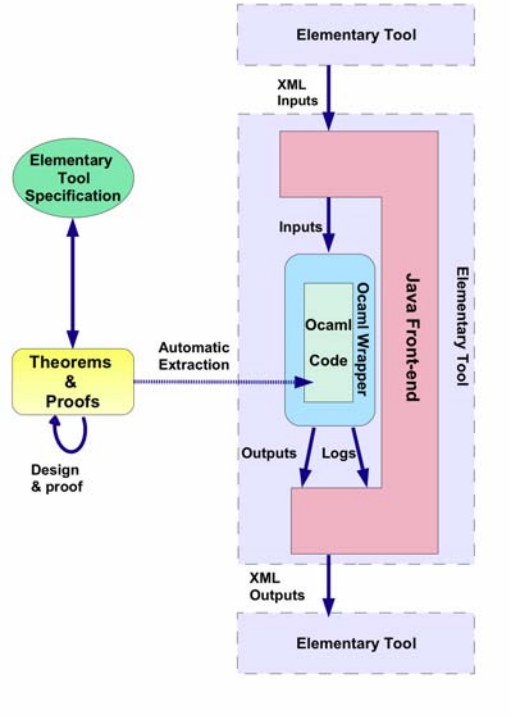
- Tool implementation: The tool developers can relies on any technologies that fits their need but they must show that they handle any risks related to that technology. For example, Java was used as programming language which relies on a virtual machine for the execution of the toolset, thus each virtual machine that will be used by the tool users must be qualified to ensure that it will behave as the one used to verify the toolset; the Xerces XML library was used to load and store the models, thus it was required to check that all errors that had been identified for Xerces and were currently not corrected could not have an impact on the toolset. All the implementation artifacts are explicitly linked to the tool requirements that they are refined from. If some additional artefact's are expressed (called derived requirements) independently of the user requirements, they must be clearly motivates as they cannot be verified using user provided tests. If some external libraries are used, their user and tool requirements, and the corresponding tests, must be provided.

- Tool verification: All the tests that have been provided at the tool requirements step are applied. The test coverages are computed and additional test scenarios are added to reach the expected level. All tests must be related to the requirements, if development artifact cannot be covered by these tests, it means that they are not related to the requirements and thus must be removed from the software. The proofreading activities are conducted.

- User validation and verification: All the tests that have been provided at the tool requirements step are applied. The test coverages are computed and additional test scenarios are added to reach the expected level. The proofreading activities are conducted.

4 Proof assistant based development and verification process

We proposed in the GeneAuto project to adapt the previous process in order to integrate formal specification and verification technologies. More precisely, we focused on the use of proof assistants for the specification of the requirements and implementation and for the verification of the compliance of the implementation to the requirements. We chose to rely on the Coq[4] toolset as it proved successful in many previous research project such as CompCert and was well known by the academic partners in GeneAuto. As it was only a preliminary experiment that must be accepted by both the industrial partners and the certification authorities, we chose to focus on the implementation of a single elementary tool, the block sequencer and to develop the other ones using the classical approach. The block sequence experiment was chosen as it is acknowledged by the current qualified tool developer at Airbus SAS to be the most difficult to verify with proofreading and tests. One driving point in the definition of the process was thus the ability to combine some elementary tools implemented in a classical way using various programming languages including Java and others implemented with proof assistant technologies. Thus the definition of the interfaces between elementary tools and the management of inputs and outputs had to be handled in a classical manner. GeneAuto relies on XML in order to store source, target and intermediate models and a model reader and writer has been developed in Java and verified with classical proofreading and testing activities. As the added value of formal verification technologies for XML model reading and writing was not significant with respect to its costs and this qualified implementation in Java was available inside the project, we decided to use the same library for integrating the elementary tools developed with Coq as shown in the following figure:



GeneAuto elementary tool structure

[4] http://coq.inria.fr

Here are the key steps of the process and the associated verification activities:

- Specification of usual elementary tool requirements using natural language. This is exactly the step conducted in the classical process;
- Design and implementation of the Java front-end and OCaml wrapper. As we decided to rely on the Java implementation for reading and writing the models we need to design for each elementary tool implemented with proof assistant technologies three very simple languages (input, output, errors) with right linear grammars to import and export exactly the data manipulated in the elementary tool requirements. The use of right linear grammars allows to have a very simple lexer and parser that can be verified by proofreading of the code;
- Formal specification of the elementary tool requirements. This specification describes the function provided by the elementary tool, the input and output data and their properties (pre and post conditions for the function) called tool properties, and specification related properties such as completeness, consistency, termination, ... called requirement properties and the proofs that can be conducted at that step. The properties that cannot be proved at that level are declared as axioms. These elements are mapped from the natural language requirements through explicit comments in the specification that are verified by proofreading of both documents. The proof are verified by the proof checker provided by the proof assistant toolset. This specification can be validated with respect to the tool user provided deployment and functional tests, by extracting the various properties regarding the input and output and checking that they hold for each test scenario. The proof checker is a verification tool that should be qualified as such. Currently, we proposed to rely on the implementation of Coq in Coq[5] as proposed by Barras et al. [2] but we did not use it at that step of the experiment to ease the development by relying on mainstream Coq;
- Formal specification of the elementary tool design. This specification refines the tool requirements in order to introduce all the required intermediate functions that leads to a full implementation. Each intermediate function is specified through the description of their input and output data and their properties called architecture properties. The specification also provides design related properties such as completeness, consistency, termination, ... called design properties and the proof that all properties from the requirements and design hold. The design should only introduce functions and properties that are required to satisfy the requirements that is for the implementation of the functions and the proof of the tool and requirement properties. This is verified by removing each element and applying the proof checker;
- OCaml code extraction and optimisation. All the specifications and proofs are done in a constructive logic that allows to extract the programs satisfying the prop-

[5] `http://www.lix.polytechnique.fr/~barras/coq-implicit`

erties from the specification. We have chosen to rely on the Coq proof assistant and to extract OCaml source code. The extraction process must be configured in order to produce efficient OCaml code (mainly by using primitive data types instead of inductive ones and array based cache to avoid computing several time the same values) and the OCaml wrapper must be connected to the extracted functions. The program extractor is a kind of automated code generator. It should thus also be qualified. In a first step, we propose to avoid it by relying on proof-reading of the generated code with respect to the specification. In this end, we had to avoid using the *proof as program* paradigm and the dependent types that can lead to extracted code structurally different from the specification. Thus, we write explicitly the functions in Coq and then we prove that they satisfy the requirements separately. In the long run, we should rely on an implementation of Coq in Coq by Letouzey and Glondu [24, 16] to bootstrap the qualification of tools;

- Elementary tool delivery. This is exactly the step conducted in the classical process.

This process was presented to the certification authorities that found it well designed and sensible and allowed us to go on in the development of the toolset. This was only an early evaluation that did not foresee anything regarding the final acceptation for the use of the toolset for a specific system.

The process was applied successfully to the block sequencer elementary tool that is currently the reference implementation provided with the publicly available GeneAuto toolset[6].

We had a lot of exchange during the GeneAuto project with the various industrial partners in order to check the acceptance of the use of formal methods, explicit semantics based activities and proof assistants. The two first points were accepted but the last one has several drawbacks related to its use by current industrial software engineering teams that develops automated model transformations using classical technologies:

- The technology is quite difficult to access for common software engineers and the tools are still academic prototypes event if some industrial success stories are available;
- It is quite difficult to give an accurate estimation of the development time based on the early user requirements in natural language;
- Qualified Coq proof checker and program extractor are currently only available as preliminary academic prototypes that may evolve regularly and does not fit industrial tool development;
- The development of scalable algorithms able to handle real industrial models are much costlier than the first basic inductive prototypes.

[6] `http://www.geneauto.org`

The GeneAuto experiments are still going on in the ITEA2 OPEES in order to check the commonalities between various elementary tools to have a better assessment of the development costs and to carry on the exchange with certification authorities.

But we concluded in GeneAuto that we currently have two distinct jobs: on the one hand, academic partners that are accustomed to this kind of approaches but whose purpose is not to develop industrial toolsets; and on the other hand, industrial partners that seems not to be ready currently for the widespread use of proof assistants, and in general formal verification methods that are not fully automated (even if early experiments show that things are changing on that subject).

We propose in the next section an approach for bridging the gap between both communities.

5 A structural approach for separating preoccupations

We propose to rely on standard formal modeling languages such as MOF [28] and OCL [29] for the specification of user and tool requirements instead of natural languages. This will allow to give a formal specification of the transformation requirements and to check that each application of the transformation is correct by evaluating the OCL constraints on the source and target of the transformation. One key point is to use explicit traceability links in order to ease the writing of the specification and the execution of the verification. This proposal is derived from existing work that uses OCL for expressing preconditions and postconditions for transformations [9, 8] without relying on traceability links, and work conducted by the Triple Graph Grammar community in the same purpose [36, 37] and the relational part of the QVT standard [22, 30].

The use of traceability links, or model weaving [13, 14], allows to relate explicitly the source and target elements and to express the correctness of the transformation as a property of the links.

The use of explicit traceability links instead of implicit ones that can be generated automatically by transformation language as ATL[7] or Kermeta[8] [20, 44, 15] or by extended model manipulation libraries [1], allows to focus on specification and design level information that are related to the semantics of the transformation instead of implementation details provided by automatic generation of traceability links.

As a preliminary, a specification of the input and output languages using MOF and OCL must be available. This specification usually contains the abstract syntax of the modeling language expressed in MOF and its static semantics defined in OCL. But, in order to be able to verify the semantics soundness of the transformation, the specification of the semantics of the languages should also be provided. To stay in the same technical space, this can be done using MOF to extends the usual metamodel with the execution related information as proposed by Combemale et al. [10, 11]. Then traceability links and OCL can be used in the same manner to define the execution relation for a given language.

[7] http://www.obeo.fr/pages/obeo-traceability/en
[8] http://www.kermeta.org/mdk/traceability

These specifications should be shared between the various transformations as standard references, which is already the case for the abstract syntax and the static semantics as can be seen in the latest OMG specifications.

Then, the development of the transformations is split in several phases conducted by different teams:

- Specification of the transformation using MOF and OCL that defines:
  - Traceability link metaclasses relating the source and target metamodels;
  - OCL properties that must hold for each traceability link metaclass expressing the semantics relations between the source and target.
- Implementation of the transformation that must produce both the target and the traceability links with the source;
- Proof of semantics soundness of the transformation specification with respect to the input and output languages.

This proposal establishes a separation of concerns between:

- the specification activities that must be conducted by a joint team with domain experts and language designers with the occasional help of semantics experts;
- the implementation activities that can be conducted by any development team without any knowledge of semantics engineering, using any kind of programming languages;
- the verification of semantics soundness of the specification that will be conducted by semantics experts and formal methods engineers.

The second and third concerns can be handled independently by different teams in a different time frame.

The third activities can also be conducted using other verification technologies than semantics proof, such as automated generation of tests that satisfies the MOF and OCL specifications [5, 38] (source, trace and target) and oracles for executing the source, the target and comparing the results thanks to the traceability links.

This proposal has been experimented in OPEES as a followup to GeneAuto for the generation of C code from sequenced and typed Simulink models; and for the flattening of hierarchical Simulink and Stateflow models.

The first experiment produced very good results that fit exactly the way qualified code generators are currently specified in natural language by industrial partners such as Airbus SAS. However, it is a structural transformation: the source and target are structurally similar, more precisely an element in the source is translated to several elements in the target, the specification is thus quite simple.

The purpose of the second one is to remove the structure from the source and produce a flattened target. We experimented two different approaches:

- first we wrote a full specification independent of the implementation strategy. It required a recursive traversal of the tree structure of hierarchical models that could be expressed with OCL extended with recursive helpers in ATL, or with transitive closure operators. But this specification was very complex, far more complex than its natural language counterpart, and did not fit at all our purpose of bridging the current industrial practise and formal technologies;

- then, we decided to specify the transformation as a transitive closure of the flattening of one layer elementary transformation. Both specifications of the one layer flattening and the transitive closure were simple enough to fit our needs. This compromise led the specification to be tainted with design level elements. However, the implementation is not required to be done like that, it must only produce the traceability links in this way.

These experiments were done using ATL to verify at runtime the results of the transformation.

We are now conducting larger scale experiments on the full GeneAuto language and exchanging with our industrial partners to improve the acceptability of the approach. This work is being done inside the ITEA2 OPEES project for GeneAuto and FRAE quarteFt project for AADL and will be partly supported by a PhD thesis in coooperation with Airbus SAS.

In order to fully bridge the gap, it is now mandatory to rely on:

- a systematic mapping between MOF and OCL specifications for the languages and the transformations and formal technologies such as proof assistants. A lot of activities have already been conducted and are still being done in that direction as various encodings are possible that do not hallow the same kind of proof strategies [35, 40, 7, 32, 6, 34];

- a qualified OCL verifier to check the results of each transformation use. This verifier could be developed either using the approach advocated in this contribution as the operational semantics of OCL is itself a transformation from OCL expressions to OCL values; or using the process design in GeneAuto relying on proof assistants.

6 Conclusion

We have presented the development process defined in the GeneAuto project for the specification, implementation and verification of automated model transformations that led to fruitful exchange with the certification authorities. We have shown how this process encountered difficulties for its acceptation by the industrial partners development teams and we have proposed a pragmatic structural approach for separating the various concerns and bridging the gap between semantics and structural concerns. This proposal relies on component of the shelf standard technologies such as MOF and OCL that are well known by industrial partners and allow to choose any formal technology in parallel for proving the semantics soundness of the specification.

References

[1] Bastien Amar, Hervé Leblanc, and Bernard Coulette. A traceability engine dedicated to model transformation for software engineering. In Jon Oldevik, Goran K. Olsen, Tor Neple, and Richard Paige, editors, *ECMDA Traceability Workshop 2008, Berlin, 12/06/08-12/06/08*, pages 7–16, http://www.springerlink.com, june 2008. Springer.

[2] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto M. Amadio, editor, *FoSSaCS*, volume 4962 of *Lecture Notes in Computer Science*, pages 365–379. Springer, 2008.

[3] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a c compiler front-end. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[4] Jan Olaf Blech, Sabine Glesner, Johannes Leitner, and Steffen Mülling. Optimizing code generation from ssa form: A comparison between two formal correctness proofs in isabelle/hol. In *Proceedings of the COCV-Workshop (Compiler Optimization meets Compiler Verification), 8th European Conferences on Theory and Practice of Software (ETAPS 2005)*, pages 33–51. Elsevier, April 2005.

[5] Erwan Brottier, Franck Fleurey, Jim Steel, Benoit Baudry, and Yves Le Traon. Metamodel-based test generation for model transformations: an algorithm and a tool. In *ISSRE*, pages 85–94. IEEE Computer Society, 2006.

[6] Achim D. Brucker and Burkhart Wolff. An extensible encoding of object-oriented data models in hol. *J. Autom. Reasoning*, 41(3-4):219–249, 2008.

[7] Achim D. Brucker and Burkhart Wolff. Hol-ocl: A formal proof environment for uml/ocl. In José Luiz Fiadeiro and Paola Inverardi, editors, *FASE*, volume 4961 of *Lecture Notes in Computer Science*, pages 97–100. Springer, 2008.

[8] Eric Cariou, Nicolas Belloir, Franck Barbier, and Nidal Djemam. Ocl contracts for the verification of model transformations. In *OCL workshop of MoDELS*, oct 2009.

[9] Eric Cariou, Raphael Marvie, Lionel Seinturier, and Laurence Duchien. Ocl for the specification of model transformation contracts. In *Workshop OCL and Model Driven Engineering of the Seventh International Conference on UML Modeling Languages and Applications (UML 2004)*, oct 2004.

[10] Benoît Combemale, Sylvain Rougemaille, Xavier Crégut, Frédéric Migeon, Marc Pantel, Christine Maurel, and Bernard Coulette. Towards rigorous metamodeling. In Luís Ferreira Pires and Slimane Hammoudi, editors, *MDEIS*, pages 5–14. INSTICC Press, 2006.

[11] Benoît Combemale, Xavier Crégut, Jean-Pierre Giacometti, Pierre Michel, and Marc Pantel. Introducing Simulation and Model Animation in the MDE TOPCASED Toolkit. In *Proceedings of the 4th European Congress* EMBEDDED REAL TIME SOFTWARE *(ERTS)*, Toulouse, France, January 2008.

[12] Maulik A. Dave. Compiler verification: a bibliography. *ACM SIGSOFT Software Engineering Notes*, 28(6):2, 2003.

[13] Marcos Didonet Del Fabro, Jean Bézivin, Frédéric Jouault, and Patrick Valduriez. Applying generic model management to data mapping. In Véronique Benzaken, editor, *BDA*, 2005.

[14] Marcos Didonet Del Fabro and Patrick Valduriez. Towards the efficient development of model transformations using model weaving and matching transformations. *Software and System Modeling*, 8(3):305–324, 2009.

[15] Jean-Remy Falleri, Marianne Huchard, and Clementine Nebut. Towards a traceability framework for model transformations in kermeta. In *ECMDA Traceability Workshop 2006, Bilbao, 10/07/06-13/07/06*, july 2006.

[16] Stéphane Glondu. Extraction certifiée dans coq-en-coq. In *Journées Francophones des Langages Applicatifs (JFLA)*, Saint-Quentin sur Isère, France, January 2009.

[17] Nassima Izerrouken, Marc Pantel, and Xavier Thirioux. Machine-checked sequencer for critical embedded code generator. In Karin Breitman and Ana Cavalcanti, editors, *ICFEM*, volume 5885 of *Lecture Notes in Computer Science*, pages 521–540. Springer, 2009.

[18] Nassima Izerrouken, Marc Pantel, Xavier Thirioux, and Olivier Ssi Yan Kai. Integrated formal approach for qualified critical embedded code generator. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *FMICS*, volume 5825 of *Lecture Notes in Computer Science*, pages 199–201. Springer, 2009.

[19] Nassima Izerrouken, Xavier Thirioux, Marc Pantel, and Martin Strecker. Certifying an automated code generator using formal tools : Preliminary experiments in the geneauto project. In *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, page (electronic medium), http://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[20] Frédéric Jouault. Loosely coupled traceability for atl. In *Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability*, Nuremberg, Germany, 2005.

[21] Maher Lamari. Towards an automated test generation for the verification of model transformations. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *SAC*, pages 998–1005. ACM, 2007.

[22] Kevin Lano and David Clark. Model transformation specification and verification. In Hong Zhu, editor, *QSIC*, pages 45–54. IEEE Computer Society, 2008.

[23] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In J. Gregory Morrisett and Simon L. Peyton Jones, editors, *POPL*, pages 42–54. ACM, 2006.

[24] Pierre Letouzey. Extraction in coq: An overview. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *CiE*, volume 5028 of *Lecture Notes in Computer Science*, pages 359–369. Springer, 2008.

[25] Michael Leuschel. Towards demonstrably correct compilation of java byte code. In Frank S. de Boer, Marcello M. Bonsangue, and Eric Madelain, editors, *FMCO*, volume 5751 of *Lecture Notes in Computer Science*, pages 119–138. Springer, 2008.

[26] Yuehua Lin, Jing Zhang, and Jeff Gray. *Model-Driven Software Development*, chapter A Testing Framework for Model Transformations. Springer Verlag, 2005.

[27] Robin Milner and R. Weyhrauch. Proving compiler correctness in a mechanised logic. *Machine Intelligence*, (7), 1972.

[28] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Core Specification*, January 2006. Final Adopted Specification.

[29] Object Management Group, Inc. *Object Constraint Language (OCL) 2.0 Specification*, May 2006. Final Adopted Specification.

[30] Object Management Group, Inc. *Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) 1.0 Specification*, April 2008. Final Adopted Specification.

[31] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In Bernhard Steffen, editor, *TACAS*, volume 1384 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 1998.

[32] Iman Poernomo. Proofs-as-model-transformations. In Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors, *ICMT*, volume 5063 of *Lecture Notes in Computer Science*, pages 214–228. Springer, 2008.

[33] Xavier Rival. Symbolic transfer function-based approaches to certified compilation. In Neil D. Jones and Xavier Leroy, editors, *POPL*, pages 1–13. ACM, 2004.

[34] Jose E. Rivera, Francisco Duran, and Antonio Vallecillo. Formal Specification and Analysis of Domain Specific Models Using Maude. *SIMULATION*, 85(11-12):778–792, 2009.

[35] José Raúl Romero, José Eduardo Rivera, Francisco Durán, and Antonio Vallecillo. Formal and tool support for model driven engineering with maude. *Journal of Object Technology*, 6(9):187–207, 2007.

[36] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *WG*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[37] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *ICGT*, volume 5214 of *Lecture Notes in Computer Science*, pages 411–425. Springer, 2008.

[38] Sagar Sen, Benoit Baudry, and Jean-Marie Mottu. Automatic model generation strategies for model transformation testing. In Richard F. Paige, editor, *ICMT*, volume 5563 of *Lecture Notes in Computer Science*, pages 148–164. Springer, 2009.

[39] Susan Stepney, Dave Whitely, David Cooper, and Colin Grant. A demonstrably correct compiler. *Formal Asp. Comput.*, 3(1):58–101, 1991.

[40] Xavier Thirioux, Benoit Combemale, Xavier Cregut, and Pierre-Loic Garoche. A framework to formalise the mde foundations. In *Proc. of the 1st Intl. Workshop on Towers of Models*, june 2007.

[41] Andres Tooms, Nassima Izerrouken, Tonu Naks, Marc Pantel, and Olivier Ssi-Yan-Kai. Towards reliable code generation with an open tool: Evolutions of the geneauto toolset. In *European symposium on Real Time Software and Systems (ERTS^2), Toulouse, 29/01/08-01/02/08*, page (electronic medium), http://www.sia.fr, 2010. Société des Ingénieurs de l'Automobile.

[42] Andres Tooms, Tonu Naks, Marc Pantel, Marcel Gandriau, and Indra Wati. Geneauto - an automatic code generator for a safe subset of simulink-stateflow and scicos. In *European symposium on Real Time Systems (ERTS), Toulouse, 29/01/08-01/02/08*, page (electronic medium), http://www.sia.fr, 2008. Société des Ingénieurs de l'Automobile.

[43] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In George C. Necula and Philip Wadler, editors, *POPL*, pages 17–27. ACM, 2008.

[44] Andrés Yie and Dennis Wagelaar. Advanced traceability for ATL. In *Proceedings of the 1st International Workshop on Model Transformation with ATL (MtATL 2009)*, page 78–87, Nantes, France, 2009.

[45] Anna Zaks and Amir Pnueli. Program analysis for compiler validation. In Shriram Krishnamurthi and Michal Young, editors, *PASTE*, pages 1–7. ACM, 2008.

Les plates-formes d'exécution dans l'IDM : Quelles modélisations pour quelles utilisations ?

