Actes des 1ères journées du GDR CNRS Génie de la Programmation et du Logiciel (GPL 2009), Toulouse, 28-30 janvier 2009

Marc Pantel, Yves Ledru

Actes des journées du GDR CNRS Génie de la Programmation et du Logiciel

28 au 30 janvier 2009, ENSEEIHT-IRIT, Toulouse

RESUME

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.

Editeurs : Yves LEDRU et Marc PANTEL

Table des matières

Session action AFSEC

Session groupe de travail Transformation

Session groupe de travail FORWAL

Session groupe de travail MFDL

Session groupe de travail COSMAL

Session action IDM

Session groupe de travail LAMHA

Session groupe de travail LTP

Session groupe de travail MTV2

Session groupe de travail RIMEL

Préface

C'est avec plaisir que je vous accueille aux premières Journées Nationales du GDR GPL. Le GDR Génie de la Programmation et du Logiciel (GPL), créé en 2008 pour une durée de 4 ans par le CNRS (GDR 3168), a vocation à structurer et animer la communauté scientifique dans le domaine du Génie Logiciel et des Langages de Programmation. Plusieurs actions seront également entreprises en direction des jeunes chercheurs.

Le Génie Logiciel et la Programmation sont au coeur de l'activité informatique et, comme elle, sont en constante évolution. D'une part, l'avénement de nouveaux domaines d'application et de nouveaux problèmes (systèmes embarqués, informatique ambiante, services sur le réseau, mobilité, sécurité, autonomie,...) fait naître de nouveaux besoins pour maîtriser la conception et la réalisation de tels systèmes. Cette maîtrise passe par la définition de méthodes et techniques de conception et de validation, ainsi que de nouveaux langages dédiés. D'autre part, dans les domaines plus classiques de l'informatique, comme le développement de systèmes d'information, la compétition internationale et la croissance continue de la taille des applications exigent chaque jour des gains en productivité et en qualité, qui sont autant de défis pour les concepteurs de langages de programmation et d'outils automatisés de conception et validation. Les progrès dans ces domaines ont des répercussions au delà de la communauté informatique : dans de nombreux secteurs industriels, le progrès est conditionné par la mise en oeuvre de solutions informatiques au point que le développement de la partie logicielle de ces produits est un facteur prépondérant dans le temps de développement de produits innovants. Dans ce contexte, la création d'un GDR dédié à ces thématiques me semble pleinement d'actualité.

Ces premières journées nationales clôturent une première année d'activité du GDR GPL où la plupart des groupes de travail ont organisé 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.

Les premières journées nationales constituent un premier temps fort dans la vie du GDR GPL. 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, une table ronde, 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. Enfin, je remercie les groupes de travail qui ont organisé au début de cette semaine plusieurs événements scientifiques, tous très intéressants : atelier AFADL, journées du groupe COSMAL et des actions IDM et AFSEC.

Nous espérons ainsi réunir une assemblée significative et représentative des activités du GDR GPL et contribuer à la renaissance d'une communauté nationale.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit d'Alessandro Cimatti (Center for Information Technology, Fondazione Bruno Kessler , Trento, Italie), de Paul Klint (Centrum Wiskunde & Informatica (CWI) , Amsterdam, Pays-Bas) et de Xavier Leroy (INRIA Paris-Rocquencourt, France). 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. En 2008, seulement 5% des candidats à la qualification comme maître de conférences correspondaient à la thématique "génie logiciel et programmation", ce qui peut paraître surprenant pour une discipline qui est au coeur de la science informatique. Il est donc essentiel de préparer de nouvelles générations de chercheurs et d'enseignants-chercheurs et de porter l'effort tant sur leur quantité que sur leur qualité. L'Ecole des Jeunes Chercheurs en Programmation 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.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de ces premières 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 toulousains ont relevé le défi d'organiser cet événement scientifique sans réellement savoir combien de participants il attirerait. Parmi eux, je remercie chaleureusement Marc Pantel et Christel Seguin 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 2009 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Yves Ledru (Président), LIG, Université de Grenoble-1

Yamine Ait Ameur, LISI / ENSMA
Franck Barbier, LIUPPA, Université de Pau et des Pays de l'Adour
Mireille Blay-Fornarino, I3S, Polytech'Nice
Dominique Cansell, LORIA, Université de Metz
Pierre Castéran, LABRI, Université de Bordeaux I
Jean-Michel Couvreur, LIFO, Université d'Orléans
Catherine Dubois, CEDRIC, ENSIIE
Hubert Dubois, CEA-LIST
Laurence Duchien, LIFL, Université des Sciences et Technologies de Lille
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
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
Pierre-Etienne Moreau, LORIA, INRIA
Mourad Oussalah, LINA, Université de Nantes
Marc Pantel, IRIT, INPT
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

Marc Pantel (IRIT)
Christel Seguin (ONERA-DCSD)
Jean-Paul Bodeveix (IRIT)
Mamoun Filali (IRIT)
Pierre Maurice (IRIT)
Pierre Michel (ONERA-DTIM)

Nous tenons à remercier chaleureusement la direction de l'ENSEEIHT pour avoir accepté d'accueillir les journées, les services communications et financiers de l'IRIT ainsi que les services entretiens et techniques de l'ENSEEIHT pour leur assistance dans l'organisation des journées.

Programme préliminaire

Mercredi 28 janvier 14h-19h30

<table>
<tr><td>14h-15h30</td><td colspan="2">Conférencier invité : Alessandro Cimatti, Center for Information Technology, Fondazione Bruno Kessler , Trento, Italie
Titre : Software Model Checking via SMT solving</td></tr>
<tr><td></td><td colspan="2">Pause café et posters</td></tr>
<tr><td>16h-17h30</td><td>Session Action AFSEC

Titre : Modéliser et vérifier les systèmes embarqués : pourquoi et comment ?
Auteur :
Jean-Pierre Elloy, Ecole Centrale / IRCCyN, Nantes

Titre : Dépliages de réseaux d'automates et application à la supervision
Auteur :
Claude Jard, ENS Cachan, IRISA, Université Européenne de Bretagne

Titre : Expressivité comparée de modèles temporisés.
Auteur :
Olivier H. Roux, IRCCyN, Université de Nantes</td><td>Session GT Transformation

Titre : Génération de canevas de programmation dédiés pour les applications de téléphonie avancées
Auteurs :
Wilfried Jouve INRIA/LaBRI,
Nicolas Palix, DIKU, Université de Copenhague,
Charles Consel, INRIA/LaBRI,
Patrice Kadionik, IMS, Université de Bordeaux

Session GT FORWAL

Titre : Approximations régulières pour l'analyse d'accessibilité
Auteurs :
Yohan Boichut, LIFO, Université d'Orléans
Roméo Courbis, LIFC, INRIA-CASSIS, Université de Franche-Comté
Pierre-Cyrille Héam, LSV, CNRS-INRIA, Ecole Normale Supérieure de Cachan
Olga Kouchnarenko3, LIFC, INRIA-CASSIS, Université de Franche-Comté

Titre : Vérification et systèmes de réécriture : de plus en plus efficace
Auteurs :
Emilie Balland, équipe PAREO, INRIA Lorraine, Nancy
Yohan Boichut, équipe PRV, LIFO, Orléans
Pierre-Etienne Moreau, équipe PAREO, INRIA Lorraine, Nancy
Thomas Genet, équipe LANDE, INRIA Rennes</td></tr>
<tr><td>17h45-19h30</td><td colspan="2">Réunion Bureau GDR + Responsables des Groupes de Travail + Comité scientifique
Bilan première année GDR GPL et projets pour la 2e année.</td></tr>
</table>

Jeudi 29 janvier 9h-12h30

<table>
<tr><td>9h-10h30</td><td colspan="2">Conférencier invité : Paul Klint (Centrum Wiskunde & Informatica (CWI) , Amsterdam)
Titre : Grammarware is everywhere and what to do about that?</td></tr>
<tr><td></td><td colspan="2">Pause café et posters</td></tr>
<tr><td>11h-12h30</td><td>Session GT MFDL

Titre : Un cadre formel pour le contrôle d'accès
Auteur : Mathieu Jaume - SPI - LIP6 - Université Paris 6

Titre : Une approche incrémentale combinant test et extraction de modèles
Auteurs :
Roland Groz, LIG, Grenoble Universités
Muzammil Shahbaz, France Telecom R&D
Keqin Li, LIG, Grenoble Universités

Titre : Développement formel par composants : assemblage et vérification à l'aide de B
Auteurs :
Arnaud Lanoix, COLOSS/LINA,
Samuel Colin et Jeanine Souquières (DEDALE/LORIA)</td><td>Session GT COSMAL

Titre : Construction dynamique d'annuaires de composants par classification de services utilisant l'analyse formelle de concepts
Auteurs:
Gabriela Arévalo, LIFIA - Facultad de Informática (UNLP), La Plata, Argentina
Nicolas Desnos, LGI2P - Ecole des Mines d'Alès, Nîmes
Marianne Huchard - LIRMM - UMR 5506 - CNRS and Univ. Montpellier 2
Christelle Urtado - LGI2P - Ecole des Mines d'Alès, Nîmes
Sylvain Vauttier - LGI2P - Ecole des Mines d'Alès, Nîmes

Titre : Vers la génération de modèles de sûreté de fonctionnement
Auteurs :
Xavier Dumas, Claire Pagetti, Laurent Sagaspe, Pierre Bieber, ONERA-CERT
Philippe Dhaussy, ENSIETA - LiSyC - DTN

Titre : Composition et expression qualitative de politiques d'adaptation pour les composants Fractal
Auteurs :
Franck Chauvel, Peking University, Chine
Olivier Barais, Noel Plouzeau, IRISA (INRIA & Université de Rennes 1)
Isabelle Borne, VALORIA (Université de Bretagne Sud)
Jean-Marc Jézéquel, (INRIA & Université de Rennes 1)</td></tr>
<tr><td></td><td colspan="2">Repas</td></tr>
</table>

Jeudi 29 janvier 14h-18h30

<table>
<tr><td rowspan="2">14h-15h30</td><td>Session Action IDM

Titre : Retours sur Models 2008
Auteur :
Xavier Blanc, INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, Université des Sciences et Technologies de Lille

Titre : Des plate-formes pour l'IDM: "OPEES, OpenEmbeDD, Papyrus, TOPCASED..."
Auteur :
Sébastien Gérard, CEA LIST

Titre : Alignement de métamodèles pour la génération automatique de transformation de modèles
Auteurs :
Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade et Clémentine Nebut, LIRMM, CNRS et Université de Montpellier 2</td><td>Session GT LaMHa

Titre : Sémantiques formelles d'un mini-langage impératif BSP - Application à la preuve de programmes et à l'optimisation
Auteurs:
Jean Fortin et Frédéric Gava, LACL, Université Paris Est

Titre : Functional meta-programming for parallel skeletons
Auteurs :
Jocelyn Serot, LASMEA, CNRS/U. Blaise Pascal
Joel Falcou, LRI, Université Paris-Sud

Titre : Placement d'applications distribuées hétérogènes sur des architectures de type grappe
Auteurs :
Sylvain Jubertie, Emmanuel Melin, Jérémie Vautard, Arnaud Lallouet, Laboratoire d'Informatique Fondamentale d'Orléans</td></tr>
<tr><td></td><td>Pause café et posters</td></tr>
<tr><td>16h-17h30</td><td colspan="2">Session Posters et démos</td></tr>
<tr><td>17h30-18h30</td><td colspan="2">Table ronde (Véronique Donzeau-Gouge, Luis Farinas, Jean-Louis Giavitto) animée par Yves Ledru
Thème : Quel rôle pour les conférences francophones?</td></tr>
<tr><td></td><td colspan="2">Dîner</td></tr>
</table>

Vendredi 30 janvier 9h-12h30

<table>
<tr><td>9h-10h30</td><td colspan="2">Conférencier invité : Xavier Leroy, INRIA Paris-Rocquencourt
Titre : Vérification formelle d'un compilateur C réaliste</td></tr>
<tr><td></td><td colspan="2">Pause café et posters</td></tr>
<tr><td>11h-12h30</td><td>Session GT LTP
Titre : Classes de types de première classe
Auteurs :
Matthieu Sozeau, LRI, INRIA/Université Paris Sud
Nicolas Oury, Université de Nottingham

Titre: Code porteur de preuve pour la vérification de bytecode Java
Auteurs :
Gilles Barthe, IMDEA Software, Madrid, Spain
Pierre Crégut, France Télécom, France
Benjamin Grégoire, INRIA Sophia-Antipolis Méditerranée, France
Thomas Jensen, IRISA/CNRS, France
David Pichardie, IRISA/INRIA Rennes Bretagne Atlantique, France

Titre : Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe
Auteurs :
Sandrine Blazy, Benoît Robillard et Éric Soutif, Laboratoire CEDRIC</td><td>Session GT MTV2

Titre : Constraint-Based Software Testing
Auteurs :
Sebastian Bardin, Bernard Botella, CEA LSL
Frédéric Dadeau, University of Franche-Comté, LIFC / INRIA projet CASSIS
Florence Charreteur, Arnaud Gotlieb, INRIA Rennes-Bretagne Atlantique, projet LANDE
Bruno Marre, CEA LSL
Claude Michel, Michel Rueher, University of Nice-Sophia Antipolis, projet CeP
Nicky Williams, CEA LSL

Titre : Simulated annealing applied to test generation: landscape characterization and stopping criteria
Auteurs :
Hélène Waeselynck, Pascale Thévenod-Fosse, Olfa Abdellatif-Kaddour, LAAS-CNRS.

Titre : Coverage-biased Random Exploration of Models
Auteurs :
Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Université de Paris-Sud 11 & CNRS
Richard Lassaigne, Université Paris Diderot 7 & CNRS
Johan Oudinet, Sylvain Peyronnet, Université de Paris-Sud 11 & CNRS</td></tr>
<tr><td></td><td colspan="2">Repas</td></tr>
</table>

Vendredi 30 janvier 14h-16h

14h-15h30		Session GT RIMEL Titre : On the Use of Formal Techniques to Support Model Evolution Auteurs : Tom Mens, Service de Génie Logiciel, Université de Mons-Hainaut, Belgique Ragnhild Van Der Straeten, Systems and Software Engineering Lab, Vrije Universiteit Brussel, Belgique Titre : Conciliating Property Stability and System Evolution through Software Model Analysis Auteur : Ilham Alloui, LISTIC - Polytech'Savoie Titre : Le contrat d'évolution d'architectures : un outil pour le maintien de propriétés non fonctionnelles Auteurs: Régis Fleurquin, VALORIA, Université Bretagne-Sud et IRISA, INRIA Rennes, Chouki Tibermacine, LIRMM, Université Montpellier II Salah Sadou, VALORIA, Université Bretagne-Sud
15h30-16h	Clôture : proclamation du meilleur poster et du nouveau logo du GDR	

Conférences invitées

Software Model Checking via SMT solving

Alessandro Cimatti
Center for Information Technology
Fondazione Bruno Kessler
Trento, Italie

Software model checking is a fully automated approach to the verification of software that has received substantial interest in the last decade. In this talk, I will first review the field, focussing on different forms of software model checking based on predicate abstraction. Then, I will present some recent improvements, showing how the use of modern Satisfiability Modulo Theories (SMT) procedures can increase the accuracy and the efficiency of the analysis.

Grammarware is everywhere and what to do about that?

Paul Klint
Centrum Wiskunde & Informatica (CWI)
Amsterdam, Pays-bas

Many aspects of software engineering are based on grammars albeit that they occur in disguise. From the grammars for mainstream programming languages that are hidden in compilers to the grammars for Domain-specific Languages that are part of dedicated DSL implementations. From XML schemas or MDA models to Application Programming Interfaces for large systems. In all these cases large amounts of software may depend on such grammars and will be affected when the grammar is erroneous or is subject to evolution.

While software engineers use rigorous engineering principles for developing source code, this is much less the case when developing grammars in their various forms. What is the quality of a grammar? What is the effect of grammar modifications on the software that depends on it? What is the effect of API changes? How to refactor DSL programs?

In the first part of the talk I will sketch these problems and identify grammarware engineering as a promising research area that contains many interesting problems [1].

In the second part of the talk I will present some of our results in this area. After a brief overview of the Meta-Environment [2] I zoom in on current work on the design and implementation of the Rascal language, a special purpose scripting language for source code analysis and transformation [3]. Rascal provides many fundamental concepts like parsing, regular/list/set matching, term rewriting, backtracking and relational calculus in an integrated form that is as unintimidating as possible for the average programmer without a formal background. Design goals, the language itself and the current state of its implementation will be briefly discussed. We believe that this approach will enable the development of new tools not only for grammarware engineering but also for other problems in refactoring, software analysis and software transformation.

[1] Paul Klint, Ralf Lämmel and Chris Verhoef, Toward an engineering discipline for grammarware, ACM Transactions on Software Engineering Methodology, 2005(14), 331--380.
[2] Meta-Environment Home Page, http://www.meta-environment.org.
[3] Joint current research with Jurgen Vinju and Tijs van der Storm.

Vérification formelle d'un compilateur C réaliste

Xavier Leroy
INRIA Paris-Rocquencourt
Paris, France

Travail commun avec
Yves Bertot (INRIA Sophia-Antipolis),
Sandrine Blazy (ENSIIE),
Pierre Letouzey (Université Paris 7), et
Laurence Rideau (INRIA Sophia-Antipolis).

Les compilateurs sont des logiciels fort complexes qui parfois contiennent des erreurs entraînant la production de code exécutable faux à partir d'un programme source correct. L'existence de tels bugs introduits par le compilateur affaiblissent les garanties que l'on peut obtenir par l'application de méthodes formelles au niveau du programme source.

Cet exposé présente le projet CompCert: une expérience en cours consistant à développer et à prouver correct un compilateur réaliste, modérément optimisant, produisant de l'assembleur PowerPC à partir d'un large sous-ensemble du langage C. La preuve de correction, menée sur machine avec l'assistant de preuve Coq, établit que le code PowerPC engendré se comporte exactement comme prescrit par la sémantique du source C, éradiquant ainsi toute possibilité de bugs introduits par le compilateur et établissant un niveau de confiance sans précédent envers ce compilateur.

Table Ronde

Quel rôle pour les conférences francophones ?

Participants :

Véronique Donzeau-Gouge
Luis Fariñas del Cerro
Jean-Louis Giavitto

Animateur : Yves Ledru

Nous avons connu en 2008 une diminution des soumissions aux conférences francophones proches du GDR GPL. D'aucuns y voient la conséquence d'une dépréciation des publications francophones. En effet, les critères retenus pour déclarer un chercheur comme « publiant » font la part belle aux revues et conférences internationales. D'autres y voient un signe de l'internationalisation de nos laboratoires où beaucoup de nos doctorants ne sont pas francophones.

Dans ce contexte quelle est aujourd'hui le rôle des conférences et des revues francophones ? Sont-elles vouées à disparaître ? Ont-elles vocation à devenir le petit village gaulois où se trouvent les irréductibles défenseurs de la langue française ? Doivent-elles se regrouper en un unique événement annuel pour atteindre une masse critique ? Faut-il adopter l'anglais pour ainsi attirer des contributeurs venus de toute la planète à l'instar de ce qui se fait chez nos voisins allemands ? Faut-il privilégier l'événement social où germent les projets nationaux et où les jeunes chercheurs préparent leur mobilité ?

Cette table ronde donnera la parole à des représentants du CNRS et de l'AERES pour qu'ils précisent le rôle qu'ils voient jouer à ces conférences et revues, ainsi qu'au GDR GPL, dans une organisation de la recherche en profond renouvellement.

Session action AFSEC

Approches Formelles des Systèmes Embarqués Communicants

Modéliser et vérifier les systèmes embarqués : pourquoi et comment ?

Jean-Pierre Elloy

Janvier 2009

Le particularisme des systèmes embarqués, en particulier la finitude du nombre de leurs objets constitutifs a permis l'émergence de démarches spécifiques pour leur conception ainsi que pour vérifier la conformité du produit obtenu aux exigences initiales de l'application. Dans ces démarches, la modélisation est l'abstraction charnière qui : décrit le comportement du système à développer, sert de support à son développement, et doit reconnaître les propriétés du système développé. Mais pour atteindre ces objectifs, la modélisation d'un système embarqué doit être soigneusement élaborée. Il faut identifier avec précision le périmètre du modèle à établir, puis l'immerger dans un environnement le saturant de toutes les circonstances qui peuvent l'affecter. Il faut recenser les actions pertinentes du modèle au regard des propriétés qu'il doit vérifier, pour ensuite en démontrer leur satisfaction. Il faut s'assurer que les opérateurs de manipulation des modèles respectent la sémantique des mêmes opérations effectuées sur les systèmes modélisés, pour autoriser une conception incrémentale du système à construire. Il faut enfin inscrire avec soin les phases d'exploitation d'un modèle dans le processus général de conception du système.

On examinera ces différents aspects dans le cas où le système à concevoir est le dispositif de pilotage d'un équipement embarqué. On indiquera quelles sont les types de fautes qui peuvent en "distordre" le fonctionnement, puis on se focalisera sur la modélisation comportementale (causale) de ce système de pilotage. On montrera ainsi qu'il est nécessaire de porter une attention soutenue à cette modélisation pour aboutir à un système embarqué aux prestations réellement sûres de fonctionnement.

En conclusion, on affirmera que les méthodes formelles d'analyse de modèles se révèlent être un moyen de garantir la sûreté de fonctionnement des systèmes embarqués. Les travaux actuels sur leur élargissement théorique, sur l'optimisation de leurs techniques de mise en œuvre, sur leur outillage, sur leur intégration dans les démarches d'ingénierie système, leur confèrent une crédibilité applicative grandissante. Le succès de leurs applications actuelles en vraie grandeur en confirme la richesse et les potentialités. Leur avenir est maintenant ouvert...

Expressivité comparée de modèles temporisés

Olivier (H.) Roux

IRCCyN, CNRS UMR 6597, Nantes, France
Olivier-h.Roux@irccyn.ec-nantes.fr

Abstract. Il y a en général une opposition entre l'*expressivité* d'un modèle, c'est-à-dire sa capacité à représenter un nombre important de caractéristiques d'un système, et sa *simplicité* en termes de vérification ou de contrôle (décidabilité et complexité algorithmique). Dans cet article, nous nous intéressons à des modèles dits *temporisés* pour lesquels le temps est manipulé de manière explicite et, plus particulièrement, aux automates temporisés et à différentes classes de réseaux de Petri temporels. Nous donnons les principaux résultats de décidabilité sur ces modèles et nous comparons leur expressivité en termes de bisimulation temporelle.

Key words: réseaux de Petri temporels, automates temporisés, expressivité, bisimulation

1 Introduction

La classe des systèmes *temps réel* et *embarqués* regroupe les programmes informatiques qui pilotent une application en réagissant à des stimuli reus d'un environnement évolutif. Cela leur vaut également le nom de systèmes *réactifs*. Ces systèmes doivent réagir aux évolutions d'un procédé qu'ils contrôlent ou qu'ils suivent avec un temps de réponse suffisamment petit par rapport à sa dynamique interne. Ils sont soumis à des contraintes temporelles fortes. Il est donc important de s'assurer de leur correction non seulement fonctionnelle, mais aussi temporelle.