Jérôme Delatour, Matthias Brun, Guillaume Savaton, Jonathan Ilias-Pillet, Cédrick Lelionnais

[1] Groupe ESEO, Équipe de recherche TRAME
4 rue Merlet de la Boulaye, BP 30926, 49009 Angers cedex 01
{prenom.nom@eseo.fr}

Abstract. L'un des objectifs de l'Ingénierie Dirigée par les Modèles (IDM) est de mettre en œuvre un même modèle de fonctionnalités sur différents supports technologiques, ces derniers sont souvent désignés comme des plates-formes d'exécution. Toutefois, peu de travaux ont été menés, à la fois pour préciser comment modéliser ces plates-formes et comment les mettre en œuvre dans des transformations. L'objectif de cette présentation est de proposer des éléments de réponse à ces questions.

Keywords: Plate-forme d'exécution, Métamodèle de plate-forme, modèle de plate-forme, Langage de Modélisation Dédié

1 Introduction

L'un des soucis constants de l'industrie informatique dans le développement de logiciels est de s'abstraire des considérations technologiques. Différentes couches d'abstractions (des systèmes d'exploitation aux intergiciels, en passant par les machines virtuelles) ont ainsi été développées pour que le logiciel applicatif, ou sa modélisation, soit de plus en plus indépendant des technologies de mise en œuvre. Une telle séparation permet d'espérer une meilleure réutilisation du logiciel applicatif, en particulier de la connaissance métier qu'il capture, sur les supports technologiques sous-jacents. Malgré de nombreux efforts liés à la normalisation, aux méthodologies et aux outillages, beaucoup de temps est encore dépensé à l'adaptation des applicatifs aux supports technologiques, supports en évolution constante.
L'un des points essentiels pour parvenir à une plus grande séparation des préoccupations est la modélisation explicite des aspects technologiques indépendamment des aspects applicatifs.
L'Ingénierie Dirigée par les Modèles (IDM) a très tôt identifié cette problématique de séparation des préoccupations technologiques. En particulier, l'approche MDA (Model Driven Architecture) a proposé un cadre méthodologique différenciant des modèles indépendants des plates-formes (PIM : Platform-Independant Model) de ceux en dépendant (PSM : Platform-Specific Model). Le terme de plate-forme est

utilisé pour caractériser les technologies sous-jacentes. Des transformations de modèles, plus ou moins automatiques, sont alors définies pour spécialiser un PIM en un PSM.
Actuellement, peu de travaux ont été menés, à la fois pour préciser ce qu'est un modèle de plate-forme (parfois identifié dans le MDA comme PDM – Platform Description Model) et définir comment mettre en œuvre de tels modèles dans des transformations. Un ensemble de questions corollaires se posent : Quelle représentation d'une plate-forme est la plus adéquate ? Une représentation est-elle préférable pour un contexte d'utilisation ? Peut-on espérer capitaliser des transformations et les réutiliser dans des contextes différents ?

2 Le temps réel comme domaine d'étude

Pour établir des éléments de réponse à ces questions, nous avons expérimenté et évalué différentes propositions, à la fois sur la représentation d'une plate-forme d'exécution et sur son utilisation dans des transformations.
Ces expérimentations ont été faites dans le domaine du temps réel embarqué. En effet, les supports d'exécution y jouent un rôle fondamental. Les considérations inhérentes à ce domaine (respect des contraintes temporelles, limitation des ressources disponibles, accès à des ressources partagées, sûreté de fonctionnement, etc.) dépendent étroitement de ce support. La plate-forme d'exécution joue un rôle tôt dans le cycle de développement, depuis l'analyse des propriétés fonctionnelles et temporelles d'une application jusqu'au déploiement de l'application sur la plate-forme. En outre, les besoins de réutilisation et de capacité de portage d'une application sur différentes plates-formes orientent le développement logiciel vers des conceptions indépendantes de toute plate-forme d'exécution. Cette séparation se retrouve souvent dans les pratiques métiers et les formalismes utilisés dans ce domaine. Pour toutes ces raisons, le domaine du temps réel embarqué nous semblait un bon candidat d'étude pour les plates-formes d'exécution.

3 Travaux effectués

Lors de nos travaux [1][2][3], il a été identifié trois grandes approches de modélisation et d'utilisation par transformation des plates-formes d'exécution :

- Une approche dite enfouie, où les informations sur la plate-forme sont directement présentes dans la transformation.
- Une approche dite implicite, où les éléments caractérisant une plate-forme d'exécution sont donnés dans un métamodèle.
- Une approche dite explicite, où la plate-forme est décrite dans un modèle conforme à un métamodèle de plate-forme d'exécution.

Pour chacune de ces approches, des expérimentations (écriture des modèles et des transformations associées permettant la génération de code) ont été effectuées. Ces expérimentations ont été évaluées afin d'identifier le degré de réutilisation des modèles, méta-modèles et transformations associés, de quantifier le temps de développement de ces différents éléments, déterminer leurs facilités de développement et d'utilisation. Ce travail a permis d'identifier des contextes d'utilisation et des recommandations sur les approches à employer suivant ce contexte.
Les expérimentations ont été menées, tour à tour, dans un contexte de développement utilisant des DSML (Domain Specific Modeling Language) mais aussi dans un contexte GPML (Generic Purpose Modeling Language) basé sur l'utilisation d'UML et ses formes profilées (utilisation du profil MARTE et participation à la définition de son paquetage SRM). En effet, nous voulions nous assurer que nos conclusions ne dépendaient pas du contexte de développement.

4 Plan de la présentation

Suite à une première partie introductive, un rappel sur la notion de plate-forme dans l'IDM sera effectué. Nous présenterons ensuite les grandes approches identifiées de modélisation des plates-formes. Pour chacune d'elles, nous expliquerons leurs représentations, ainsi que leur utilisation dans le déploiement d'applications (transformations d'un PIM vers un PSM). Suite à une rapide présentation des expérimentations menées et leurs évaluations, nous donnerons un premier ensemble de recommandations sur l'utilisation de telles ou telles approches. Cette présentation se conclura sur les perspectives de travail et les extensions en cours à d'autres domaines d'étude.

Références

Cette présentation est basée sur les publications suivantes :

1. Thomas F., Delatour J., Gérard S., Brun M., Terrier F., « Contribution à la modélisation explicite des plates-formes d'exécution pour l'IDM », L'Objet, Hermes-Lavoisier, vol. 13, n°4, ISSN:1262-1137, 2007, p. 9-31.
2. Thomas F., Delatour J., Terrier F., Gérard S., « Towards a framework for explicit platform based transformations », ISORC '08 : Proceedings of the 11th IEEE International Symposium on Object-oriented Real-time distributed Computing, Orlando, FL, Etats-Unis, 2008, p. 211-218.
3. M. Brun, J. Delatour, Y. Trinquet, F. Thomas, S. Gérard, « Étude comparative pour la modélisation de plates-formes d'exécution », TSI, numéro spécial "Ingénierie dirigée par les modèles", vol 29, 2010

Défis et table ronde pour le Génie de la Programmation et du Logiciel à échéance de 2020

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du génie de la programmation et du logiciel à échéance de 5 à 10 ans ?

De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable et les économies d'énergie.

Les cinq défis seront présentés lors de deux sessions. Ils portent sur la (re)modularisation du logiciel, la mise en place de fermes de composants et de services, l'évaporation des langages, la modélisation pour l'utilisateur final et la réification de l'énergie au niveau des systèmes et des langages. Une table ronde permettra ensuite de débattre de ces sujets. Lors de cette table ronde, Bertrand Braunschweig fera le point sur la perception de notre domaine au sein de l'ANR.

Laurence Duchien

Software (re)modularization: Fight against the structure erosion and migration preparation

N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet
RMoD INRIA
stephane.ducasse@inria.fr

R. Ducournau, R. Giroudeau, M. Huchard, J.C. König, D. Seriai
LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
huchard@lirmm.fr

I. CONTEXT

Software systems, and in particular, Object-Oriented systems are models of the real world that manipulate representations of its entities through models of its processes. The real world is not static: new laws are created, concurrents offer new functionalities, users have renewed expectation toward what a computer should offer them, memory constraints are added, etc. As a result, software systems must be continuously updated or face the risk of becoming gradually out-dated and irrelevant [34]. In the meantime, details and multiple abstraction levels result in a high level of complexity, and completely analyzing real software systems is impractical. For example, the Windows operating system consists of more than 60 millions lines of code (500,000 pages printed double-face, about 16 times the Encyclopedia Universalis). Maintaining such large applications is a trade-off between having to change a model that nobody can understand in details and limiting the impact of possible changes. Beyond maintenance, a good structure gives to the software systems good qualities for migration towards modern paradigms as web services or components, and the problem of architecture extraction is very close to the classical remodularization problem.

II. A SIGNIFICANT PROBLEM

Most of the effort while developing and maintaining a software system is spent in supporting its evolution [50]. It is well-known that up to 80% of the total cost of software development project is spent in maintenance and evolution of existing applications [15], [19]. Large IT applications including applications running central and critical business (*e.g.,* command tracking, banking, railway) have to run and evolve over decades.

Such situations are crystallized in the following two laws of Lehman and Belady. These laws, known as the laws of software evolution, stress the fact that software *must* continuously evolve to stay useful and that this evolution is accompanied by an increase of complexity.

> **Continuous Changes.** *"an E-type program—i.e., a software system that solves a problem or implements a computer application in the real world—that is used must be continually adapted else it becomes progressively less satisfactory"* [34]
>
> **Increasing Complexity.** *"As a program is evolved its complexity increases unless work is done to maintain or reduce it."*[34]

From a business perspective, maintenance is mandatory and vital. It is worth to note that the Year 2000 Bug was also revealing some business occasions. For example in France SOPRA which was mainly an SSII, has since the year 2000 developed its TMA (Tierce Maintenance Applicative) branch to get into the maintenance business. SOPRA TMA now represents 2500 developers working on maintenance on a total of 11000 developers.

If many organizations use third party software (e.g. COTS) which they don't have to maintain, this is less true for their core business applications where their market differential must be implemented in the applications that help running day to day activities. As highlighted by the law of Lehman and Belady, these in-house applications must evolve and in this process will extend far beyond their initial structure, in many independent (and sometimes even conflicting) directions. After some time, it becomes mandatory to restructure the application to federate these evolutions inside a more general and renewed architecture to foster possible future evolutions.

Supporting evolution and prepare migration of applications will be *always* mandatory. Different programming paradigms have been invented to cope with changes: late-binding, the cornerstone of object-oriented programming, is a typical illustration. But we think that no paradigm will eradicate the need for evolution and changes and that the only possible approach is to guide evolution or to repair the damages caused by an inevitable erosion.

III. THE FAILURE OF CURRENT SOLUTIONS

Whereas software (re-)modularization is a relatively old research field in the context of C or Cobol, it is still really important and requires *innovative approaches* to deal with the complexity of modern systems especially those

developed in OOP languages. Our analysis — also confirmed in the recent literature (e.g., [1], [7], [47]) — is that this failure is a direct consequence of: (1) the *complexity of the manipulated concepts and the variety of modular abstractions* (subsystems, packages, classes, class hierarchies, late-binding, aspects, various import relationships....) as well as (2) a monolithic approach: the use of *one* type of algorithms (clustering) and *one* kind of system representation (software components interactions found in the source code).

A. Modular Abstractions

Modular constructs have been the focus of a large body of research. Here we give a non-exhaustive list. A lot of work is currently underway in the context of aspect-oriented programming. Module and package systems have been the focus on a large amount of work. The recent work on Units [22], Jiazzi [42], Mixjuice [28], MJ [14], JAM [52], mixin layers [48] shows that this topic is a crucial research area. Bergel *et al.* conducted a survey on modular language constructs that reflects such a diversity [6].

Modular constructs have also been considered at a lower level than classes, *e.g.,* with mixins [9], traits [17]. These constructs denote, here, sets of properties that somewhat represent the *differentia* in the Aristotelian *genus-differentia* definition. Overall, remodularization must address both modular construct levels, by clustering related properties for defining classes and related classes for defining packages. Recursively nested class models have also been proposed [20], [44]—however they cannot be considered as long as the point with non-recursive models is not fixed.

Component-based software approach proposed to build software systems by assembling prefabricated reusable components [54]. Assembly consists in connecting matching interfaces of the components: in a connection, a required interface (describing the services a component needs) is connected to a provided interface (describing the services another component offers). Extracting a component architecture from an object-oriented software has many in common with remodularization because classes need to be grouped based on their dependencies to form the components [11].

Current Issues – *The class notion is the only universally accepted modular abstraction. Higher and lower level abstractions are often still in flux. Different languages use different concepts such as modules, packages, namespaces. Hence, current issues involve both identifying adequate abstractions and adapting remodularization algorithms to the various alternative abstractions. Moreover, assessing the modularity of software requires specific tools (e.g., metrics, visualization) that must be adapted to each modular abstraction.*

B. Remodularization Approaches

Class hierarchy analysis: Class hierarchy analysis has been largely investigated, for restructuring purposes or finding separate concerns (aspects, traits). As it has been shown in [26], most of the refactorization approaches use explicitly or implicitly substructures of those obtained by Formal Concept Analysis (FCA).

The application of clustering algorithms for software remodularization has been intensively studied [2]. Thousands of experiments were conducted to compare different clustering algorithms, different representation schemes and different coupling metrics between files. Although the experiments used procedural systems, many conclusions may be applied to OO systems as well. The extraction of class views based on Formal Concept Analysis has been proposed in [3]. They evaluated how FCA supports the identification of traits in existing hierarchies [8], [35]. Godin [24] developed FCA algorithms to infer a non-redundant form for implementation and interface hierarchies and carried out experiments on several Smalltalk applications. For dealing with UML models, Relational Concept Analysis, an extension of FCA, takes the relations into account [27]. Other approaches analyze class hierarchies using access or usage information. [49], [51] analyze the *usage* of the hierarchy by a set of client programs. Mining aspects has been considered in the context of FCA [10].

Current Issues – *Factorization is by nature a combinatorial process. Recent studies [21] show that Relational Concept Analysis, applied to rich UML descriptions including references between concepts, produces a huge number of artefacts which is quite impossible to analyze by hand. Execution time can become a problem for large size software, but the actual difficulty is the result size.*

Other modular construct discovery: Clustering approaches are, by far, the preferred algorithmic approach to the problem. They have been proposed to identify modules in applications that are not specifically object-oriented (e.g. [29], [30], [38], [43]).

Finding components in object-oriented software is proposed in [12]. Simulated annealing is used to gather classes into components by optimizing metrics measuring cohesion and coupling. The problem has similarity with package mining.

It is a well-known practice to layer applications with bottom layers being more stable that top layers [41]. Until now few works have been done in practice to identify layers: Mudpie [55] is a first cut at identifying cycles between packages as well as package groups potentially representing layers. DSM (dependency structure matrix) [53], [46] are adapted for such a task but there is a lack of detail information. From the side of remodularization algorithms, a lot of them were defined for procedural languages [31]. However object-oriented programming languages bring some specific problems linked with late-binding and the fact that a package does not have to be systematically cohesive since it can be an extension of another one [18], [57].

Current Issues – *These approaches are often not customized for object-oriented applications. Existing solutions propose modules at a very low level of abstraction that do not reduce enough the size of the system comprehension problem. Solutions that may offer larger (potentially more abstract) modules, result in modules that have no meaning for the software engineers.*

C. Techniques for module assessment

Software Metrics: Re-modularization of software systems is geared toward producing highly cohesive and loosely coupled modules. Many different cohesion/coupling metrics were proposed (including a study by [2]). In the more specific case of object-oriented programming, assessing cohesion and coupling has been the focus of several metrics. However their success is rather mitigated as the number of critics raised. For example, LCOM [13] has been highly criticized [4]. Other approaches have been proposed such as RFC and CBO [13] to assess coupling between classes. However, many other metrics have not been the subject of careful analysis such as Data Abstraction Coupling (DAC) and Message Passing Coupling (MPC) [5], or some metrics are not clearly specified (MCX, CCO, CCP, CRE) [36]. New cohesion measures were proposed [40], [45] taking class usage into account. The Cohesion/Coupling dogma, however, started to receive critics in recent times [1], [47]. People argue that software engineers do not base clustering on this criterion but rather use more semantical approaches.

Software Visualization: There is a significant effort to create efficient software visualizations to support the understanding and analyses of applications [32], [37], [39], [56]. Lanza and Ducasse worked on system level understanding combining metrics and visualization [33] and class understanding support [16].

Current Issues – *Existing cohesion and coupling metric resulting values are difficult to map back to the actual situation, they lead to packages that seem artificial and are not understood by experts of the systems. There is a lack for package cohesion and coupling software metrics in presence of late-binding promoted by object-oriented programming. There is a need for program visualization to support the understanding of packages and procedural code. In addition, there is a need for new metrics that would yield more "natural" packages.*

IV. SCIENTIFIC CHALLENGES AND TRACKS OF RESEARCH

The main scientific challenges are as follows.

Abstraction Diversity: One of the major problems to solve when tackling remodularization of object-oriented systems is the choice of good abstractions and the appropriate relations between them.

Complementary Remodularization Algorithms: There is a need for a global remodularization infrastructure in terms of analyses (algorithms, information presentation, metrics) that can take into account the diversity of the abstractions in presence (different module semantics, different abstractions and relationships including different levels, functions, classes, packages, etc.).

Complexity and approximation: In the point of view of graph theory, the central problem seems to be close to the classic k-cuts problem [23], [25]. Nevertheless, we must add several new criteria, like the *quality* of the proposed solution. The first step of the research will be the characterization of an optimization criterion. We also think useful to study and analyse the sensibility of the problem with respect to the operations: add/delete of vertices/edges. It is a major challenge to be able to propose robust approximations.

Scalability: Computational complexity of algorithms has a limit, already known for Formal Concept Analysis (FCA/RCA) and foreseeable for exact methods. Checking the scalability of these algorithms is thus an additional challenge. Another issue is the combinatorial explosion which may occur in the remodularization results. Because of the size of current applications, presenting these results to the engineers and guiding them to take a decision is an additional challenge.

Reengineer inputs and quality of the solution: Engineers should drive the remodularization. Fully automated approaches are applicable only to a very limited context. In reality, external constraints have to be specified and taken into account by the remodularization algorithms (such as the inclusion of a class in a specific package). Software engineers should guide the process possibly confronted to different solutions and their relative impacts. Often favoring minimum impact on existing code has to be considered. Finally the quality of the resulting modularizations has to be taken into account.

As a conclusion: The problems of software evolution are many and varied, in this proposal we plan to consider one of these problems: software remodularization. We think urgent to drive such a complete study of the problem, both "vertically" by studying all the aspects of the modularization problem (modeling of the software, modularization quality metrics, modularization algorithms, presentation of the results), and, "horizontally", by considering different modularization approaches. The solution will not apply one single method, but a combination of various skills in different research domains. Such a research would also be guided by platforms for testing ideas on real-world applications.

REFERENCES

[1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, Mar. 2001.

[2] N. Anquetil and T. Lethbridge. Comparative study of clustering algorithms and abstract representations for software remodularization. *IEE Proceedings - Software*, 150(3):185–201, 2003.

[3] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of 18th Conference on Automated Software Engineering (ASE'03)*, pages 267–270. IEEE Computer Society, Oct. 2003. Short paper.

[4] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.

[5] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, 11(8):47–52, Jan. 1999.

[6] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, Nov. 2005.

[7] P. Bhatia and Y. Singh. Quantification criteria for optimization of modules in oo design. In *Software Engineering Research and Practice*, pages 972–979, 2006.

[8] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, Oct. 2003.

[9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.

[10] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, 2006.

[11] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.

[12] S. Chardigny, A. Seriai, M. C. Oussalah, and D. Tamzalit. Extraction d'Architecture à Base de Composants d'un Système Orienté Objet. In *INFORSID*, pages 487–502, 2007.

[13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[14] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 241–254. ACM Press, 2003.

[15] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.

[16] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.

[18] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.

[19] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[20] E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.

[21] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization (short paper). In *ASE 2008: 23th IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[22] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.

[23] A. Freize and M. Jerrum. Improved approximation algorithm for $\max -k$-cut and $\max$ bisection. *Algorithmica*, 18:67–81, 1997.

[24] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.

[25] O. Goldschmidt and D. Hochbaum. Polynomial algorithm for the k-cut problem. In I. C. Society, editor, *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 444–451, 1988.

[26] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify building class hierarchies algorithms. *ITA*, 34(6):521–548, 2000.

[27] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007.

[28] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of *LNCS*, Malaga, Spain, June 2002. Springer Verlag.

[29] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, 1988.

[30] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[31] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.

[32] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.

[33] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[34] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.

[35] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.

[36] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.

[37] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, May 2001.

[38] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.

[39] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.

[40] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.

[41] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.

[42] S. McDirmid, M. Flatt, and W. Hsieh. Jiazzi: New age components for old fashioned Java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.

[43] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.

[44] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.

[45] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.

[46] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.

[47] R. Sindhgatta and K. Pooloth. Identifying software decompositions by applying transaction clustering on source code. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.

[48] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth.*, 11(2):215–255, 2002.

[49] G. Snelting and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.

[50] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

[51] M. Streckenbach and G. Snelting. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.

[52] R. Strniša, P. Sewell, and M. Parkinson. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM.

[53] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.

[54] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[55] D. Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.

[56] R. Wettel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.

[57] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

Component and Service Farms

Gabriela Arévalo
LIFIA - Facultad de Informática (UNLP) - La Plata (Argentina)
garevalo@sol.info.unlp.edu.ar

Zeina Azmeh, Marianne Huchard, Chouki Tibermacine
LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
{*azmeh, huchard, tibermacin*}*@lirmm.fr*

Christelle Urtado, Sylvain Vauttier
LGI2P - Ecole des Mines d'Alès - Nîmes (France)
{*Christelle.Urtado, Sylvain.Vauttier*}*@mines-ales.fr*

I. CONTEXT

Software components and web services are software building blocks that are used in the composition of modern software applications. They both provide functionalities that require to be advertised by registries in order to be discovered and reused during software building processes[1] [1], [2], [3], [4], [5]. Building or evolving existing software implies assembling software components. This task is not trivial because it requires to select the adequate component or service that provides some part of the desired application functionality and connects easily (with minimum adaptations) to other selected components.

Within this context, we identified two main issues: (1) finding appropriate components from huge databases, and (2) creating and maintaining distributed applications.

Finding appropriate components from huge databases: A huge number of components already exist. For example, the Seekda[2] web service search engine has a catalog of more than 28.000 references, and the OW2 consortium[3] groups around 40 open-source projects in the domain of middleware technology. In Seekda, functionalities are rather directly exposed, but in projects, components are sometimes buried into application code repositories and should be properly extracted before being publicly advertised. However, in both cases the task of finding and adapting an appropriate component is hard and time-consuming, and this will surely worsen when more components become available.

Creating and maintaining distributed applications: There is a growing need for being able to create and maintain distributed applications assembled from third party components as network, middleware and deployment technologies now are mature enough. Reduced cost, increased quality and decreased dependence to a given provider also contribute to popularize this trend. Paradigms that are based on this principle are numerous: component-based development, web 2.0, mashups, cloud computing, software as a service, etc. State-of-the-practice technologies such as OSGi[4], that enables the deployment and redeployment of software (for example, in remotely administered internet boxes), or upcoming technologies such as Google Chrome OS[5], that aims to put web-apps at the center of net-books' operating system, also illustrate this trend.

II. INNOVATIVE REGISTRIES

In this context, it is necessary to design innovative software component registries to advertise components and assist users when they need to search, select, adapt and connect a component to others [6], [7]. We even could think of the automation of these tasks as much as possible, in order to adapt to open and dynamic contexts (remotely administered embedded devices, pervasive computing, open and extensible applications, mobile computing, etc.).

Then, the challenge consists in proposing an online architecture for a component service registry, with two functionalities (a) gives efficient access to adequate components, and (b) provides life-cycle long support to developers and applications (in automatic mode). This registry will be a platform for developers to share their knowledge and experiences on the components they use (what runs and what does not, what are ideal contexts for running some given component, what adaptations have already been performed or tested on some component, what are user ratings on components, etc.), and improve development efficiency as well as running software reliability or liveliness.

As an extension of the component search engine, we think of a *component farm*, where components could be either physically located (component repository model) or solely referenced in an adequately organized component

[1] In the following, we will often use the term *components* as a generic name for both web services and software components.

[2] http://webservices.seekda.com/

[3] http://www.ow2.org/

[4] http://www.osgi.org

[5] http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html

advertisement directory). The component farm (*cf.* Fig. 1) should offer multiple views on components and tailored efficient retrieval mechanisms.

Developed software applications that use components from the farm could either simply use the farm's public facilities or register to benefit from increased community-related capabilities. In the latter case, the added value would be provided by the exchange of component-related data between registered software applications and the farm, in both directions:

- *from software applications to the farm.* Applications could register components developed for their own purposes into the farm directory or share usage information on components they have previously retrieved from the farm, etc. This means that developers should contribute as in collaborative web.
- *from the farm to software applications.* The farm should provide several operations supported by efficient tools for searching and selecting components, providing usage feedback on components, informing the developer of other developers' experiences regarding new products, found problems with components, problem solving issues, possible adaptations, etc.

Our proposal focuses on two challenging issues, each of which is developped in one of the followig sections: (1) the adequacy and efficiency of the organization of the component directory, (2) the proposal of tailored support to applications.

III. Efficient organization

Many components exist but are not always available in public servers/directories. Providing abstract views and adequate services, a global organization could help to stimulate their sharing and be profitable increasing cross-fertilization between projects. The first aim of the registry is to efficiently organize components for several development and maintenance tasks. The components can be physically contained in the repository or just accessible from the directory, through hypertext links, from other external locations. An efficient organization is based on an efficient classification of the components. For example, some approaches [8], [9], [10], [11], [12], [13] have already investigated formal concept analysis (FCA) [14], [15], an approach that rigorously classifies data in structures that have strong mathematical properties. Other approaches should nevertheless be studied.

Several aspects can be used to classify components: syntactical, semantic and pragmatic [16], [17]. All of them could be used complementarily. At the syntactical level, component external descriptions include ports, interfaces, functionality signatures and parameter types. At the semantic level, components can be documented (with plain or structured text, with interaction protocols, etc.), described by keywords (either normalized through ontologies, for example, or not), and by any information that conveys the meaning of the component (for what purpose it has been developed, by whom, etc.). At the pragmatic level, components are documented by information that provides feedback from its usage, in assemblies, choreographies or orchestrations: which functionalities are used, which functionalities are never used, how components are connected (with which adaptations), good and bad experiences, etc.

IV. Application support

The second aim of the registry is software application support. Applications using the components from the registry are invited to register so as the directory can receive feedback. Thus, the directory stores information on which application uses which components and documentation about this use, in order to share usage information among developers.

The registry memorizes the manual or automatic adaptations that are necessary when a faulty component is replaced by another. This information is used to optimize future replacements and is capitalized to be used by other registered applications.

The registry prepares backups (sets of possible substitutes) for each used component in order to ensure rapid repairing of a software application in case one of its components fails [18]. These backups are organized in small classifications for efficiency purposes. They can be used manually or sometimes automatically in restricted cases where either exact matching between the used component and its potential substitute exists, or automatic adaptations are known and can be applied. These backups can be stored inside the repository or uploaded on demand by applications. Backups can also be used by designers to make the software evolve when the application designer wants to add extra functionalities or improve the software quality attributes.

Candidate backup components and possible adaptations are dynamically updated, as components become unavailable or are upgraded. Software applications that use them are also dynamically informed of these updates.

V. Starting point

In previous work, we have started to imagine partial solutions to the issues identified here.

- We have developed an automated process for classifying components from their external descriptions. This process is based on type-theory (we only use syntactic information) and uses FCA to iteratively build lattices that provide functionality signature classifications, interface classifications and component classifications [8], [9], [10].
- We have prototyped the CoCoLa tool [10] that implements the aforementioned process. Thanks to a pivot meta-model, component descriptions from various formats are translated into comparable models (instances of the common meta-model). These descriptions are then processed to build context tables and lattices.

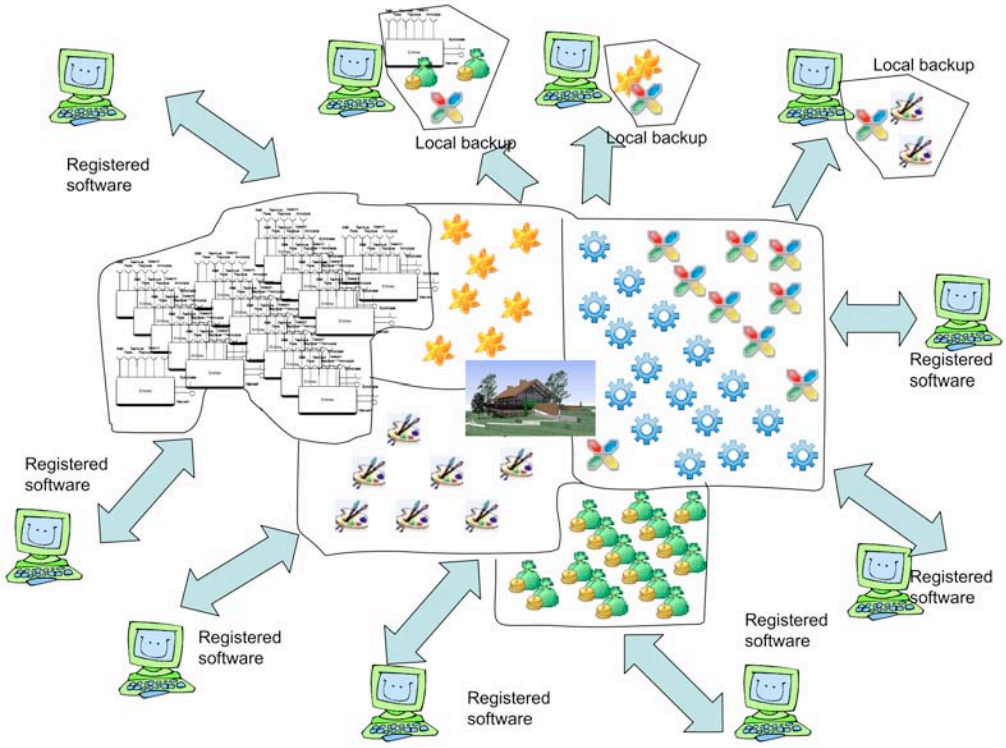


Figure 1. Overview of the component farm

Experiments have been run on the Dream[6] component library (that comes from a real-world component-based framework). They show the feasibility of our approach as the produced lattices enables us to identify possible component substitutions and provides a readable component classification.

- We also have proposed an approach based on formal concept analysis (FCA) for classifying web services [19], [18]. A web service lattice reveals the invisible relations between web services in a certain domain, showing the services that are able to replace other ones. This facilitates service browsing, selection and identification of possible substitutions. We explained how to exploit the resulting lattices to build web services orchestrations and support them with backup services.

VI. MAIN CHALLENGES

There still are numerous challenges to overcome in order to build such a component farm:

- combine syntactical, semantic and pragmatic views of components in order to be able to efficiently search and select appropriated components (pertinence of the classification).
- automate component indexing and classification as well as feedback collection in order not to put the burden on component developers or component users. This task could need to exploit automatically all the available semantic information available (such as functionality names, documentations, interaction protocols, etc.) using text mining techniques [20].
- try and capitalize coarser grained software parts by classifying whole component assemblies,
- think about building techniques and browsing techniques for each of the provided classifications but also inter-classifications.
- propose replacement scenarios and tools,
- define the services that would motivate users in collaboratively providing usage feedback information,
- find means to conduct early experimentations of the component farm. This is not easy as very few component directories are available online, as for now.

As a response to the globalization challenge, software engineering has the capability to positively react in providing software component catalogs, thus increasing software quality while decreasing costs and time-to-market. As an immaterial product market-place, the software component industry benefits from great assets: software components must not be stocked, they can be transported at no cost and instantaneously, they are not perishable. To take advantage of this, efforts should be focused not only on development,

[6]http://dream.ow2.org/

as traditionally done, but also on better diffusion, sharing, deployment and maintenance. We believe that component farms could contribute to this objective.

REFERENCES

[1] W. Hoschek, "The web service discovery architecture," in *Int'l. IEEE/ACM Supercomputing Conference (SC 2002)*, I. C. S. Press, Ed., 2002.

[2] OMG, "Trading Object Service Specification (TOSS) v1.0," 2000, http://www.omg.org/cgi-bin/doc?formal/2000-06-27.

[3] *ebXML Registry Services Specification (RS) v3.0*, http://www.oasis-open.org/, May 2005.

[4] L. Clement, A. Hately, C. von Riegen, and T. Rogers, "Uddi version 3.0.2. uddi spec technical committee draft , dated 20041019. http://uddi.org/pubs/uddi_v3.htm," Tech. Rep. [Online]. Available: http://uddi.org/pubs/uddiv3.htm

[5] Information Technology Open Distributed Processing, "ODP Trading Function Specification ISO/IEC 13235-1:1998(E)," December 1998, http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf.

[6] L. Iribarne, J. M. Troya, and A. Vallecillo, "A trading service for COTS components," *The Computer Journal*, vol. 47, no. 3, pp. 342–357, 2004.

[7] S. Dustdar and M. Treiber, "A view based analysis on web service registries," *Distributed and Parallel Databases*, vol. 18, pp. 147–171, 2005.

[8] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, "Precalculating component interface compatibility using FCA," in *Proceedings of the 5^{th} international conference on Concept Lattices and their Applications (CLA 2007), CEUR Workshop Proceedings Vol. 331*, J. Diatta, P. Eklund, and M. Liquière, Eds., Montpellier, France, October 2007, pp. 241–252. [Online]. Available: http://ceur-ws.org/Vol-331/

[9] ——, "FCA-based service classification to dynamically build efficient software component directories," *Int. Journ. of General Systems*, vol. 38, no. 4, pp. 427–453, May 2009.

[10] N. A. Aboud, G. Arévalo, J.-R. Falleri, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, "Automated architectural component classification using concept lattices," in *In proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 (WICSA/ECSA'09)*. Cambridge, UK: IEEE Computer Society Press, September 2009.

[11] A. M. Zaremski and J. M. Wing, "Specification matching of software components," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 4, pp. 333–369, 1997.

[12] B. Fischer, "Specification-based browsing of software component libraries," in *Proc. of the 13^{th} IEEE int. conf. on Automated Software Engineering (ASE'98)*, 1998, pp. 74–83.

[13] B. Sigonneau and O. Ridoux, "Indexation multiple et automatisée de composants logiciels orientés objet," in *AFADL — Approches Formelles dans l'Assistance au Développement de Logiciels*, J. Julliand, Ed. Besançon, France: RTSI, Lavoisier, Juin 2004.

[14] M. Barbut and B. Monjardet, *Ordre et Classification*. Hachette, 1970.

[15] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.

[16] M. Á. Corella and P. Castells, "Semi-automatic semantic-based web service classification," in *Business Process Management Workshops*, ser. LNCS 4103, J. Eder and S. Dustdar, Eds. Springer, 2006, pp. 459–470.

[17] M. Bruno, G. Canfora, M. D. Penta, and R. Scognamiglio, "An approach to support web service classification and annotation," in *Proc. of the IEEE Int. Conf. on e-Technology, e-Commerce and e-Service (EEE'05)*, 2005, pp. 138–143.

[18] Z. Azmeh, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, "Using concept lattices to support web service compositions with backup services," in *To appear in Proceedings of the 5^{th} International Conference on Internet and Web Applications and Services (ICIW 2010)*, Barcelona, Spain, May 2010.

[19] ——, "WSPAB: A tool for automatic classification & selection of web services using formal concept analysis," in *Proceedings of the 6^{th} IEEE European Conference on Web Services (ECOWS 2008)*. Dublin, Ireland: IEEE, November 2008, pp. 31–40.

[20] J.-R. Falleri, Z. Azmeh, M. Huchard, and C. Tibermacine, "Automatic tag identification in web service descriptions," in *To appear in proceedings of the International Conference on Web Information Systems and Technology (WEBIST'10)*, Valencia, Spain, April 2010.

End-User Modelling

Albert Patrick *(IBM, Paris)*, Blay-Fornarino Mireille *(I3S, Sophia-Antipolis)*, Collet Philippe *(I3S, Sophia-Antipolis)*, Combemale Benoit *(IRISA, Rennes)*, Dupuy-Chessa Sophie *(LIG, Grenoble)*, Front Agnès *(LIG, Grenoble)*, Grost Anthony *(ATOS, Lille)*, Lahire Philippe *(I3S, Sophia-Antipolis)*, Le Pallec Xavier *(LIFL, Lille)*, Ledrich Lionel *(ALTEN Nord, Lille)*, Nodenot Thierry *(LIUPPA, Pau)* et Pinna-Dery Anne-Marie *(I3S, Sophia-Antipolis)* et Rusinek Stéphane *(Psitec, Lille)*

1 Contexte et domaines applicatifs

En 2006, le *Standish Group* indiquait que l'amélioration du taux de réussite des projets informatiques (passant de 16 à 32%) provenait de différents facteurs, dont le plus important était l'implication grandissante des utilisateurs finaux. Le mouvement syndicaliste scandinave (début 70) nommé “conception participative” avait déjà influencé l'ingénierie logicielle dans les années 80 dans ce sens: l'adhésion des employés pour un nouvel outil peut s'obtenir par leur participation à sa conception. Ce sont les démarches agiles, officialisées en 2001, qui ont popularisé ce principe d'implication des utilisateurs dans la conception logicielle. Parallèlement, l'augmentation de la production logicielle et de la taille des applications informatiques a accentué la préoccupation de réutilisation logicielle. Une réponse à celle-ci a été d'accorder plus d'importance à la modélisation dans la conception logicielle : un modèle fournit un meilleur support pour la discussion, la compréhension d'une architecture ou d'un composant et demeure moins impacté par les évolutions technologiques.

L’utilisation de plus en plus intensive des « wiki » et la mise à disposition d’outils google témoignent de l’intérêt de donner aux utilisateurs les moyens de « programmer » leurs propres outils. Cette thématique se retrouve au sein d’éditeurs d'applications *mash up* comme Yahoo Pipes, MS Pop Fly. Ces éditeurs, très en vogue actuellement, sont souvent basés sur des éditeurs de modèles graphiques très accessibles et en principe au moins lisible par un grand nombre d'utilisateurs “avertis”. De manière plus industrielle, la construction d’usines logicielles et les environnements de modélisation dédiés à des domaines spécifiques entrent également dans ce cadre, en donnant à l’utilisateur des langages de modélisation adaptés à son métier. Le défi proposé ici est donc : *comment donner aux experts d’un domaine les moyens de construire leur propre système informatique en utilisant des techniques de modélisation ?*

Les enjeux sont une plus grande productivité et une meilleure adéquation des produits aux problèmes. En particulier en permettant aux utilisateurs d'adapter le logiciel, une plus grande réactivité/agilité est obtenue. Ces enjeux ne seront atteints que si les artefacts de modélisation donnés sont adéquats et si les produits résultants sont fiables et opérationnels.

Les risques sont la définition de méga-métamodèles non exploitables, l’obtention de produits inutiles, non performants, non réutilisables.

Nous proposons d’aborder ce défi par la construction d’environnements de modélisation dédiés à des domaines spécifiques. Des systèmes tels que CENTAUR [1] dans les années 80-90 avaient des objectifs similaires mais étaient orientés langages de programmation. Le défi que nous proposons en s’appuyant sur ces acquis envisage une appréhension de la “programmation” dirigée non pas par la syntaxe mais par les concepts, les usages et les contraintes du métier. Le défi est de faciliter la construction de ces environnements en utilisant des techniques de modélisation, en particulier la modélisation sous la forme de diagrammes.

2 Verrous

Les verrous identifiés se situent au niveau (i) des représentations à donner aux multiples profiles d'utilisateurs qui doivent être en adéquation avec leur usage, (ii) de l’interprétation de ces notations par le système pour assurer l’aide, la validation et la collaboration et l’exécution éventuelle des systèmes produits, (iii) de l’impact sur les démarches usuelles de développement.

1. Représentation métier :
   - comment éviter la surcharge cognitive au niveau des syntaxes concrètes dédiées aux utilisateurs ?
   - comment supporter la montée en compétences (connaissances du système) d'un utilisateur ?
   - comment supporter les multi-représentations lors de travaux collaboratifs ?
2. Interprétation des modélisations métier :
   - comment supporter plusieurs représentations d'un même système (multi-utilisateurs, multi-niveaux) ?
   - comment rendre opérationnelles ces représentations en limitant les coûts de développements et en assurant la cohérence des systèmes construits ? (intégration de l'existant, incrément, agilité des développements ?
   - comment permettre à l'utilisateur final d'interagir avec le système de manière à en adapter le comportement (e.g., systèmes adaptables) ?
   - Un des verrous à lever est le juste appariement entre des approches dîtes "formelles" indispensables à ancrer le procédé dans une logique de fiabilité, et des approches de modélisation plus flexibles qui supportent une expression plus "naturelle" des domaines métiers.
3. Démarches de développement :
   - comment supporter l'agilité de la construction, la robustesse des environnements construits tout en acceptant les incohérences nécessaires au développement collaboratif, …

3 Fondements

L'IDM apporte un ensemble de fondements permettant d'aborder la problématique soulevée par ce défi [2]. En effet, la montée en abstraction supportée par l'IDM pour la construction de systèmes complexes réduit l'écart entre la spécification et la conception/développement d'un système. En premier lieu, la prise en compte des différents profils d'utilisateurs finaux nécessitent de définir des représentations adéquates pour chacun d'entre eux. Cette adéquation nécessite une étude des propriétés cognitives des formalismes utilisés. Pour cela, nous pourrons nous baser sur les synthèses de Bernard Morand [16] sur les propriétés cognitives des diagrammes (du point de vue de la psychologie cognitive, de la linguistique…) comme sur les travaux de Gerald Lohse [17] qui sont plus spécifiques aux diagrammes représentant des systèmes informatiques. D'autres travaux ont porté sur la compréhensibilité globale d'une notation et ont permis d'aboutir à des protocoles expérimentaux réutilisables et utilisées dans le monde industriel [18]. Dans tous ces cas, l'approche utilisée est celle des expériences utilisateurs.

Mais une autre approche, plus automatisable, est possible pour évaluer la qualité d'un modèle ou d'un langage : elle se base sur l'utilisation de métriques qui peuvent être définies sur les modèles [19] ou les métamodèles [20]. Il existent donc de nombreuses possibilités pour appréhender ce qui a été définie dans les frameworks de qualité [21][22] comme la qualité pragmatique, c'est-à-dire la qualité dépendant de l'interprétation des utilisateurs et des concepteurs.

La définition d'un langage dédié (*Domain Specific Language*, ou DSL), et plus récemment de langage de modélisation dédié (*Domain Specific Modeling Language*, ou DSML) a été étudié au travers de techniques telles que les profils UML (consistant à spécialiser UML pour des besoins particuliers), ou la métamodélisation offrant toute l'ingénierie pour la définition d'un nouveau langage [9]. L'outillage de celui-ci peut par ailleurs être facilité à l'aide d'approches génératives ou génériques, telle que par exemple GMF[1] qui permet de générer automatiquement des outils de modélisation métier graphique pour un langage dédié donné. Le traitement automatique, à base de modèles, de langue naturelle ou de règles métiers est également un axe de recherche intéressant pour la résolution de ce défi. La publication récente par l'OMG de la norme *Semantics for Business Vocabulary and Rules* (SBVR)[2] permet par exemple d'interpréter formellement sous la forme de modèle un anglais structuré [23]

Les travaux sur les approches par points de vues devront également être pris en considération de manière à bien préciser l'intention de chaque type d'utilisateurs [15], de même que les travaux sur les usines logicielles [8]. D'autres travaux sur l'extension des métamodèles, tant syntaxique que comportementale (e.g., la modélisation orientée aspect [3,4]) offre également des moyens d'adapter ou de préciser un langage existant pour des besoins spécifiques.

[1] Cf. http://www.eclipse.org/gmf/

[2] *Semantics of Business Vocabulary and Business Rules* 1.0 specification (2008), http://www.omg.org/spec/SBVR/1.0/

D'autre part, des techniques de composition [5,6,10] et de transformation de modèles [7] permettent de prendre en compte les modèles issus de différentes préoccupations de manière à obtenir un modèle unique du système complet. Ces techniques permettent ainsi de rendre productif les modèles métiers et de les considérer comme des artefacts terminaux du processus de développement. Le couplage transparent des représentations métiers avec les méthodes formelles [13] devra également être pris en considération afin d'assurer la fiabilité des systèmes construits.
L'utilisation de ces techniques doit toutefois s'intégrer dans une démarche agile favorisant un cycle court de manière à permettre à l'utilisateur de voir très rapidement le résultat de son travail. Des adaptations dynamiques de l'environnement doivent également être envisagées [24,25]. La nécessité d'intégrer un processus itératif est donc indispensable et doit s'appuyer pour cela sur la traçabilité entre les modèles métiers et le système final obtenu. Cette traçabilité peut par exemple être utilisée pour indiquer les impacts sur le système final de chaque modification d'un modèle métier.

Enfin, l'assistance à la modélisation peut s'appuyer d'une part sur les modèles paramétrés pour permettre une modélisation par la réutilisation de briques génériques [14], et d'autre part sur le typage de modèle [11] pour permettre la généricité de certain traitement [12].

4 Usages et impacts sociétaux

- La programmation des applications est donnée aux experts des domaines, la construction des environnements est donnée aux experts logiciels. Mais les environnements eux-mêmes doivent pouvoir évoluer par des cycles courts.
- Les services informatiques représentent une activité économique importante (> 700Md $ en 2008). C'est pourtant un secteur dont la majorité des projets sont des échecs. L'implication des utilisateurs est le facteur de réussite préconisé mais sa mise en œuvre reste difficile. Un objectif majeur du défi est de diminuer voir supprimer cette difficulté.