Les modèles présentés dans cet article manipulent des actions considérées comme instantanées et identifiées comme pertinentes (qui représentent en général les événements associés aux commandes des actionneurs et aux mesures des capteurs du procédé) et des variables à valeurs réelles représentant l'écoulement de durées. Ils décrivent ainsi des sous-classes des systèmes temps réel embarqués.

1.1 Les modèles temporisés

Étant donnée une propriété que l'on veut vérifier (ou garantir par un contrôleur) sur un système (respectivement sur un procédé), un modèle peut être défini comme une abstraction du système telle que la valeur de vérité de la satisfaction de la propriété est la même pour le système et son modèle. Cela implique qu'un modèle n'est valable que pour un ensemble de propriétés et ne reflète pas

complètement le comportement réel du système. Par ailleurs, il y a en général une opposition entre l'*expressivité* d'un modèle, c'est-à-dire sa capacité à représenter un nombre important de caractéristiques du système, et sa *simplicité* en termes de vérification ou de contrôle (décidabilité et complexité algorithmique).

D'un autre coté, choisir un modèle *suffisamment* expressif permet d'éviter le pessimisme d'un modèle surévaluant[1] le comportement réel du système. En effet, ce pessimisme peut, d'une part, conduire à une propriété de sûreté décidée fausse sur le modèle alors qu'elle est vraie sur le système réel et, d'autre part, conduire à un espace d'états infini alors que celui d'un modèle plus fin serait fini (ou aurait une abstraction finie). En particulier le modèle discret d'une application peut avoir un espace d'états non calculable alors que l'ajout des paramètres temporels de l'application, en restreignant les comportements, peut borner le nombre d'états discrets du modèle et permettre ainsi sa vérification.

De plus, dans certaines applications, le cahier des charges spécifie certaines propriétés dont la détermination nécessite la connaissance de durées séparant certaines transitions ou actions. Il est alors nécessaire de prendre en compte des caractéristiques temporelles autres que le temps logique (capturé par exemple par un modèle du type automate fini ou réseau de Petri) et de modéliser les dynamiques du procédé ou la réactivité du système de contrôle afin de vérifier des propriétés telles qu'un temps de réponse quantifié ou de faire apparatre des causes précises de dysfonctionnement.

Les modèles Les modèles très généraux (et de très bas niveau) tels que les systèmes de transitions (temporisés ou non) ne sont pas directement utilisables en vérification ou en contrôle car ils peuvent représenter n'importe quel système, sans restriction, ce qui a en général pour conséquence qu'il n'est pas possible de représenter ces systèmes de manière finie. Nous nous intéressons donc à des modèles de plus haut niveau que l'on peut représenter de manière finie mais dont on donne la sémantique par un système de transitions.

Dans le cadre non temporisé, le plus simple des modèles est sans doute celui des automates finis qui n'est rien d'autre que l'ensemble des systèmes de transitions ayant un nombre fini d'états. Les deux autres principaux modèles sont les réseaux de Petri et les algèbres de processus comme CCS (*Calculus of Communicating Systems*) [1]. Ces modèles peuvent être décrits de manière finie, même si leurs dépliages en systèmes de transitions sont généralement infinis.

Dans le cadre temporisé, ces modèles ont été étendus en leur adjoignant des notions de temps conduisant ainsi à des modèles dits de *haut niveau*. Parmi ces modèles, nous pouvons par exemple citer les automates temporisés [2] (*Timed Automata*, TA), qui proviennent du modèle des automates finis, les réseaux de Petri *temporisés* [3] (*Timed Petri Nets*) ou *temporels* [4] (*Time Petri Nets*, TPN) et les algèbres de processus temporisées, comme TCCS, l'une des extensions temporisées de CCS [5].

[1] C'est à dire un modèle grossier ne permettant pas de distinguer des comportements fins dont certains sont en réalité impossibles.

Dans la suite de cet article nous nous intéressons à ces modèles dits *temporisés* pour lesquels le temps est manipulé de manière explicite et, plus particulièrement, aux automates temporisés et aux réseaux de Petri temporels

Enfin, ces modèles peuvent être étendus en mesurant l'écoulement du temps par des *chronomètres* à la place des *horloges* ce qui permet de modéliser des actions que l'on interrompt puis que l'on reprend plus tard. Les dérivées par rapport au temps des variables peuvent alors prendre deux valeurs exprimant la progression (1) ou la suspension (0). Ces modèles entrent dans la classe des modèles hybrides. Ils permettent de modéliser les phases d'exécution et d'arrêt des activités d'un système ce qui permet de décrire finement des phases d'exécution et de suspension d'un programme, comme celles qui sont décidées par un ordonnanceur.

2 Notations, langages et systèmes de transitions temporisés

Notations générales

- L'ensemble $\mathbb{B}$ désigne les valeurs booléennes tt et ff ;
- $\mathbb{N}, \mathbb{Q}, \mathbb{R}$ désignent respectivement les ensembles des entiers naturels, des rationnels et des réels ;
- $\mathbb{R}_{\geq 0}$ désigne l'ensemble des réel positifs ou nuls et $\mathbb{R}_{>0} = \mathbb{R}_{\geq 0} \setminus \{0\}$ est l'ensemble des réels strictement positifs ; il en est de même pour les ensembles $\mathbb{N}$ et $\mathbb{Q}$;
- Soit $n \in \mathbb{N}$. $\mathbb{R}^n$ désigne l'espace réel à n dimensions ;
- Soit un ensemble fini E. Nous notons $|E|$ le cardinal de E ;
- B^A désigne l'ensemble des applications de A dans B. Si A est fini et $|A| = n$, un élément de B^A est aussi un vecteur dans B^n. Les opérateurs usuels $+, -, <, \leq, >, \geq$ et $=$ sont étendus (composante par composante) aux vecteurs de A^n avec $A = \mathbb{N}, \mathbb{Q}, \mathbb{R}$;
- Une *valuation* ν sur un ensemble de variables X est un élément de $\mathbb{R}_{\geq 0}^X$. Pour $\nu \in \mathbb{R}_{\geq 0}^X$ et $d \in \mathbb{R}_{\geq 0}$, $\nu + d$ désigne la valuation $(\nu + d)(x) = \nu(x) + d$, et pour $X' \subseteq X$, $\nu[X' \mapsto 0]$ désigne la valuation ν' avec $\nu'(x) = 0$ si $x \in X'$ et $\nu'(x) = \nu(x)$ sinon. $\mathbf{0}$ désigne la valuation telle que $\forall x \in X, \nu(x) = 0$;
- Soit X un ensemble de variables, une *contrainte atomique* sur X est une formule de la forme $x \sim c$ avec $x \in X$, $c \in \mathbb{Q}_{\geq 0}$ et $\sim \in \{<, \leq, \geq, >\}$. $\mathcal{C}(X)$ désigne l'ensemble des *contraintes* sur l'ensemble de variables X constitué de la conjonction des contraintes atomiques sur X. De plus, nous notons $\mathcal{C}_{dbm}(X)$ l'ensemble des combinaisons booléennes (avec les opérateurs logiques $\vee$, $\wedge$ et $\neg$) de termes de la forme $x - x' \sim c$ ou $x \sim c$, avec $x, x' \in X$, $\sim \in \{<, \leq, =, \geq, >\}$ et $c \in \mathbb{Q}$;
- Pour une contrainte $\varphi \in \mathcal{C}(X)$ et une valuation $\nu \in \mathbb{R}_{\geq 0}^X$, nous notons $\varphi(\nu) \in \mathbb{B}$ la valeur de vérité de φ obtenue en substituant chaque occurrence de x dans φ par $\nu(x)$. Nous notons ainsi $[\![\varphi]\!] = \{\nu \in \mathbb{R}_{\geq 0}^X \mid \varphi(\nu) = \mathsf{tt}\}$.

Mots et langages temporisés Soit Σ un ensemble fini appelé *alphabet*. Σ^* (respectivement Σ^ω) est l'ensemble des suites finies (respectivement infinies) d'éléments de Σ et $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$. Nous parlerons de *mots finis* sur l'alphabet Σ pour les éléments de Σ^* et de *mots infinis* sur l'alphabet Σ pour les éléments de Σ^ω.

Nous notons $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ avec $\varepsilon \notin \Sigma$, où ε représente une action (lettre) particulière dite *silencieuse* ou *non observable*.

Definition 1 (Mots temporisés). *Un* mot temporisé *w sur un alphabet Σ est une séquence finie ou infinie $w = (a_0, d_0)(a_1, d_1)\cdots(a_n, d_n)\cdots$ telle que pour tout $i \geq 0$, $a_i \in \Sigma$, $d_i \in \mathbb{R}_{\geq 0}$ et $d_{i+1} \geq d_i$.*

Un mot temporisé $w = (a_0, d_0)(a_1, d_1)\cdots(a_n, d_n)\cdots$ sur Σ est une paire $(a, d) \in \Sigma^\infty \times \mathbb{R}^\infty_{\geq 0}$ avec $|a| = |d|$. La valeur d_i donne la date absolue (en considérant que l'instant initial est la date 0) de l'action a_i.

Nous notons $\mathit{Untimed}(w) = a_0 a_1 \cdots a_n \cdots$ pour la partie non temporisé de w, et $\mathit{Duration}(w) = \sup_{d_i \in d} d_i$ pour la durée de w.

Definition 2 (Langages temporisés). *Notons $\mathcal{TW}^*(\Sigma)$ (respectivement $\mathcal{TW}^\omega(\Sigma)$) l'ensemble des mots temporisés finis (respectivement infinis) sur Σ et $\mathcal{TW}^\infty(\Sigma) = \mathcal{TW}^*(\Sigma) \cup \mathcal{TW}^\omega(\Sigma)$. Un* langage temporisé *L sur Σ est un sous-ensemble de $\mathcal{TW}^\infty(\Sigma)$.*

Systèmes de transitions temporisés Nous nous intéressons à des systèmes pouvant être décrits par un système de transitions, c'est-à-dire un ensemble (quelconque) d'états et de transitions étiquetées par des actions entre les états. Lorsque l'on se trouve dans l'un des états d'un système de transitions, il est possible de changer d'état en effectuant l'action qui étiquette l'une des transitions sortant de l'état. Une exécution dans un système de transitions est alors une séquence d'actions qui est souvent notée sous la forme d'un mot représentant la succession des actions.

Les systèmes de transitions temporisés (*Timed Transition System* en anglais, ou TTS) sont des systèmes de transitions particuliers pour lesquels deux types de transitions sont possibles : des transitions d'action et des transitions de temps modélisant respectivement des évolutions *discrètes* et des évolutions *continues* du système.

Definition 3 (Système de transitions temporisé). *Un* système de transitions temporisé (TTS) *sur un ensemble d'action Σ est un quadruplet $S = (Q, Q_0, \Sigma, \longrightarrow)$ où Q est un ensemble d'état, $Q_0 \subseteq Q$ est un ensemble d'états initiaux, Σ est un ensemble fini d'actions (disjoint de $\mathbb{R}_{\geq 0}$), $\longrightarrow \subseteq Q \times (\Sigma \cup \mathbb{R}_{\geq 0}) \times Q$ est une relation de transition (ensemble d'arcs). Si $(q, e, q') \in \longrightarrow$, nous notons aussi $q \xrightarrow{e} q'$. La relation de transition se décompose en une relation de transition* continue $\xrightarrow{d \in \mathbb{R}_{\geq 0}}$ *et une relation de transition* discrète $\xrightarrow{a \in A}$.

Nous faisons les hypothèses habituelles suivantes sur les TTS:

- 0-DÉLAI : $q \xrightarrow{0} q'$ ssi $q = q'$,

- ADDITIVITÉ: si $q \xrightarrow{d} q'$ et $q' \xrightarrow{d'} q''$ avec $d,\ d' \in \mathbb{R}_{\geq 0}$, alors $q \xrightarrow{d+d'} q''$,
- CONTINUITÉ: si $q \xrightarrow{d} q'$, alors pour tout d' et d'' dans $\mathbb{R}_{\geq 0}$ tel que $d = d' + d''$, il existe q'' tel que $q \xrightarrow{d'} q'' \xrightarrow{d''} q'$,
- DÉTERMINISME TEMPOREL: si $q \xrightarrow{d} q'$ et $q \xrightarrow{d} q''$ avec $d \in \mathbb{R}_{\geq 0}$, alors $q' = q''$.

Definition 4 (Exécution (run) d'un TTS). *Une* exécution ρ *d'un TTS* S *est une séquence finie ou infinie de transitions continues et discrètes de* S. *L'ensemble des exécutions d'un TTS* S *est noté* $[\![S]\!]$.

Une exécution peut toujours s'écrire sous la forme d'une alternance de transitions continues (éventuellement de durée 0) et de transitions discrètes :

$$\rho = q_0 \xrightarrow{d_0} q'_0 \xrightarrow{a_0} q_1 \xrightarrow{d_1} q'_1 \xrightarrow{a_1} \cdots q_n \xrightarrow{d_n} q'_n \cdots$$

Pour une exécution de taille n, nous notons $\mathit{Untimed}(\rho) = a_0 a_1 \cdots$ et $\mathit{Duration}(\rho) = \sum_{i=0}^{n} d_i$.

Definition 5 (Trace). *La* trace *d'une exécution* $\rho = q_0 \xrightarrow{d_0} q'_0 \xrightarrow{a_0} q_1 \cdots q_k \xrightarrow{d_k} q'_k \cdots$ *d'un TTS est le mot temporisé* $\mathit{trace}(\rho) = (a_0, \delta_0) \cdots (a_k, \delta_k) \cdots$ *avec* $\delta_k = \sum_{i=0}^{k} d_i$.

Un mot temporisé $w = (a_i, d_i)_{0 \leq i \leq n}$ est *accepté* par le système de transition $S = (Q, Q_0, \Sigma, \longrightarrow)$ si il existe une exécution de S à partir d'un état initial $q_0 \in Q_0$ et de trace w. Le *langage temporisé* $\mathcal{L}(S)$ accepté par S est l'ensemble des mots temporisés acceptés par S. C'est-à-dire qu'on considère ici que tous les états du système de transition sont accepteurs.

Le *langage non temporisé* de S est constitué des mots de S dans lesquels les actions continues ont été abstraites.

Definition 6 (TTS ε-abstrait). *Soit* $S = (Q, Q_0, \Sigma_\varepsilon, \longrightarrow)$ *un TTS. Nous définissons le TTS* $S^\varepsilon = (Q, Q_0^\varepsilon, \Sigma, \longrightarrow_\varepsilon)$ *dans lequel les actions* ε *ont été abstraites de* S *par :*

- $q \xrightarrow{d}_\varepsilon q'$ *avec* $d \geq \mathbb{R}_{\geq 0}$ *ssi il existe une exécution* $\rho = q \rightarrow q'$ *avec* $\mathit{Untimed}(\rho) = \varepsilon^*$ *et* $\mathit{Duration}(\rho) = d$,
- $q \xrightarrow{a}_\varepsilon q'$ *avec* $a \in \Sigma$ *ssi il existe une exécution* $\rho = q \rightarrow q'$ *avec* $\mathit{Untimed}(\rho) = \varepsilon^* a \varepsilon^*$ *and* $\mathit{Duration}(\rho) = 0$,
- $Q_0^\varepsilon = \{q \mid \exists q' \in Q_0 \mid q' \rightarrow q \text{ et } \mathit{Duration}(\rho) = 0 \wedge \mathit{Untimed}(\rho) = \varepsilon\}$.

Enfin si S est défini sur Σ_ε, le langage accepté par S est constitué des mots dans lesquels les actions ε ont été abstraites.

3 Modèles temporisés

Les systèmes de transitions sont très généraux et donc de très (trop) bas niveau. D'une manière générale, il n'est pas possible de représenter de tels systèmes de manière finie. Ils ne sont par conséquent pas directement utilisables pour la vérification ou le contrôle. Nous nous intéressons donc à des modèles de plus haut niveau que l'on peut représenter de manière finie mais dont la sémantique sera donnée par un système de transitions.

3.1 Automates temporisés

Présentation Les automates temporisés, ou *Timed Automata* (TA) en anglais, étendent les automates finis classiques avec des horloges explicites. Ils ont été introduits par Alur et Dill en 1994 [2] et étendus avec la notion d'invariant par Henzinger *et al.* dans [6]. Le temps est ajouté au modèle classique des automates sous la forme d'horloges et de prédicats sur ces horloges. Ces prédicats sont de deux types : les *gardes*, associées aux transitions discrètes, donnent des contraintes sur les horloges, à respecter pour pouvoir exécuter cette transition discrète. Les *invariants*, associés aux localités, donnent des contraintes qui doivent être respectées dans les localités.

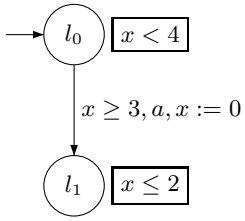


Fig. 1. Un automate temporisé

La figure 1 donne un exemple très simple d'automate temporisé. La localité initiale est l_0. L'automate dispose d'une seule horloge : x. x est nul à l'instant initial et l'invariant de l_0 indique donc que l'on pourra rester dans l_0 strictement moins de 4 unités de temps. Dès que 3 unités de temps se sont écoulées, la garde de la transition entre l_0 et l_1 est vérifiée (et l'invariant de l_1 est vérifié après franchissement de cette transition) et la transition peut donc être franchie. Si cette transition est franchie, l'horloge x est remise à zéro et l_1 devient la localité active.

Définitions

Definition 7 (Automate temporisé). *[6] Un* automate temporisé *est un 6-uplet* $(L, l_0, X, \Sigma, E, Inv)$ *où*

- L *est un ensemble fini de* localités *;*
- l_0 *est la* localité initiale *;*
- X *est un ensemble fini d'*horloges *à valeurs réelles positives ;*
- Σ *est un ensemble fini d'*actions *;*
- $E \subset L \times \mathcal{C}(X) \times \Sigma \times 2^X \times 2^{X^2} \times L$ *est un ensemble fini d'*arêtes*. Soit* $e = (l, \delta, \alpha, R, l') \in E$*. e est l'arête reliant la localité* l *à la localité* l'*, avec la* garde δ *et l'action* α*, l'ensemble d'horloges à remettre à zéro* R*.*
- $Inv \in \mathcal{C}(X)^L$ *associe un* invariant *à chaque localité.*

Nous définissons la sémantique des automates temporisés sous la forme d'un système de transitions temporisé.

Definition 8 (Sémantique d'un automate temporisé). *La sémantique d'un automate temporisé* $\mathcal{A}$ *est définie sous la forme d'un système de transitions temporisé* $\mathcal{S}_{\mathcal{A}} = (Q, Q_0, \Sigma, \rightarrow)$ *où*

- $Q = L \times (\mathbb{R}^+)^X$ *;*
- $Q_0 = (l_0, \mathbf{0})$ *;*
- $\rightarrow \in Q \times (\Sigma \cup \mathbb{R}) \times Q$ *est la relation définie pour* $a \in \Sigma$ *et* $d \in \mathbb{R}^+$ *par :*
  - *la relation de transition* discrète *:* $(l, \nu) \xrightarrow{a} (l', \nu')$ *ssi* $\exists (l, \delta, a, R, l') \in E$ *tel que*
    $$\begin{cases} \delta(\nu) = \texttt{true}, \\ \nu' = \nu[R \leftarrow 0], \\ Inv(l')(\nu') = \texttt{true} \end{cases}$$
  - *la relation de transition* continue *:*
    $(l, \nu) \xrightarrow{\epsilon(d)} (l, \nu')$ *ssi* $\begin{cases} \nu' = \nu + d, \\ \forall d' \in [0, d], Inv(l)(\nu + d') = \texttt{true} \end{cases}$

3.2 Réseaux de Petri T-temporels

Les réseaux de Petri et le temps Les deux extensions temporelles principales des réseaux de Petri sont les réseaux de Petri *temporisés* [3] et les réseaux de Petri *temporels* [4]. Pour les premiers, le temps est représenté par des durées minimales (ou exactes dans le cas d'un fonctionnement *au plus tôt* du réseau) de franchissement des transitions. Pour les seconds, le temps prend la forme d'un intervalle contraignant les instants de tir des transitions. Par ailleurs le temps peut être associé aux transitions, aux places, aux jetons, aux arcs... Les classes de réseaux de Petri temporisés qui en résultent sont incluses dans les classes de réseaux de Petri temporels correspondantes [7]. Les principales classes de réseaux de Petri temporels sont les réseaux de Petri T-temporels [8], P-temporels [9] et A-temporels [10] où un intervalle de temps est associé respectivement aux transitions, aux places et aux arcs. De plus, il existe deux sémantiques pour ces modèles : la sémantique faible pour laquelle les bornes temporelles supérieures peuvent être dépassées et la sémantique forte permettant la modélisation de l'urgence qui est une caractéristique essentielle pour les systèmes temps réel. Pour les réseaux de Petri A-temporels et P-temporels, la sémantique forte conduit à des jetons qui ne sont plus utilisables et qu'il faut faire *mourir* ce qui est parfois

difficile à interpréter. Les réseaux de Petri T-temporels sont ainsi les plus utilisés pour la modélisation des systèmes temps réel et ce sont ceux qui nous étudions dans ce livre.

Dans cet article, nous nous intéressons au modèle le plus courant, les réseaux de Petri T-temporels en sémantique forte, que nous notons $\overline{T\text{-}TPN}$ Cependant, par souci de concision et lorsque cela n'introduit pas d'ambiguté, nous utiliserons souvent le terme *réseaux de Petri temporels* - que nous noterons simplement TPN - pour désigner Les réseaux de Petri T-temporels en sémantique forte.

Présentation informelle Les réseaux de Petri T-temporels [4], ou *Time Petri Nets* (TPN) en anglais, étendent les réseaux de Petri avec des intervalles $[\alpha(t), \beta(t)]$ associés à chaque transition t du réseau Pour être tirée, une transition t doit non seulement être sensibilisée mais également l'avoir été continûment pendant une durée comprise entre $\alpha(t)$ et $\beta(t)$.

Définitions

Definition 9 (Réseau de Petri T-temporel). *Un réseau de Petri T-temporel est un 7-uplet* $\mathcal{N} = (P, T, {}^\bullet(.), (.)^\bullet, \alpha, \beta, M_0)$*, où*

- $P = \{p_1, p_2, \ldots, p_m\}$ *est un ensemble fini et non vide de* places *;*
- $T = \{t_1, t_2, \ldots, t_n\}$ *est un ensemble fini et non vide de* transitions $(T \cap P = \emptyset)$ *;*
- ${}^\bullet(.) \in (\mathbb{N}^P)^T$ *est la* fonction d'incidence amont *;*
- $(.)^\bullet \in (\mathbb{N}^P)^T$ *est la* fonction d'incidence aval *;*
- $M_0 \in \mathbb{N}^P$ *est le* marquage initial *du réseau ;*
- $\alpha \in (\mathbb{R}^+)^T$ *et* $\beta \in (\mathbb{R}^+ \cup \{\infty\})^T$ *sont les fonctions donnant pour chaque transition, respectivement son instant de tir* au plus tôt *et* au plus tard*(*$\alpha \leq \beta$*).*

Un marquage M du réseau est un élément de $\mathbb{N}^P$ tel que $\forall p \in P, M(p)$ est le nombre de jetons dans la place p.