5 Jalons, démarche

- Circonscrire la complexité par des études de cas :
  - Expérimentations sur des classes d'applications pour lesquelles les langages/formalismes sont maîtrisables par les "personnes métier" (ex. : Business Rules, modélisation de workflows en analyse d'images médicales, modélisation de systèmes de diffusions d'informations en milieu scolaire, domotique),
- Définition des artefacts de modélisation (formalismes) cognitivement adaptés aux différents rôles (ou personnes) impliquées
  - Processus de construction attendu, validations, tests : les artefacts se définissent relativement à leur usage.
  - Les experts d'un domaine peuvent adapter ou faire évoluer l'application uniquement sur la partie métier.
- Corrélation entre modélisation métier et mise en œuvre au niveau des plates-formes :
  - Dans un premier temps, des experts en programmation définissent les méthodologies, processus, outils de tests et de validation qui permettent de construire, modifier, valider des "modèles métiers" et d'assurer la cohérence des systèmes construits : "*Controlled Agility*".
  - Dans un second temps, des patrons (transformations, mécanismes de profiling) sont utilisés pour, sur la base de la définition de métamodèles métiers, obtenir les outils associés (validation, tests, …). Partiellement atteint dans le cadre des langages de programmation, cet objectif devrait bénéficier des avancées en matière de métamodélisation.
  - En obtenant une certaine automatisation de la production des outils, les utilisateurs finaux devraient pouvoir faire évoluer leur propre environnement de modélisation dans la limite des contraintes nécessaires pour assurer la cohérence des environnements et des systèmes construits.

References

1. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, *Centaur: the system* , Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3), Boston, USA, 1988.
2. Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. L'ingénierie dirigée par les modèles : au-delà du MDA. Hermes-Lavoisier, Cachan, France, February 2006.
3. Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. Transactions on Aspect-Oriented Software Development (TAOSD), LNCS 4620 :167–199, 2007.
4. Jean-Marc Jézéquel. – Model driven design and aspect weaving. – Journal of Software and Systems Modeling (SoSyM), 7(2):209–218, may 2008.
5. Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In EDOC'07 (Entreprise Distributed Object Computing Conference), Annapolis, MD, USA, 2007.
6. Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson, Siobhan Clarke, *Composing Multi-View Aspect Models*, 7th IEEE International Conference on Composition-Based Software Systems (ICCBSS) (2008)
7. Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. A canonical scheme for model composition. In Arend Rensink and Jos Warmer, editors, ECMDA-FA, volume 4066 of Lecture Notes in Computer Science, pages 346–360. Springer, 2006.
8. Carlos Parra , Rafael Leano , Xavier Blanc, Laurence Duchien, Nicolas Pessemier, Chantal Taconet and Zakia Kaziaoul, *Dynamic Software Product Lines for Context-Aware Web Services*., in Enabling Context-Aware Web Services: Methods, Architectures, and Technologies. Chapman and all, July 2009
9. F. Barbier, *UML 2 et MDE - Ingénierie des modèles avec études de cas*, Dunod, 2005
10. Mireille Blay-Fornarino, *"Interprétations de la composition d'activités"*, HDR Thesis University of Nice-Sophia Antipolis, 1-201 pages, Sophia Antipolis, France, apr 2009
11. Jim Steel and Jean-Marc Jézéquel. – On model typing. – Journal of Software and Systems Modeling (SoSyM), 6(4):401–414, December 2007.
12. Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. – Generic Model Refactorings. – In ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, Colorado, USA, Oct 2009.
13. B. Combemale, X. Crégut, P.-L. Garoche and X. Thirioux – Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification – Journal of Software (JSW), 4(6), December 2009
14. Alexis Muller, Olivier Caron, Bernard Carré, Gilles Vanwormhoudt, *On Some Properties of Parameterized Model Application* in Proceedings of ECMDA-FA'2005: First European Conference on Model Driven Architecture - Foundations and Applications, Nuremberg, November 2005, LNCS 3748, A. Hartman and D. Kreische Eds.
15. Adil Anwar, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile. A Rule-Driven Approach for composing Viewpoint-oriented Models. Dans : Journal of Object Technology, ETH Swiss Federal Institute of Technology, p. 1-26, 2010
16. B. Morand, Logique de la Conception, Figures de sémiotique générale d'après Charles S. Peirce, L'Harmattan, Collection L'Ouverture Philosophique, 294 p.
17. G. L. Lohse, D. Minb and J. R. Olsonc, Cognitive evaluation of system representation diagrams, Information & Management, Volume 29, Issue 2, August 1995, Pages 79-94
18. (Patig 2008) : S. Patig, A practical guide to testing the understandability of notations, Conferences in Research and Practice in Information Technology Series; Vol. 325, Proceedings of the fifth on Asia-Pacific conference on conceptual modelling - Volume 7, 2008, pp 49-58
19. (Lange 2005) : C. Lange, M. Chaudron, Managing Model Quality in UML-Based Sotware Delevopment, Proc. of the 13th workshop on software Tachnology and Engineering Practice (STEP'05), 2005, pp. 7-16.
20. (Rossi 1996) M. Rossi et S. Brinkkemper, Complexity metrics for systems development methods and techniques, Information Systems, Vol. 21, num 2, 1996, pp. 209-227.
21. (Linbdland 1994) O. Lindland, G. Sindre, A. Solvberg, Understanding Quality in Conceptual Modeling, IEEE Software, Vol. 11, 1994, pp. 42-49.
22. (Krogstie 1998) J. Krogstie, Integrating the Understanding of Quality in Requirements Specification and Concptual Modeling, Software Engineering Notes, ACM SIGSOFT, Vol 23, num 1, 1996, pp. 86-91
23. Mathias Kleiner, Patrick Albert and Jean Bézivin, Parsing SBVR-based controlled languages, In ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), empirical track, Denver, Colorado, USA, Oct 2009, pages 122-136.
24. Reza Razavi, Noury Bouraqadi, Joseph W. Yoder, Jean-François Perrot, Ralph E. Johnson: *Language support for adaptive object-models using metaclasses*. Computer Languages, Systems & Structures 31(3-4): 199-218 (2005)
25. Razavi, Reza, Kirill Mechitov, Gul Agha, Jean-Francois Perrot. "Ambiance: A Mobile Agent Platform for End-User Programmable Ambient Systems," J.C. Augusto and D. Shapiro (eds.), Advances in Ambient Intelligence, Frontiers in Artificial Intelligence and Applications (FAIA), vol. 164, IOS Press, 2007

Towards Disappearing Languages

Charles Consel,
INRIA / University of Bordeaux
Charles.Consel@inria.fr

This paper explores challenges that need to be addressed to make the Domain-Specific Language (DSL) approach successful. In particular, we argue that a DSL should be blended with a domain process and used by domain experts in support of their job. We call such languages *disappearing languages*.

The DSL approach has long been used with great success in both historical domains, such as telephony, and recent ones, such as Web application development. And yet, from software engineering to programming languages, there is a shared feeling that there is still much work to do to make the DSL approach successful.

Unlike General-Purpose programming Languages (GPLs) that target trained programmers, a DSL revolves around a domain: it originates from a domain and targets members of this domain. Thus, a successful DSL should be some kind of a *disappearing language*; that is, one that is blended with some domain process. In doing so, programming expands its scope to reach end users. That is, users for which writing programs come in support of achieving their primary job function. Well-known examples include Excel and MatLab.

Creating a disappearing language critically relies on an analysis phase of the target domain. The result of this analysis then fuels a design phase of the language. Practical experience shows that these two phases are time consuming, human intensive and high risk.

- What kind of tools could assist in performing these two phases?
- How much improvement can we expect from a tool-assisted process?

A successful DSL is above all one that is being used. To achieve this goal, the designer may need to downgrade, simplify and customize a language. In doing so, DSL development contrasts with programming language research where generality, expressivity and power should characterize any new language. As a consequence, programming language experts may not be the perfect match for developing a DSL.

- Does this mean that, for a given domain, its members should be developing their own DSL?
- Or, should there be a new community of *language engineers* that bridge the gap between programming language experts and members of a domain?

In many respects, a DSL is often an over-simplified version of a GPL: customized syntactic constructs, simple semantics, and by-design verifiable properties. These key differences may raise concerns about a lack of tool support for DSL development. Yet, there are many program manipulation tools (parser generators, editors, IDEs) that can be easily customized for new languages, whether textual or graphical. Furthermore, for a large class of DSLs,

compilation amounts to producing code for a domain-specific programming framework, and enabling the use of high-level transformation tools. Lastly, properties can often be checked by generic verification tools.

- Then, what is missing to develop DSLs?
- Do we need to have an integrated environment for DSL development, orchestrating a library of tools?
- Should there be a new breed of compiler and analysis generators matching the requirements of DSLs?

The research community in programming languages has always overwhelmingly focused on the design and implementation of languages. Very little effort has been devoted to understanding how humans use programming languages. This avenue of research is a key to expanding programming beyond computer scientists.

- How do we design languages that are easy to use?
- How do we measure the productivity benefit of the use of a language?
- How do we assess the software quality of DSL programs?

Vers une réification de l'énergie dans le domaine du logiciel

L'énergie comme ressource de première classe

Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé
Pierre Cointe, Rémi Douence, Mario Südholt

Équipe ASCOLA (EMNantes-INRIA, LINA)
prénom.nom@emn.fr

Résumé

En quelques années, le problème de la gestion de l'énergie est devenu un enjeu de société. En informatique, les principaux travaux se sont concentrés sur des mécanismes permettant de maîtriser l'énergie au niveau du matériel. Le renforcement du rôle de l'informatique dans notre société (développement des centres de données, prolifération des objets numériques du quotidien) conduit à traiter ces problèmes aussi au niveau du logiciel.

Dans ce papier, nous nous posons la question de la réification de l'énergie comme fut posée en son temps celle de la réification de la mémoire (l'espace) et de l'interpréteur (la machine d'exécution). Le défi est d'abord de sensibiliser l'utilisateur final au problème de la consommation énergétique en visualisant, grâce à des mécanismes d'introspection, sa consommation, à l'image de ce qui se fait aujourd'hui dans le domaine automobile (consommation instantanée d'essence). Il s'agira ensuite de proposer aux développeurs des mécanismes d'intercession leur permettant de contrôler cette consommation énergétique. Ces mécanismes réflexifs devront concerner l'ensemble du cycle de vie du logiciel.

1 L'énergie comme nouvelle préoccupation ?

En 2008, 2% de la production électrique mondiale a été consommée par les centres de données utilisés pour l'hébergement de l'Internet. La problématique est d'importance car, d'un point de vue mondial, et selon l'*Environmental Protection Agency* (EPA), il a été estimé qu'entre 2007 et 2012 les serveurs et les centres de données vont doubler leurs besoins en énergie pour atteindre 100 milliards de kWh. Cette augmentation des besoins est liée à l'accroissement du nombre d'internautes [1], des terminaux d'accès [2], des données disponibles [3], et à l'apparition de nouveaux usages comme le *Cloud Computing*, ou l'Internet des Objets [4]. À ce rythme, dans 25 ans, Internet consommera autant d'énergie que l'humanité tout entière en 2008.

D'un point de vue plus global, l'impact des TIC

1. Principalement dans les pays du BRIC : Brésil, Russie, Inde, Chine.

2. Le nombre d'ordinateurs est estimé à 1 milliard en 2008, 2 milliards en 2013, sans compter le nouveau marché des téléphones intelligents (77 millions vendus sur les trois premiers mois de 2009 soit 14 par seconde).

3. 161 exaoctets de données ont été créées en 2006, soit approximativement 3 millions de fois l'information contenue dans tous les livres jamais écrits.

4. D'ici à quelques années, le nombre d'appareils communiquants pourrait être de plusieurs dizaines de milliards, soit plus que le nombre d'êtres humains.

dans la consommation électrique est de l'ordre de 10 à 15% dans les pays développés. Pour la France, l'estimation de la consommation électrique est comprise entre 55 et 60 TWh par an, soit 13,5% de la consommation française. Au niveau des particuliers, les TIC dans leur globalité représentent 30% [5] de l'électricité consommée. Depuis 10 ans, cette consommation a triplé. Cette progression risque d'augmenter fortement dans les prochaines années avec l'apparition d'équipements toujours plus « énergivores » (écrans LCD, passerelles domestiques, etc.).

Par conséquent, une rupture technologique est essentielle pour relever le défi majeur de la maîtrise énergétique des équipements informatiques. Les fondeurs, constructeurs et installateurs de matériels informatiques ont déjà travaillé sur ce problème. Par exemple, dans les centres de données, l'unité d'accueil de nouvelles machines n'est pas le nombre de serveurs à installer ou leur type, mais la puissance électrique consommée. Si des modèles de consommation d'énergie au niveau matériel sont aujourd'hui disponibles, il n'existe pas l'équivalent au niveau logiciel.

Dans ce papier, nous explorons les possibilités d'intégrer l'énergie en tant que ressource de première classe pour le logiciel, et donc de disposer de tels modèles dans le domaine du logiciel, manipulables par le logiciel. Cette réification permettra de maîtriser l'empreinte énergétique d'une application tout au long de son cycle de vie.

2 L'énergie, nouvelle ressource physique ?

En matière de gestion explicite des ressources physiques et depuis la fin des années 40, l'informatique est passée d'un extrême à l'autre. Pendant longtemps, le programmeur a été comptable de toutes les ressources qu'il utilisait : mémoire, nombre de cycles du processeur, nombre d'accès au disque, etc. Ainsi sous DOS, la mémoire était organisée en blocs de 64 Ko et cela expliquait la taille maximale d'un tableau dans des langages comme Pascal. Pour le langage Lisp, les programmeurs attachaient une grande importance au contrôle de la consommation de la mémoire qui se traduisait dans le nombre de doublets alloués et la taille de la pile. D'où l'utilisation des fonctions de chirurgie (par exemple `nconc` versus `append` pour la concaténation) permettant d'altérer physiquement les listes. « *Ces fonctions sont d'un maniement délicat faisant passer LISP d'un langage d'expressions à un langage de manipulation de pointeurs où les programmes peuvent s'automodifier* » (chapitre 2 de [2]). Avec l'usage des macros, ces fonctions ont donné lieu aux travaux sur la métaprogrammation et la réflexion, dont 3LISP, dans le but initial d'autodécrire des mécanismes de base de l'interpréteur dont la gestion de la mémoire et celle des continuations [4].

À partir des années 80, la mise en œuvre de mécanismes de gestion des ressources, comme les ramasse-miettes, a permis de décharger les développeurs de la contrainte d'une gestion fine des ressources. Ce détachement du programmeur vis-à-vis des ressources a été amplifié par la montée en puissance des capacités des machines suivant la loi de Moore. Les méthodes de programmation et de génie logiciel se sont affranchies (presque) complètement des notions physiques de la machine pour traiter de la rapidité de développement (objets, programmation agile, patrons de conception, usine logicielle) et la facilité de maintenance et d'évolution (architectures à base de composants, d'aspects et de services).

Grâce à ces techniques logicielles, et à la loi de Moore, nous avons pu construire des applications complexes à forte fonctionnalité ajoutée. Mais, alors que la demande continue à croître, il est impératif d'enrayer l'envolée de la consommation énergétique associée. On va donc rapidement se retrouver dans la situation des programmeurs des années 70, avec des contraintes énergétiques plutôt que des contraintes sur la consommation de la mémoire. Peut-on appliquer à l'énergie les mêmes stratégies que celles utilisées pour gérer la consommation de la mémoire ? Contrairement à la mémoire, la consommation énergétique propre d'une application n'est pas directement accessible. Le premier verrou à lever consiste à réifier cette ressource particulière.

5. Étude REMODECE 2007, ADEME, électricité spécifique consommée.

3 Comment réifier l'énergie ?

Pour réifier l'énergie, il faut dans un premier temps la quantifier, donc disposer d'outils logiciels ou matériels permettant de mesurer la consommation électrique d'un quantum d'exécution (instruction de la machine, pseudo-code, exécution d'un service, suivant la granularité choisie). Cependant mesurer la consommation énergétique d'un quantum d'exécution complexe, comme un service, est une tâche ardue. En effet, l'énergie consommée par un quantum d'exécution est la somme des énergies consommées par les ressources physiques utilisées par ce quantum d'exécution (CPU, mémoire, disque, réseau, etc.).

Dans un deuxième temps, il est nécessaire de construire un modèle théorique de la consommation énergétique qui permette d'attacher des quanta d'énergie aux quanta d'exécution. Pour être utilisable, ce modèle doit être compositionnel et réaliste, c'est-à-dire qu'il doit être capable de prédire avec suffisamment de précision la consommation d'un programme par composition simultanée des quanta d'exécution et des quanta d'énergie. Ces besoins peuvent conduire à choisir des quanta de grain assez gros. Travailler à un niveau de pseudo-code semble prometteur [3]. Une première difficulté peut être, suivant le modèle de programmation, la prise en compte de la composition parallèle de programmes. Certains modèles de programmation, comme le modèle de programmation synchrone, peuvent être plus favorable que d'autres. Une deuxième difficulté est, dans un contexte distribué, de prendre en compte les coûts de communication et plus généralement d'infrastructure (assurant la disponibilité des services).

Cette quantification énergétique des applications sera utilisée à la fois statiquement par le programmeur et le compilateur, assistés de systèmes d'analyse statique quantitative (voir, par exemple, [5]), et dynamiquement par le système d'exécution.

Réifier l'énergie au niveau langage permettra à un programmeur d'introspecter son code sous l'angle énergétique, et de le modifier en changeant ses fonctionnalités ou son implémentation, par exemple en développant des algorithmes de type *anytime* [6]. Les algorithmes *anytime* sont des algorithmes dont la qualité des résultats augmente avec le temps de calcul. Ce type d'algorithme permet de contrôler l'énergie affectée à un calcul en interrompant ce calcul lorsque son quota est épuisé.

Réifier l'énergie au niveau du système d'exécution permettra d'attribuer l'énergie disponible aux applications et d'arbitrer ces allocations en cas de pénurie (énergétique) comme cela est fait pour d'autres ressources [1]. Des stratégies d'allocations spécifiques à la gestion énergétique seront probablement à développer.

4 Conclusion

L'informatique au sens large a et aura un impact significatif sur la consommation électrique mondiale. La poursuite de son développement passe par une prise en charge généralisée de la contrainte énergétique au-delà des domaines habituels du matériel et de certains secteurs de l'informatique embarquée. Pour ce faire, il est nécessaire de réifier l'énergie en faisant de celle-ci une ressource accessible statiquement et dynamiquement au niveau de chaque composant applicatif. Il sera alors possible, dans un premier temps, de sensibiliser de manière efficace à la fois les utilisateurs et les programmeurs finaux aux coûts énergétiques et, dans un deuxième temps, de fournir des concepts [6] et des outils de développement ainsi que des systèmes d'exécution susceptibles de significativement réduire ces coûts.

Références

[1] Luc Morau and Christian Queinnec. Resource aware programming. *ACM Transactions on Programming Languages and Systems*, 27(3) :441–476, May 2005.

[2] Christian Queinnec. *LISP Mode d'emploi.* Eyrolles, 1984.

[3] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-Level Energy Consumption Esti-

6. On peut par exemple penser au développement, dans la continuité des algorithmes *anytime*, d'une algorithmique ≪ verte ≫.

mation for Distributed Java-Based Software Systems. In *Component-Based Software Engineering*, pages 97–113. Springer-Verlag, 2008.

[4] Brian Cantwel Smith. Reflection and semantics in LISP. Technical report, Palo Alto Research Center, 1984.

[5] Pascal Sotin, David Cachera, and Thomas Jensen. Quantitative static analysis over semirings : Analysing cache behaviour for Java Card. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Quantitative Aspects of Programming Languages (QAPL '06)*, 2006.

[6] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3) :73–83, 1996.

Posters et démonstrations

SPaCIFY

Projet ANR 06 TLOG 17

http://spacify.gforge.enseeiht.fr/

AGENCE NATIONALE DE LA RECHERCHE ANR

Ingénierie Dirigée par les Modèles et Méthodes Formelles pour les Systèmes embarqués critiques

Contact : cortier@irit.fr

Partenaires Industriels : CNES, Thales Alenia Space, EADS Astrium, Geensys, Anyware Technologies

Partenaires académiques : IRIT-ACADIE, Labri, Télécom Bretagne - CAMA, IRISA-ESPRESSO

Objectifs du projet

Développer une méthodologie et un environnement de conception pour les systèmes embarqués (Logiciel de Vol) basée sur :

- l'**Ingénierie Des Modèles**(IDM)
- les **Méthodes Formelles**

- Définition d'un langage métier : le langage synchrone Synoptic
- Outil de conception graphique (TOPCASED, OpenEmbeDD)
- Vérification & Validation, génération de code
- Plateforme d'exécution (intergiciel) : architecture et services

Ingénierie des Modèles et Méthodes formelles

Ingénierie des modèles :

- placer les modèles au coeur du processus de conception
- modèle = vecteur de communication
- gestion de l'information, automatisation de certaines tâches

Modèles et techniques formelles :

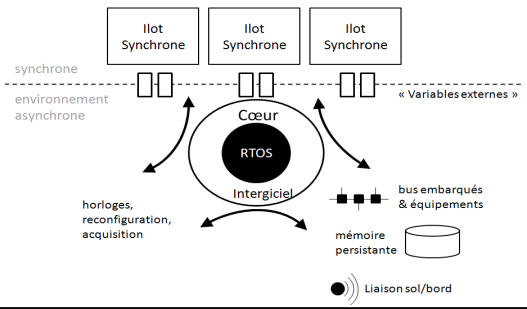
- effectuer des analyses mathématiques rigoureuses
- détecter des erreurs de modélisation au plus tôt
- vérifier certaines propriétés du système

Le langage de modélisation Synoptic

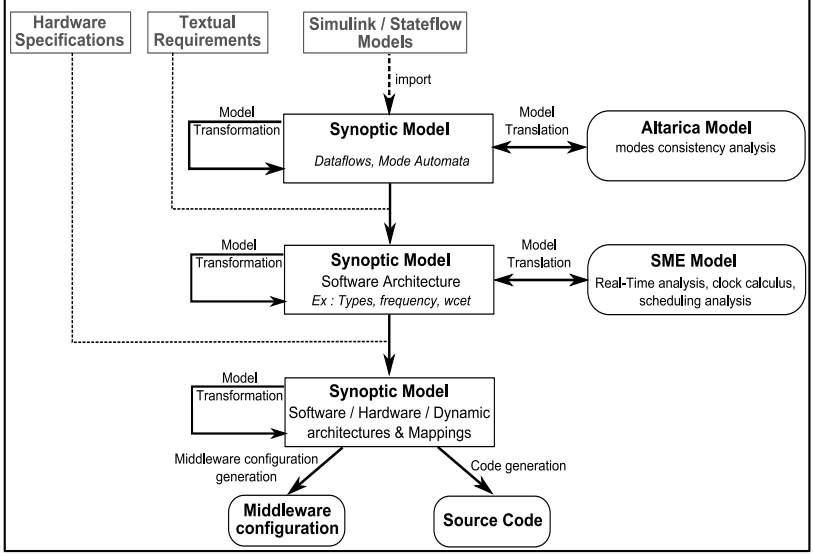
- **Langage métier** dédié à l'aérospatial pour la modélisation des systèmes embarqués critiques
- **Langage pivot du processus SPaCIFY**
  - inspiré de Simulink/Stateflow, AADL& StateCharts
- Modélisation des **apsects fonctionnels, architecturaux, dynamiques et temps-réel**
- Le langage Synoptic est **doté d'une sémantique formelle** :
  - sémantique synchrone multi-horloge
  - objectifs : **vérifications** (model-checking), **validations de transformations** **génération de code** (Approche Geneauto)

Code genéré et Intergiciel : système GALS

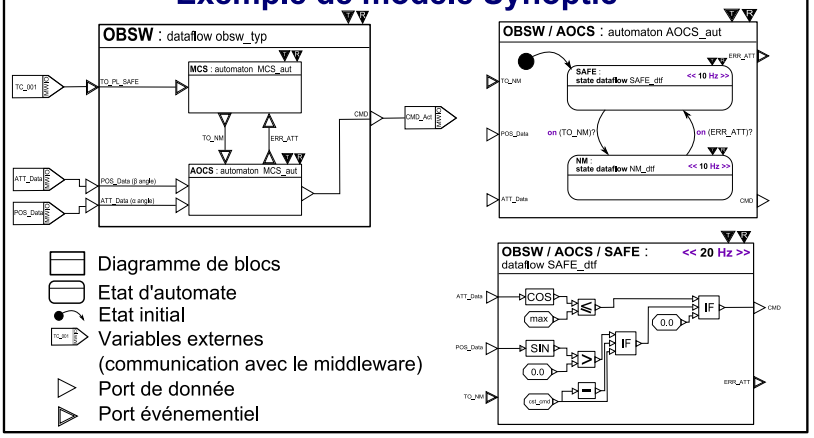
- GALS : **Globalement Asynchrone Localement Synchrone**
- Code généré = ensemble d'ilôts synchrones
- Spécification de l'Intergiciel = architecture et services devant être fournis par une plateforme d'exécution pour supporter l'exécution temps réel du code généré
- Caractéristiques de l'Intergiciel : distribution, partitionnement, adaptation dynamique



Processus de développement SPaCIFY



Exemple de modèle Synoptic



Réalisation et travaux en cours

Réalisations :

- Définition du langage de modélisation dédié Synoptic et de sa sémantique synchrone multi-hotloge
- Développement d'un éditeur textuel et graphique
- Transformation par traduction Synoptic → SME (Signal Meta under Eclipse)
- Définition de l'architecture et des services du middleware
- Réalisation d'une étude de cas pilote

Travaux en cours :

- Transformation par traduction Synoptic → Altarica
  model-checking : vérification de cohérence des modes
- Génération de code modulaire pour systèmes GALS (Polychrony)
- Validation formelle de transformations métiers
- Outils de simulation
- Etudes de cas industrielles :
  - partie contrôle/commande d'un GCU
  - modélisation d'un Système de Contrôle d'Orbite et d'Attitude
  - modélisation d'un système de vol en formation

Modelling Search Strategies in Rules2CP

François Fages, Julien Martin

INRIA Rocquencourt, BP105, 78153 Le Chesnay Cedex, France.
http://contraintes.inria.fr

In this abstract, we present a rule-based modelling language for constraint programming, called Rules2CP [1], and a library PKML for modelling packing problems. Unlike other modelling languages, Rules2CP adopts a single knowledge representation paradigm based on logical rules without recursion, and a restricted set of data structures based on records and enumerated lists given with iterators. We show that this is sufficient to model constraint satisfaction problems together with search strategies, where *search trees* are expressed by *logical formulae*, and *heuristic choice criteria* are defined by *preference orderings on variables and formulae*. Rules2CP statements are compiled to constraint programs over finite domains (currently SICStus-prolog and soon Choco-Java) by term rewriting and partial evaluation.

The Packing Knowledge Modelling Language (PKML) is a Rules2CP library developed in the European project Net-WMS for dealing with real size non-pure bin packing problems in logistics and automotive industry. PKML refers to shapes in $\mathbb{Z}^K$. A *point* in this space is represented by the list of its K integer coordinates. A *shape* is a *rigid assembly of boxes*, represented by a record. A *box* is an orthotope in $\mathbb{Z}^K$. An *object* , such as a bin or an item, is a record containing one attribute `shapes` for the list of its *alternative shapes*, one *origin* point, and some optional attributes such as weight, virtual reality representations or others. The alternative shapes of an object may be the discrete rotations of a basic shape, or different object shapes in a configuration problem. The end in one dimension and the volume of an object with alternative shapes are defined with reified constraints.

PKML uses Allen's interval relations in one dimension, and the topological relations of the Region Connection Calculus in higher-dimensions, to express placement constraints. Pure *bin packing problems* can be defined as follows:

```
non_overlapping(Items,Dims) --> forall(O1,Item, forall(O2,Items,
        uid(O1)<uid(O2) implies notoverlap(O1,O2,Dims))).
containmentAE(Items,Bins,Dims) -->
        forall(I,Items, exists(B,Bins, contains_rcc(B,I,Dims))).
bin_packing(Items,Bins,Dims) --> containmentAE(Items,Bins,Dims)
        and non_overlapping(Items,Dims) and labeling(Items).
```

Other rules about weights, stability, as well as specific packing business rules can be expressed, e.g.

```
gravity(Items) --> forall(O1,Items, origin(O1,3)=0
        or exists(O2,Items, uid(O1)\#uid(O2) and on\_top(O1,O2))).
```

The search is specified here with a simple labeling of the (coordinate) variables of the items. Heuristics can be defined as preference orderings criteria, using the expressive power of the language by pattern matching. For instance, the statement `variable_ordering([greatest(volume(^ )), is(z(^ ))])` expresses the choice of the variables belonging to the objects of greatest volume first, and among them, the z coordinate first.

On Korf's benchmarks of optimal rectangle packing problems [2] (i.e. finding the smallest rectangle containing n squares of sizes $S_i = i$ for $1 \leq i \leq n$), compared to the SICStus Prolog program of Simonis and O'Sullivan [3] which improved best known runtimes up to a factor of 300, the SICStus Prolog program generated by the Rules2CP compiler explores exactly the same search space, and is slower by a factor less than 3 due to the interpretation overhead for the dynamic search predicates. The follwoing table shows the compilation and running times in seconds:

N	R2CP compilation	Rules2CP	Original
18	0.266	13	6
20	0.320	20	10
22	0.369	364	197
24	0.443	5230	1847
25	0.509	52909	17807

Rules2CP differs from OPL, Zinc and Essence modelling languages in several aspects among which: the use of logical rules, the absence of recursion, the restriction to simple data structures of records and enumerated lists, the specification of search by logical formulae, the specification of heuristics as preference orderings, and the absence of program annotations. The expression of complex search strategies and heuristics is currently not expressible in Zinc and Essence, and can be achieved in OPL in a less declarative manner by programming. On the other hand, we have not considered the compilation of Rules2CP to other solvers such as local search, or mixed integer linear programs, as has been done for OPL and Zinc systems.

As for future work, a large subset of PKML rules has been shown in [4] to be compilable with indexicals *within* the geometrical kernel of the global constraint `geost` providing better performance than by reification. The generality of this approach will be explored with greater generality for Rules2CP. The specification of search strategies in Rules2CP will also be explored more systematically, possibly with adaptive strategies in which the dynamic criteria depend on execution profiling criteria.

1. Fages, F., Martin, J.: From rules to constraint programs with the Rules2CP modeling language. In: Proc. 13th International Workshop on Constraint Solving and Constraint Programming CSCLP'08. To appear in Recent Advances in Constraints. Lecture Notes in Artificial Intelligence, Roma, Italy, Springer-Verlag (2008)
2. Korf, R.E.: Optimal rectangle packing: New results. In: ICAPS. (2004) 142–149
3. Simonis, H., O'Sullivan, B.: Using global constraints for rectangle packing. In: Proceedings of the first Workshop on Bin Packing and Placement Constraints BPPC'08, associated to CPAIOR'08. (2008)
4. Carlsson, M., Beldiceanu, N., Martin, J.: A geometric constraint over k-dimensional objects and shapes subject to business rules. In Stuckey, P.J., ed.: Proceedings of CP'2008. Volume 5202 of LNCS., Springer (2008) 220–234

EUclide is a Constraint Language based on Imperative DEfinition

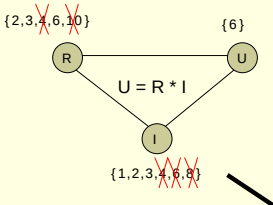
INRIA

Supported by ANR CAVERN

http:// cavern.inria.fr

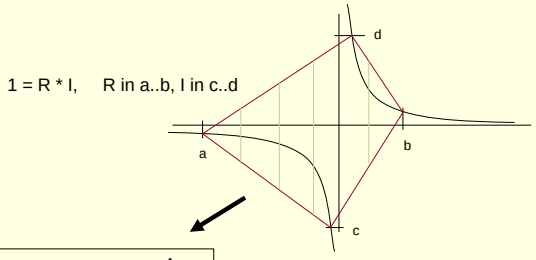
Constraint Programming

Exploit relations (constraints) to infer new informations on objects that represent unknowns (variables)

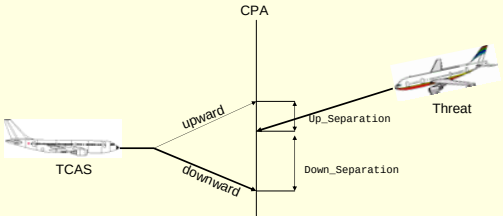


Abstractions

Over-approximate the computation of relations to benefit from powerful solving techniques (e.g. Linear Programming)



Automatic test data generation for critical C programs



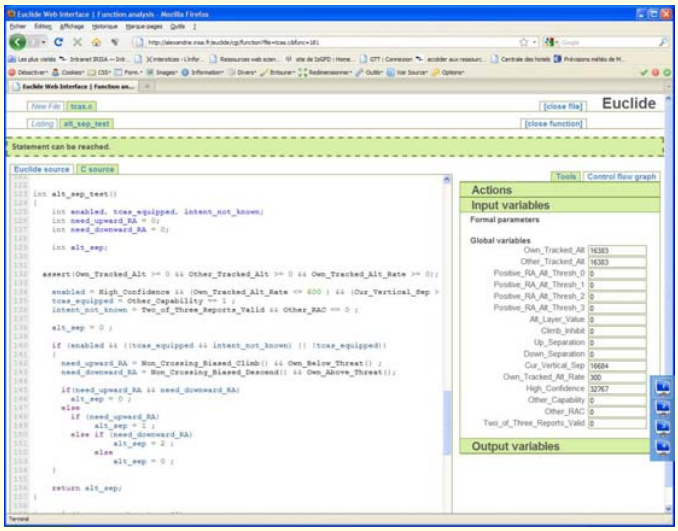
Num.	Property	Explanation	ACSL specification
P1a	Safe advisory selection	An downward RA is never issued when an downward maneuver does not produce an adequate separation	assumes Up_Separation >= Positive_RA_Alt_Thresh && Down_Separation < Positive_RA_Alt_Thresh; ensures result != need_Downward_RA;
P1b	Safe advisory selection	An upward RA is never issued when an upward maneuver does not produce an adequate separation	assumes Up_Separation < Positive_RA_Alt_Thresh && Down_Separation >= Positive_RA_Alt_Thresh; ensures result != need_Upward_RA;
		A downward RA is never issued when neither climb or descend ma-	assumes Up_Separation < Positive_RA_Alt_Thresh

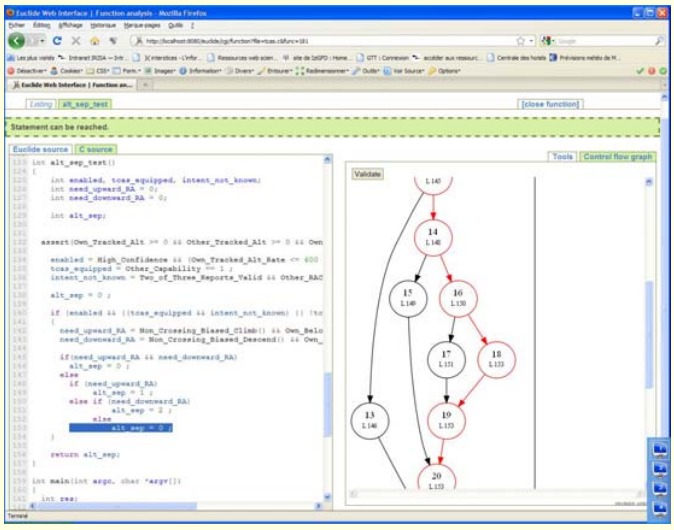
Are properties
P1a, P1b, etc. satisfied
by this implementation ?

```
int alt_sep_test()
{
1.    bool enabled, tcas_equipped, intent_not_known;
2.    bool need_upward_RA, need_downward_RA;
3.    int alt_sep;

4.    enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV)
                && (Cur_Vertical_Sep > MAXALTDIFF);
5.    tcas_equipped = (Other_Capability == TCAS_TA);
6.    intent_not_known = (Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT);

7.    alt_sep = UNRESOLVED;

8.    if (enabled && ((tcas_equipped && intent_not_known) ||!tcas_equipped))
      {
9.        need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
10.       need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
11.       if (need_upward_RA && need_downward_RA)
              // unreachable: Own_Below_Threat and Own_Above_Threat can't be both true
              alt_sep = UNRESOLVED;
12.       else if (need_upward_RA)
13.           alt_sep = UPWARD_RA;
14.       else if (need_downward_RA)
15.           alt_sep = DOWNWARD_RA;
16.       else
17.           alt_sep = UNRESOLVED;
      }
18.   return alt_sep;
      }
```

EUCLIDE is a free open-source software

A web-based interface, accessible online http://euclide.gforge.inria.fr/





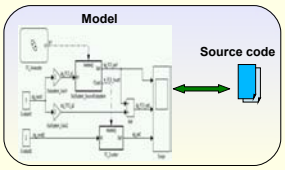
Contact: **Arnaud.Gotlieb@inria.fr** **(INRIA Rennes)**

Integrated Formal Approach for a Qualified Critical Code Generator

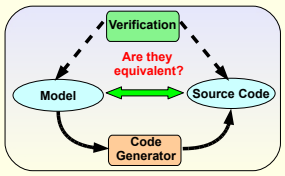
N. Izerrouken[1,2](nizerrou@N7.fr), M. Pantel[2], X. Thirioux[2] and O. Ssi Yan Kai[1]
[1]Continental Automotive, Innovation Center, Toulouse, France
[2] Institut de Recherche en Informatique de Toulouse, Institut National Polytechnique de Toulouse, France

Problem Statement

- Code generators reduce development time and verification costs (compliance of source code w.r.t. model-based design)
- Industry is aware that critical systems require more rigorous verification than classical testing
  => formal specification & verifiaction

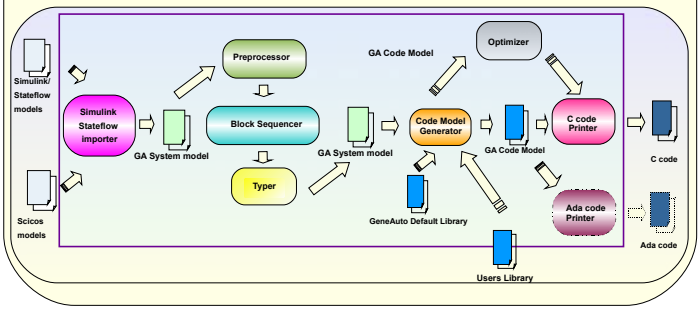


- Formal verification of:
  - Input model
  - Generated source code
  - Compliance generated code/model
- Complex analysis of verification failures
  => Verification of the Code generator itself



Context: GeneAuto

- GeneAuto : Automatic code Generator for critical embedded systems dedicated to transportation domain.
- GeneAuto is split into elementary tools
  - Easier to specify, verify and validate
  - Several implementations can be provided



Main goals

- Reduction of industrial unit testing costs
- Qualification of GeneAuto using DO178B/ED-12B recommendations
- Pragmatic integration of formal technologies into develpment tools for safety critical systems

Development Process

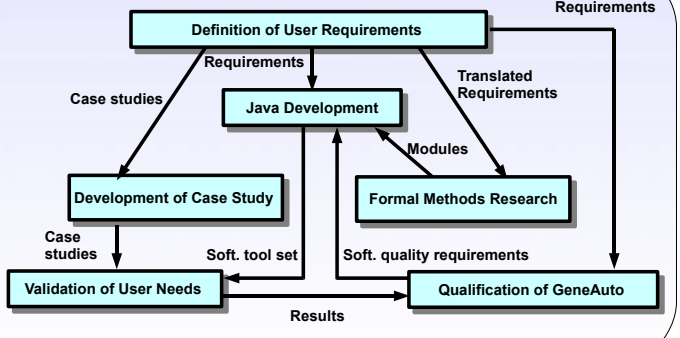
Choices

- Specification, verification and validation using the Coq proof assistant
- Integration of the formal elementary tools to the GeneAuto tool chain
- Qualification of the development process of GeneAuto containing classical Java and formal elementary tools (Coq/OCaml)

For each elementary tool

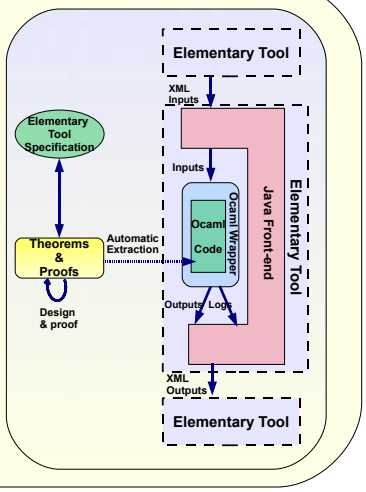
- Translation of user/tool requirements from natural to formal language (complex task, human proof reading)
- Formal specification of the tool requirements and design
- Formal verification of specified properties (correctness of Block Sequencer,Typer, etc.)

GeneAuto workflow



Integration of Formal Methods

- Each elementary tool
  - is developed and verified in Coq;
  - is verified and extracted in OCaml: extracted code preserves the properties proved in Coq.
- Java front-end
  - reads input XML models;
  - executes extracted Ocaml Wrapper;
  - writes output XML models.
- Ocaml Wrapper
  - reads input models;
  - executes the extracted OCaml code (sequencer, typer, etc.);
  - writes the output result (execution order, types, etc.).



Qualification Concerns

Example of translation of requirements

- From natural language

```
F6.1 Sort blocks based on data-flow constraints.
F6.3 Sort blocks with partial ordering according to priority from the input
model.
F6.4 Sort blocks that are still partially ordered according to their graphical
position in the input model.
```

- To Coq language

```
Definition correct_execution_order_dataflow
(m : ModelType) (s : SequencedModelType) : Prop :=
  forall (d : nat), (0 < d) /\ (d <= m.signalsNumber) ->
    ((s.signalKind = DataSignal) ->
    (~ (isControlled s.src m)) ->
    (~ (isControlled s.dst m)) ->
    (s.dst.blockKind = CombinatorialBlock) ->
    (s.src.blockKind = CombinatorialBlock) ->
    ((s.sequencedBlocks d.src) = (Position _)) ->
    ((s.sequencedBlocks d.dst) = (Position _)) ->
      let (Position posSrc) = (s.sequencedBlocks d.src) in
      let (Position posDst) = (s.sequencedBlocks d.dst) in
        posSrc < posDst.
```

Qualification process

- Qualification of the development process of Java components
  - Detailed documented development process using DO178B/ED-12B
  - Validation process done through testing and cross-reading
- Qualification of the formal elementary tools
  - Coq proof checker partially verified
  - Coq extractor generates Ocaml code structurally similar to Coq specification
  - Removal of unit & integration test phase from the formally developed elementary tools in DO178B/ED-12B

Main Results

- Mixing classical and formal development
- Development of correct-by-construction components
  - ~4500 lines of Coq code and more than 130 proved theorems for the Block Sequencer
- Block Sequencer case study successfully integrated into GeneAuto
- Application to Real-size systems from transportation domains

Case Study	Satellite Orbit Control	"Knock" reduction Software	Airpline Flight Control System	Satellite Agile Control System	Sensor Networks
Model blocks	1085	5793	2800	1931	1108
Depth	8	9	7	6	7

- Runtime cost is comparable with similar tools (eg. Mathworks RTW)
- Qualification of the development process
  - Classical development
  - Formal components (Block Sequencer case study)

Integration of Agent and Component approaches by Service Oriented vision using Model Driven Engineering

Service Oriented Architecture

(SOA)

Composants Based Software Engineering

(CBSE)

Multi Agents System

(MAS)

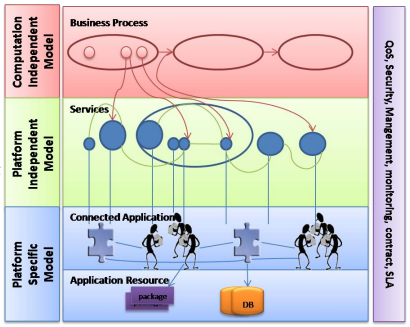
Model Driven Engineering

(MDE)

LIUPPA

Context

- Component based Software Engineering interests
  - **Reutilisation, Composition,...**
- System Multi Agent interests
  - **Autonomy, sociability, flexibility,...**
- Service Oriented Approach
  - **Loosely coupled, Autonomous Service, Location Transparency**
- Service Oriented Computing calls for more abstract software development technologies.



Objective

- Integration of these two approaches (CBSE, SMA) to benefit from their interests in applications development.

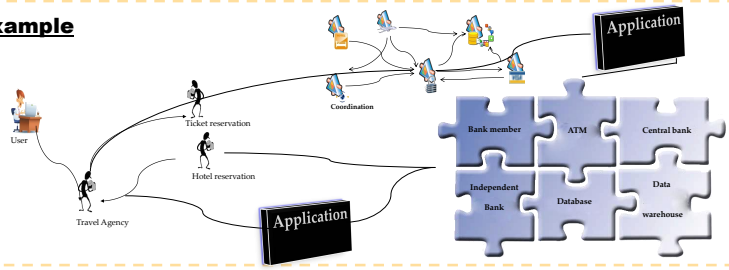
Method

- Service Oriented Architecture vision via a MDE approach.

Solution

- Component Agent Service Oriented Model "CASOM" which is mixed from Components and MAS models.

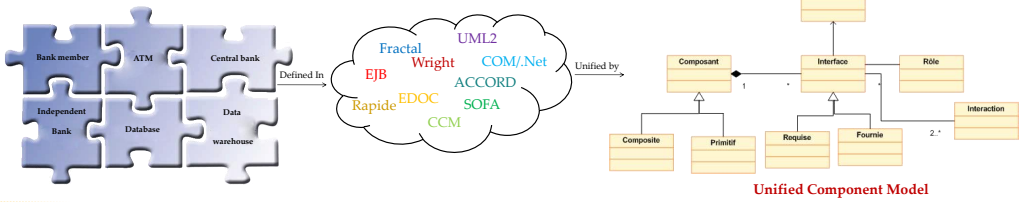
Example



Different applications based on different approaches exchange services as providers or consumers.