Une transition t est dite *sensibilisée* par le marquage M si $M \geq {}^\bullet t$, c'est-à-dire si le nombre de jetons, pour M, de chaque place amont de t est plus grand ou égal à la valuation de l'arc entre cette place et la transition. Nous notons $t \in enabled(M)$.

Une transition t est dite *nouvellement sensibilisée* par le tir de la transition t' à partir du marquage M, ce que nous noterons $\uparrow enabled(t, M, t')$, si t est sensibilisée par le nouveau marquage $M - {}^\bullet t' + t'^\bullet$ mais ne l'était pas par le marquage $M - {}^\bullet t'$. Formellement,

$$\uparrow enabled(t, M, t') = ({}^\bullet t \leq M - {}^\bullet t' + t'^\bullet) \wedge ((t = t') \vee ({}^\bullet t > M - {}^\bullet t'))$$

De la même faon, t est dite *désensibilisée* par le tir de t' à partir du marquage M, et nous notons $disabled(t, M, t')$, si t est sensibilisée par M mais ne l'est plus par $M - {}^\bullet t'$.

Par extension, nous notons $enabled(M, t')$ (respectivement $disabled(M, t')$) l'ensemble des transitions nouvellement sensibilisées (respectivement désensibilisées) par le tir de t' depuis M.

Nous définissons la sémantique des réseaux de Petri T-temporels sous la forme d'un système de transitions temporisé (TTS).

Definition 10 (Sémantique d'un TPN). *La sémantique d'un réseau de Petri T-temporel $\mathcal{N}$ est définie sous la forme d'un système de transitions temporisé $\mathcal{S}_{\mathcal{N}} = (Q, q_0, \Sigma, \rightarrow)$ tel que :*

- $Q = \mathbb{N}^P \times (\mathbb{R}^+)^T$;
- $q_0 = (M_0, \mathbf{0})$;
- $\Sigma = T$;
- $\rightarrow \in Q \times (T \cup \mathbb{R}) \times Q$ *est la relation de transition incluant des transitions continues et des transitions discrètes :*
  - *la relation de transition* continue *est définie* $\forall d \in \mathbb{R}^+$ *par :*

$$(M, \nu) \xrightarrow{d} (M, \nu') \text{ ssi } \begin{cases} \nu' = \nu + d, \\ \forall t_k \in T, M \geq {}^\bullet t_k \Rightarrow \nu'(t_k) \leq \beta(t_k). \end{cases}$$

  - *la relation de transition* discrète *est définie* $\forall t_i \in T$ *par :*

$$(M, \nu) \xrightarrow{t_i} (M', \nu') \text{ ssi } \begin{cases} M \geq {}^\bullet t_i, \\ \alpha(t_i) \leq \nu(t_i) \leq \beta(t_i), \\ M' = M - {}^\bullet t_i + t_i^\bullet, \\ \forall t_k, \nu'(t_k) = \begin{cases} 0 \text{ si } \uparrow enabled(t_k, M, t_i), \\ \nu(t_k) \text{ sinon.} \end{cases} \end{cases}$$

Quand une transition discrète est possible depuis l'état $s = (M, \nu)$ (de la sémantique) du réseau, nous dirons que la transition correspondante est *tirable*. Formellement :

Definition 11 (Transition tirable). *Soit $s = (M, \nu)$ un état de la sémantique d'un réseau de Petri T-temporel. Une transition t est dite* tirable *dans s si $M \geq {}^\bullet t$ et $\alpha(t) \leq \nu(t) \leq \beta(t)$.*

Nous pouvons remarquer que, dans cette sémantique, si une place contient plusieurs jetons pouvant sensibiliser une ou des transitions sortantes, seul le nombre de jetons indiqué sur l'arc sera considéré. Les autres seront utilisés pour les tirs suivants éventuels des transitions sortantes. C'est une hypothèse *monoserveur*. Cette sémantique est illustrée par la figure 2 : supposons que t_1 est tirable. Sur la figure 2a, t_1 et t_2 sont sensibilisées par le marquage M et par le marquage $M' = M - {}^\bullet t_1 + t_1^\bullet$ mais pas par $M - {}^\bullet t_1$. Les transitions t_1 et t_2 sont donc *nouvellement* sensibilisées par le tir de t_1.

Sur la figure 2b, t_1 et t_2 sont sensibilisées par le marquage M et par le marquage $M' = M - {}^\bullet t_1 + t_1^\bullet$, mais aussi par $M - {}^\bullet t_1$. Dans ce cas, t_1 est *nouvellement* sensibilisée par le tir de t_1 (car elle est la transition tirée) mais pas t_2: t_2 *reste* sensibilisée.

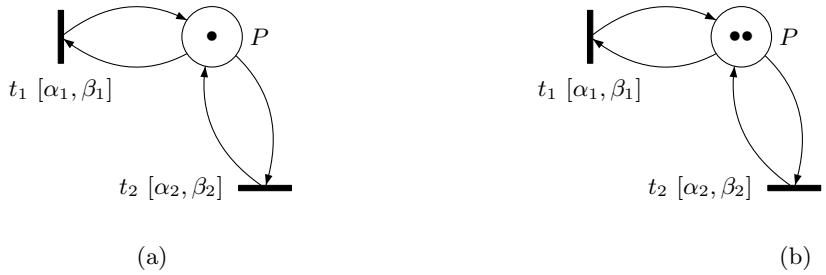


Fig. 2. Exemple de transitions nouvellement sensibilisées.

On peut complexifier la sémantique de faon à ce qu'une transition puisse être sensibilisée plusieurs fois simultanément [11]. On considère alors une hypothèse *multiserveur* et le nombre d'horloges du réseau devient potentiellement infini.

Definition 12 (Réseau de Petri temporel étiqueté). *Un réseau Petri temporel étiqueté (*Labelled Time Petri Net*)* $\mathcal{N}$ *est un 9-uplet* $\mathcal{N} = (P, T, \Sigma_\varepsilon, {}^\bullet(.), (.)^\bullet, \alpha, \beta, M_0, \Lambda)$ *tel que :* $(P, T, {}^\bullet(.), (.)^\bullet, , \alpha, \beta, M_0)$ *est un TPN,* Σ *est un ensemble fini d'*actions *et* $\Lambda : T \to \Sigma_\varepsilon$ *est une fonction de nommage. Un TPN non étiqueté est donc un TPN étiqueté tel que* $\Sigma = T$ *et* $\Lambda(t) = t$ *pour tout* $t \in T$*.*

Classiquement (et c'est la convention que nous utilisons dans ce livre), par TPN, nous désignons implicitement des TPN étiquetés avec $\Sigma = T$ et $\Lambda(t) = t$.

Enfin nous pouvons étendre très naturellement les définitions 9, 10 et 12 pour considérer aussi des contraintes strictes dans les intervalles de tir des transitions.

3.3 Décidabilité des problèmes classiques

Étant donné un réseau de Petri T-temporel $\mathcal{N}$ et $\mathcal{S}_\mathcal{N}$ sa sémantique, plusieurs problèmes peuvent être étudiés, notamment :

- l'*accessibilité de marquages* : étant donné un marquage M, $\exists (M', \nu') \in \mathcal{S}_\mathcal{N}, M = M'$;
- la *bornitude* : $\exists b \in \mathbb{N}, \forall (M, \nu) \in \mathcal{S}_\mathcal{N}, \forall p \in P, M(p) \leq b$;
- la *k-bornitude* : étant donné $k \in \mathbb{N}$, $\forall (M, \nu) \in \mathcal{S}_\mathcal{N}, \forall p \in P, M(p) \leq k$;
- l'*accessibilité d'états* : étant donné un état s, $s \in \mathcal{S}_\mathcal{N}$;
- la *vivacité* : $\forall t \in T, \forall s \in \mathcal{S}_\mathcal{N}, \exists \sigma \in T^*, s' \in \mathcal{S}_\mathcal{N}, s \xrightarrow{\sigma.t} s'$.

L'accessibilité de marquages est un problème indécidable [12], ce qui implique que la bornitude, l'accessibilité d'états et la vivacité sont également des problèmes indécidables. En revanche, la k-bornitude est décidable et peut être décidée par le calcul d'une abstraction finie de l'espace d'états telle que le graphe des classes d'états de [8].

Dans le cas des réseaux de Petri T-temporels bornés, l'accessibilité de marquages, l'accessibilité d'états et la vivacité sont décidables.

A part la bornitude et la k-bornitude, les mêmes problèmes se posent sur les automates temporisés. A cela on peut ajouter un autre problème important

qui est celui de la décidabilité du model-checking en particulier de propriétés exprimées dans la logique temporelle quantitative TCTL. Enfin, du point de vue langage temporisé, les problèmes classiques sont le *langage vide*, *l'universalité* (équivalent au problème de l'inclusion de langage) et les propriétés de fermeture.

Nous résumons les résultats de décidabilité dans le tableau suivant :

Problème	**automates temporisés**	**B-TPN (TPN bornés)**
Accessibilité langage vide	Décidable [2]	Décidable [8]
Universalité inclusion de langage	Indécidable [2]	Indécidable [13]
Propriétés de fermeture	Clos par $\cap, \cup$ mais pas par compl. [2]	Clos par $\cap, \cup$ mais pas par compl. [13]
Model-checking de TCTL	Décidable [2]	Décidable [14, 15]

3.4 Expressivité comparée de différentes classes de modèles temporisés

Definition 13 (Simulation temporelle forte). *Soient deux systèmes de transitions temporisés* $S_1 = (Q_1, Q_0^1, \Sigma, \longrightarrow_1)$ *et* $S_2 = (Q_2, Q_0^2, \Sigma, \longrightarrow_2)$ *sur* Σ *et* $\sqsubseteq$ *une relation binaire sur* $Q_1 \times Q_2$. *Nous écrivons* $s \sqsubseteq s'$ *pour* $(s, s') \in \sqsubseteq$. $\sqsubseteq$ *est une* relation de simulation temporelle forte *de* S_1 *par* S_2 *si les trois assertions suivantes sont vérifiées :*

1. *si* $s_1 \in Q_0^1$, *alors il existe* $s_2 \in Q_0^2$ *tel que* $s_1 \sqsubseteq s_2$;
2. *si* $s_1 \xrightarrow{d}_1 s_1'$ *avec* $d \in \mathbb{R}_{\geq 0}$ *et* $s_1 \sqsubseteq s_2$ *alors il existe* $s_2 \xrightarrow{d}_2 s_2'$ *tel que* $s_1' \sqsubseteq s_2'$;
3. *si* $s_1 \xrightarrow{a}_1 s_1'$ *avec* $a \in \Sigma$ *et* $s_1 \sqsubseteq s_2$ *alors il existe* $s_2 \xrightarrow{a}_2 s_2'$ *tel que* $s_1' \sqsubseteq s_2'$.

Un TTS S_2 simule fortement S_1 *si il existe une relation de simulation forte de* S_1 *par* S_2. *Nous notons alors* $S_1 \preceq_{\mathcal{S}} S_2$.

Definition 14 (Simulation temporelle faible). *Soient deux systèmes de transitions temporisés* $S_1 = (Q_1, Q_0^1, \Sigma_\varepsilon, \longrightarrow_1)$ *et* $S_2 = (Q_2, Q_0^2, \Sigma_\varepsilon, \longrightarrow_2)$ *sur* Σ_ε *et* $\sqsubseteq$ *une relation binaire sur* $Q_1 \times Q_2$. $\sqsubseteq$ *est une* relation de simulation temporelle faible *de* S_1 *par* S_2 *si c'est une relation de simulation temporelle forte entre ces deux TTS* ε*-abstraits. Un TTS* S_2 simule faiblement S_1 *si il existe une relation de simulation faible de* S_1 *par* S_2. *Nous notons alors* $S_1 \preceq_{\mathcal{W}} S_2$.

Definition 15 (Bisimulation temporelle). *Deux TTS* S_1 *et* S_2 *sont en relation de bisimulation temporelle forte (respectivement faible) si il existe une relation de simulation forte (respectivement faible)* $\sqsubseteq$ *de* S_1 *par* S_2 *et si* $\sqsubseteq^{-1}$ *est aussi une relation de simulation forte*[2] *(respectivement faible) de* S_2 *par* S_1. *Nous notons alors* $S_1 \approx_{\mathcal{S}} S_2$ *(respectivement* $S_1 \approx_{\mathcal{W}} S_2$*).*

[2] $S_2 \sqsubseteq^{-1} S_1$ ssi $S_1 \sqsubseteq S_2$.

Expressivité des modèles temporisés Soient $\mathcal{C}$ et $\mathcal{C}'$ deux classes de modèles temporisés.

Definition 16 (Expressivité en termes d'acceptation de langage temporisé). *La classe $\mathcal{C}$ est* plus expressive *que la classe $\mathcal{C}'$ en termes d'acceptation de langage temporisé si pour tout $B' \in \mathcal{C}'$ il existe $B \in \mathcal{C}$ tel que $\mathcal{L}(B) = \mathcal{L}(B')$. Nous notons alors $\mathcal{C}' \subseteq_{\mathcal{L}} \mathcal{C}$. De plus si il existe $B \in \mathcal{C}$ tel qu'il n'existe pas $B' \in \mathcal{C}'$ avec $\mathcal{L}(B) = \mathcal{L}(B')$, alors $\mathcal{C}' \subset_{\mathcal{L}} \mathcal{C}$ (strictement moins expressive). Si on a $\mathcal{C}' \subseteq_{\mathcal{L}} \mathcal{C}$ et $\mathcal{C} \subseteq_{\mathcal{L}} \mathcal{C}'$ alors $\mathcal{C}$ et $\mathcal{C}'$ sont d'expressivité égale en termes d'acceptation de langage temporisé, et nous notons $\mathcal{C} =_{\mathcal{L}} \mathcal{C}'$.*

Definition 17 (Expressivité en termes de bisimulation temporelle). *La classe $\mathcal{C}$ est* plus expressive *que la classe $\mathcal{C}'$ en termes de bisimulation temporelle forte (respectivement faible) si pour tout $B' \in \mathcal{C}'$ il existe $B \in \mathcal{C}$ tel que $B \approx_{\mathcal{S}} B'$ (respectivement $B \approx_{\mathcal{W}} B'$). Nous notons alors $\mathcal{C}' \leq_{\mathcal{S}} \mathcal{C}$ (respectivement $\mathcal{C}' \leq_{\mathcal{W}} \mathcal{C}$). De plus, si il existe $B \in \mathcal{C}$ tel qu'il n'existe pas $B' \in \mathcal{C}'$ avec $B \approx_{\mathcal{S}} B'$ (respectivement $B \approx_{\mathcal{W}} B'$), alors $\mathcal{C}' <_{\mathcal{S}} \mathcal{C}$ (respectivement $\mathcal{C}' <_{\mathcal{W}} \mathcal{C}$). Si on a $\mathcal{C}' \leq_{\mathcal{S}} \mathcal{C}$ et $\mathcal{C} \leq_{\mathcal{S}} \mathcal{C}'$ (respectivement $\leq_{\mathcal{W}}$) alors $\mathcal{C}$ et $\mathcal{C}'$ sont d'expressivité égale en termes de bisimulation temporelle forte (respectivement faible) et nous notons $\mathcal{C} \approx_{\mathcal{S}} \mathcal{C}'$ (respectivement $\mathcal{C} \approx_{\mathcal{W}} \mathcal{C}'$).*

Expressivité comparée de différentes classes de TPN Nous donnons ici une comparaison de l'expressivité de différentes classes de réseaux de Petri temporels en termes de bisimulation temporelles que l'on peut trouver dans [16, 17].

Les classes de modèle considérées sont:

- réseaux de Petri temporels avec temps sur les places (*P-TPN*), les arcs (*A-TPN*) ou les transitions (*T-TPN*).
- sémantiques forte ($\overline{\textit{P-TPN}}$, $\overline{\textit{A-TPN}}$ et $\overline{\textit{T-TPN}}$) ou faible ($\underline{\textit{P-TPN}}$, $\underline{\textit{A-TPN}}$ et $\underline{\textit{T-TPN}}$)
- réseaux saufs (1-bornés)
- contraintes temporelles quelconques (larges ou strictes)
- réseaux avec étiquettes et *epsilon*-transitions

Cependant les *T-TPN* sont beaucoup plus faciles à utiliser que les *A-TPN* pour modéliser les synchronisations

Expressivité : $\overline{\textit{T-TPN}}$ *vs* *TA*

Classes de TPN et de TA Nous notons $\mathcal{TA}$ la classe des TA (automates temporisés) et $\overline{\textit{T-TPN}}$ la classe des TPN (réseaux de Petri temporels) en semantique forte avec le temps associé aux transitions.

Definition 18 (Classes de réseaux de Petri temporels bornés). *Soit k-$\overline{\textit{T-TPN}}$, l'ensemble des TPN k-bornés.*
Soit 1-$\overline{\textit{T-TPN}}$, l'ensemble des TPN 1-bornés c'est-à-dire saufs.
Soit B-$\overline{\textit{T-TPN}} = \{\mathcal{N} \mid \exists k \geq 0 \mid \mathcal{N} \in k\text{-}\overline{\textit{T-TPN}}\}$, l'ensemble des TPN bornés.

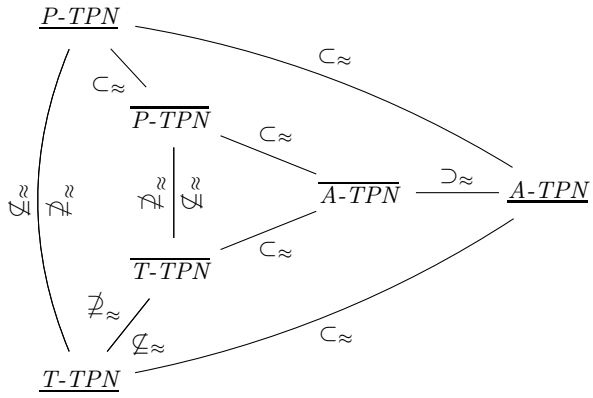


Fig. 3. Expressivité comparée des reseaux de Petri temporels.

La comparaison de l'expressivité des réseaux de Petri temporels par rapport à celle des automates temporisés est étudiée dans [13] et peut être schématisée par la figure suivante :

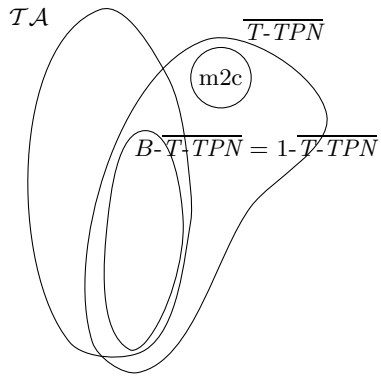


Fig. 4. Expressivité comparée des reseaux de Petri temporels et des automates temporisés.

4 Conclusion

Nous avons proposé une comparaison de l'expressivité de differentes classes de modèles temporisé et en particulier des automates temporisés et des réseaux de Petri temporels. Notons qu'en terme d'acceptation de langages temporisés, en revanche, les réseaux de Petri temporels bornés et les automates temporisés sont d'expressivité égale [13].

Enfin le lecteur trouvera des informations complémentaires dans [18–20]

References

1. Milner, R.: Communication and Concurrency. Prentice Hall International (1989)
2. Alur, R., Dill, D.: A theory of timed automata. Theoretical Computer Science **126**(2) (1994) 183–235
3. Ramchandani, C.: Analysis of asynchronous concurrent systems by timed Petri nets. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA (1974) Project MAC Report MAC-TR-120.
4. Merlin, P.: A study of the recoverability of computing systems. PhD thesis, Department of Information and Computer Science, Univ. of California, Irvine (1974)
5. Yi, W.: Ccs + time = an interleaving model for real time systems. In: ICALP. (1991) 217–228
6. Henzinger, T., Nicollin, X., Sifakis, J., Yovine, S.: Symbolic model checking for real-time systems. Information and Computation **111**(2) (1994) 193–244
7. Pezze, M., Toung, M.: Time Petri nets: A primer introduction. Tutorial presented at the Multi-Workshop on Formal Methods in Performance Evaluation and Applications, Zaragoza, Spain (september 1999)
8. Berthomieu, B., Diaz, M.: Modeling and verification of time dependent systems using time Petri nets. IEEE trans. on software engineering **17**(3) (1991) 259–273
9. Khansa, W., Denat, J.P., Collart-Dutilleul, S.: P-Time Petri Nets for manufacturing systems. In: International Workshop on Discrete Event Systems, WODES'96, Edinburgh (U.K.) (august 1996) 94–102
10. de Frutos Escrig, D., Ruiz, V.V., Alonso, O.M.: Decidability of properties of timed-arc Petri nets. In: ICATPN'00. Volume 1825 of Lecture Notes in Computer Science., Aarhus, Denmark (june 2000) 187–206
11. Berthomieu, B.: La méthode des classes d'états pour l'analyse des réseaux temporels. In: Modélisation des Systèmes Réactifs (MSR'01), Toulouse, France, Hermes (october 2001) 275–290
12. Jones, N., Landweber, L., Lien, Y.: Complexity of some problems in Petri nets. Theoretical Computer Science 4 (1977) 277–299
13. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: Comparison of the expressiveness of timed automata and time Petri nets. In: 3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2005). Lecture Notes in Computer Science, Uppsala, Sweden, Springer (sep 2005)
14. Cassez, F., Roux, O.H.: Structural translation from time petri nets to timed automata. In: The 4th International Workshop on Automated Verification of Critical Systems (AVoCS 2004), London, United Kingdom (september 2004)
15. Cassez, F., Roux, O.H.: Structural translation from Time Petri Nets to Timed Automata – Model-Checking Time Petri Nets via Timed Automata. The journal of Systems and Software **79**(10) (2006) 1456–1468
16. Boyer, M., Roux, O.H.: Comparison of the expressiveness of arc, place and transition time Petri nets. In: ICATPN'07. Volume 4546 of LNCS. (2007) 63–82
17. Boyer, M., Roux, O.H.: On the compared expressiveness of arc, place and transition time Petri nets. Fundamenta Informaticae **88**(3) (2008) 225–249
18. Berthomieu, B., Peres, F., Vernadat, F.: Bridging the gap between timed automata and bounded time petri nets. In: FORMATS'06. Volume 4202 of LNCS. (2006)
19. Srba, J.: Comparing the expressiveness of timed automata and timed extensions of petri nets. In: FORMATS'08. Volume 5215 of LNCS. (2008)
20. Bérard, B., Cassez, F., Haddad, S., Lime, D., Roux, O.H.: When are timed automata weakly timed bisimilar to time petri nets? Theoretical Computer Science **403**(2-3) (2008) 202–220

Dépliages de réseaux d'automates et application à la supervision

Claude Jard

ENS Cachan, IRISA, Université Européenne de Bretagne
Campus de Ker-Lann, F-35170 Bruz, France
Claude.Jard@bretagne.ens-cachan.fr

Résumé Cet article présente des techniques de dépliages de réseaux d'automates. Elles permettent de mettre en évidence les relations causales d'ordre partiel existant entre les événements d'un modèle des comportements des systèmes répartis. Elles sont donc particulièrement adaptées pour traiter la question de la supervision. Etant donné une séquence d'actions observées durant l'exécution d'une application répartie, il s'agit, à l'aide d'un modèle du système, de déterminer les transitions du modèle ayant produit ces actions et d'inférer les relations de causalité ou d'indépendance qui les relient. Nous explorons notamment le contexte original de la supervision de systèmes modélisés par des réseaux d'automates temporisés et montrons comment on peut aussi inférer les dates possibles d'occurrence des actions à l'aide de contraintes symboliques.