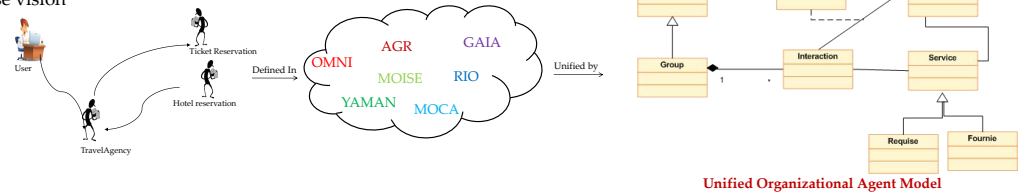
Our Approach

• Definition of a Unified Component model with service and interaction vision



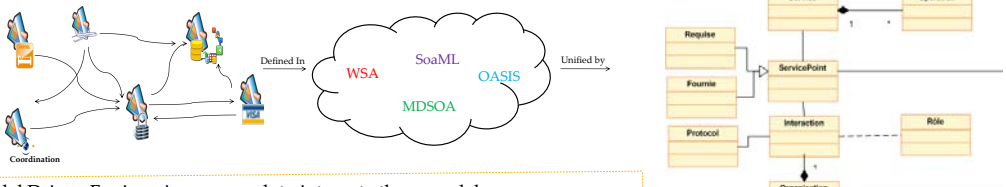
Unified Component Model

• Definition of a Unified Multi Agent System model with service, composition and reuse vision



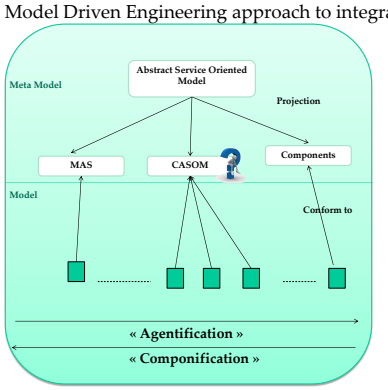
Unified Organizational Agent Model

• Definition of a Unified Service model with interaction vision



Unified Service Model

• Model Driven Engineering approach to integrate these models.



- Projection of the abstract model towards any of Component, agent or CASOM models.
- Easy transformations among all of component, MAS, CASOM models.
- Ability of mixing Agents and Components according to application requirements, either by:
  - Agentification, adds agents features to existing components.
  - Compontification, adds components features to existing agents.

UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

Nour Alhouda ABOUD
Doctorante
Nour-alhouda.Aboud@univ-pau.fr

Philippe ANIORTÉ
Directeur de thèse
Philippe.Aniorte@iutbayonne.univ-pau.fr

Eric CARIOU
Co-encadrant
Eric.Cariou@univ-pau.fr

Eric GOUARDERES
Co-encadrant
Eric.Gouarderes@univ-pau.fr

Extending PlusCal: A Language for Describing Concurrent and Distributed Algorithms

Sabina Akhtar, Stephan Merz, Martin Quinson

LORIA – INRIA Nancy Grand Est & Nancy University, Nancy, France
{Sabina.Akhtar,Stephan.Merz,Martin.Quinson}@loria.fr

1 Context

The design of correct concurrent and distributed algorithms is notoriously difficult, and they are prone to errors such as deadlocks and race conditions. In fact, it is often non-trivial to precisely state the assumptions under which the algorithms are expected to operate and the exact properties they are expected to guarantee. Model checking [1] is one of the most successful formal techniques in this area: properties (usually expressed in temporal logic) can be verified automatically over instances of algorithms whose state space is finite (or finitely representable). In case the property does not hold, the model checker produces a counter-example, whose inspection assists in finding the cause of the error.

Model checkers expect that algorithms be expressed in a formal modeling language. For example, TLC [6] accepts models written in TLA$^+$ [4], a specification language based on mathematical set theory and the Temporal Logic of Actions. Such languages have a very different flavor from the (pseudo-)programming languages that algorithm designers typically use to express algorithms, and that mismatch creates a barrier for the routine use of model checking. Recognizing this problem, Lamport introduced PLUSCAL [5], a high-level language for describing concurrent and distributed algorithms, from which TLA$^+$ models are generated and then analyzed using TLC.

Unfortunately, PLUSCAL suffers from a number of limitations that restricts its usefulness as an abstraction layer for the underlying TLA$^+$ formalism. Most importantly, assumptions (such as fairness) and correctness properties cannot be expressed in PLUSCAL but have to be stated in terms of the TLA$^+$ model that results from the translation. Hence, the user not only has to be knowledgeable in TLA$^+$, but also has to understand the translation of PLUSCAL to TLA$^+$. In turn, the translation has to be relatively straightforward so that its result remains human-readable, and this motivates several restrictions of PLUSCAL. For example, PLUSCAL algorithms are restricted to only top-level processes, whereas many distributed algorithms are naturally expressed in terms of nodes communicating by message passing, each of which contains several threads that access shared memory. PLUSCAL does not enforce variable scoping, which may result in uncaught modeling errors such as nodes inadvertently accessing variables of distant nodes.

One of the fundamental concepts of parallelism is to indicate the unit of atomicity. PLUSCAL employs a simple, but powerful idea: statements can be labeled, and all code appearing between two labels is assumed to be executed atomically. However, labels are also introduced for the purposes of compilation, for example when translating loops or procedure calls, and this leads to rather complex rules governing where labels can and must be placed.

We propose a variant of PLUSCAL that preserves the basic objectives of the language while overcoming the restrictions mentioned above. In the remainder of this contribution we sketch the main design decisions for our language extensions.

2 Our Contributions

Our language, just like PLUSCAL, is based on TLA$^+$. In particular, the objects and data structures that algorithms manipulate are represented by TLA$^+$ expressions. We found that working with set-theoretical abstractions helps clarifying the fundamental concepts underlying concurrent and distributed algorithms. The resulting expressive power is incomparably higher than that afforded by conventional modeling languages such as PROMELA [2]. Unlike PLUSCAL, representations of algorithms using our system are en-

tirely self-contained.[1] Fairness hypotheses are indicated by appropriate annotations of processes or labeled statements, and the compiler generates corresponding formulas in the TLA$^+$ specification. Also, our system allows models to contain an *instance section* that defines the finite instance to be verified by TLC, and from which a configuration file is generated. Third, correctness properties can be stated at the level of individual processes (verified for each instance of the process) and for the entire algorithm.

Processes can be arbitrarily nested in our language, and the compiler enforces static scoping of variable or procedure declarations, as well as of TLA$^+$ operator definitions appearing in processes. (Global variables and procedures are accessible throughout an algorithm.) Hierarchical processes more naturally reflect the structures of actual algorithms; static scoping helps limiting modeling errors and opens the way to optimizations during verification such as partial-order reduction. These changes complicate the translation and make the resulting TLA$^+$ model harder to read and understand. Because models using our language extensions are self-contained, we do not expect users to read or edit the generated TLA$^+$ specification.

Our language extension retains the basic idea of indicating atomicity via labels. Whenever additional labels are required for the purposes of translation, the compiler adds them and informs the user. We have added an explicit *atomic* construct that makes a group of statements execute without interruption by other processes, even in the presence of intervening labels.

We introduced a number of relatively minor differences to PLUSCAL. For example, we allow several assignments to the same variable to appear within a group of statements without an intervening label. Also, we have added a *for* statement for iterating over the elements of a finite set in an unspecified order. Our language is not entirely backward-compatible with PLUSCAL: for example, accesses to variables that are not in static scope are rejected. We have also replaced the general, but unscoped macro facility of PLUSCAL by a more restrictive concept of locally scoped *definitions*. Nevertheless, we have found that existing PLUSCAL algorithms can be adapted with minor effort.

We have implemented a compiler that translates algorithms written in our language to TLA$^+$ and have successfully encoded and verified a number of representative algorithms. We found it easy to describe algorithms and specify their properties using our language extensions. The fact that fairness annotations appear within the algorithms gives particular expressive power to designers and lets them explore the liveness properties guaranteed by algorithms.

In case a property fails to hold, the counter-example is currently presented by TLC in terms of the TLA$^+$ model that results from translation. Understanding counter-examples could already be difficult for the original PLUSCAL, it is even more so for our extensions, because the compilation is more involved. We plan to translate counter-examples back to the PLUSCAL level and to present them within the GUI that is currently being developed for the TLA$^+$ tools. We also interact with the authors of PLUSCAL to integrate our changes into their public distribution. Another important problem we are working on is addressing the problem of state space explosion based on partial order reduction techniques. Static scoping of variables local to (nested) processes is a prerequisite here because it helps us to detect locality and independence of TLA$^+$ actions that represent a block of PLUSCAL statements.

References

1. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, Mass., 1999.
2. G. J. Holzmann. *The SPIN Model Checker*. Addison-Wesley, 2003.
3. L. Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.
4. L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.
5. L. Lamport. Checking a multithreaded algorithm with +CAL. In S. Dolev, editor, *20th Intl. Symp. Distributed Computing (DISC 2006)*, volume 4167 of *LNCS*, pages 151–163, Stockholm, Sweden, 2006. Springer.
6. Y. Yu, P. Manolios, and L. Lamport. Model checking TLA+ specifications. In L. Pierre and T. Kropf, editors, *Correct Hardware Design and Verification Methods (CHARME'99)*, volume 1703 of *LNCS*, pages 54–66, Bad Herrenalb, Germany, 1999. Springer.

[1] A complete example of a model using our language extensions appears in the appendix.

Example: Representation of the FastMutex algorithm

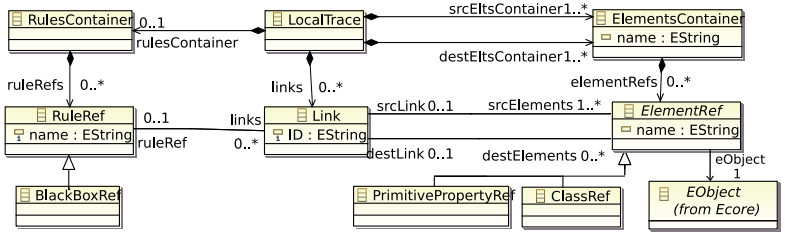
The following is a model of Lamport's fast mutual exclusion algorithm [3] in our extension of PLUSCAL. Note in particular the use of fairness annotations for processes and the fact that properties are given within the model.

```
1  algorithm FastMutex
2  extends Naturals                    (* Standard module of TLA+ *)
3  constants
4     N   (* Number of processes *)
5  variables
6        x = 0,
7        y = 0,
8        b = [id ∈Peer ↦ FALSE]
9  fair process Peer[N]
10    begin
11       ncs:  loop
12                skip ;              (* Non-critical Section *)
13       start:    b[self] := TRUE ;
14       l1:       x := self ;
15       l2:       if y # 0 then
16       l3:          b[self] := FALSE ;
17       l4:          when y = 0;
18                    goto start;
19                end if;
20       l5:       y := self ;
21       l6:       if x # self then
22       l7:          b[self] := FALSE ;
23       l8:          for j ∈ 1 .. N
24                        when ¬b[j];
25                    end for;
26       l9:          if y ≠ self then
27       l10:             when y = 0; goto start;
28                    end if;
29                end if;
30       cs:       skip;               (* Critical Section *)
31       l11:      y := 0 ;
32       l12:      b[self] := FALSE
33             end loop;
34    end process
35 end algorithm
36 (* Assert: No two processes simultaneously execute the critical section *)
37 invariant ∀ i, k ∈ 1..N: i ≠ k ⇒ ¬(Peer[i]@cs ∧ Peer[k]@cs)
38 (* Liveness: Each request for the lock is eventually granted *)
39 temporal ∀ p ∈ Peer: □◊¬b[p]
40 (* Instantiating the model for 3 processes. *)
41 constants
42       N = 3
```

Traceability Mechanism in Transformations Chains Dedicated to Model and Transformation Debugging

Vincent Aranega
Université Lille 1, LIFL
vincent.aranega@lifl.Fr

Traceability and Transformations Chains



Local Trace Metamodel

The local trace metamodel catches information relative to the transformation execution, as the manipulated objects/properties and the applied rules. The metamodel is independent from the used transformation language.

Transformation Traceability

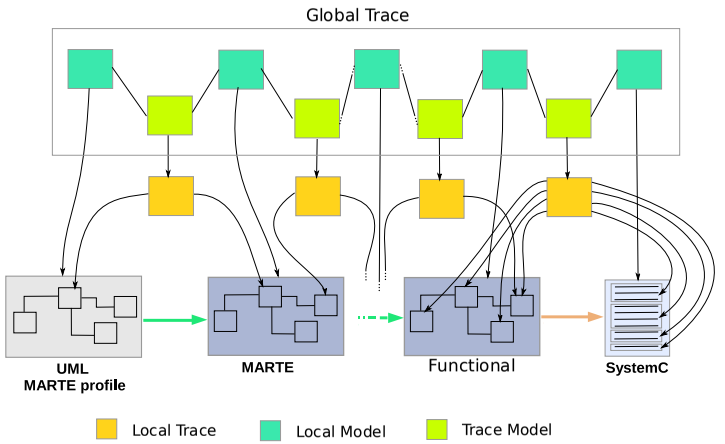
The transformation traceability defines links that keep the relationship between the input elements manipulated by the transformation and the produced output ones.

Model to Model Traceability

For a simple model to model transformation, the local trace metamodel provides these kinds of relationship as presented on the left figure.

Transformations Chain Traceability

A transformation chain occurs when transformation outputs are directly consumed by another transformation. To manage the links between the produced and consumed models, a global trace is added. The global trace allows the navigation between models and traces in the whole transformation chain.

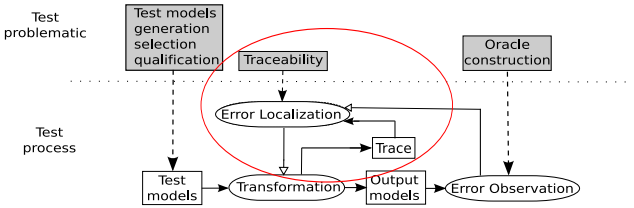


Example of Traces on Transformation Chain

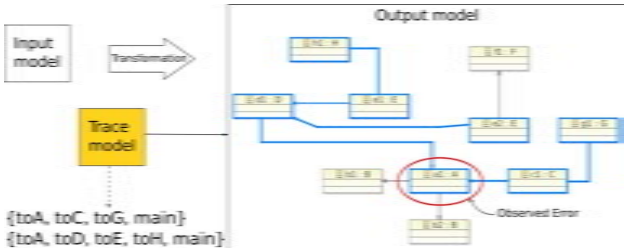
Error Localization

Traceability Goal

During the transformation writing process, the transformation is often tested and debugged. However, once a fault is observed in the output model, the debugging part is often manually performed and the error in the transformation rules can be difficult to identify.

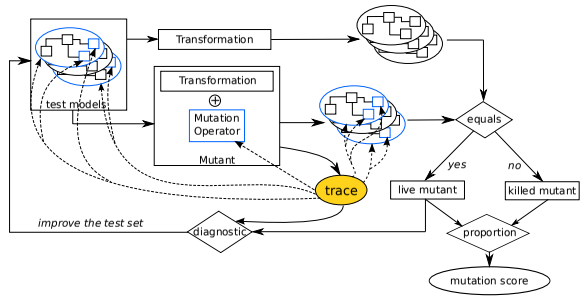


The provided algorithm uses the trace, the output model and the element on which the error had been observed. It computes sets of rules that handled the output model faulty part. The sets represent the rules calls sequence. The developer can now easily reduces its search field and identify the erroneous rule.

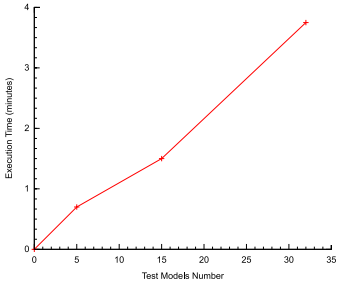


Help for the Mutation Analysis Process

Traceability Goal



When a transformation is tested, a set of test models is used. To check the test models capacity to emphasize the transformation errors, mutation analysis dedicated to model transformation is used.

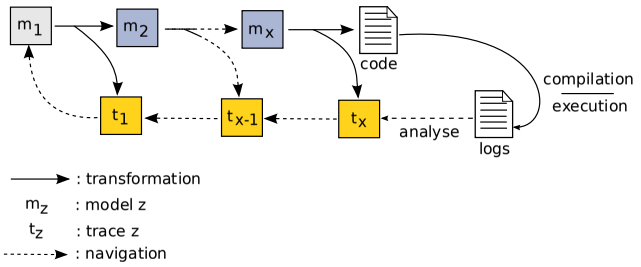


If the test model set is not enough to highlight all the errors voluntary introduced in the transformation, it has to be improved. This activity remains manual and is often a though job. The trace is used here as an information collector. Pertinent models to modify are identified based on trace information. A little benchmark has been performed in order to measure the required time to give the information to the tester. It shows that the answer time is only dependent of the test model number

Model Debugging and Profiling

Traceability Goal

A trustworthy transformation chain doesn't implies that the input models are designed with the wanted meaning. Although an erroneous or incoherent model can generate compilable and executable code, the behavior observed during the execution can be not wished. To identify the model faulty part, we use the trace produced as a path from the specific faulty sketch of code to its referred input elements.



The program execution produced several logs that are analyzed. When a piece of information is detected, the responsible elements in the input models are identified, and the information is reported on these elements. The user can easily debugg its own models interactively. The information added are not always relative to an identified fault but can be also information upon specific point (as time measurement or power consumption).

PERSO

project JC07_186508
grant ANR-07-JCJC-0155-01
Nov. 2007 - Oct. 2010

ANR
National Agency for Research
Young Researchers Program

PERvasive Service cOmposition

Service Composition for Pervasive Systems

pervasive computing

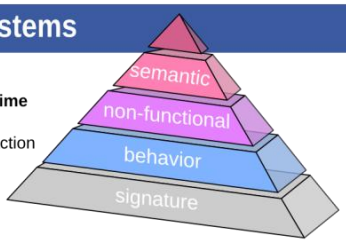
- access to information and computing **anywhere and at anytime**
- **user-centric vision**
  end-user programming - intuitive, pleasant and natural interaction

service-oriented architectures and computing

- **services**
  reusable, loosely coupled, and distributed entities
- **service composition**
  fosters the creation of value-added composite applications

Web services and interface levels

- **widely accepted service-oriented architecture**
  standardized languages
- **four interface description levels**
  1- signature (operations)
  2- behavior (protocols)
  3- non-functional (quality of service, transactional properties)
  4- semantic (capabilities)



Objectives / Research Issues

multi-level service composition

- support **different description levels**
- **solve mismatch** between requirements and services
  abstraction level mismatch, protocol mismatch, etc.
- **satisfy** non-functional and semantic **user requirements**

highly dynamic pervasive environment

- **run time** self-adaptive service composition
- **distributed** service composition (choreography)
- execution on **low-resource** devices (PDA, smart phones)

Innovative Character

combine Model Driven Engineering and formal methods

- models increase approach **genericity**
- formality enables the development of **rigorous algorithms**

bringing "exotic" techniques into mainstream software engineering

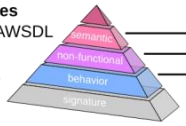
- combinatorial optimization techniques support **expressive non-functional requirements**
- networking analysis techniques provide **compositionality** at the **quality of service** level

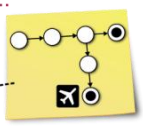
Principles of Model-Based Service Composition

Step 1 - Model Generation

formal models are generated from service description languages ...

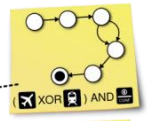
- **languages**
  OWL+SAWSDL
  WSQDL
  (A)BPEL
  WSDL
- **models**
  semantic annotations
  QoS attributes (price & duration)
  transition systems (STS)

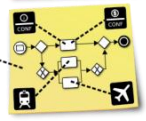




... and from end-user requirements

- **languages**
  ABPEL (operation-based)
  YAWL, BPMN (capability-based)
- **models**
  transition systems (STS)
  workflows

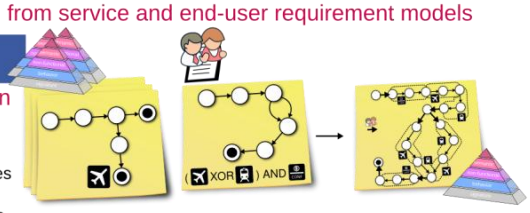




+ **semantic annotations**
+ **transactional properties**
  atomic (e.g., (AND) OR NONE) / compensatable
+ **QoS satisfaction properties**
  MAX[(0.4)MIN(price), (0.6)MIN(duration)]

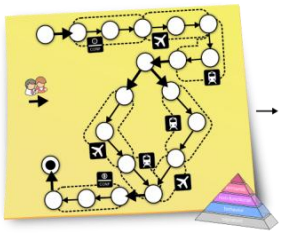
Step 2 - Composition / Adaptation

composition / adaptor models are generated from service and end-user requirement models



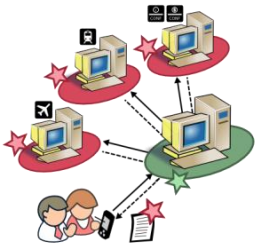
Step 3 - Model Implementation

composition / adaptor implementations are generated from composition / adaptor models

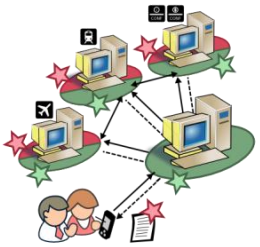


```
<?xml version="1.0" encoding="UTF-8"?>
<process ...>
  <import ... />
  <partnerLinks>
    <partnerLink name="user" .../>
    <partnerLink name="flight" .../> ...
  </partnerLinks>
  <variables> ... </variables>
  <correlationSets>
    <correlationSet name="USER_CS" .../>
  </correlationSets>
  <sequence>
    <receive createInstance="yes"
             partnerLink="user"
             operation="getInfo" ....>
      <correlations>
        <correlation set="USER_CS" .../>
      </correlations>
    </receive>
  ... </sequence>
... </process>
```

BPEL



centralized composition



distributed composition

generated adaptation and composition code

run time requirement/service change
resulting re-adaptation and re-composition

Early Results

operation-based composition

- **contract-based adaptation**
  builds adaptors solving mismatch between requirements and services
- **distributed composition**
  computes distributed orchestrators using operation semantic annotations

capability-based composition

- **non-functional service discovery**
  ensures non-functional preferences and transactional properties
- **adaptive service planning**
  generates all service invocation orderings suiting user requirements

implementation / assessment

- **lightweight orchestration**
  language and execution engine based on Java and WS-BPEL
- **composition performance analysis**
  computes mean response time and QoS reliability for orchestrations

Selected Project References

[B1] C. Canal, P. Poizat and G. Salaün. Model-based Adaptation of Behavioural Mismatching Components. IEEE Transactions on Software Engineering, 34(4):546-563. 2008

[B2] R. Mateescu, P. Poizat, and G. Salaün. Adaptation of Service Protocols using Process Algebra and On-the-Fly Reduction Techniques. ICSOC'08, LNCS 5364:84-99.

[NF1] J. El Haddad, M. Manouvrier, G. Ramirez, and M. Rukoz. QoS-driven Selection of Web Services for Transactional Composition. ICWS'08, pp 653-660.

[NF2] J. Ben Othman, L. Mokdad, M. Ould Cheikh, and M. Sene, Performance analysis of composite web services using Stochastic Automata Networks over IP network. ISCC'09, pp 92-97

[S1] T. Melliti, P. Poizat and S. Ben Mokhtar. Distributed Behavioural Adaptation for the Automatic Composition of Semantic Services. FASE'2008, LNCS 4961:146-162.

[S2] S. Beauche, and P. Poizat. Automated Service Composition with Adaptive Planning. ICSOC'08, LNCS 5364:530-537.

Envisioned Contributions

service-oriented computing

language extensions, models, and algorithms for:

- **adaptive** composition
- **distributed** composition
- **run time** re-composition

proof of concept

- case-study and prototypes

software industry

- potential **benefit** to service engineering and cloud computing

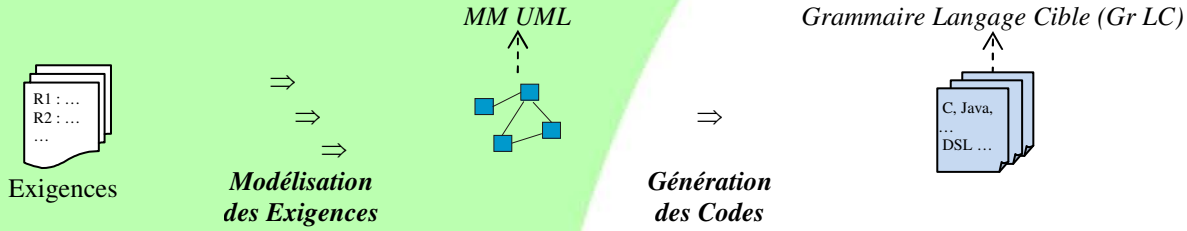
http://anr-perso.ibisc.univ-evry.fr/
contact: pascal.poizat@lri.fr

INRIA ARLES project-team INRIA
ibisc IBISC Univ. Evry Val d'Essonne
LAMSADE Univ. Paris Dauphine

L'Activité de Génération de Codes Dirigée par les Modèles

Thanh-Thanh LE THI

Les deux activités de base d'un processus IDM



- La ***Modélisation des Exigences***, basée sur des démarches formelles, est l'activité essentielle des processus IDM
- Assuré de la qualité des modèles, l'***Activité de Génération des codes*** est réduite à une simple création des codes

→ **Objectif de l'étude** : ***Réaliser l'Activité de Génération de codes dans le cadre d'un processus IDM***

I) Modélisation en UML/OCL d'un langage et de ses propriétés : MM L *[J. Malenfant, PA. Muller, …]*

- Constructions syntaxiques de la grammaire : Diagramme de classes, Invariants OCL
- Propriétés de typage, comportementales, axiomatiques : OCL, Langage d'actions (Kermeta, USE, …)

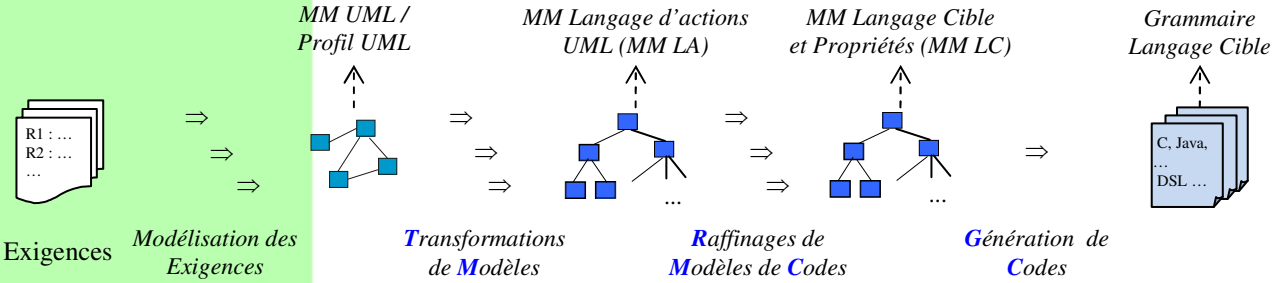
→ Exécution du modèle d'un programme, Intégration de propriétés (Métier, Règles de programmation, …), …

II) Propriétés sur les modèles liées à un composant de Transformation IDM (Prédicats de Conformité

[M. Blay, …]

- Pattern du Composant IDM : $C\_IDM = < MM\ S , PC_{MM\ S} , Tr_{MM\ S \rightarrow MM\ C} , PC_{MM\ S \rightarrow MM\ C} , MM\ C, PC_{MM\ C} >$

III) Processus incrémental dédié à l'activité de Génération des Codes *[G. de Fombelle, Y. Bertot, …]*



- Processus IDM définissant l'activité de Génération de codes sous la forme de compositions de Patterns :
  - $TM = < MM\ UML , PC_{MM\ UML} , Tr_{MM\ UML \rightarrow MM\ LA} , PC_{MM\ UML \rightarrow MM\ LA} , MM\ LA , PC_{MM\ LA} >$
  - $RMC = < MM\ LA , PC_{MM\ LA} , Tr_{MM\ LA \rightarrow MM\ LC} , PC_{MM\ LA \rightarrow MM\ LC} , MM\ LC , PC_{MM\ LC} >$
  - $GC = < MM\ LC , PC_{MM\ LC} , Tr_{MM\ LC \rightarrow Gr\ LC} >$

Bilan et Perspectives

- Homogénéité du formalisme tout au long du processus (Modèles UML, OCL, langage d'actions)
- Visibilité des Modèles pour les Analystes/Concepteurs, à chaque étape du processus
- Projet ANR DOMINO'2006 : Lot 4 (Abstraction et Modélisation de DSL et de leurs propriétés)

→ Ré-utilisation des technologies des traducteurs pour vérifier la préservation de la sémantique initiale des Modèles et de la sémantique des Patterns : GrammarWare ←→ ModelWare

Thanh-Thanh Le Thi - IRIT / UPS Toulouse (lethi@irit.fr 05 61 55 86 32)

WebMov : Modélisation,Test et Validation de Services Web

WebMoV
http://webmov.lri.fr/

AGENCE NATIONALE DE LA RECHERCHE ANR
Projet labellisé le 21 novembre 2007

Contexte

- **Vision SOA**
  - développement moderne de logiciels = réutiliser + composer
  - des ressources/données partagées de manière flexible, interopérable et standardisée ⇒ modulaire et unités faiblement couplées : service
- **Services Web**

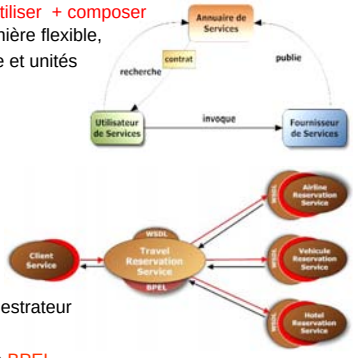
respectent des standards, principalement:

- description des données : XML Schema
- description des messages : SOAP
- interface des services: WSDL

au travers d'un protocole Internet (HTTP, SMTP, ...)

Composition centralisée de services par un orchestrateur *vs* composition distribuée (chorégraphie).

Webmov s'intéresse à l'orchestration décrite en BPEL



Exemple d'un service composé : **Travel Reservation System (TRS)**

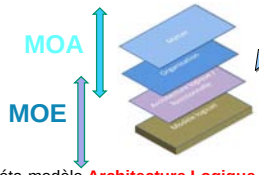
Objectifs

L'objectif principal de WebMov est de contribuer à la **conception**, la **composition** et la **validation** de **services Web** à travers une vue abstraite de haut niveau et une vision **SOA** (architecture orientée services) basée sur une **architecture logique**. Ce projet doit permettre :

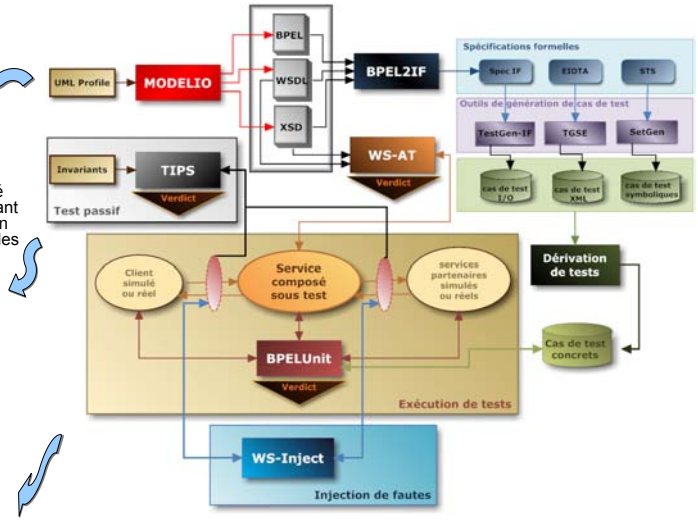
1. de définir une méthodologie de conception de services Web composés basée sur une architecture logique
2. de concevoir des méthodes automatisées de dérivation et d'exécution des cas de test pour les services Web. L'orchestration ainsi que ses partenaires sont testés
3. de concevoir des méthodes de monitorage basées sur des techniques formelles de test passif
4. de développer une chaîne logicielle intégrée qui couvre les différentes étapes du cycle de vie des services Web (modélisation, déploiement, génération de test et supervision)
5. de mener des expérimentations sur des exemples de taille réelle afin d'évaluer les méthodes et outils proposés dans le cadre du projet

Architecture logicielle

Architecture Logique



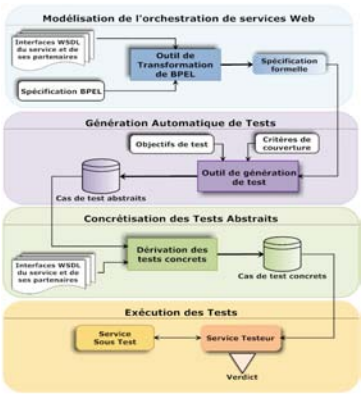
Le méta-modèle **Architecture Logique** a été conçu dans le but d'offrir un support permettant de fixer les grandes décisions de structuration du système d'information indépendamment des solutions techniques.



- **Test passif basé sur les invariants**
  - les invariants portent sur les échanges de messages entre le service composé sous test et son environnement (ses partenaires). Ces invariants doivent être satisfaits pour émettre un verdict positif
  - Collection des traces à l'aide d'un sniffer XML/SOAP. Prise en compte du temps et des données
- **Test de robustesse par injection de fautes :**
  - perturbation du fonctionnement des services Web (environnement hostile)
  - injection de deux types de fautes au niveau des messages SOAP
    - Fautes d'interfaces
    - Fautes de communication

- Tous les outils développés dans le cadre de **WebMov** sont à la fois autonomes et complémentaires
- Ces outils peuvent former plusieurs chaînes logicielles couvrant le cycle de vie d'un service Web (modélisation, déploiement, génération et exécution de tests, supervision).
- Cette chaîne d'outils sera intégrée dans une plate-forme (e.g. Eclipse).

- Génération de tests unitaires, d'intégration et de robustesse à partir de modèles formels
- Génération de tests selon une architecture de test et des niveaux de contrôle et d'observation. Deux types d'approches sont définis : approche boîte noire, approche boîte grise
- Les différentes techniques proposées sont complémentaires (temps, couverture de BPEL,...). Elles couvrent les étapes de la modélisation à l'exécution des tests



Innovations

- Prise en compte de plusieurs niveaux de description (interface, comportemental) dans les méthodologies développées
- Définition de nouveaux modèles formels pour prendre en compte les spécificités des services Web composés
- Combinaison de différentes techniques de test : actif/passif, interface/comportement, boîte noire/boîte grise pour les services Web
- Technique d'injection de fautes pour le test de robustesse
- Approche de bout en bout traitant les différentes étapes du cycle de vie d'un service Web
- Expérimentations sur des cas d'études réels : tous les outils ont été appliqués au cas d'étude TRS (service de réservation de voyage)

Partenariat

- **Académiques** :
  - **LRI** - Université Paris-Sud XI: FORTESSE
  - **LaBRI** - Université Bordeaux I : Langages, Systèmes et Réseaux
  - **Telecom SudParis** : équipe Validation des protocoles et des services
  - **Unicamp**, Brésil : Institute of Computing

Equipe associée :
- **LIMOS** – Universités Clermont Ferrand 1 et 2 : Architectures et Réseaux Mobiles

- **Industriels** :
  - **SOFTEAM** : Modélisation d'architecture SOA
  - **Montimage** : Outil de monitorage de services Web et cas d'études industriels

Retombées

- Méthodes formelles et semi-formelles applicables par les ingénieurs chargés du développement de services Web et les testeurs chargés de valider leur déploiement
- Outils pour la gestion des services Web dans ses différentes phases de développement
- Intégration des résultats d'architecture logique et de génération de BPEL à partir de BPMN à l'outil Modelio[1] de Softeam
- Participation active à la standardisation à l'OMG de SoaML[2]

[1] http://www.modeliosoft.com/modules/modelio-ea-bpm-modeler.html
[2] http://www.soaml.org

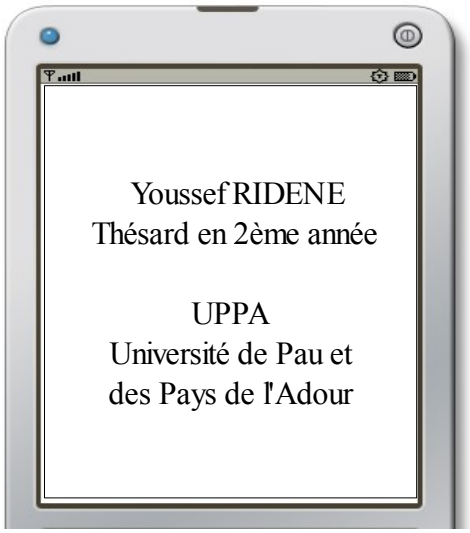
12 publications et 6 thèses en cours dont une est déjà soutenue

Sélection de publications :

[1] L. Bentakouk, P. Poizat, and F. Zaidi. A Formal Framework for Service Orchestration Testing based on Symbolic Transition Systems. In TESTCOM 2009,16-32, Springer, 2009.
[2] F. Bessayah, A. Cavalli and E. Martins. A Formal Approach for Specification and Verification of Fault Injection Process. In ACM Proceedings of ICIS 2009, 883-890.
[3] D. Cao, P. Félix, R. Castanet and I. Berrada. Testing Web Services Composition using TGSE tool. In WS-Testing 2009.
[4] S. Salva and I. Rabhi. Automatic web service robustness testing from WSDL descriptions. In 12th European Workshop on Dependable Computing, EWDC 2009.
[5] M. Lallali, F. Zaidi, A. Cavalli and I. Hwang. Automatic Timed Test Case Generation for Web Services composition. In the 6th IEEE European Conference on Web Services (ECOWS'08).

Contact : Fatiha ZAIDI, Université Paris-Sud XI, LRI UMR 8623 – Fatiha.Zaidi@lri.fr (+33 1 72 92 59 62)

TELECOM SudParis · UNICAMP · montimage · SOFTEAM Think Object



Youssef RIDENE
Thésard en 2ème année

UPPA
Université de Pau et
des Pays de l'Adour

Définition d'un langage de modélisation spécifique (Domain Specific Modeling Language) au test d'applications embarquées sur téléphones mobiles

LIUPPA
Laboratoire d'Informatique
de l'Université de Pau et
des Pays de l'Adour

Problématique

- Comment définir, exécuter et adapter un **scénario de test** pour une application embarquée sur téléphone mobile?
- Comment **automatiser** les tests d'applications mobiles dans leurs environnements de déploiement?
- Comment prendre en compte les **différences** entre téléphones portables?

En pratique

Test 1: Est-ce que l'application s'exécute correctement?
Test 2: Est-ce que l'image est bien centrée?
Test 3: Est-ce que la musique est jouée normalement?
…
Test 48: Y a t-il des fautes d'orthographe?

Je n'ai plus que les 48 tests à refaire sur 350 autres mobiles!!!

Objectif

Définir et mettre en place un langage de modélisation pour définir les scénarios de test en prenant en compte les différences entre les mobiles.

Youssef Ridene Doctorant	Franck Barbier Directeur de thèse	Nadine Couture Co-directrice de thèse	Nicolas Belloir Co-directeur de thèse
youssef.ridene@univ-pau.fr	franck.barbier@univ-pau.fr	n.couture@estia.fr	nicolas.belloir@univ-pau.fr

Journées Nationales du GDR Génie de la Programmation et du Logiciel

Taming Orchestration Design Complexity through the ADORE Framework

Demonstration submitted to the "journées 2010 du GDR GPL"

Sébastien Mosser and Mireille Blay–Fornarino

University of Nice Sophia – Antipolis
Cnrs, I3s Laboratory, Modalis team
Sophia Antipolis, France
{mosser,blay}@polytech.unice.fr

Abstract. The Service Oriented Architecture (Soa) paradigm supports the assembly of atomic services to create applications that implement complex business processes. Assembly is accomplished by service orchestrations defined by Soa architects. The Adore framework allows Soa architects to model complex orchestrations of services by composing models of smaller orchestrations involving subsets of services. The smaller orchestrations are called orchestration *fragments* and encapsulates new concerns. Adore is then used to weave fragments into existing application models. This demonstration illustrates how the Adore framework can be used to model a Soa using fragments composition. We illustrate it using several implemented case studies.

Work Context: SOA & Business Processes Design

An application in the Service Oriented Architecture (Soa, [1]) paradigm is an assembly of services that implements a business process. Soa applications can be defined as orchestrations of services [2]. A Soa application is typically defined by business specialists and can involve many services that are orchestrated in a variety of ways. Furthermore, the need to extend an Soa application with new business features (to follow market trends or adapt a normalized process from a company to another one) arises often in practice. Existing tools and formalisms (*e.g.* Bpmn notation [3], Bpel industrial language [4]) are technologically–driven. They use a *design–in–the–large* approach and considerable effort can be expended when using them to develop and adapt large applications involving many services that are orchestrated in a variety of ways.

We propose a *design–in–the–small* driven framework called Adore[1] to tame the complexity of orchestration design. Experts focus on the design of small process *fragments*, and let the complexity of composing all the fragments into a final application to dedicated algorithms.

Relations to Aspect Oriented Modeling Approaches

Several approaches fill the gap between orchestrations and Aop (*e.g.*, [5], [6]). These approaches rely on the Bpel language and impose to use dedicated Bpel execution engines to interpret the aspects. Adore preaches technological independence and exposes itself as a *model* to support composition [7]. Instead of interpreting *aspectized* Bpel code, Adore aims to generate complete orchestrations of services, executable in any industrial engine.

One of the strength of Adore is to focus on the so–called *Shared Join Points* (Sjp, [8]) interactions spawn. We develop a set of rules to identify conflicting interactions between orchestration fragment at composition time. Instead of *re–ordering* the aspects to deal with conflicts around a Sjp, we use an *order–independent* composition process. When interactions are detected, the user

[1] *Activity moDel supOrting oRchestration Evolution*, http://www.adore-design.org

will enter knowledge at a fine–grained level (where coarse–grained is fragment re–ordering) to solve the conflict and then ease the interaction.

In [9], authors propose a way to weave multiple aspects in UML sequence diagrams. They propose a very precise way to identify join points and express pointcuts, but their weaving methodology relies on a sequential aspect composition, where ADORE uses an order independent composition process.

Inspired by grid–computing community, ADORE proposes an algorithm (fully described in [10]) to automatically enhance a process with *set* concerns. Considering a process p handling a scalar data d, the algorithm can automatically transform p into a process handling a set of data $d^\star \equiv \{d1, \ldots, n_n\}$.

Moreover, ADORE allows users or programs to extract metrics from its internal representation, inspired by well-known indicators like [11].

Underlying Implementation

ADORE user interface is implemented as an EMACS major mode. This mode hides in an user–friendly way the set of shell scripts used to interact with the underlying engine. The engine is implemented as a set of logical rules, using the PROLOG language. A dedicated compiler implements an automatic transformation between ADORE concrete syntax and the associated PROLOG facts used internally by the engine. As visualizing processes is important in design phase, ADORE provides a transformation from its internal facts model to a GRAPHVIZ code. It produces as a result a graphical representation of ADORE models. Visualization tools and graphs screenshots are available on the project website.

Relation to other industrial or research efforts

ADORE was partially funded (2005 – 2009) by the French Research Agency (ANR) through the FAROS consortium. The work of the FAROS consortium (including both industrial– Orange Labs, EDF & Alicante – and academic – IRISA, I3S & LIFL– partners) was to propose a model–driven methodology to build reliable SOA. The ADORE framework is one of the platforms targeted by the FAROS methodology.

References

1. MacKenzie, M., Laskey, K., McCabe, F., Brown, P., Metz, R.: Reference Model for Service Oriented Architecture 1.0. Technical Report wd-soa-rm-cd1, OASIS (February 2006)
2. Peltz, C.: Web Services Orchestration and Choreography. Computer **36**(10) (2003) 46–52
3. White, S.A.: Business Process Modeling Notation (BPMN). IBM Corp. (May 2006)
4. OASIS: Web Services Business Process Execution Language Version 2.0. Technical report, OASIS (2007)
5. Charfi, A., Mezini, M.: Aspect-Oriented Web Service Composition with AO4BPEL. In: ECOWS. Volume 3250 of LNCS., Springer (2004) 168–182
6. Courbis, C., Finkelstein, A.: Weaving Aspects into Web Service Orchestrations. In: ICWS, IEEE Computer Society (2005) 219–226
7. Mosser, S., Blay-Fornarino, M., Riveill, M.: Web Services Orchestration Evolution : A Merge Process For Behavioral Evolution. In: 2nd European Conference on Software Architecture(ECSA'08), Springer LNCS (September 2008)
8. Nagy, I., Bergmans, L., Aksit, M.: Composing Aspects at Shared Join Points. In Hirschfeld, R., Kowalczyk, R., Polze, A., Weske, M., eds.: NODe/GSEM. Volume 69 of LNI., GI (2005) 19–38
9. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving Multiple Aspects in Sequence Diagrams. Transactions on Aspect-Oriented Software Development (TAOSD) **LNCS 4620** (2007) 167–199
10. Mosser, S., Blay-Fornarino, M., Montagnat, J.: Orchestration Evolution Following Dataflow Concepts: Introducing Unanticipated Loops Inside a Legacy Workflow. In: International Conference on Internet and Web Applications and Services(ICIW), IEEE Computer Society (May 2009)
11. Laue, R., Gruhn, V.: Complexity Metrics for Business Process Models. In Abramowicz, W., Mayr, H.C., eds.: BIS. Volume 85 of LNI., GI (2006) 1–12

SELKIS

AGENCE NATIONALE DE LA RECHERCHE
ANR
Systèmes Embarqués et Grandes Infrastructures 2008

Une méthode de développement de systèmes d'information médicaux sécurisés : de l'analyse des besoins à l'implémentation.