1 Introduction

La notion de dépliage pour les modèles comportementaux des systèmes répartis [12] est apparue pour doter ces modèles d'une sémantique dite "d'ordre partiel" [8,9]. Par opposition aux sémantiques séquentielles habituelles données en terme de systèmes de transitions [2] et traditionnellement utilisées dans les activités de vérification, de contrôle et de test, une sémantique d'ordre partiel définit les exécutions comme des ordres partiels sur des événements. Elle permet donc de repérer dans une exécution les dépendances causales entre événements et par complément d'identifier la production d'événements indépendants ou parallèles ("concurrent" en anglais). La superposition des ensembles d'événements pour l'ensemble des exécutions possibles peut constituer ce que l'on appelle une *structure d'événements* [13] ou encore dépliage. Une structure d'événements est un ensemble d'événements structuré par une relation d'ordre partiel et une relation de conflit permettant de retrouver les exécutions valides. Cette notion a essentiellement été développée dans le cadre du modèle des réseaux de Petri [14]. Nous la présentons ici dans le cadre de modèles de réseaux d'automates [2,9].

La question de la supervision est un problème pratique qui se pose pour suivre en ligne l'activité de systèmes réels et corréler entre eux les événements que produit le système [4,15]. A cette fin, des sondes sont implantées dans le système et fabriquent des événements portant des informations utiles pour le superviseur. Ces événements sont ensuite acheminés (souvent à travers un réseau

dans le cas des systèmes répartis) pour être traités en séquence par un module appelé le superviseur. On considère que l'ordre d'arrivée des événements devant le superviseur n'est pas significatif puisqu'il est en général le résultat d'un processus asynchrone de collecte. Le superviseur doit alors reconstruire les dépendances qui ne peuvent pas être retrouvées directement à partir des informations locales associées aux événements. Pour celà, on considère une approche "fondée modèle" dans laquelle le superviseur utilise un modèle de comportement du système. Ce modèle comporte des transitions étiquetées par les informations qui peuvent être observées.

Notre approche est résumée dans la figure 1 suivante. La séquence observée est une suite de symboles. Le superviseur produit un ensemble d'explications puisque plusieurs trajectoires du modèle peuvent produire le même ensemble de symboles. Une explication est un ordre partiel d'événements. Un événement correspond à l'exécution d'une transition du modèle, transition étiquetée par le symbole observé qu'il faut expliquer. Le superviseur infère donc à partir du modèle les dépendances possibles entre les symboles observés. Nous étendons l'approche avec des contraintes temporelles [11] à l'intérieur du modèle (le modèle de réseaux d'automates temporisés), ce qui permet en outre d'inférer les dates possibles de production des symboles observés.

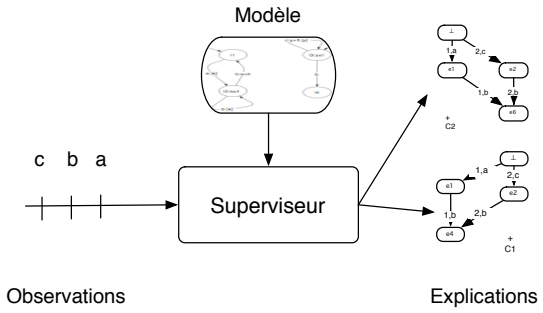


Fig. 1. La question de supervision.

Nous avons développé ces dernières années une approche de la supervision fondée sur le dépliage de réseaux de Petri [14], essentiellement dans un cadre non temporisé [3], avec une motivation particulière pour la répartition du calcul lui-même de la supervision [10], et plus récemment en abordant les questions de contraintes symboliques [6,7].

La principale nouveauté apportée par l'article est de reprendre la démarche et de la présenter de façon unifiée à travers le modèle des réseaux d'automates temporisés [1]. L'utilisation explicite des réseaux est originale car bien différente de l'approche habituelle dans laquelle on calcule d'abord l'automate unique correspondant à la mise en réseaux des automates composants avant de procéder à

l'analyse. La complexité algorithmique est plus grande, mais est largement compensée par la compacité en mémoire de la représentation du parallélisme (on refuse de calculer l'ensemble des entrelacements d'actions dues au parallélisme). Surtout, c'est le moyen naturel d'expliciter les relations de causalité qui nous semblent centrales pour l'activité de supervision et de diagnostic.

Le reste de l'article est structuré ainsi. On commence par présenter le concept de dépliage à travers le modèle bien connu des automates finis, puis des réseaux d'automates finis. On décrit alors la question de la supervision et une méthode permettant de produire les explications d'une séquence d'observations pour un modèle non temporisé. Le modèle temporisé est ensuite introduit et une méthode pour produire les explications temporisées est décrite.

2 Automates finis

Définition 1 (Automate fini) *Un* automate fini A *est défini par un quadruplet* $A = (Q, q_0, \Sigma, T)$*, où* Q *est un ensemble fini d'états,* $q_0 \in Q$ *l'état initial,* Σ *l'alphabet (un ensemble fini de symboles) et* $T \subseteq Q \times \Sigma \times Q$ *l'ensemble des transitions (ou arcs).*

Un exemple de description graphique est donné dans la figure 2.

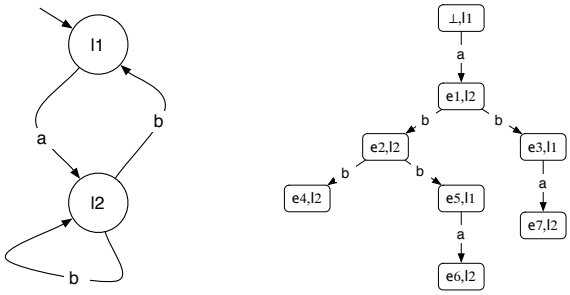


Fig. 2. Un automate fini (à gauche). l_1 est l'état initial, $\Sigma = \{a, b\}$. A droite : un préfixe de son dépliage.

Les exécutions d'un tel automate sont données par les chemins étiquetés du graphe commençant dans l'état initial. Elles peuvent être formellement définies en utilisant la notion de dépliage. Une transition sera notée par $t = (\alpha(t), \lambda(t), \beta(t))$.

Le *dépliage* d'un automate A, noté $\mathcal{U}(A)$ est donné par un ensemble d'événements. Un *événement* représente l'occurrence d'une transition. Il est défini par un couple $e = (\pi(e), \tau(e))$, où $\pi(e)$ est l'événement qui précède e dans le dépliage, et $\tau(e)$ est la transition considérée. Il est pratique de considérer un événement initial fictif $\perp$, avec $\beta(\tau(\perp)) = q_0$.

$\mathcal{U}(A)$ peut être défini inductivement ainsi.

Définition 2 (Dépliage d'un simple automate) *Etant donné un automate fini $A = (Q, q_0, \Sigma, T)$, le dépliage de A, noté $\mathcal{U}(A)$, est le plus petit ensemble tel que :*

- $\bot \in \mathcal{U}(A)$,
- $\{\exists \pi(e) \in \mathcal{U}(A) \wedge \exists t \in T \wedge \alpha(t) = \beta(\tau(e))\} \Rightarrow (e, t) \in \mathcal{U}(A)$.

Soient deux événements e et e' de $\mathcal{U}(A)$, e précède immédiatement e' (noté par $e \rightarrow e'$) si $\pi(e') = e$. La *causalité* entre événements est définie comme la clôture transitive de la relation $\rightarrow$ (notée $\rightarrow^*$). Pour un événement e, son ensemble de prédécesseurs causaux est noté $\downarrow e = \{f \mid f \rightarrow^* e\}$. La notation est étendue aux ensembles : $\downarrow E = \bigcup_{e \in E} \downarrow e$.

Les dépliages sont en général des ensembles infinis (dès qu'il existe une boucle dans l'automate). La figure 2 à droite montre un sous-ensemble fini clos par précédence causale (ce que l'on appelle un préfixe) du dépliage de l'exemple de l'automate de gauche. Graphiquement, un événement $(\pi(e), \tau(e))$ est représenté en traçant un arc étiqueté par $\lambda(\tau(e))$, du nœud $\pi(e)$ au nœud e. Les nœuds sont dessinés comme des rectangles contenant le nom de l'événement et l'état atteint $\beta(\tau(e))$.

Les préfixes des dépliages des automates finis sont des arbres de degré borné.

3 Réseau d'automates

Nous suivons la même approche que précédemment pour définir la notion de dépliage d'un réseau.

Définition 3 (Réseau d'automates finis) *Un* réseau d'automates finis *est un ensemble $\{A_1, ..., A_n\}$ de n automates finis avec $A_i = (Q_i, q_{0i}, \Sigma_i, T_i)$. Un* état global *du réseau est un n-uplet $Q_1 \times Q_2 ... \times Q_n$. L'état global initial est $(q_{01}, ..., q_{0n})$. L'activité des automates est synchronisée sur les transitions de même étiquette. On définit l'ensemble des synchronisations $Sync$ comme l'ensemble des vecteurs de transitions $(t_1, ..., t_n) \in (T_1 \cup \{\epsilon\}) \times ... \times (T_n \cup \{\epsilon\})$ tels que $(t_1, ..., t_n) \neq (\epsilon, ..., \epsilon)$ et qu'il existe a tel que $\forall t_i \neq \epsilon, \lambda(t_i) = a \wedge \forall t_i = \epsilon, a \notin \Sigma_i$. Pour une transition globale $t \in Sync$, on notera $\lambda(t)$ l'action a.*

Un exemple de tel réseau est donné dans la figure 3.

Comme précédemment le *dépliage* d'un réseau N, noté $\mathcal{U}(N)$ est donné par un ensemble d'événements. Un événement est un vecteur $e = (e_1, ..., e_n)$ où $e_i = (\pi_i(e), \tau_i(e))$, dans lequel $\pi_i(e)$ désigne le dernier événement qui a concerné l'automate A_i, et $(\tau_1(e), ..., \tau_n(e)) \in Sync$. Dans le cas où l'automate A_i n'est pas concerné, on définit $\pi_i(e) = \epsilon$ ne désignant aucun événement du dépliage. L'événement initial $\bot$ est tel que $\beta(\tau_i(\bot)) = q_{0i}$.

Etant donnés deux événements e et e', e précède immédiatement e' sur l'automate A_i (noté par $e \rightarrow_i e'$) si $\pi_i(e') = e$. Un ensemble d'événements E est en *conflit* ssi $\exists e \neq e' \in E, i \in [1, n]$ tel que $\pi_i(e) = \pi_i(e')$. Du point de vue de la causalité, chaque événement doit avoir un passé causal sans conflit sinon cela voudrait dire qu'il contient deux événements exclusifs (résultant d'un choix local dans un automate).

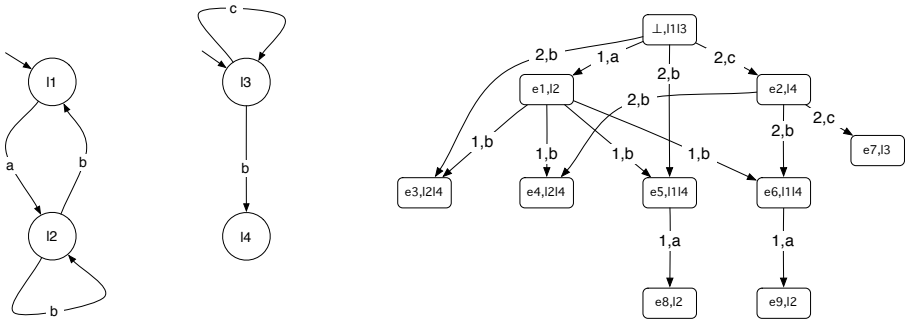


Fig. 3. Un réseau d'automates à gauche. $Sync = \{(a,\epsilon),(b,b),(c,\epsilon),(b,b)\}$. A droite : un préfixe de son dépliage.

Définition 4 (Dépliage d'un réseau) *Etant donné un réseau N, $\mathcal{U}(N)$ est le plus petit ensemble satisfaisant :*

- $\bot \in \mathcal{U}(N)$
- $$\left\{ \begin{array}{l} (\tau_1(e),...,\tau_n(e)) \in Sync \\ \forall i \in [1,n] \left\{ \begin{array}{l} \tau_i(e) = \epsilon \Rightarrow \pi_i(e) = \epsilon \\ \tau_i(e) \neq \epsilon \Rightarrow \left\{ \begin{array}{l} \pi_i(e) \in \mathcal{U}(N) \\ \alpha(\tau_i(e)) = \beta(\tau_i(\pi_i(e))) \end{array} \right. \end{array} \right. \\ \downarrow e \textit{ est sans conflit} \end{array} \right\} \Rightarrow e \in \mathcal{U}(N)$$

La figure 3 à droite montre un préfixe du dépliage du réseau de gauche. Graphiquement, un événement e est dessiné en traçant un arc du nœud $\pi_i(e)$ au nœud e, étiqueté par $i, \lambda(\tau_i(e))$. Chaque nœud est représenté par un rectangle, étiqueté avec le nom de l'événement e et les états locaux qui ont été positionnés par la transition.

Les préfixes des dépliages des réseaux sont des graphes acycliques dont le degré peut être non borné. La définition inductive donne directement un algorithme dans lequel les événements obéissant à la définition sont placés un par un dans le dépliage si ils n'y sont pas déjà.

4 Supervision par réseau d'automates

La question de la supervision s'exprime ainsi. On considère une séquence d'observations $\sigma \in \Sigma^*$. Le problème est de construire toutes les exécutions du réseau "expliquant" la séquence. On voit donc qu'il s'agit de construire un préfixe du dépliage, contraint à "coller" aux observations. La séquence d'observations étant finie, le dépliage le sera aussi.

L'idée est que l'ordre donné par la séquence d'observations σ n'est pas forcément pertinent et est juste le résultat du procédé d'observation à l'intérieur du système réparti que l'on observe. C'est le rôle du superviseur, en utilisant le modèle, de proposer les relations de causalité possibles entre les observations. Il est

clair par contre que les actions du même type sont totalement ordonnées puisque, soit cela correspond à une action locale à un automate, soit il s'agit d'une synchronisation. La bonne méthode pour guider le dépliage est donc de suivre la fonction de Parrikh de la séquence σ. La fonction de Parrikh $\varpi : \Sigma^* \rightarrow \mathbb{N}^{|\Sigma|}$ compte le nombre d'occurrences de chaque symbole dans la séquence. On va étendre l'information portée par les événements e par le vecteur de Parrikh $\varsigma(e)$ de la séquence reconnue par l'ensemble des prédécesseurs causaux de e. Ceci en exigeant de ne jamais dépasser le vecteur de Parrikh de la séquence σ. Pour une action $a \in \Sigma$, on notera χ_a le vecteur de Parrikh ayant toutes ses composantes à 0, sauf celle correspondante à l'action a, égale à 1. Les événements sont notés $e = ((\pi_i, \tau_i)_{i\in[1,n]}, \varsigma(e))$.

Le dépliage guidé par l'observation, noté $\mathcal{E}(N, \sigma)$ peut donc être défini ainsi :

Définition 5 (Supervision d'un réseau) *Etant donné un réseau N et une séquence d'observations σ, $\mathcal{E}(N, \sigma)$ est le plus petit ensemble satisfaisant :*

- $\perp \in \mathcal{E}(N, \sigma)$ *avec* $\varsigma(\perp) = 0$
- $$\left\{ \begin{array}{l} t = (\tau_1(e), ..., \tau_n(e)) \in Sync \\ \forall i \in [1, n] \left\{ \begin{array}{l} \tau_i(e) = \epsilon \Rightarrow \pi_i(e) = \epsilon \\ \tau_i(e) \neq \epsilon \Rightarrow \left\{ \begin{array}{l} \pi_i(e) \in \mathcal{E}(N) \\ \alpha(\tau_i(e)) = \beta(\tau_i(\pi_i(e))) \end{array} \right. \end{array} \right. \\ \downarrow e \textit{ est sans conflit} \\ \varsigma(e) = \sum_{f\in\downarrow e\setminus\{e\}} \varsigma(f) + \chi_{\lambda(t)} \leq \varpi(\sigma) \end{array} \right\} \Rightarrow e \in \mathcal{E}(N, \sigma)$$

La figure 4 montre la supervision obtenue pour la séquence d'observations abc. Le résultat de cette supervision représente en fait les deux explications suivantes : les actions a et c ont été effectuées indépendamment puis l'action b a été effectuée, par le tir de l'une ou l'autre des transitions correspondantes dans le premier automate. Ces explications sont obtenues en :

- extrayant les exécutions du dépliage. Une exécution étant définie par un sous-ensemble E du dépliage $\mathcal{E}(N, \sigma)$ qui est clos par causalité ($\downarrow E = E$) et sans conflit ;
- demandant que ces exécutions expliquent toute la séquence d'observations : $\sum_{e\in E} \varsigma(e) = \varpi(\sigma)$.

Non seulement donc on infère les trajectoires des automates, mais aussi les liens possibles de causalité entre les observations. C'est qui donne toute sa signification à la supervision par modèle, ajoutant de l'information à la séquence observée.

5 Automate temporisé

On explore maintenant un modèle plus riche dans lequel des contraintes sont ajoutées sur l'écoulement du temps. Ceci permet de restreindre les comportements du modèle non temporisé sous-jacent. A l'automate fini sous-jacent, on rajoute donc des horloges, des invariants sur les états, des gardes et des remises à zéro sur les transitions [1].

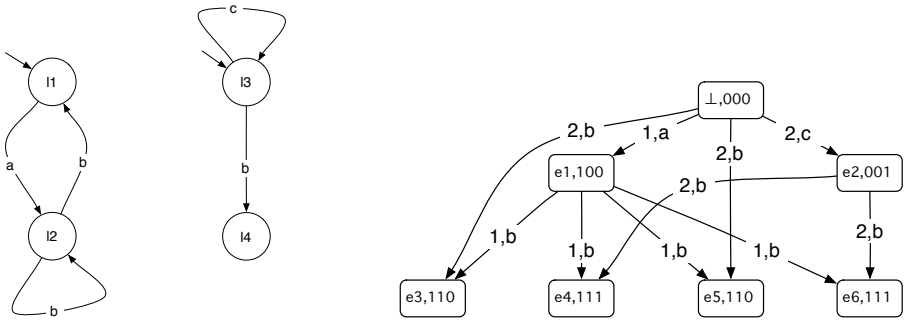


Fig. 4. Un réseau d'automates à gauche et son dépliage contraint par la séquence d'observations abc à droite. Le vecteur $\varsigma(e)$ est montré sur chaque événement e.

Définition 6 (Automate temporisé) *Un* automate temporisé *(AT) est un sextuplet* $\mathcal{A} = (Q, q_0, \Sigma, T, X, I)$ *où*

- Q *est un ensemble fini de positions,* $q_0 \in Q$ *est la position initiale ;*
- Σ *est l'alphabet des actions ;*
- X *est un ensemble fini d'horloges ;*
- $T \subseteq Q \times \mathcal{C}(X) \times \Sigma \times 2^X \times Q$ *est un ensemble fini de transitions. Nous désignerons aussi une transition* $t = (\alpha(t), \gamma(t), \lambda(t), \rho(t), \beta(t))$*, où* $\gamma(t) \in \mathcal{C}(X)$ *est une* garde*,* $\lambda(t) \in \Sigma$ *une action, et* $\rho(t) \subseteq X$ *un ensemble de* remises à zéro (raz)*.* $\mathcal{C}(X)$ *est l'ensemble des conjonctions de contraintes de la forme* $x \bowtie c$ *où* $x \in X$*,* $c \in \mathbb{R}$ *et* $\bowtie \in \{=, <, \leq, >, \geq\}$*.* $\mathcal{C}_<(X)$ *est l'ensemble des conjonctions de* $(x \leq c)$ *ou* $(x < c)$ *;*
- $I : Q \to \mathcal{C}_<(X)$ *sont les invariants.*

Comme précédemment, on peut former des réseaux de tels automates en considérant une synchronisation par action partagée.

Définition 7 (Réseau d'automates temporisés) *Un* réseau d'automates temporisés *(RAT) est un ensemble* $\{A_1, ..., A_n\}$ *de* n *automates temporisés avec* $A_i = (Q_i, q_{0i}, \Sigma_i, T_i, X_i, I_i)$*. On fait l'hypothèse de non-partage des horloges (*$\forall i \neq j, X_i \cap X_j = \emptyset$*). L'ensemble des synchronisations* $Sync$ *est défini comme dans le cas non temporisé.*

La figure 5 montre un tel RAT.

La sémantique opérationnelle d'un tel système est définie ainsi. L'état global du système est composé de l'état local de chaque automate et d'une date globale. On note un tel état $((q_i, drz_i)_{i\in[1,n]}, \theta)$. L'état local (l_i, drz_i) d'un automate est la donnée d'une position locale et de la dernière date de raz des horloges de l'automate ($drz_i : X_i \to \mathbb{R}$). Pour tirer une transition globale $(t_i)_{i\in[1,n]}$ à la date $\theta' \geq \theta$, θ étant la date courante, on demande que les invariants des positions de tous les automates soient encore vrais à la date θ' (c'est-à-dire qu'ils soient vrais pour les valeurs d'horloge $\theta' - drz$) et tous les automates qui doivent emprunter

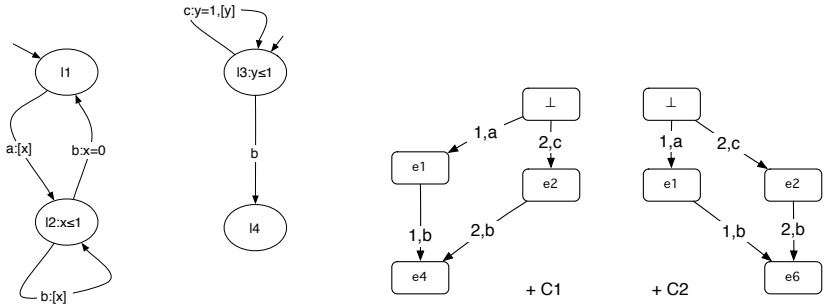


Fig. 5. Un automate temporisé (à gauche) possédant deux horloges x et y. Les positions et les invariants sont notés dans des cercles. Les transitions sont étiquetées par leur action (a, b ou c), la garde sur les horloges et l'ensemble des raz (entre crochets). A droite : les deux explications possibles, accompagnées des contraintes temporelles requises pour les dates d'occurrence des événements.

une transition locale t aient le droit de le faire : c'est-à-dire que la position de départ de la transition t soit la position courante de l'automate et que la garde soit satisfaite à l'instant θ'. L'état global résultant est obtenu en changeant les positions locales des automates conformément aux transitions locales considérées et en effectuant les raz des horloges mentionnées dans les transitions. On doit aussi s'assurer que les invariants des nouvelles positions obtenues sont vrais à la date θ'.