Objectifs

Concevoir une méthode de construction de systèmes d'information sécurisés et fiables en suivant une approche MDA :

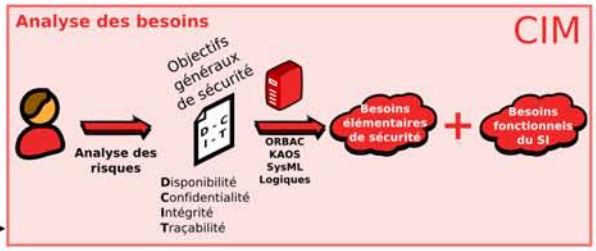
- Spécifier les besoins fonctionnels et de sécurité de manière indépendante
- Modéliser des politiques de sécurité à un haut niveau d'abstraction
- Vérifier les propriétés de sécurité et la cohérence globale du système au niveau PIM
- Implémenter les mécanismes de sécurité séparément de l'application
- Définir des liens de traçabilité entre l'implémentation et la spécification

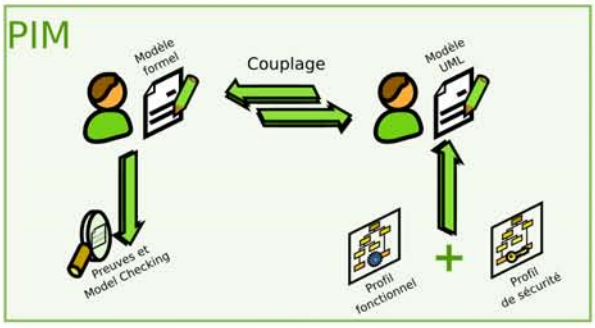
Points forts

- Prise en compte de la sécurité tout au long du développement d'un SI : de l'analyse des besoins à l'implémentation
- Utilisation conjointe de méthodes formelles et semi-formelles pour la gestion de la sécurité
- Étude de cas de taille réelle

CIM : Computation Independent Model

Définition d'un framework formel pour :
1) Spécifier les objectifs généraux de sécurité et les besoins élémentaires
2) Définir une méthodologie pour assurer la cohérence des besoins élémentaires par rapport aux objectifs de sécurité





PIM : Platform Independent Model

1) Définition d'un profil UML pour la sécurité

2) Adaptation et extension des méthodes formelles existantes (langages et outils) pour prendre en compte les aspects de sécurité

PSM : Platform Specific Model

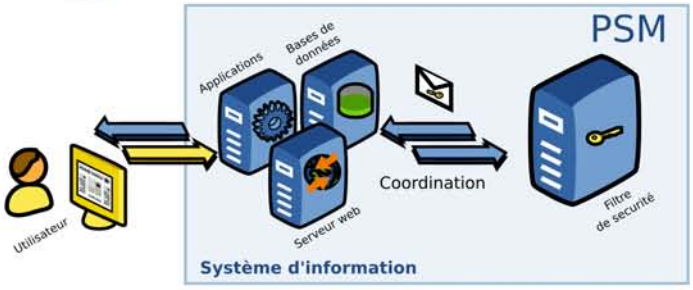
1) Architecture SOA + BD relationnelles + fichiers XML

2) 3 niveaux de granularité pour la gestion de la sécurité :
- Actions de base (requêtes SQL)
- Services atomiques (transactions SQL)
- Composition de services (workflows)

3) Définition et implémentation d'un filtre de sécurité (Policy Enforcement Manager)

Définition des métamodèles PIM et PSM et des règles de traduction

Processus de raffinement formel



Domaines d'application

1) Médecine d'urgence en haute montagne : adaptation aux contraintes notamment d'isolement

2) Partage d'informations médicales : dossiers électroniques des patients accessibles et modifiables à distance par les médecins autorisés

laleau@univ-paris12.fr
Régine Laleau (LACL - Université Paris-Est Créteil)
Site web du projet : http://lacl.univ-paris12.fr/selkis/

Ce travail a bénéficié du soutien de l'Agence Nationale de la Recherche portant la référence ANR-08-SEGI-018

Réalisation du Poster : Jérémy Milhau (LACL - IUT Sénart/Fontainebleau)

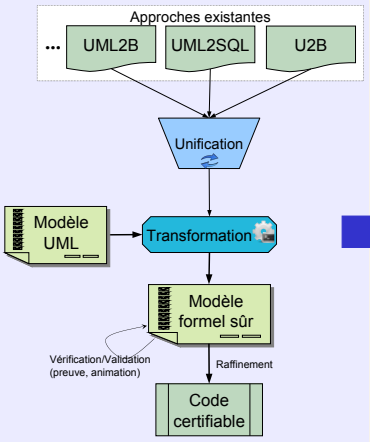
Approche orientée modèles pour une intégration efficace de B et UML

Akram Idani, Mohamed-Amine Labiadh, Yves Ledru
Laboratoire d'Informatique de Grenoble
Equipe VASCO
{Akram.Idani, Mohamed-Amine.Labiadh, Yves.Ledru}@imag.fr

Conception UML certifiée

❑ **Contexte :**

Plusieurs efforts ont été déployés afin de proposer des environnements de spécification combinant les notations B et UML. Cet intérêt est justifié par les aspects complémentaires et les apports croisés de ces techniques. L'objectif en est de préciser la sémantique d'UML, de compléter ce langage pour augmenter son pouvoir d'expression, et de traduire les spécifications semi-formelles en B pour profiter des outils de preuve et d'animation du langage B. Cependant, ces outils restent spécifiques et difficilement utilisables à grande échelle.



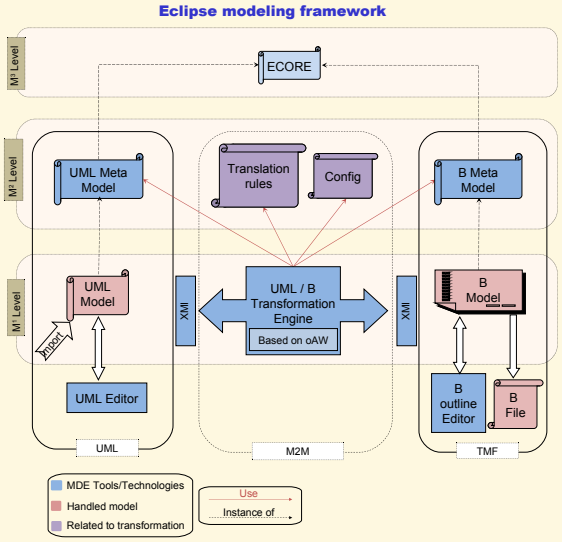
❑ **Objectifs**

1. Unifier les approches et les outils existants (UML2B, UML2SQL, U2B, ArgoUML+B) ;
2. Développer un environnement IDM configurable dédié au couplage UML/B.

Défis:

- ❑ Capitaliser le savoir-faire des ingénieurs analystes en méthodes formelles
- ❑ Maîtriser la complexité des parties formelles et semi-formelles du système
- ❑ Gérer la traçabilité et la cohérence des différents documents de spécification

Architecture UML / B mise en œuvre



Règles de transformation configurables selon le niveau d'abstraction :

Au niveau M2

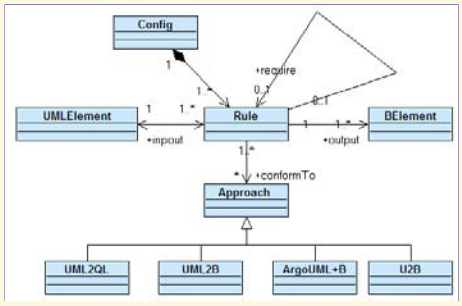
- ❑ Projections déterministes exprimées à partir des constructions des méta- modèles B et UML.
- ❑ Règles appliquées automatiquement sur toutes les constructions UML prises en compte
- ❑ Spécification adaptée pour la réutilisation des diverses approches de transformation de UML vers B

Au niveau M1

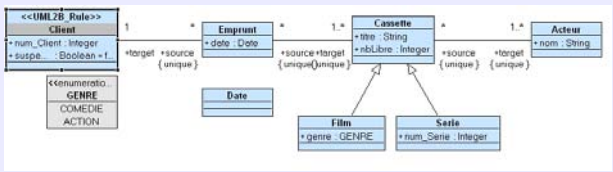
- ❑ Spécification de transformations applicables à un élément donné
- ❑ Utile pour combiner des règles de différentes approches

Approche mixte M1 / M2

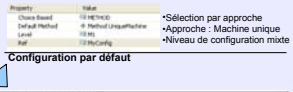
- ❑ Personnalisation du niveau M2 pour l'extension et l'adaptation d'approches existantes



Exemple: video club



Configuration de la classe « Client » : machine distincte



Configuration par défaut

```
1  MACHINE  Client
2  SETS  CLIENT
3  VARIABLES
4     Client, num_Client, suspendu
5  INVARIANT
6     Client ⊆ CLIENT
7     ∧ num_Client ∈ Client → Z
8     ∧ suspendu ∈ Client → Bool
9  END
```

```
1  MACHINE   VideoClub
2  INCLUDES   Client
3  SETS
4     EMPRUNT ; CASSETTE ; ACTEUR ; DATE;
5     GENRE = {ACTION,COMEDIE}
6  VARIABLES
7     Emprunt, Cassette, Acteur, Film, num_Serie,
8     Serie, Date, ClientEmprunt, CassetteEmprunt,
9     ActeurCassette, date, titre, nbLibre, nom, genre
10 INVARIANT
11    Emprunt ⊆ EMPRUNT
12    ∧ Cassette ⊆ CASSETTE
13    ∧ Acteur ⊆ ACTEUR
14    ∧ Film ⊆ Cassette  ∧ Serie ⊆ Cassette
15    ∧ Date ⊆ DATE
16    ∧ ClientEmprunt ∈ Emprunt → Client
17    ∧ CassetteEmprunt ∈ Emprunt ↔ Cassette
18    ∧ ActeurCassette ∈ Cassette ↔ Acteur
19    ∧ date ∈ Emprunt → Date
20    ∧ titre ∈ Cassette → String
21    ∧ nbLibre ∈ Cassette → Z
22    ∧ nom ∈ Acteur → String
23    ∧ genre ∈ Film → GENRE
24    ∧ num_Serie ∈ Serie → Z
25 END
```

Perspectives

* **Prise en charge des profils UML dans la traduction (par exemple SecureUML).**

* **Gestion de la cohérence entre les règles issues de différentes approches.**

Cette recherche est soutenue par le projet SELKIS ANR-08-SEGI-018.

Fouille de données pour la localisation de fautes dans les programmes

Peggy Cellier – GREYC – Caen
Mireille Ducassé, Sébastien Ferré, Olivier Ridoux – IRISA – Rennes

Contexte

Localisation dynamique de fautes

- **Traces d'exécution des programmes**
  - Contiennent des informations expliquant pourquoi le programme ne renvoie pas le résultat attendu (Défaillances)
  - Sont volumineuses

⇒ **Besoin d'outils d'analyse pour ces traces d'exécution**

Fouille de données (FD)

- **Traitement de grandes quantités d'information**
- **Extraction de régularités dans les données** (règles d'association, ...)
- **Calcul de clusters de données** (analyse de concepts formelle (FCA), ...)
- ...

IDÉE : Utiliser la fouille de données pour analyser les traces d'exécution des programmes

Localisation de fautes avec la FD

Construction du contexte des traces

- **Exécutions du programme contenant une/des fautes**

```
public int Trityp(){
[57] int trityp ;
[58] if ((i == 0) || (j == 0) ||
       (k == 0))
[59]   trityp = 4 ;
[60] else
[61]  {
[62]    trityp = 0 ;
[63]    if ( i == j)
[64]       trityp = trityp + 1 ;
[65]    if ( i == k)
[66]       trityp = trityp + 2 ;
[67]    if ( j == k )
[68]       trityp = trityp + 3 ;
[69]    if (trityp == 0)
[70]    {
[71]      if ((i+j <= k) ||
              (j+k <= i) ||
              (i+k <= j))
[72]          trityp = 4 ;
[73]      else
[74]          trityp = 1 ;
[75]    }
[76]    else
[77]    {
[78]      if (trityp > 3)
[79]          trityp = 3 ;
[80]      else
[81]          if ((trityp == 1)
                  && (i+j > k))
[82]            trityp = 2 ;
[83]          else
// FAUTE (trityp == 2) --> (trityp == 3)
[84]            if ((trityp == 3)
                    && (i+k > j))
[85]              trityp = 2 ;
[86]            else
[87]              if((trityp == 3)
                      && (j+k > i))
[88]                trityp  = 2 ;
[89]              else
[90]                trityp = 4 ;
[91]    }
[92]   }
[93]   return(trityp) ;}
static public
string conversiontrityp(int i){
[97]  switch (i){
[98]     case 1:
[99]         return "scalen";
[100]    case 2:
[101]        return "isosceles";
[102]    case 3:
[103]        return "equilateral";
[104]    default:
[105]        return "not a ";}}
```

- **Contexte des traces**
  - **Objets :** traces d'exécution
  - **Attributs :** lignes du programme + verdict de l'exécution
  - **Description d'une trace d'exécution :** les lignes exécutées et le verdict de l'exécution
  - **Exemple :** la 1ière exécution exécute les lignes 66, 68, ... est donne un résultat correct : trace e_1

	l_66	l_68	l_81	l_84	l_85	l_87	l_90	$\cdots$	*Succes*	*Echec*
e_1	×	×							×	
e_4	×		×	×		×	×		×	
...										
e_{108}		×	×	×	×					×
...										
e_{400}									×	

Construction du contexte des défaillances

- **Calcul des règles d'association :** *line* i, $\cdots$, *line* j $\rightarrow$ *Echec*
  - "Quand les lignes i, $\cdots$, j sont exécutées ensemble ***la plupart du temps*** l'exécution produit une défaillance"
  - **Indices statistiques pour mesurer *la plupart du temps***
    - **Support** : fréquence de la règle
    - **Lift** : comment l'exécution de l'ensemble des lignes augmente la probabilité d'avoir une défaillance
- **Contexte des défaillances**
  - **Description des règles par leur prémisse**
  - Règle r_2 :
    line 81, *line* 84, *line* 87, *line* 90, *line* 78, *line* 112, $\cdots$, *line* 93 $\rightarrow$ *Echec*

	l_81	l_84	l_87	l_90	l_105	l_66	l_78	l_112	$\cdots$	l_93
r_2	×	×	×	×	×	×	×	×		×
r_3	×	×	×	×			×	×		×
...										
r_{10}							×	×		×

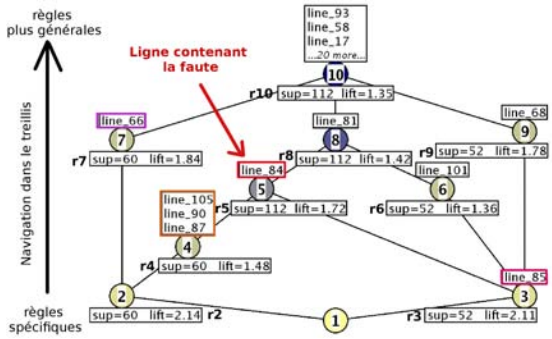
Treillis des défaillances

Beaucoup de règles générées et partiellement ordonnées

⇒ Compréhension des liens entre règles difficile à faire manuellement

IDÉE : Utilisation de la FCA pour construire un treillis

- **Treillis des défaillances à partir du contexte des défaillances**
  - **Concept (cercle) du treillis** : couple (L,R)
    - L : lignes communes aux règles de R
    - R : règles ayant toutes les lignes de L dans leur prémisse
  - Prémisse d'une règle : lignes étiquetant les concepts au dessus du concept étiqueté par la règle
  - Prémisse de r_3 : lignes 85, 84, 68, 101,81, 93, 58, 17, plus les 20 lignes du concept 10.



- **Localisation des fautes**
  - Navigation dans le treillis du bas vers le haut
  - Examen des lignes par un expert pendant la navigation
- **Interprétation de l'exemple**
  - Chemin Concept 2 : `trityp`= 2 et `(i+k>j)`
    → Concept 7 - ligne 66 : initialisation `trityp` à 2
    → Concept 4 - lignes 87, 90, 105 : branche else de la conditionnelle fausse
    ⇒ Résultat : `trityp`= 4 au lieu de 2
  - Chemin Concept 3 : `trityp`= 3 et `(i+k>j)` et `(j+k≤i)`
    ligne 85 : branche then de la conditionnelle fausse
    ⇒ Résultat : `trityp`= 2 au lieu de 4

Avantages de cette approche

- **Pas d'hypothèse sur le type des événements de la trace**
  - Lignes exécutées (exemple ici) mais aussi valeurs des variables, ...
- **Plusieurs exécutions avec des défaillances traitées en une fois**
- **Événements suspects**
  - Apparaissent **souvent** dans des exécution en échec
  - Apparaissent **rarement** dans des exécutions en succès
- **Ordre partiel sur les événements des traces**
  - Prise en compte des dépendances entre les événements des traces

Perspectives

- Utlisation de l'arbre de syntaxe abstraite pour compacter l'information
- Comparaison de suites de tests par comparaison des treillis

[1] Peggy Cellier, Mireille Ducassé, Sébastien Ferré, et Olivier Ridoux, **DeLLIS: a Data Mining Process for Fault Localization**, *SEKE'09*

Vers une méthode de validation des modèles formels AltaRica

Romain ADELINE

J. CARDOSO (ISAE), C. SEGUIN (ONERA), S. HUMBERT & P. DARFEUIL (TURBOMECA)

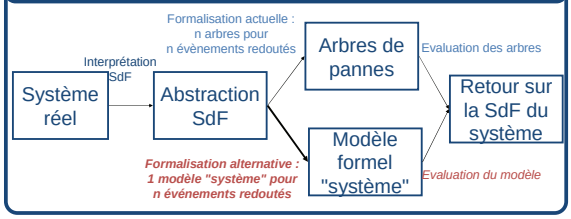
Présentation du besoin

Sûreté de Fonctionnement (SdF)

« Propriété d'un produit telle que ses utilisateurs puissent accorder, sur son cycle de vie, une confiance justifiée dans le service qu'il délivre. ». J.C Laprie

Besoin d'analyse de SdF pour garantir le niveau de sécurité d'un système

Vers un processus SdF fondée sur l'utilisation de modèles



Besoin : Pouvoir placer en ces modèles formels, supportant l'analyse de plusieurs événements redoutés, une confiance justifiée

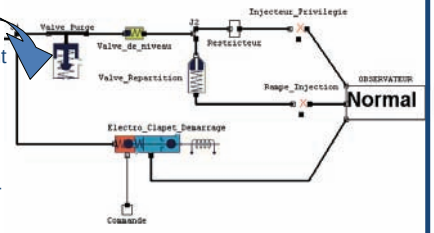
➡ **Une méthodologie de validation de ces modèles est nécessaire**

Modélisation "système" AltaRica

Modèle "système" :

-Une *architecture* interconnectant des composants (cf la vue graphique ci-contre)

-Une définition formelle AltaRica de la *dynamique* pour chaque composant (cf la vue textuelle ci-dessous)

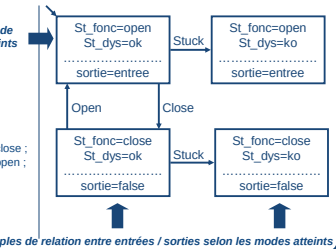


Le langage AltaRica, par l'exemple : modèle d'un composant "valve"

Sémantique : Automate de mode (AdM)

```
node valve
  flow
    entree : bool : in;
    sortie : bool : out ;
  state
    St _fonc: {open,close} ;
    St_dys : {ok,ko} ;
  init
    St_fonc:=open, St_dys:=ok;
  event
    Stuck, open, close ;
  trans
    St_fonc=open & St_dys=ok |- close -> St_fonc:=close ;
    St_fonc=close & St_dys=ok |- open -> St_fonc:=open ;
    St_dys=ok |- Stuck -> St_dys:=ko ;
  assert
    sortie = case { St_fonc=open : entree,
                    else false } ;
edon
```

Exemples de modes atteints

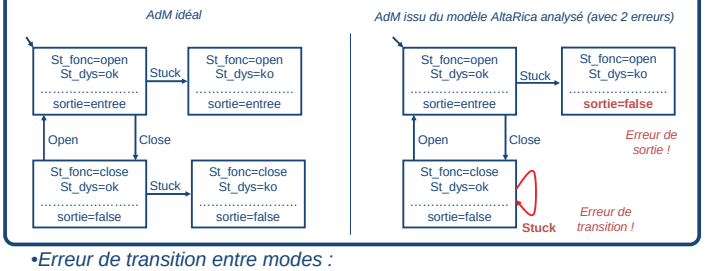


Exemples de relation entre entrées / sorties selon les modes atteints

Approche de validation des modèles AltaRica

1 : Classes d'erreur recherchée sur l'AdM d'un modèle AltaRica

Classes d'erreurs : Erreur de sortie / Erreur de transition



•Erreur de transition entre modes :

- *Une transition attendue est manquante dans l'AdM*
- *Une transition non souhaitée est présente dans l'AdM*

•Erreur de sortie :

- *Dans une combinaison de modes considérée, une variable de sortie peut prendre une valeur non souhaitée*

2 : Processus de recherche des erreurs

2.1. Création de check-lists par les experts du domaine

• Constitution d'une liste de contraintes de référence entre modes, entrées, sorties, transitions, séquences d'évènements ....

•Formalisation de ces contraintes de référence sous forme d'AdM observateurs

2.2. Recherche d'erreur dirigée par des check-lists

Pour chacune des contraintes formulées par les experts

• Evaluer si l'AdM satisfait chacune des contraintes

•Si la contrainte peut-être violée

- *Identifier les séquences minimales d'évènements conduisant à la violation*
- *Localiser les portions de code AltaRica à incriminer*

2.3. Analyse de la complétude des check-lists et recherche complémentaire

•Localisation des portions de codes non analysées par les check-lists

•Proposition de scénarios couvrant le code non analysé

•Recueil de l'avis des experts sur les scénarios proposés

3 : Méthodes et outils supportant le processus

•Evaluation d'AdM observateurs :

- *model-checker disponible*

•Génération de séquences minimales conduisant à une observation :

- *générateur de séquence disponible*

•Localisation des portions de code AltaRica sollicités par une séquence :

- *analyseur de couverture des transitions / assertions pour AltaRica: en cours*

Résumé

Ce document contient les actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées à l'université de Pau du 10 au 12 Mars 2010.

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 et de posters 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.