Définition 8 (Sémantique opérationnelle) *La sémantique opérationnelle d'un RAT est définie par le système de transitions suivant :*

- *L'état global initial est* $((q_{0i}, X_i \times \{0\})_{i\in[1,n]}, 0)$ *;*
- *Les transitions sont définies par la règle SOS suivante :*

$$\frac{\forall i \in [1,n] \begin{cases} I_i(q_i)(\theta' - drz_i) \wedge \\ (t_i \neq \epsilon) \Rightarrow \begin{cases} \alpha(t_i) = q_i \wedge \gamma(t_i)(\theta' - drz_i) \\ q'_i = \beta(q_i) \wedge I_i(q'_i)(\theta' - drz'_i) \\ drz'_i(x) = (\theta' \ \textit{si}\ x \in \rho(t_i), drz_i(x)\ \textit{sinon}) \end{cases} \end{cases}}{((q_i, drz_i)_{i\in[1,n]}, \theta) \xrightarrow{(t_i)_{i\in[1,n]} \in \mathrm{Sync}} ((q'_i, drz'_i)_{i\in[1,n]}, \theta')}$$

Dans l'exemple de RAT donné en figure 5, considérons que le système a produit les actions a, b et c. Essayons de deviner les dates possibles de production de ces actions (notées $\delta(a)$, $\delta(b)$ et $\delta(c)$). L'action c a forcément été produite à la date 1. Si l'action b a été produite par la transition remontant à la position l_1 dans le premier automate, alors $\delta(b) = \delta(a)$ à cause de la garde. $\delta(b) \leq 2$ sinon on aurait un autre c. Comme le temps doit progresser, on peut en déduire que la date de $\delta(b) \in [1, 2]$. Autrement, si l'action b était le résultat du tir de la transition bouclette du premier automate, on aurait forcément $\delta(b) \leq 2$ comme précédemment, mais aussi $\delta(b) \leq \delta(a) + 1$ parce que sinon un autre c aurait été

produit. On en déduit en fait que $max(\delta(a), 1) \leq \delta(b) \leq min(\delta(a) + 1, 2)$. C'est ce type d'information que l'on va essayer d'inférer automatiquement pendant la supervision.

6 Supervision temporisée

La question de la supervision temporisée peut se poser dans les mêmes termes que dans le cas non temporisé. Imaginons une séquence d'observations formée seulement par une suite de symboles de l'alphabet Σ. Le problème est de trouver les exécutions du modèle qui peuvent expliquer cette séquence. L'intérêt principal étant d'inférer les relations causales induites par le modèle.

Une première idée est de procéder comme dans le cas non temporisé et de définir donc la notion de dépliage d'un RAT. La séquence d'observations pouvant naturellement être vue aussi comme un AT sans contrainte de temps. La notion de dépliage temporisé a été introduite récemment [5]. Son calcul est complexe et produit une structure d'événements dont les événements portent des expressions symboliques sur les dates possibles de tir des transitions. La notion de conflit doit être aussi affaiblie en la notion de conflit asymétrique pour expliciter le plus possible de concurrence.

On propose ici une approche plus simple pour résoudre la question de supervision. Par définition, l'introduction des contraintes de temps restreint les exécutions possibles du modèle. On peut donc considérer les explications produites pour le modèle non temporisé sous-jacent (c'est-à-dire le dépliage guidé par les observations) et faire le tri en considérant les contraintes temporelles. On verra de façon plus intelligente que cette phase de post-sélection va pouvoir inférer les dates possibles des observations effectuées. Il s'agit d'une information potentiellement riche pour l'activité de supervision.

Considérons donc un RAT de réseau non temporisé sous-jacent N et une séquence d'observations σ et prenons l'ensemble des explications non temporisées possibles, c'est-à-dire tous les ensembles d'événements $E \subseteq \mathcal{E}(N, \sigma)$ tels que $\downarrow E = E$ est sans conflit et $\sum_{e \in E} \varsigma(e) = \varpi(\sigma)$.

Prenons une explication E. On note E_i l'ensemble de ses événements concernés par l'automate A_i (c'est-à-dire les événements e tels que $\tau_i(e) \neq \epsilon$). On sait que la fermeture $\rightarrow_i^*$ est un ordre total sur E_i puisque les processus sont séquentiels. On notera $\uparrow_i E$ l'événement maximum pour cette relation. Pour chaque événement, on notera $\delta(e)$ la date de cet événement et par $drz_i(e)$ la date de raz des horloges (X_i) de l'automate A_i après que l'événement s'est produit. $drz_i(e)$ est défini par :

$$\forall x \in X_i, \forall e \in E, drz_i(e)(x) \stackrel{\text{def}}{=} \begin{cases} \delta(e) & \text{si } x \in \rho(\tau_i(e)) \\ drz_i(\pi_i(e))(x) & \text{sinon} \end{cases}$$

Définition 9 (Validité temporelle) *Une explication non temporisée E est valide par rapport aux contraintes temporelles du réseau ssi :*

$$\forall i \in [1,n] \begin{cases} \forall e \neq \perp \in E_i \begin{cases} \delta(\pi_i(e)) \leq \delta(e) \\ I_i(\alpha(\tau_i(e)))(\delta(e) - drz_i(\pi_i(e))) \\ \gamma(\tau_i(e))(\delta(e) - drz_i(\pi_i(e))) \end{cases} \\ I_i(\beta(\tau_i(\uparrow_i E)))(max_{f \in E}\delta(f) - drz_i(\uparrow_i E)) \end{cases}$$

La définition de la validité temporelle reprend la définition de la sémantique séquentielle, à savoir, ligne par ligne :

- le temps ne peut pas régresser entre le prédécesseur causal d'un événement et lui-même ;
- les invariants des positions de départ de la transition sont satisfaits au moment du tir de la transition ;
- les gardes des transitions locales formant la transition de l'événement considéré sont aussi satisfaites au moment du tir ;
- Les invariants des positions d'arrivée de la transition sont satisfaits à la fin de l'explication.

Ces conditions permettent donc d'écarter les explications non valides d'un point de vue temporel. Au delà, on peut passer à une représentation symbolique des contraintes en considérant que pour chaque événement e, $\delta(e)$ est une variable réelle désignant les dates possibles de tir de la transition correspondant à l'événement e.

Pour notre exemple de la figure 5, les deux explications possibles sont assorties des contraintes suivantes :

- $C_1 \equiv \begin{cases} (\delta(\perp) \leq \delta(e_1)) \wedge (\delta(e_1) \leq \delta(e_4)) \\ (\delta(e_4) - \delta(e_1) \leq 1) \\ (max(\delta(e_1), \delta(e_2), \delta(e_4)) - \delta(e_1) \leq 1) \\ (\delta(\perp) \leq \delta(e_2)) \wedge (\delta(e_2) \leq \delta(e_4)) \\ (\delta(e_2) - \delta(\perp) \leq 1) \wedge (\delta(e_2) - \delta(\perp) = 1) \\ (\delta(e_4) - \delta(e_2) \leq 1) \end{cases}$
- $C_2 \equiv \begin{cases} (\delta(\perp) \leq \delta(e_2)) \wedge (\delta(e_2) \leq \delta(e_4)) \\ (\delta(e_4) - \delta(e_1) \leq 1) \wedge (\delta(e_4) - \delta(e_1) = 0) \\ (\delta(\perp) \leq \delta(e_2)) \wedge (\delta(e_2) \leq \delta(e_4)) \\ (\delta(e_2) - \delta(\perp) \leq 1) \wedge (\delta(e_2) - \delta(\perp) = 1) \\ (\delta(e_4) - \delta(e_2) \leq 1) \end{cases}$

En simplifiant et compte tenu que $\delta(\perp) = 0$, on obtient

$$C_1 \equiv (\delta(e_2) = 1) \wedge (1 \leq \delta(e_4) \leq 2) \wedge (\delta(e_1) \leq \delta(e_4) \leq \delta(e_1) + 1)$$

et

$$C_2 \equiv (\delta(e_2) = 1) \wedge (\delta(e_4) = \delta(e_1)) \wedge (1 \leq \delta(e_4) \leq 2)$$

confirmant bien ainsi l'analyse informelle qui avait été faite du fonctionnement du modèle temporisé.

7 Conclusion

Nous avons présenté dans cet article une méthode originale utilisant le modèle des réseaux d'automates temporisés pour produire des explications temporisées à une séquence d'actions produites par un système réparti sous surveillance. De nombreuses applications possibles peuvent être envisagées comme la corrélation d'alarmes et la détection des fautes premières, la surveillance de motifs comportementaux pour détecter des intrusions par exemple, la surveillance de propriétés temporelles non fonctionnelles, etc... Des études algorithmiques plus fines seront nécessaires. Plusieurs extensions peuvent aussi être naturellement explorées comme la répartition effective de la supervision ou l'utilisation de modèles de temps différents et plus réalistes.

Références

1. R. Alur, D. Dill. *A theory of timed automata*. Theoretical Computer Science, 126(2) :183-235, 1994.
2. A. Arnold. *Finite Transition Systems*, Prentice Hall, 1992.
3. A. Benveniste, E. Fabre, C. Jard, S. Haar. *Diagnosis of asynchronous discrete event systems, a net unfolding approach*, IEEE Transactions on Automatic Control, 48(5) :714-727, May 2003.
4. C. Cassandras, S. Lafortune. *Introduction to Discrete Event Systems*, Kluwer Academic Publishers, 1999.
5. F. Cassez, T. Chatain, C. Jard. *Symbolic unfoldings for networks of timed automata*, In ATVA, LNCS 4218, pp. 307-321, 2006.
6. T. Chatain. *Symbolic Unfoldings of High-Level Petri Nets and Application to Supervision of Distributed Systems*, PhD thesis, University of Rennes 1, November 2006.
7. T. Chatain, C. Jard. *Time Supervision of Concurrent Systems using Symbolic Unfoldings of Time Petri Net*, International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS'05), September 26-28, 2005, Uppsala, Sweden, pp. 196-210, LNCS 3829, P. Pettersson, W. Yi, eds.
8. J. Esparza. *Model checking using net unfoldings*, Science of Computer Programming 23, pp. 151-195, 1994.
9. J. Esparza, S. Römer. *An unfolding algorithm for synchronous products of transition systems*, in Proc. of CONCUR'99, LNCS 1664, Springer Verlag, 1999.
10. E. Fabre. *Bayesian Networks of Dynamic Systems*, Habilitation Thesis, University of Rennes 1, 2007.
11. G. Lamperti, M. Zanella. *Diagnosis of discrete-event systems from uncertain temporal observations*, Artif. Intel l. 137(1-2) : 91-163 (2002).
12. L. Lamport, N. Lynch. *Distributed Computing : Models and Methods*, in Handbook of Theoretical Computer Science, vol. B : Formal Models and Semantics, Jan van Leeuwen ed., Elsevier (1990), pp. 1157-1199.
13. M. Nielsen, G. Plotkin, G. Winskel. *Petri nets, event structures and domains*, Theoretical Computer Science 13(1), 1981, pp. 85-108.

14. W. Reisig. *Petri Nets*, Springer Verlag, 1985.
15. M. Sampath, R. Sengupta, K. Sinnamohideen, S. Lafortune, D. Teneketzis. *Failure diagnosis using discrete event models*, IEEE Trans. on Systems Technology, vol. 4(2), pp. 105-124, March 1996.

Remerciements : Ce travail contribue au projet national de l'ANR DOTS ANR-06-SETI-003.

Session groupe de travail Transformations

Transformations de programmes et de modèles

Génération de canevas de programmation dédiés pour les applications de téléphonie avancées

Wilfried Jouve[1], Nicolas Palix[1], Charles Consel[1], and Patrice Kadionik[2]

[1] INRIA / LaBRI, [2] IMS / Université de Bordeaux,
[1]{jouve,palix,consel}@labri.fr, [2]kadionik@enseirb.fr

Résumé Nous proposons l'approche DiaSpec fondée sur le protocole SIP pour faciliter la programmation et permettre l'exécution des applications de téléphonie avancées. Un canevas de programmation dédié est généré pour chaque application cible et fournit des opérations haut niveau faisant abstraction des technologies sous-jacentes.

Introduction. La récente convergence des réseaux de télécommunications et des réseaux informatiques permet d'envisager de nouvelles évolutions pour les applications de téléphonie. L'intégration de nouveaux objets communicants et de nouveaux modes de communication plus riches demandent aux applications de téléphonie de s'adapter en permanence et d'interagir avec des ressources toujours plus hétérogènes, comme des services Web, des agenda ou des bases de données. La programmation des applications de téléphonie en est profondément transformée. Elle demande dorénavant de coordonner des entités hétérogènes et d'échanger des types de données arbitrairement complexes. Les entités peuvent être matérielles comme des téléphones ou logicielles comme des services Web. Par exemple, une application de suivi automatique pourrait utiliser les haut-parleurs et les microphones dispersés dans chaque pièce d'une maison pour qu'un utilisateur puisse se déplacer, tout en continuant sa conversation téléphonique. L'hétérogénéité des objets communicants impliqués rend les solutions actuelles inadaptées.

Pour gérer la complexité des nouvelles applications de téléphonie, les plateformes fondées sur le protocole SIP *(Session Initiation Protocol)*[1] ont, par définition de ce protocole, le potentiel pour fournir des modes d'interaction variées : le mode d'interaction par message instantané, le mode d'interaction événement et le mode d'interaction session. En particulier, les messages instantanés sont échangés de façon synchrone entre deux entités ; ils peuvent être utilisés, par exemple, pour afficher des informations concernant un appel manqué. Le mode d'interaction événement est un mode de communication asynchrone vers plusieurs entités ; il est particulièrement utile pour propager des informations sur le contexte d'un utilisateur comme, par exemple, son statut. Finalement, le mode d'interaction session permet un échange de données entre deux entités, sur une période de temps bien définie ; il est généralement utilisé pour l'échange de flux multimédia.

Des couches logicielles ont été ajoutés au-dessus de SIP pour faciliter le développement des applications (par exemple, JAIN, SIP Servlets). Cependant, ces couches fournissent un niveau d'abstraction limité et nécessitent une grande expertise dans les technologies sous-jacentes incluant l'interface de programmation de la couche protocolaire, les protocoles réseaux et la programmation des systèmes distribués. Au delà des couches logicielles, d'autres approches comme LESS et VisuCom ont introduit des langages de programmation dédiés à la création de services de téléphonie. Toutefois leur expressivité se limite aux services pour utilisateurs finaux. D'autres approches, comme SPL, proposent des langages à script ; ils n'expriment cependant que la logique de routage des appels.

Notre approche. L'approche *DiaSpec* propose de répondre aux nouveaux défis des applications de téléphonie en élevant le niveau d'abstraction des interfaces de programmation. Pour cela, nous fournissons aux développeurs un langage déclaratif, appelé *DiaSpec*, pour spécifier les types d'entités composant l'application cible. Une entité est caractérisée par des attributs et par des fonctionnalités reposant sur les modes d'interaction SIP de type message, événement et session. Pour

[1] Rosenberg, J. et al. SIP : Session Initiation Protocol. RFC 3261, IETF, Juin 2002.

intégrer des ressources non téléphoniques, notre approche fournit un mécanisme de type RPC pour commander des entités (par exemple, pour afficher un message sur un écran) et obtenir des informations (par exemple, pour interroger un système de localisation). Une spécification DiaSpec est analysée et vérifiée par un compilateur qui génère un canevas de programmation dédié à l'application modélisée. Ce canevas de programmation fournit aux développeurs des abstractions pour découvrir, connecter et interagir avec les entités déclarées dans les spécifications DiaSpec. Nous avons étendu SIP pour gérer l'hétérogénéité et la dynamicité des nouvelles applications de téléphonie. Ces extensions utilisent les messages SIP en conformité avec le protocole. Le but est ici de permettre l'intéropérabilité des entités DiaSpec avec les infrastructures SIP existantes, constituées notamment de terminaux téléphoniques, de caméras et de serveurs gérant la mobilité. Nous détaillons maintenant les contributions principales de notre approche.

Canevas de programmation dédiés. Nous avons développé un générateur de canevas de programmation Java, compatibles avec SIP, pour abstraire la complexité des entités composant les applications de téléphonie avancées. Les canevas de programmation abstraient la complexité des technologies sous-jacentes comme les interfaces de programmation du protocole SIP et l'encodage des données. Les développeurs programment les applications de téléphonie avec des méthodes haut niveau et automatiquement générées. Ces méthodes reposent sur des opérations SIP. Notre approche générative permet aux canevas de programmation d'être typés par rapport aux spécifications DiaSpec. Les canevas de programmation s'appuient alors sur les environnements de développement comme Eclipse pour guider les développeurs en permettant, par exemple, la complétion pour spécifier les paramètres d'une requête de découverte de services.

Service de découverte. Nous introduisons un service de découverte dans des canevas de programmation reposant sur le protocole SIP rendant explicite les propriétés des entités SIP. SIP abstrait la mobilité des utilisateurs et des entités par l'introduction d'un mécanisme d'enregistrement de noms logiques appelés SIP URI. Nous avons développé une surcouche logicielle à ce mécanisme pour gérer les propriétés des entités. En utilisant le service de découverte, les applications de téléphonie peuvent sélectionner dynamiquement des entités en fonction de leurs fonctionnalités et de leurs attributs, comme par exemple la localisation.

Modes d'interaction haut niveau. Nous fournissons aux développeurs des modes d'interaction haut niveau, élevant le niveau d'abstraction des opérations SIP et introduisant un mécanisme uniforme pour l'invocation des ressources non SIP. Les nouvelles applications de téléphonie doivent intégrer de nouvelles entités toujours plus hétérogènes et dynamiques. Pour cela, DiaSpec utilise le protocole SOAP pour encoder les données échangées dans les messages SIP. L'extensibilité et la notoriété du protocole SOAP satisfait aux nouveaux besoins des applications de téléphonie. En s'appuyant sur SOAP, DiaSpec étend SIP pour gérer des flux de données arbitrairement complexes, comme par exemple, des flux de mesures pour suivre les déplacements d'une personne. De façon similaire, DiaSpec étend SIP pour la souscription et la publication d'événements arbitraires. Enfin, les canevas de programmation générés abstraient l'hétérogénéité et l'encodage des données échangées en permettant aux développeurs de s'appuyer directement sur des types de données Java.

Validation. Notre approche a été implantée et testée au travers de multiples applications. Dans le cadre d'une collaboration avec une compagnie de télécommunications, nous avons notamment développé des applications combinant la téléphonie et la domotique. L'implantation de l'approche DiaSpec est disponible en téléchargement[2].

[2] https ://diaspec.bordeaux.inria.fr/

Session groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

Approximations régulières pour l'analyse d'accessibilité

Y. Boichut[1] and R. Courbis[3] and P.-C. Héam[2] and O. Kouchnarenko[3]

(1) LIFO, Université d'Orléans, Rue Léonard de Vinci
BP 6759, 45067 Orleans Cedex 2
(2) LSV, CNRS-INRIA, Ecole Normale Supérieure de Cachan,
81 avenue du Président Wilson, 94230 Cachan
(3) LIFC, INRIA-CASSIS, Université de Franche-Comté,
16, route Gray, 25030 Besançon Cedex

Les travaux présentés ici ont été acceptés pour publication dans *Nordic Journal of Computing* sous le titre *Approximation-based Tree Regular Model-Checking* et dans la Conférence *RTA 2008*, sous le titre *Finer is better : Absrtaction Refinement for Tree Automata Completion*, LNCS 5117, pages 48-62.

1 Introduction

L'informatique est aujourd'hui une ressource critique dans de nombreux domaines de la société. Dans ce cadre, la réalisation de logiciels fiables est un enjeu majeur. De très nombreuses techniques, souvent complémentaires, co-existent pour garantir la qualité des systèmes. Les travaux présentés ici se placent dans le cadre du model-checking : le système est modélisé par un objet mathématique (automate fini, système de transitions, etc.), ainsi que la spécification souhaitée. Des techniques algorithmiques permettent alors de prouver la concordance du modèle du système vis-à-vis du modèle de la spécification.

A la fin des années 90 fut introduite la notion de model-checking régulier, où le système est modélisé par un langage régulier L de configurations initiales et par une relation R modélisant son évolution. On se donne aussi un langage régulier L_p codant des configurations que le système n'est pas supposé atteindre. Mathématiquement le système vérifie la propriété si $R^*(L) \cap L_p$ est vide. Un nombre très important de résultats ont été obtenus pour le model-checking régulier sur les mots. Récemment, les travaux de cette communauté se sont orientés vers des structures plus complexes, notamment les arbres, rejoignant ainsi des problématiques de la réécriture.

2 Contributions

Dans un cadre général, et même dans de nombreux cas restreints, on ne sait pas décider si $R^*(L) \cap L_p$ est vide. Nous nous sommes donc intéressés à une technique semi-algorithmique par approximation. Notre but est de prouver que $R^*(L) \cap L_p = \emptyset$, alors que $R^*(L)$ n'est pas calculable. Nous proposons donc, en s'appuyant sur une technique développée par Th. Genet (IRISA), de calculer automatiquement un langage régulier d'arbre K tel que $R^*(L) \subseteq K$ et tel que $K \cap L_p = \emptyset$.

2.1 Généralisation de la technique de complétion

La technique proposée par Th. Genet se restreignait aux systèmes de réécriture linéaires à gauche, c'est-à-dire dans lesquels une variable ne peut pas apparaître plusieurs fois dans une partie gauche de règles. C'est en pratique une forte restriction qui empêche de modéliser des comparaisons de valeurs. Nous avons montré comment étendre la technique à tout système de réécriture. Cette extension engendrant des calculs supplémentaires, nous avons aussi montré comment optimiser ceux-ci en utilisant des structures de données adaptées.

2.2 Automatisation

L'approche par complétion s'appuie sur des fusions d'états dans les automates. Conçue initialement comme une méthode d'assistance à la preuve, ces fusions étaient faites *à la main* par un expert. Nous avons développé des heuristiques afin de les rendre automatiques, heuristiques qui se sont montrés particulièrement efficaces dans le cadre de la vérification de protocoles de sécurité.

2.3 Raffinement

Lors du calcul, il se peut que le langage K sur-approximant $R^*(L)$ soit trop gros, c'est-à-dire que $K \cap L_p \neq \emptyset$. Dans ce cas, on ne peut pas conclure. En suivant le paradigme CEGAR (*Counter-Exemple Guided Abstraction Refinement*,) nous avons proposé un semi-algorithme permettant de calculer un langage K_1, plus petit que K (pour l'inclusion), et sur-approximant aussi $R^*(L)$. La procédure peut être éventuellement répétée et l'on sait qu'elle s'arrêtera si $R^*(L) \cap L_p$ est non vide.

2.4 Applications

Les contributions ont été implémentées et font, pour la plupart, partie de l'outil AVISPA[1] de vérification de protocoles de sécurité. Nous avons pu automatiquement vérifier plus d'une vingtaine de protocoles de communications sécurisées Internet, dans des temps d'au plus quelques minutes et dans un cadre où le problème de vérification est indécidable.

3 Conclusion

Dans le cadre du projet ANR-RAVAJ, ces techniques sont actuellement en cours d'extension ; l'implémentation dans l'outil de réécriture TOM[2] (LORIA-INRIA) est en train d'être finalisée. L'objectif étant d'appliquer l'approche par approximations à la vérification de ByteCode Java pour la téléphonie mobile.

[1] http ://www.avispa.org
[2] http//tom.loria.fr

Vérification de systèmes et réécriture : de plus en plus efficace

E. Balland[2], Y. Boichut[3], Th. Genet[1], and P.-E. Moreau[2]

[1] Irisa, INRIA Rennes-Bretagne Atlantique, France
[2] Loria, INRIA Nancy – Grand Est, France
[3] LIFO, Université d'Orléans, France

Résumé Les systèmes de réécriture (TRSs) sont de plus en plus utilisés pour la modélisation d'applications ou de systèmes. Pour ces modèles, l'analyse d'atteignabilité, i.e. prouver qu'un terme est atteignable par réécriture à partir d'un ensemble de termes donné et pour un système de réécriture donné, n'est rien d'autre qu'une forme de technique de vérification.

Par l'utilisation d'une technique de complétion sur des automates d'arbres, il a été démontré que la non-atteignabilité d'un terme t peut être vérifiée par le calcul d'une sur-approximation de l'ensemble des termes réellement atteignable et par la non-appartenance de t à cette sur-approximation. Puisque la vérification de programmes ou de systèmes concrets donne lieu à des modèles de réécriture de tailles significatives, des implantations efficaces de la complétion dans notre cadre sont essentielles.

L'objectif principal de [1] est de présenter une transformation de systèmes de réécriture qui, dans un premier temps préserve l'analyse d'atteignabilité, puis dans un second temps donne naissance à des systèmes de réécriture beaucoup plus simples à appliquer que les originaux. Cette approche a été implantée en Tom, un langage ajoutant des primitives de réécriture au langage Java. Les premières expérimentations sont très prometteuses en comparaison avec l'outil de référence pour cette technique : Timbuk.

1 Introduction

Pour la vérification de systèmes infinis, une approche consiste en l'utilisation de systèmes de réécriture (TRSs) comme modèles et de l'analyse d'atteignabilité en tant que technique de vérification. En comparaison avec d'autres techniques de modélisation, l'avantage des systèmes de réécriture est qu'ils peuvent être exécutés et vérifiés. D'un côté, comparer l'exécution d'un TRS à l'exécution d'un programme permet de vérifier la cohérence entre le modèle formel et le programme à vérifier. D'un autre côté, la plupart des techniques de vérification ont une facette réécriture : *model-checking* [5], analyse statique et interprétation abstraite [8,7,12] ou également preuves intéractives [3]. De plus, puisque toutes ces techniques opèrent sur un modèle formel commun, i.e. TRS, cela peut faciliter la combinaison de ces différentes approches.

Cependant, des problèmes d'efficacité naissent face à l'application à des cadres concrets. Cela est par exemple le cas pour la vérification de programmes Java par *model-checking* avec des TRSs [11,6] ou pour l'analyse statique de protocoles de sécurité [9] ou encore de programmes *Bytecode* Java [2]. Dès lors, des techniques de vérification efficaces sont indispensables pour que les systèmes de réécriture deviennent une technique de modélisation légitime.

Dans [1], nous avons amélioré de manière significative le côté analyse statique par réécriture. La technique que nous utilisons, appelée *complétion d'automates d'arbres* [8], construit une sur-approximation des termes atteignables. A partir d'un ensemble initial de termes (représentant par exemple les configurations initiales d'un système), l'algorithme de complétion calcule un sur-ensemble régulier des termes atteignables par réécriture (des configurations atteignables du système étudié). Ainsi, il est possible de vérifier des propriétés liées à l'analyse d'atteignabilité (en particulier des propriétés de sûreté) sur ces approximations calculées. Notre contribution dans [1], décrite également par ces quelques lignes, a été d'améliorer l'implantation de cette approche d'un facteur 10 d'efficacité en général, et jusqu'à un facteur 100 pour la vérification de programmes Java. Pour arriver à ce résultat, nous avons dans un premier temps proposé une transformation des systèmes de réécriture en des systèmes de réécriture beaucoup plus simples à gérer. Puis dans un second temps, nous avons proposé un codage original de l'implantaton en Tom/Java [1].

Après avoir présenté succinctement l'approche classique dans la section 2, dans la section 3 la transformation de TRSs et ses propriétés sont présentées. Enfin, nous décrivons l'implantation et les résultats obtenus.

2 Calculs de sur-approximation par complétion d'automates

Un automate d'arbres est une structure qui permet de reconnaître un ensemble fini ou non de termes, voir [4] pour plus de détails. Nous donnons en exemple l'automate ci-dessous.

Exemple 1 *Soit $\mathcal{A}$ l'automate contenant les transitions suivantes où f, a, b, c, d, g et h sont des symboles fonctionnels à arité fixe, q_i pour $i = 0, \ldots, 7$ sont les états de l'automate et l'état q_0 est un état final : $a \to q_4$, $b \to q_5$, $c \to q_6$, $d \to q_7$, $f(q_4, q_5) \to q_1$, $h(q_6) \to q_2$, $h(q_7) \to q_3$, $g(q_1, q_2) \to q_0$, $g(q_1, q_3) \to q_0$. Le langage de $\mathcal{A}$, noté $\mathcal{L}(\mathcal{A})$, est l'ensemble des termes qui peuvent être réécrits en q_0 en utilisant des transitions de l'automate. Dans le cas présent, $\mathcal{L}(\mathcal{A}) = \{g(f(a,b), h(c)),\ g(f(a,b), h(d))\}$.*

Une règle de réécriture notée $l \to r$ spécifie qu'un terme l peut être réécrit en un terme r. Par exemple, prenons le terme $t = g(f(a,b), h(c))$ et la règle $g(f(a,x), h(y)) \to g(f(a,\ f(a,x)), h(h(y)))$ où x et y sont deux variables. Si nous remplaçons x par b et y par c, alors $g(f(a,b), h(c))$ devient

[1] Voir [1] pour plus de détails.

$g(f(a, f(a, b)), h(h(c)))$. Naturellement, nous pouvons alors appliquer cette règle sur le terme t et nous obtenons alors le terme $g(f(a, f(a, b)), h(h(c)))$. Clairement, nous pouvons appliquer de nouveau la règle sur le terme obtenu en substituant x par $f(a, b)$ et y par $h(c)$ et nous devinons que nous pouvons le faire indéfiniment. Le langage engendré par ce processus est non régulier dans ce cas. En effet, les termes atteignables ont la forme suivante : $g(f^n(a, b), h^n(c))$ avec $n > 0$. Donc à partir du langage $\{g(f(a, b), h(c)),\ g(f(a, b), h(d))\}$, l'ensemble des termes atteignables par cette règle de réécriture ont la forme suivante : $g(f^n(a, b), h^n(\{c, d\}))$ avec $n > 0$

Pour généraliser cette approche sur un nombre non borné de termes, dans [8], l'auteur propose une construction fondée sur les automates d'arbres permettant de calculer une sur-approximation de l'ensemble des termes atteignables.

L'approche est la suivante. Etant donné un automate d'arbres $\mathcal{A}$, un TRS $\mathcal{R}$ et une fonction d'abstraction γ (dont nous préciserons le rôle dans quelques lignes), la complétion, proposée dans [8,7], calcule une séquence d'automates par des applications successives de $\mathcal{R}$. Pour l'automate donné dans l'exemple 1 et la règle de réécriture donnée précédemment, la première étape est de chercher toutes les instances possibles du terme $g(f(a, x), h(y))$[2] en substituant les variables par des états de l'automate. La recherche des instances est guidée par les symboles fonctionnels de $g(f(a, x), h(y))$. Les deux instances possibles ici sont $g(f(a, q_5), h(q_6))$ et $g(f(a, q_5), h(q_7))$ car ces deux termes peuvent se réécrire en utilisant les transitions de l'automate en un simple état (q_0 pour les deux instances dans le cas présent). Ensuite, deux nouvelles transitions sont construites à partir 1) des deux substitutions résultant des instances, 2) de la partie droite de la règle ($g(f(a, f(a, x)), h(h(y)))$) et 3) de l'état reconnaissant ici les deux instances (q_0) : $g(f(a, f(a, q_5)), h(h(q_6))) \rightarrow q_0$ et $g(f(a, f(a, q_5)), h(h(q_7))) \rightarrow q_0$. Notons que ces deux transitions n'ont pas la même forme que celles listées dans l'exemple 1. Justement, le rôle de la fonction d'abstraction γ est de découper cette transition en plusieurs transitions de la forme souhaitée[3]. Prenons une fonction d'abstraction γ projettant $h(q_7)$ sur l'état q_6, $h(q_6)$ sur l'état q_6, a sur l'état q_4 et $f(q_4, q_5)$ sur q_5. En appliquant γ sur la partie gauche de la transition $g(f(a, f(a, q_5)), h(h(q_6))) \rightarrow q_0$, l'ensemble de transitions S_1 suivant est alors obtenu : $\{a \rightarrow q_4,\ f(q_4, q_5) \rightarrow q_5,\ h(q_6) \rightarrow q_6,\ g(q_5, q_6) \rightarrow q_0\}$. La même opération avec l'autre transition donnera l'ensemble de transitions S_2 suivant : $\{a \rightarrow q_4,\ f(q_4, q_5) \rightarrow q_5,\ h(q_7) \rightarrow q_6,\ h(q_6) \rightarrow q_6,\ g(q_5, q_6) \rightarrow q_0\}$. Les transitions de $S_1 \cup S_2$ sont ajoutées aux transitions de l'automate courant, ce qui constitue l'automate successeur de l'automate courant. Une nouvelle recherche des instances de $g(f(a, x), h(y))$ est effectuée sur le nouvel automate, et dans le cas présent, aucune instance non traitée auparavant n'est détectée. Le nouvel automate est donc un point fixe de notre calcul. Le lecteur attentif remarquera que le langage engendré par le dernier automate obtenu est l'ensemble des termes

[2] La partie gauche de la règle $g(f(a, x), h(y)) \rightarrow g(f(a, f(a, x)), h(h(y)))$.
[3] $f(q_1, \ldots, q_n) \rightarrow q$ avec f un symbole fonctionnel autorisant n enfants et $q_1, \ldots, q_n$ des états.

satisfaisant $g(f^n(a,b), h^m(\{c,d\}))$ avec $n, m \geq 0$. Cet ensemble contient bien l'ensemble des termes atteignables i.e. $g(f^n(a,b), h^n(\{c,d\}))$ avec $n > 0$.

De manière générale, pour un automate initial $\mathcal{A}_0$, un système de réécriture $\mathcal{R}$ et une fonction d'abstraction γ, la complétion calcule un automate point-fixe $\mathcal{A}_k$, si γ le permet. Le langage reconnu par $\mathcal{A}_k$ représente une sur-approximation de l'ensemble des termes atteignables par réécriture à partir de l'ensemble des termes reconnus par $\mathcal{A}_0$.

3 Une transformation de TRS menant à de meilleures performances

Un point sensible au coeur de la complétion avec un système de réécriture $\mathcal{R}$ est le calcul des instances de l avec $l \to r \in \mathcal{R}$ pour un automate donné $\mathcal{A}$. D'une manière générale, avec le système Timbuk, les temps de calculs n'étaient pas très satisfaisants lorsque pour une règle de réécriture $l \to r$, l est un terme profond. Et c'est d'ailleurs le cas pour la vérification de programmes Java où non seulement les termes sont profonds (en moyenne générale entre 4 et 5 étages de symboles fonctionnels) mais sont également parfois très larges (jusqu'à 8 enfants). Dans [1], nous avons proposé une transformation ϕ de systèmes de réécriture. A partir d'un système de réécriture $\mathcal{R}$, le système de réécriture $\mathcal{R}' = \phi(\mathcal{R})$ a les deux propriétés suivantes :

1. Pour toute règle $l \to r \in \mathcal{R}'$, la profondeur de l est inférieure ou égale à 2 ;
2. Pour toute règle $l \to r \in \mathcal{R}$, le terme l peut être réécrit en r en utilisant des règles de réécriture de $\mathcal{R}'$.

L'intérêt de la première propriété est que la recherche des instances des parties gauches des règles est simplifiée puisque ces termes ne sont pas profonds (au pire une profondeur de 2). La seconde propriété nous assure que si un terme est atteignable avec le système de réécriture $\mathcal{R}$, alors il le sera également avec le système de réécriture $\mathcal{R}'$. Nous avons donc une préservation de l'analyse d'atteignabilité du TRS original par transformation.

Exemple 2 *Pour le système de réécriture $\mathcal{R}$ composé de la seule règle $g(f(a,x), h(y)) \to g(f(a,f(a,x)), h(h(y)))$, selon la transformation ϕ définie dans [1], $\phi(\mathcal{R})$ contient les règles suivantes : $rl_1 : a \to C_1$, $rl_2 : f(C_1,x) \to C_2(x)$, $rl_3 : h(y) \to C_3(y)$, $rl_4 : g(C_2(x), C_3(y)) \to C_4(x,y)$ et $rl_5 : C_4(x,y) \to g(f(a,f(a,x)), h(h(y)))$ où C_1, C_2, C_3 et C_4 sont des symboles fonctionnels frais i.e. présents ni dans l'automate de départ $\mathcal{A}$, ni dans le système de réécriture $\mathcal{R}$. La caractéristique 1. mentionnée ci-dessus est triviallement vérifiée. Concernant la caractéristique 2., nous remarquons que le terme $g(f(a,x), h(y))$ peut être successivement réécrit en :*

- $g(f(C_1,x), h(y))$ *en appliquant* rl_1,
- $g(f(C_1,x), C_3(y))$ *en appliquant* rl_3,
- $g(C_2(x), C_3(y))$ *en appliquant* rl_2,
- $C_4(x,y)$ *en appliquant* rl_4 *et finalement*

– $g(f(a, f(a, x)), h(h(y)))$ *en appliquant* rl_5.

Par ces deux propriétés, en utilisant la complétion sur le système de réécriture $\mathcal{R}'$, si le calcul converge vers un point-fixe $\mathcal{A}_k$, non seulement le langage reconnu par cet automate représente une sur-approximation des termes atteignables par réécriture avec $\mathcal{R}'$, mais cela représente également une sur-approximation de l'ensemble des termes atteignables par réécriture en utilisant $\mathcal{R}$.

Nous avons implanté un outil en Tom, un langage ajoutant des primitives de réécriture au langage Java. A partir d'une spécification contenant un système de réécriture $\mathcal{R}$, un automate initial $\mathcal{A}$ et une fonction d'abstraction γ, notre implantation génère dans un premier temps un programme Tom/Java implantant la complétion pour le système de réécriture $\mathcal{R}' = \phi(\mathcal{R})$. Dans un second temps, ce programme est compilé avec Tom et génère ainsi un programme Java implantant une complétion dédiée à notre spécification de départ. Quelques résultats obtenus avec notre outil sont récapitulés dans le tableau ci-dessous.

	NSPK protocole	View-Only protocole	programme Java (listes chaînées)
Taille TRS	13	15	303
Timbuk : Temps (s)	**19.7**	**6420**	**37387**
Tom : Temps (s)	**5.9**	**150**	**303**
Timbuk/Tom	**3**	**40**	**120**

4 Conclusion

Dans [1], nous avons proposé un codage original de l'algorithme de complétion initialement introduit dans [8]. Ce codage a conduit à une implantation beaucoup plus efficace que l'outil original Timbuk [10] implanté par Th. Genet. En effet, notre implantation se montre jusqu'à 100 fois plus efficace pour la vérification de programmes Java dans le cadre de l'ANR RAVAJ[4]. Ceci représente donc un premier pas prometteur vers le chemin menant à un outil de vérification général et compétitif pour la vérification de systèmes infinis.

Références

1. E. Balland, Y. Boichut, Th. Genet, and P.-E. Moreau. Towards an efficient implementation of tree automata completion. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 67–82. Springer-Verlag, 2008.
2. Y. Boichut, T. Genet, T. Jensen, and L. Le Roux. Rewriting Approximations for Fast Prototyping of Static Analyzers. In *Proceedings of the 18th Conference on Rewriting Techniques and Applications*, volume 4533 of *LNCS*, pages 48–62, 2007.

[4] http ://www.irisa.fr/lande/genet/RAVAJ/

3. M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool : a tutorial. *J. UCS*, 12(11) :1618–1650, 2006.
4. H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. `http ://www.grappa.univ-lille3.fr/tata/`, 2002.
5. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL Model Checker and its Implementation. In *SPIN*, volume 2648 of *LNCS*, pages 230–234. Springer, 2003.
6. A. Farzan, C. Chen, J. Meseguer, and G. Rosu. Formal analysis of java programs in javafan. In *CAV*, volume 3114 of *LNCS*, pages 501–505. Springer, 2004.
7. G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *JAR*, 33 (3-4) :341–383, 2004.
8. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA Conf., Tsukuba (Japan)*, volume 1379 of *LNCS*, pages 151–165. Springer-Verlag, 1998.
9. T. Genet and F. Klay. Rewriting for Cryptographic Protocol Verification. In *Proc. 17th CADE Conf., Pittsburgh (Pen., USA)*, volume 1831 of *LNAI*. Springer-Verlag, 2000.
10. T. Genet and V. Viet Triem Tong. Timbuk 2.0 – a Tree Automata Library. IRISA / Université de Rennes 1, 2000. `http ://www.irisa.fr/lande/genet/timbuk/`.
11. J. Meseguer and G. Rosu. Rewriting logic semantics : From language specifications to formal analysis tools. In *IJCAR*, pages 1–44, 2004.
12. T. Takai. A Verification Technique Using Term Rewriting Systems and Abstract Interpretation. In *Proc. 15th RTA Conf., Aachen (Germany)*, volume 3091 of *LNCS*, pages 119–133. Springer, 2004.

Session groupe MFDL

Méthodes Formelles dans le Développement du Logiciel

Un cadre formel pour le contrôle d'accès

Mathieu Jaume

SPI - LIP6 - Université Paris 6
104 av. du Président Kennedy
F-75016 Paris, France

Abstract. Un des aspects de la sécurité en informatique concerne le contrôle des accès aux données d'un système pour lequel différentes politiques de sécurité peuvent être mises en application. Toutefois, rien ne sert de mettre en place une politique de sécurité pour gérer un système si les programmes chargés de garantir le bon fonctionnement de cette politique ne sont pas fiables. Cet article rend compte de manière informelle de différentes expériences permettant d'obtenir des développements formels de politiques de contrôle d'accès. Ces développements nous conduisent à introduire un "cadre sémantique" dans lequel il est possible de spécifier et d'implanter des politiques de contrôle d'accès. Ce cadre permet de définir des mécanismes de comparaison de modèles et d'analyser ces modèles en termes de flots d'information qu'ils autorisent.

Mots clés : Contrôle d'accès, Méthodes formelles

1 Introduction – Motivations

La protection des informations d'un système informatique est une préoccupation majeure. L'apparition de systèmes informatiques de plus en plus grands, la dissémination de l'information et le développement des réseaux, permettent dorénavant des attaques depuis l'extérieur et rendent la protection des informations de plus en plus complexe. Comme dans toutes les autres disciplines scientifiques, le besoin de recourir à des modèles et formalismes mathématiques se fait ressentir pour mieux comprendre et analyser les problèmes liés à l'informatique. C'est de ce besoin que viennent les méthodes formelles. Dans [1], P. Amey définit la "chose formelle" comme une "chose soutenue par une rigueur mathématique". Ainsi, les méthodes formelles peuvent être vues comme des "méthodes soutenues par une rigueur mathématique" dont l'absence d'ambiguïté permet de spécifier et d'implanter un système en garantissant que certaines propriétés sont respectées. Lorsqu'il s'agit de systèmes logiciels critiques, ces propriétés peuvent être vitales. Dans cet article, nous utilisons les méthodes formelles pour étudier certaines des propriétés classiques de sécurité des systèmes informatiques. Nous nous intéressons plus particulièrement au contrôle d'accès. Il s'agit de régir et de gérer les accès effectués selon certains modes (lecture, écriture, ...) par des sujets, les entités actives (processus, programmes, utilisateurs, ...) sur des objets, les entités passives (données, fichiers, programmes, ...). A plus long

terme, notre objectif est d'obtenir une bibliothèque certifiée de moniteurs de référence mettant en application différentes politiques de sécurité. En effet, le développement logiciel d'un moniteur de référence n'a de sens que s'il permet de garantir les propriétés de sécurité pour lesquelles il a été conçu. Pour atteindre de hauts niveaux de certification, il est nécessaire de fournir un modèle formel du système permettant d'obtenir des preuves formelles mécanisées. Nous présentons ici plusieurs expériences menées pour atteindre cet objectif. Comme nous allons le voir, trois difficultés sont à prendre en compte dans ce travail. La première est classique et provient de l'activité même de formalisation : le passage de l'informel au formel nécessite d'identifer les hypothèses implicites et d'expliciter totalement le système à modéliser. Afin de valider la formalisation obtenue, il s'agit alors de la "mécaniser" (i.e. de l'implanter). C'est la deuxième difficulté : certaines preuves "triviales" à obtenir sur le papier le sont beaucoup moins avec un assistant à la preuve. Quoi qu'il en soit, conduire un développement formel de cette nature est une activité chronophage. Il faut, dans la mesure du possible, factoriser les spécifications et les preuves formelles afin de faciliter la réutilisation de ces développements. Bien sûr, l'utilisation d'un atelier de développement formel muni de mécanismes facilitant l'écriture modulaire de spécifications, de définitions et de preuves permet d'atteindre une certaine "réutilisabilité" des développements conduits. Ce n'est toutefois pas suffisant. Il est en effet souhaitable de concevoir un cadre formel uniforme dans lequel puissent s'exprimer les modèles de contrôle d'accès que nous envisageons. C'est la troisième difficulté : il s'agit à la fois d'identifier les "ingrédients" communs aux politiques de contrôle d'accès, d'exprimer les propriétés génériques qu'ils vérifient, d'en prouver certaines et de formaliser les politiques envisagées comme des instances du cadre générique. Un tel cadre procure un formalisme commun pour décrire des modèles de contrôle d'accès et permet de dégager des techniques d'analyse de ces modèles (comparaison, flots d'informations, ...). Cet article a pour objectif de fournir une présentation synthétique des divers travaux que nous avons réalisés sur la formalisation et l'implantation des politiques de contrôle d'accès dans un cadre formel [12, 18–22, 14, 17].

2 Descriptions formelles de modèles de contrôle d'accès

Cette section rend compte de manière informelle de différentes expériences de développements formels de modèles de contrôle d'accès dont l'objectif est de certifier des programmes en charge de la sécurité dans un système d'information. Plus précisément, il s'agit de garantir qu'un moniteur de référence chargé du contrôle des accès dans un système maintient une politique de contrôle d'accès donnée. Cette propriété est bien sûr cruciale pour la plupart des systèmes. Comme nous l'avons suggéré dans l'introduction, la première difficulté provient du passage de l'informel au formel. En effet, bon nombre de politiques sont exprimées de manière informelle dans la littérature. Pour illustrer cela, considérons l'exemple classique de la politique de la Muraille de Chine, introduite par Brewer et Nash [6], pour résoudre les problèmes de conflit d'intérêt dans le monde des

consultants. Chaque objet du système appartient à un ensemble de données d'une compagnie, et chaque compagnie appartient à une unique classe de conflit d'intérêt. Ces classes de conflit correspondent à des milieux professionnels distincts, comme par exemple les banques ou les compagnies pétrolières. Selon cette politique, un consultant peut travailler en même temps pour une banque et une compagnie pétrolière, mais ne peut pas le faire pour deux banques ou deux compagnies pétrolières. Il existe de plus une classe de conflit spéciale qui ne contient qu'une seule compagnie et qui contient les informations "sanitisées", c'est-à-dire celles qui peuvent être lues par tout le monde sans provoquer de conflit d'intérêt. La politique est énoncée comme la combinaison de deux propriétés de sécurité. Nous répétons volontairement en anglais l'une d'entre elles, afin de permettre au lecteur de mieux percevoir le niveau de formalisation de la description originale de cette politique.

> [6] *Write access is only permitted if (a) access is permitted by the simple security rule, and (b) no object can be read which is in a different company dataset to the one for which write access is requested and contains unsanitized information.*

D'une part, cette propriété est difficilement compréhensible à la première lecture, du fait de la structure de la phrase employée. D'autre part, certaines notions restent indéfinies. Quel est le statut exact d'un objet qualifié de "*object can be read*" ? Est-ce un objet que le sujet est en train de lire ? ou bien un objet qu'il lira dans l'avenir ? cela signifie-t-il qu'il a les droits pour le faire ? Pour formaliser cette propriété, il faut faire des choix quant à l'interprétation de ces concepts. Le danger provient de ces choix qui permettent plusieurs formalisations non équivalentes de cette politique. Enfin, l'étape de formalisation permet de distinguer clairement d'une part les propriétés de sécurité souhaitées et d'autre part le moniteur de référence chargé de les faire respecter. En effet, la présentation informelle de la politique de la Muraille de Chine peut s'apparenter à une description algorithmique de la fonction d'autorisation des accès. Cela peut amener à considérer immédiatement le "comment", donc l'implantation, au lieu de réfléchir d'abord au "quoi", c'est-à-dire à la spécification de la politique de sécurité.

2.1 Formalisation du modèle de Bell & LaPadula avec Coq

Nous présentons ici brièvement une formalisation du modèle de Bell et LaPadula [23, 3] dans le système Coq à partir de laquelle un programme certifié implantant un moniteur de référence pour cette politique a été extrait. Ce travail est décrit dans [12].

Politique de Bell & LaPadula La politique de Bell et LaPadula est habituellement décrite par une machine à états [23, 3]. Elle dépend d'un ensemble $\mathcal{S}$ de sujets, d'un ensemble $\mathcal{O}$ d'objets, d'un ensemble $\mathcal{A}$ de modes d'accès et d'un treillis fini $(\mathcal{L}, \preceq, \curlyvee, \curlywedge)$ de niveaux de sécurité. Un état du système est décrit par un

quadruplet (m, D, f_s, f_o) où m est l'ensemble des accès courants effectués dans le système, D est l'ensemble des droits d'accès et $f_s : \mathcal{S} \to \mathcal{L}$ (resp. $f_o : \mathcal{O} \to \mathcal{L}$) est une fonction associant un niveau de sécurité aux sujets (resp. aux objets). Les éléments de m et de D sont des accès représentés par des triplets de la forme (s, o, a) : $(s, o, a) \in m$ signifie qu'un sujet s accède à un objet o selon le mode d'accès a tandis que $(s, o, a) \in D$ signifie qu'un sujet s dispose des droits (discrétionnaires) pour accéder à un objet o selon le mode d'accès a. Les trois propriétés de sécurité définies dans la politique de Bell et LaPadula sont les suivantes.

- Propriété DAC (*Discretionnary Access Control*) : La propriété DAC exprime que tout accès courant est conforme aux droits d'accès : $m \subseteq D$
- Propriétés MAC et MAC* (*Mandatory Access Control*) :
  - La propriété MAC ("*no read-up property*") exprime qu'un sujet ne peut accéder en lecture à un objet que si son niveau de sécurité est supérieur à celui de l'objet accédé : $(s, o, \mathsf{read}) \in m \Rightarrow f_s(s) \succeq f_o(o)$
  - La propriété MAC* ("*no write-down property*") permet d'éviter qu'un sujet recopie de l'information sensible à un niveau de sécurité inférieur :

$$((s, o_1, \mathsf{read}) \in m \wedge (s, o_2, \mathsf{write}) \in m) \Rightarrow f_o(o_1) \preceq f_o(o_2) \qquad (1)$$

Certification d'un moniteur de référence Nous esquissons ici les grandes lignes de la formalisation du modèle de Bell et LaPadula dans le système Coq. Ce développement est paramétré par $\mathcal{S}$, $\mathcal{O}$, $\mathcal{A}$ et un treillis $(\mathcal{L}, \preceq, \curlyvee, \curlywedge)$. Ce treillis est en fait obtenu à partir de deux paramètres : un ensemble $\mathcal{K}$ de "domaines" (connu sous le nom de "*needs-to-know*"), comme par exemple {nucléaire, médical, ... }, et un ensemble $\mathcal{C}$ de classifications, comme par exemple {Top-secret, secret, public, ... }, muni d'une relation d'ordre total. $\mathcal{L}$ est alors défini comme le treillis produit $\mathcal{C} \times T_k$ où $T_k = (\wp(\mathcal{K}), \subseteq, \cup, \cap)$ est le treillis des parties de $\mathcal{K}$. A partir de la définition de l'ensemble Σ des états, les fonctions de transition sont définies comme des fonctions de $\mathcal{R} \times \Sigma$ dans $\mathcal{D} \times \Sigma$ où $\mathcal{R}$ est un ensemble de requêtes et $\mathcal{D} = \{\mathsf{yes}, \mathsf{no}\}$ contient les réponses possibles. Une fonction de transition τ, correspondant à la mise en application d'une politique de sécurité, sera qualifiée de fonction "sûre" ssi elle transforme chaque état vérifiant les propriétés DAC, MAC et MAC* en états vérifiant encore ces propriétés. Le développement réalisé consiste en une implantation dans Coq de la fonction de transition τ_{BLP} introduite par Bell et LaPadula ainsi que de la preuve mécanisée du célèbre "*Basic Security Theorem*" [3] affirmant que τ_{BLP} est "sûre". Le mécanisme d'extraction de Coq a permis, à partir de cette preuve, d'obtenir un programme certifié qui implante τ_{BLP}.

Ce travail de mécanisation de la preuve a non seulement fourni une implantation avec un niveau de confiance élevé mais a aussi permis de reconnaître, parmi les hypothèses sur la fonction de transition, celles qui sont nécessaires au maintien de la politique de sécurité. Cela permet donc d'envisager des variantes de la fonction τ_{BLP} qui préservent les propriétés de sécurité. Toutefois, si elle remplit ses objectifs en termes de garantie de correction, une telle approche peut conduire à des développements difficilement réutilisables. En effet,

la moindre modification tant sur la spécification de la politique que sur son implantation conduit à modifier une preuve de plus d'un millier de lignes. Il s'agit là d'un exercice particulièrement chronophage. Pour faciliter la réutilisation, il est nécessaire d'élaborer un cadre formel caractérisant les éléments communs aux politiques de contrôle d'accès et permettant d'en dériver progressivement la spécification complète de la politique choisie, puis différentes implantations. Il est également préférable d'utiliser un environnement de développement qui facilite l'implantation de ces différents traits. Même si le système Coq dispose à présent de mécanismes permettant de conduire des développements formels de manière modulaire, nous avons choisi d'utiliser l'atelier FoCaL, mieux adapté car offrant des traits objets.

2.2 Implantation de l'algèbre des modèles de sécurité de McLean

Afin de faciliter la réutilisation des développements formels de modèles de contrôle d'accès, l' "algèbre des modèles de sécurité" introduite par McLean [25].a été implantée avec l'environnement de développement FoCaL [28, 9]. Cette algèbre définit un cadre générique pour le contrôle d'accès qui peut ensuite être instancié afin d'en dériver une implantation de la politique de Bell et LaPadula, qui peut à son tour être utilisée pour gérer les accès à une base de données relationnelle. Nous décrivons ici l'architecture de l'ensemble de ces développements qui sont détaillés dans [26, 18, 19, 4].

Algèbre des modèles de sécurité L'"algèbre des modèles de sécurité" permet la description d'un modèle de contrôle d'accès à trois niveaux de spécification différents : les frameworks, les modèles et les systèmes. Le framework décrit l'environnement : il est paramétré par $\mathcal{S}$, $\mathcal{O}$, $\mathcal{A}$ et un treillis $\mathcal{L}$ et spécifie quels sont les ensembles de sujets qui pourront conjointement demander à accéder à un objet ou demander à changer le niveau de sécurité d'un sujet ou d'un objet (sans toutefois spécifier comment sera traitée cette demande). McLean introduit ici la notion d'accès conjoints : certaines opérations ne peuvent être autorisées que si elles sont demandées par un certain groupe de sujets. Dans le cas le plus général, les ensembles de sujets qui pourront conjointement soumettre une requête sont des parties non vides quelconques de $\mathcal{S}$ (toute opération est initiée par au moins un sujet) mais il peut exister des contraintes sur ces ensembles. Par exemple, certaines politiques imposent que si un groupe de sujets peut conjointement soumettre une requête, alors tout sous-ensemble de ce groupe peut également soumettre cette requête. Dans le cas le plus courant, sans accès conjoints, les seuls ensembles autorisés à soumettre une requête sont les singletons construits à partir de $\mathcal{S}$. Les différentes possibilités pour les ensembles de sujets permettent de construire une hiérarchie de frameworks permettant de factoriser les spécifications. La notion de modèle, paramétrée par celle de framework, a été introduite afin de pallier aux problèmes posés par des systèmes de contrôle d'accès ne fixant aucune règle régissant le changement de niveau de sécurité d'un sujet ou d'un objet. Un modèle spécifie quels sont les ensembles de sujets qui seront

effectivement autorisés à modifier les niveaux de sécurité. Comme pour les frameworks, les différentes propriétés vérifiées par ces ensembles de sujets permettent de construire une hiérarchie de modèles. Certaines relations et opérations sur les modèles sont définies et permettent d'implanter l'ensemble des modèles comme une instance de treillis distributif. Les états décrivent les informations relatives aux différents niveaux de sécurité associés aux sujets et aux objets ainsi que les accès courants. Une fois la notion d'état définie, on spécifie par un prédicat quels sont les états sûrs, c'est-à-dire quelle est la politique de sécurité appliquée. C'est à partir de cette notion d'état et de modèle qu'est définie la notion de système. Celle-ci décrit la fonction de transition et les propriétés qu'elle doit vérifier. Cette fonction de transition entre états décrit les changements d'états produits lors des requêtes d'accès émises par les sujets. Ces trois notions ont été mécanisées dans l'atelier FoCaL puis utilisées pour traiter le cas particulier de la politique de Bell et LaPadula.

Détection de flots d'information non autorisés Ici, l'étape de formalisation et de mécanisation de la formalisation avec FoCaL a permis de corriger la spécification initiale. En effet, dans [25], McLean illustre l'utilisation du cadre qu'il introduit en spécifiant en son sein une version enrichie de la politique de Bell et LaPadula, essentiellement en considérant la notion d'accès conjoints. La propriété de sécurité $\mathsf{MAC}^\star$ définie en (1) est alors (re)définie comme suit :

> [25] *a state is $\star$-secure if for any subjects S_1, S_2 and objects o_1, o_2, if $(S_1, o_1, \mathsf{read}) \in m$ and $(S_2, o_2, \mathsf{write}) \in m$ and the classification of o_1 dominates that of o_2, then $S_1 \cap S_2 = \emptyset$*

Ici, m est l'ensemble des accès courants, S_1 et S_2 sont des ensembles de sujets et un accès est un triplet (S, o, a) exprimant que les sujets présents dans l'ensemble S accèdent conjointement à l'objet o selon le mode a. La formalisation de cet énoncé permet d'obtenir, par contraposition, la spécification suivante :

$$((S_1, o_1, \mathsf{read}) \in m \wedge (S_2, o_2, \mathsf{write}) \in m \wedge S_1 \cap S_2 \neq \emptyset) \Rightarrow \neg(f_o(o_2) \prec f_o(o_1)) \quad (2)$$

Or, si on se limite au cas où S_1 et S_2 sont réduits à des singletons, c'est-à-dire si on ne considère pas les accès conjoints, la propriété (1) implique la propriété (2), mais la réciproque est en général fausse : (2) n'implique (1) que si l'ordre sur les niveaux de sécurité est total. En général cet ordre est partiel (puisqu'il définit une structure de treillis) et la propriété $\mathsf{MAC}^\star$ n'est alors pas vérifiée (bien que (2) soit satisfaite) comme le montre l'exemple suivant. En respectant la propriété (2), un sujet s_1 peut accéder simultanément en lecture à un objet de niveau de sécurité l_1 et en écriture à un objet de niveau de sécurité l_2 tels que l_1 et l_2 ne soient pas comparables. Un sujet s_2 peut alors lire ce dernier objet de niveau l_2 et écrire simultanément dans un objet de niveau l_3 tel que l_3 et l_2 ne soient pas comparables mais tel que l_3 soit inférieur à l_1. Il y a alors une "fuite d'information vers le bas" puisqu'il devient ainsi possible de recopier les informations d'un objet de niveau l_1 dans un objet de niveau $l_3 \preceq l_1$.

L'activité de formalisation permet de mettre en lumière de tels problèmes. Ils peuvent être découverts en essayant de prouver des "énoncés faux" (par exemple

en essayant de prouver l'équivalence entre (1) et (2) lorsque S_1 et S_2 sont des singletons), mais ils peuvent parfois aussi être découverts en utilisant des méthodes permettant de tester une propriété, par exemple une propriété assurant qu'il n'existe pas de séquence d'accès permettant de copier de l'information de niveau élevé dans des objets de niveaux inférieurs. Pour ce faire, il est possible d'utiliser, dans certaines conditions, un outil de test intégré à l'atelier FoCaL développé par M. Carlier et C. Dubois [7]. Citons aussi les travaux de C. Morisset et A. Santana de Oliveira [15, 27, 26, 8] qui décrivent un algorithme permettant de détecter la "fuite d'information" correspondant à l'exemple présenté dans cette section à partir d'une spécification de politique par un système de réécriture. La description de politiques de sécurité *via* un système de réécriture a été étudiée par A. Santana de Oliveira dans [8].

Instanciation du modèle et application En définissant, c'est-à-dire en instanciant, chacune des entités spécifiées dans le cadre générique, nous avons défini avec FoCaL la politique de Bell et LaPadula et avons fourni une implantation de la fonction de transition τ_{BLP}. Cette implantation reste encore relativement générique : elle ne spécifie ni les sujets, ni les objets. En effet, un programme qui implante une certaine politique de contrôle d'accès est *a priori* indépendant du système sur lequel il doit s'appliquer (bases de données, systèmes de gestion de fichiers, ...). Il possède des paramètres destinés à prendre en compte l'environnement concret dans lequel il va s'exécuter. Nous allons maintenant illustrer succintement l'intégration de tels programmes dans un système "concret", une base de donnée relationnelle. Il s'agit ici d'une mise à l'épreuve du terrain de la méthode toute entière.

La base de données (MySQL) contient à la fois les données des utilisateurs, et les données relatives à la sécurité (droits d'accès, accès courants, niveaux de classification, "*needs-to-know*"). Les paramètres du système sont instanciés par des objets stockés dans la partie "sécurité" de la base de données et les états de ce système sont construits à partir de ces données. Les objets du sytème sont les tables de la base de données. Une granularité plus fine pour la notion d'objet pourrait être envisagée mais soulèverait des problèmes connus dans le domaine des bases de données pour lesquels des solutions existent mais sortent du cadre de notre prototype. La figure 1 illustre le traitement des requêtes effectué par l'application obtenue. Etant donnée une requête SQL soumise par un sujet (authentifié), le programme d'analyse de requêtes SQL que nous avons développé fait appel au système de Bell et LaPadula et, selon la réponse obtenue, traite ou refuse l'exécution de la requête soumise. Pour cela, nous avons défini pour chaque requête SQL, un ensemble de requêtes pour le système de Bell et LaPadula, donnant en quelque sorte une sémantique en termes d'accès aux requêtes SQL. On remarquera que les requêtes SQL soumises par l'utilisateur utilisent la syntaxe standard de SQL, rendant ainsi transparente pour l'utilisateur la mise en œuvre de notre application, qui peut être vue comme un filtre entre l'utilisateur et le système de gestion de la base de données. Ce développement est décrit en détail dans [26, 4].

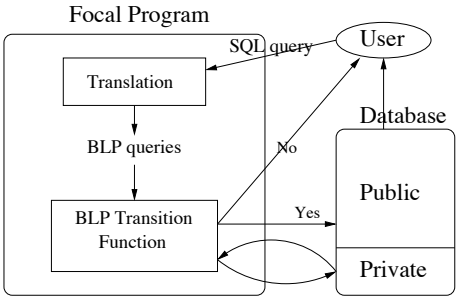


Fig. 1. Contrôle d'accès pour un SGBD

2.3 Nécessité de définir un cadre plus générique

S'il permet de caractériser certains aspects communs à différentes politiques de contrôle d'accès, le cadre introduit par McLean n'est pas encore suffisamment riche. Il est certes bien adapté à la définition de politiques basées sur un ensemble de niveaux de sécurité associés à certaines entités du système mais ne permet pas de définir des politiques comme la Muraille de Chine (dans le contexte de la Muraille de Chine nous ne disposons pas d'un treillis de niveaux de sécurité mais d'un ensemble de classes de conflits) ou encore des politiques à base de rôles (dans le contexte d'une politique à base de rôles, nous ne disposons pas d'un treillis de niveaux de sécurité mais d'un ensemble de rôles muni d'un ordre partiel). Le cadre de McLean se révèle donc à la fois trop contraignant et trop peu expressif (en termes de propriétés de comparaison de modèles, de simulation d'implantations permettant une plus grande réutilisabilité, d'analyse de flots d'informations). Nous avons alors été amenés à concevoir un cadre sémantique pour le contrôle d'accès permettant de répondre à nos objectifs. Il ne s'agit pas de la définition d'un langage mais plutôt d'une spécification de style mathématique de ce qui constitue une politique de contrôle d'accès. Ce cadre et son utilisation sont présentés dans la section suivante.

3 Un cadre formel pour le contrôle d'accès

3.1 Modèles de contrôle d'accès

Politiques de contrôle d'accès Une politique de contrôle d'accès permet de caractériser les états d'un système et de spécifier ce qu'est un état sûr en fonction d'informations de sécurité associées aux entités du système. Les entités du système peuvent être réparties dans deux ensembles : l'ensemble $\mathcal{S}$ des sujets, qui correspondent aux entités qui effectuent les actions, et l'ensemble $\mathcal{O}$ des objets, qui subissent les actions. Nous désignons par $\mathcal{A}$ l'ensemble des modes d'accès qui caractérisent les différents types d'accès effectués par les sujets sur les objets. Cet ensemble contient généralement read, write, append, etc. Une approche

classique consiste à représenter un accès par un triplet (s, o, a), signifiant que le sujet s accède à l'objet o selon le mode d'accès a. Néanmoins, d'autres approches existent (on peut par exemple regrouper les modes d'accès, ou encore considérer les accès conjoints). Afin de pouvoir prendre en compte ces différentes situations, nous nous limitons à dénoter par $\mathbb{A}$ l'ensemble de tous les accès, sans introduire une véritable définition qui serait trop précise à ce stade.

Les systèmes de contrôle d'accès sont ici modélisés sous la forme de machines à états. Un état représente le système à un instant donné et contient au moins une description de l'ensemble des *accès courants*, c'est-à-dire de tous les accès qui ont été acceptés et qui n'ont pas encore été relâchés. Ces accès sont donc supposés être effectués simultanément dans le système. L'ensemble des états est noté Σ et l'ensemble des accès courants d'un état σ est noté $\Lambda(\sigma)$. Une politique de contrôle d'accès permet de spécifier un sous-ensemble de Σ, contenant les états sûrs, c'est-à-dire les états qui vérifient la politique. Afin de déterminer si un état est sûr, les entités du système sont associées à des informations de sécurité :

- le paramètre de sécurité, noté ρ, décrit les informations de sécurité *statiques* de la politique, c'est-à-dire les informations de sécurité qui ne changeront pas durant la vie du système,
- les fonctions de sécurité décrivent les informations de sécurité *dynamiques* de la politique, c'est-à-dire les informations de sécurité susceptibles d'évoluer durant la vie du système (étant donné un état σ, on note $\Upsilon(\sigma)$ les fonctions de sécurité associées à l'état σ).

A partir de ces informations, les états sûrs sont caractérisés par un prédicat Ω sur les états. On note $\Sigma_{|\Omega}$ l'ensemble $\{\sigma \in \Sigma \mid \Omega(\sigma)\}$ des états sûrs. Toutes ces notions permettent de définir une politique de contrôle d'accès $\mathbb{P}[\rho]$, basée sur un paramètre de sécurité ρ, par un quintuplet :

$$\mathbb{P}[\rho] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma, \Omega)$$

A ce niveau de spécification, il est possible de caractériser certaines propriétés que peuvent vérifier les politiques de contrôle d'accès.

- Une politique est dite *compacte* si lorsque l'on supprime des accès de l'ensemble des accès courants d'un état sûr, l'état reste sûr :

  $$\forall \sigma_1 \in \Sigma \quad \Omega(\sigma_1) \Rightarrow (\forall \sigma_2 \in \Sigma \ (\Lambda(\sigma_2) \subseteq \Lambda(\sigma_1) \wedge \Upsilon(\sigma_1) = \Upsilon(\sigma_2)) \Rightarrow \Omega(\sigma_2))$$

  La plupart des politiques classiques (HRU, Bell et LaPadula, Muraille de Chine, RBAC, ...) sont compactes. Toutefois, il peut être utile de définir des politiques non compactes dans certains contextes. Considérons par exemple une politique de contrôle des accès aux ressources d'un système qui stipule qu'un utilisateur est autorisé à utiliser des ressources d'une équipe à laquelle il n'appartient pas, mais seulement dans le cas où toutes les ressources de son équipe sont déjà accédées. Une telle politique n'est pas compacte puisque lorsqu'un utilisateur accède à une ressource d'une équipe à laquelle

il n'appartient pas, retirer un accès sur une ressource de son équipe conduit à un état non sûr.

– Une politique est dite *libre* si lorsqu'un accès est autorisé pour un état sûr du système, il est autorisé pour tous les états sûrs[1] :

$$\forall \sigma \in \Sigma\ \forall s \in \mathcal{S}\ \forall o \in \mathcal{O}\ \forall a \in \mathcal{A}$$
$$((\exists \sigma' \in \Sigma\ (\Omega(\sigma') \wedge (s,o,a) \in \Lambda(\sigma'))) \wedge \Omega(\sigma)) \Rightarrow \Omega(\sigma \oplus (s,o,a))$$

Par exemple, HRU (uniquement basée sur un ensemble D d'accès autorisés) définit une politique libre puisque le prédicat caractérisant les états sûrs est défini en examinant chacun des accès courants indépendamment des autres accès en cours. En revanche, la politique de Bell et LaPadula n'est pas libre puisqu'il est possible qu'un sujet soit autorisé à écrire dans un objet lorsque ses accès en lecture vérifient certaines conditions et ne soit plus autorisé à écrire dans cet objet lorsque ses accès en lecture ne vérifient plus ces conditions.

Nous introduisons à présent la notion de modèle de contrôle d'accès, qui permet de spécifier comment passer d'un état du système à un autre. Pour cela, nous introduisons tout d'abord la notion de langage de requêtes.

Requêtes Une requête est soumise par un sujet afin de faire évoluer le système, soit en ajoutant ou en enlevant un accès, soit en changeant les informations de sécurité dynamiques du système. Dans ce dernier cas, on parle de *requêtes administratives*. Etant donné un ensemble de requêtes $\mathcal{R}$, il est nécessaire d'introduire une sémantique pour le langage de requêtes afin de pouvoir caractériser les modifications sur les états engendrées par l'application de ces requêtes lors d'une transition effectuée par le moniteur de référence. Le procédé le plus classique pour définir une sémantique du langage des requêtes consiste à introduire une "sémantique de transitions" *via* une relation $[\![\mathcal{R}]\!]_\Sigma \subseteq \Sigma \times \mathcal{R} \times \Sigma$. Avec une telle approche, $(\sigma_1, R, \sigma_2) \in [\![\mathcal{R}]\!]_\Sigma$ permet de spécifier les propriétés d'un état σ_2 lorsqu'il a été obtenu par application d'une requête R sur un état σ_1. Par exemple, la sémantique de la requête $\langle +, s, o, a\rangle$ de demande d'ajout d'un accès du sujet s sur l'objet o selon le mode a est spécifiée par :

$$(\sigma_1, \langle +, s, o, a\rangle, \sigma_2) \in [\![\mathcal{R}]\!]_\Sigma \Leftrightarrow (\Lambda(\sigma_2) = \{(s,o,a)\} \cup \Lambda(\sigma_1) \wedge \Upsilon(\sigma_1) = \Upsilon(\sigma_2))$$

Modèles et Implantations La donnée d'une politique $\mathbb{P}[\rho]$ et d'un langage de requêtes $\mathcal{R}$ muni d'une sémantique $[\![\mathcal{R}]\!]_\Sigma$ constitue un modèle :

$$\mathbb{M}[\rho] = (\mathbb{P}[\rho], \mathcal{R})$$

[1] $\sigma \oplus (s,o,a)$ (resp. $\sigma \ominus (s,o,a)$) dénote un état tel que :

$$\Lambda(\sigma \oplus (s,o,a)) = \Lambda(\sigma) \cup \{(s,o,a)\} \text{ et } \Upsilon(\sigma \oplus (s,o,a)) = \Upsilon(\sigma)$$
$$(\text{resp.}\Lambda(\sigma \oplus (s,o,a)) = \Lambda(\sigma) \backslash \{(s,o,a)\} \text{ et } \Upsilon(\sigma \oplus (s,o,a)) = \Upsilon(\sigma))$$

Implanter un modèle consiste à définir une paire (τ, Σ_I) où $\tau : \mathcal{R} \times \Sigma \to \mathcal{D} \times \Sigma$ est une fonction de transition entre états ($\mathcal{D}$ est l'ensemble des réponses possibles) et où Σ_I est l'ensemble des états initiaux possibles (il s'agit bien sûr d'un sous-ensemble des états sûrs). A ce niveau, il est possible de spécifier les propriétés attendues sur (τ, Σ_I). Les deux propriétés principales sont les propriétés de correction vis-à-vis de la politique (τ transforme tout état sûr en état sûr) et de la sémantique des requêtes (les états obtenus en appliquant τ à une requête R satisfont la relation $[\![\mathcal{R}]\!]_\Sigma$). Nous notons $\mathbb{M}[\rho] \vdash (\tau, \Sigma_I)$ lorsque toutes les propriétés de correction d'une implantation (τ, Σ_I) d'un modèle $\mathbb{M}[\rho]$ sont satisfaites.

Applications Le cadre formel décrit ci-dessus a été implanté avec l'atelier FoCaL et instancié pour obtenir une implantation des principales politiques que l'on peut rencontrer dans la littérature : [5] décrit une implantation de politique discrétionnaire à base de matrice de droits d'accès [16], [2] décrit les implantations d'une politique *à la* Unix intégrant la notion de groupes d'utilisateurs, d'une politique à base de "tickets" [24], et d'une politique discrétionnaire intégrant un mécanisme de délégation, [26] décrit une implantation de la politique de Bell et LaPadula [23, 3] et [13] décrit une implantation d'une politique à base de rôles (RBAC96) [10, 29].

3.2 Comparaison de modèles

Le cadre sémantique que nous venons d'introduire nous permet de formaliser les concepts nécessaires pour envisager la comparaison de modèles de contrôle d'accès. Intuitivement, un modèle de contrôle d'accès $\mathbb{M}_1[\rho_1]$ est plus restrictif qu'un modèle $\mathbb{M}_2[\rho_2]$ ssi toute implantation correcte de $\mathbb{M}_1[\rho_1]$ peut être simulée par une implantation correcte de $\mathbb{M}_2[\rho_2]$.

Préordre sur les modèles Afin de donner une définition du préordre sur les modèles, il faut préciser la notion de simulation d'implantations. Il s'agit du concept de simulation faible : la fonction de transition $\tau_2 : \mathcal{R}_2 \times \Sigma_2 \to \mathcal{D} \times \Sigma_2$ simule faiblement la fonction $\tau_1 : \mathcal{R}_1 \times \Sigma_1 \to \mathcal{D} \times \Sigma_1$, ce que nous notons $\tau_1 \overset{\kappa_\Sigma, \kappa_\mathcal{R}}{\rightharpoonup} \tau_2$, ssi il existe deux relations $\kappa_\Sigma \subseteq \Sigma_1 \times \Sigma_2$ (caractérisant les états "sémantiquement" équivalents) et $\kappa_\mathcal{R} \subseteq \Sigma_1 \times \mathcal{R}_1 \times \mathcal{R}_2^\star$ (mettant en correspondance les séquences de requêtes "sémantiquement" équivalentes) telles que :

$$\forall \sigma_1, \sigma_1' \in \Sigma_1 \quad \forall \sigma_2^0 \in \Sigma_2 \quad \forall R_1 \in \mathcal{R}_1 \quad \forall d_1 \in \mathcal{D}$$

$$\begin{pmatrix} \tau_1(R_1, \sigma_1) = (d_1, \sigma_1') \\ \wedge\ (\sigma_1, \sigma_2^0) \in \kappa_\Sigma \end{pmatrix} \Rightarrow \begin{pmatrix} \exists \sigma_2^1, \sigma_2^2, \ldots, \sigma_2^{n-1}, \sigma_2^n \in \Sigma_2 \\ \exists (R_2^1, \ldots, R_2^n) \in \mathcal{R}_2^\star\ \exists d_2^1, \ldots, d_2^n \in \mathcal{D} \\ \tau_2(R_2^1, \sigma_2^0) = (d_2^1, \sigma_2^1) \\ \wedge \cdots \wedge \tau_2(R_2^n, \sigma_2^{n-1}) = (d_2^n, \sigma_2^n) \\ \wedge\ d_1 = d_2^1 = d_2^2 = \cdots = d_2^n \\ \wedge\ (\sigma_1, R_1, (R_2^1, \ldots, R_2^n)) \in \kappa_\mathcal{R} \\ \wedge\ (\sigma_1', \sigma_2^n) \in \kappa_\Sigma \end{pmatrix}$$

Chaque transition de τ_1 doit donc pouvoir être simulée par une séquence de

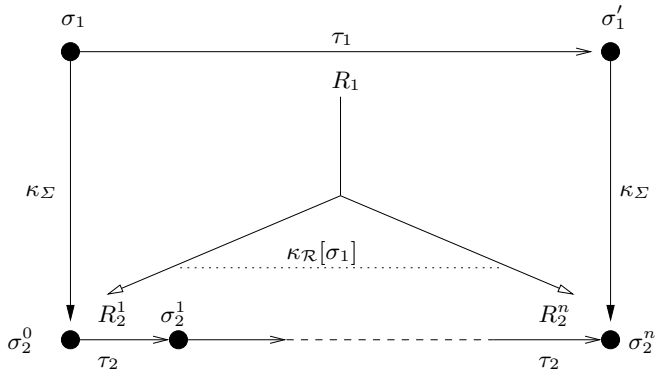


Fig. 2. Simulation faible

transitions de τ_2. En effet, selon le pouvoir d'expression des deux langages de requêtes considérés $\mathcal{R}_1$ et $\mathcal{R}_2$, il se peut que pour simuler l'effet d'une requête appartenant à $\mathcal{R}_1$, il soit nécessaire d'appliquer plusieurs requêtes de $\mathcal{R}_2$. C'est par exemple le cas pour la requête d'ajout d'un rôle autorisé pour un utilisateur dans le modèle RBAC, dont la simulation dans le modèle HRU nécessite d'ajouter plusieurs accès autorisés et donc d'appliquer plusieurs fois la requête du modèle HRU permettant d'autoriser un nouvel accès. Cette définition, illustrée sur la figure 2, est étendue aux implantations : l'implantation (τ_2, Σ_2^I) simule faiblement (τ_1, Σ_1^I), ce que nous notons $(\tau_1, \Sigma_1^I) \overset{\kappa_\Sigma, \kappa_\mathcal{R}}{\preceq} (\tau_2, \Sigma_2^I)$, ssi il existe deux relations $\kappa_\Sigma \subseteq \Sigma_1 \times \Sigma_2$ et $\kappa_\mathcal{R} \subseteq \Sigma_1 \times \mathcal{R}_1 \times \mathcal{R}_2^\star$ telles que :

$$\tau_1 \overset{\kappa_\Sigma, \kappa_\mathcal{R}}{\preceq} \tau_2 \wedge \forall \sigma_1 \in \Sigma_1^I \quad \exists \sigma_2 \in \Sigma_2^I \quad (\sigma_1, \sigma_2) \in \kappa_\Sigma$$

La relation $\kappa_\mathcal{R}$ est une relation ternaire : le jugement $(\sigma, R_1, (R_2^1, \cdots, R_2^n)) \in \kappa_\mathcal{R}$ exprime que la séquence de requêtes $(R_2^1, \cdots, R_2^n)$ simule la requête R_1 lorsque le système se trouve dans un état σ. En effet, la séquence de requêtes peut dépendre de l'état dans lequel se trouve le système. C'est par exemple le cas pour la requête d'ajout d'une permission (o, a) à un rôle r dans le modèle RBAC, dont la simulation dans le modèle HRU nécessite d'ajouter les accès (s, o, a) aux accès autorisés pour chaque sujet s ayant activé un rôle r' supérieur à r. Ici, l'ensemble des rôles activés par un sujet est défini par une fonction de sécurité, qui dépend donc de l'état du système.

Lors de la comparaison de modèles, les relations κ_Σ et $\kappa_\mathcal{R}$ utilisées doivent satisfaire certaines propriétés supplémentaires : d'une part κ_Σ et $\kappa_\mathcal{R}$ doivent être totales à gauche, et d'autre part, deux états en relation par κ_Σ doivent autoriser les mêmes accès (i.e. les ensembles d'accès que l'on peut ajouter aux accès courants de ces deux états sans violer la politique de sécurité sont identiques), on dira dans ce cas que κ_Σ est $\mathcal{W}$-préservante. Le préordre $\trianglelefteq$ sur les modèles

peut à présent être défini comme suit :

$$\mathbb{M}_1[\rho_1] \trianglelefteq \mathbb{M}_2[\rho_2]$$

$$\Leftrightarrow \left(\begin{array}{l} \forall \tau_1 : \mathcal{R}_1 \times \Sigma_1 \to \mathcal{D} \times \Sigma_1 \quad \forall \Sigma_1^I \subseteq \Sigma_1 \\ \\ \mathbb{M}_1[\rho_1] \vdash (\tau_1, \Sigma_1^I) \Rightarrow \left(\begin{array}{l} \exists \tau_2 : \mathcal{R}_2 \times \Sigma_2 \to \mathcal{D} \times \Sigma_2 \quad \exists \Sigma_2^I \subseteq \Sigma_2 \\ \exists \kappa_\Sigma \subseteq \Sigma_1 \times \Sigma_2 \; \exists \kappa_\mathcal{R} \subseteq \Sigma_1 \times \mathcal{R}_1 \times \mathcal{R}_2^\star \\ \quad \kappa_\Sigma \text{ et } \kappa_\mathcal{R} \text{ sont totales à gauche} \\ \wedge \; \kappa_\Sigma \text{ est } \mathcal{W}\text{-préservante} \\ \wedge \; \mathbb{M}_2[\rho_2] \vdash (\tau_2, \Sigma_2^I) \wedge (\tau_1, \Sigma_1^I) \; {}^{\kappa_\Sigma, \kappa_\mathcal{R}}\!\!\preceq (\tau_2, \Sigma_2^I) \end{array} \right) \end{array} \right)$$

Remarquons que la comparaison de deux modèles $\mathbb{M}_1[\rho_1]$ et $\mathbb{M}_2[\rho_2]$ nécessite la construction d'une relation de simulation pour chaque implantation correcte de $\mathbb{M}_1[\rho_1]$. Toutefois, en pratique, les relations κ_Σ et $\kappa_\mathcal{R}$ sont définies indépendamment de l'implantation considérée et il suffit alors de restreindre $\kappa_\mathcal{R}$ en fonction de la fonction de transition à simuler.

Application Un modèle pouvant admettre un nombre important d'implantations différentes (plus ou moins restrictives), la définition du préordre $\trianglelefteq$ sur les modèles est difficilement utilisable directement. Toutefois, en pratique, comparer deux modèles $\mathbb{M}_1[\rho_1]$ et $\mathbb{M}_2[\rho_2]$ conduit souvent à définir un "plongement" de $\mathbb{M}_1[\rho_1]$ vers $\mathbb{M}_2[\rho_2]$ satisfaisant de "bonnes propriétés" et à partir duquel on peut définir les relations de simulation κ_Σ et $\kappa_\mathcal{R}$. Sur tous les exemples concrets que nous avons envisagés, les relations obtenues satisfont de "bonnes propriétés" : elles préservent à la fois les propriétés de sécurité, définies par les prédicats de sécurité, et la sémantique des requêtes. Dans ce cas, il est possible de montrer directement que $\mathbb{M}_1[\rho_1] \trianglelefteq \mathbb{M}_2[\rho_2]$. Une fois établi, ce résultat générique permet donc de comparer deux modèles sans avoir à considérer les implantations de ces modèles, il suffit d'étudier les propriétés des relations de simulation.

En adoptant cette approche, nous avons prouvé que le modèle de la Muraille de Chine est strictement plus restrictif que le modèle de Bell et LaPadula, qui est lui même strictement plus restrictif que le modèle RBAC à base de rôles qui est lui même équivalent au modèle HRU à base de matrices de droits d'accès.

3.3 Analyse de flots d'informations

Dans cette sous-section, nous allons voir que le cadre formel introduit pour les modèles de contrôle d'accès permet de considérer ces modèles en termes de flots d'information qu'ils permettent. En effet, une politique de contrôle d'accès permet de spécifier quels sont les accès autorisés lorsque le système se trouve dans un certain état mais ne permet pas, tout du moins de manière explicite, de spécifier les flots d'information qui sont autorisés durant la vie du système. Ainsi, une fois qu'un sujet a pu accéder à un objet, il n'y a généralement aucun contrôle sur la propagation de l'information lue par le sujet. Par exemple, avec un modèle discrétionnaire, il est possible qu'un sujet s_1 lise un objet o_1 et recopie l'information lue dans un objet o_2 accessible en lecture par un sujet s_2 non autorisé à lire o_1. La politique de contrôle d'accès est ici respectée mais ne coïncide

pas avec son interprétation en termes de flots d'information. La cohérence entre ces deux lectures sémantiques des modèles de contrôle d'accès permet d'exprimer la correspondance entre les flots engendrés par les exécutions des implantations d'un modèle de contrôle d'accès et les politiques de confidentialité et d'intégrité induites par la politique de contrôle d'accès. Lorsque cette propriété de cohérence n'est pas vérifiée, il peut être utile de compléter le mécanisme de contrôle d'accès à l'aide d'un mécanisme de détection de flots.

Flots et Politiques de flots Nous étudions ici les flots d'information qui se produisent entre les entités du système. Nous distinguons plusieurs sortes de flots.

- Nous notons $\hookrightarrow^{OO}$ le produit cartésien $\mathcal{O} \times \mathcal{O}$. Un élément $o_1 \hookrightarrow^{OO} o_2$ de $\mathcal{O} \times \mathcal{O}$ permet d'exprimer que le contenu d'un objet o_1 est propagé dans un objet o_2. Nous caractériserons par la suite plusieurs sous-ensembles de $\hookrightarrow^{OO}$ permettant de décrire des flots d'information entre objets dans différents contextes. Une *politique de confinement* est un sous-ensemble $\rightsquigarrow^{OO}$ de $\hookrightarrow^{OO}$.
- Nous notons $\hookrightarrow^{OS}$ le produit cartésien $\mathcal{O} \times \mathcal{S}$. Un élément $o \hookrightarrow^{OS} s$ de $\mathcal{O} \times \mathcal{S}$ permet d'exprimer qu'un sujet s prend connaissance des informations contenues dans un objet o. Nous caractériserons par la suite plusieurs sous-ensembles de $\hookrightarrow^{OS}$ permettant de décrire des flots d'information des objets vers les sujets dans différents contextes. Une *politique de confidentialité* est un sous-ensemble $\rightsquigarrow^{OS}$ de $\hookrightarrow^{OS}$.
- Nous notons $\hookrightarrow^{SO}$ le produit cartésien $\mathcal{S} \times \mathcal{O}$. Un élément $s \hookrightarrow^{SO} o$ de $\mathcal{S} \times \mathcal{O}$ permet d'exprimer qu'un sujet s propage les informations dont il dispose dans un objet o. Nous caractériserons par la suite plusieurs sous-ensembles de $\hookrightarrow^{SO}$ permettant de décrire des flots d'information des sujets vers les objets dans différents contextes. Une *politique d'intégrité* est un sous-ensemble $\rightsquigarrow^{SO}$ de $\hookrightarrow^{SO}$.

Les flots d'information créés durant la vie d'un système sont caractérisés à partir d'une séquence d'états décrivant les états successifs du système. A partir de l'ensemble des accès courants qui ont lieu lorsque le système se trouve dans un état σ, nous pouvons définir les flots d'information entre objets comme suit. Le contenu d'un objet o_1 est diffusé dans un objet o_2 s'il existe un ensemble de sujets dont les accès sur les objets du système permettent de recopier l'information de o_1 dans o_2, cette information pouvant transiter par des objets intermédiaires :

$$\hookrightarrow_{\sigma}^{OO} = \left\{ o_1 \hookrightarrow^{OO} o_2 \mid \left(\begin{array}{l} \exists s_1, \cdots, s_k, s_{k+1} \in \mathcal{S}\ \exists o^1, \cdots, o^k \in \mathcal{O} \\ \left\{ \begin{array}{l} (s_1, o_1, \mathsf{read}), (s_1, o^1, \mathsf{write}), \\ (s_2, o^1, \mathsf{read}), (s_2, o^2, \mathsf{write}), \\ \cdots, \\ (s_i, o^{i-1}, \mathsf{read}), (s_i, o^i, \mathsf{write}), \\ \cdots, \\ (s_{k+1}, o^k, \mathsf{read}), (s_{k+1}, o_2, \mathsf{write}) \end{array} \right\} \subseteq \Lambda(\sigma) \\ \vee o_1 = o_2 \end{array} \right) \right\}$$

Cette définition est illustrée sur la partie gauche de la figure 3. Nous pouvons

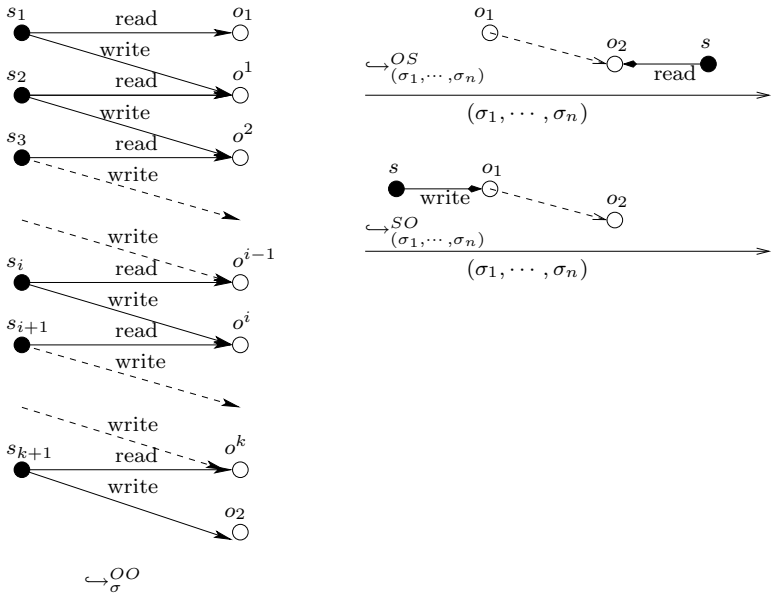


Fig. 3. Flots d'information

à présent définir les flots d'information qui ont lieu durant une séquence d'états $(\sigma_1, \cdots, \sigma_n)$. Les flots entre objets sont obtenus par composition des flots engendrés par chacun des états apparaissant dans la séquence :

$$\hookrightarrow^{OO}_{(\sigma_1,\cdots,\sigma_n)} = \begin{cases} \hookrightarrow^{OO}_{\sigma_1} & \text{si } n = 1 \\ \hookrightarrow^{OO}_{\sigma_{k+1}} \circ \hookrightarrow^{OO}_{(\sigma_1,\cdots,\sigma_k)} & \text{si } n = k+1 \end{cases}$$

Les flots d'un objet o vers un sujet s sont identifiés lorsqu'un objet o' a reçu l'information contenue dans o et que s accède ensuite en lecture à o' :

$$\hookrightarrow^{OS}_{(\sigma_1,\cdots,\sigma_n)} = \bigcup_{i=1}^{n} \left\{ o_2 \hookrightarrow^{OS} s \mid o_2 \hookrightarrow^{OO}_{(\sigma_1,\cdots,\sigma_i)} o_1 \wedge (s, o_1, \mathsf{read}) \in \Lambda(\sigma_i) \right\}$$

Enfin, les flots d'un sujet s vers un objet o sont identifiés lorsque s accède en écriture à un objet o' dont le contenu est ensuite diffusé dans o :

$$\hookrightarrow^{SO}_{(\sigma_1,\cdots,\sigma_n)} = \bigcup_{i=1}^{n} \left\{ s \hookrightarrow^{SO} o_2 \mid (s, o_1, \mathsf{write}) \in \Lambda(\sigma_i) \wedge o_1 \hookrightarrow^{OO}_{(\sigma_i,\sigma_{i+1},\cdots,\sigma_n)} o_2 \right\}$$

Ces définitions, illustrées sur la partie droite de la figure 3, sont étendues aux ensembles de séquences d'états. Si $E \subseteq \Sigma$ est un ensemble d'états et $F \subseteq E^\star$ est un ensemble de séquences d'états, on définit :

$$\hookrightarrow^{X}_{F} = \bigcup_{s \in F} \hookrightarrow^{X}_{s} \qquad \text{où } X \in \{OO, OS, SO\}$$

Une politique de contrôle d'accès $\mathbb{P}[\rho] = (\mathcal{S}, \mathcal{O}, \mathcal{A}, \Sigma, \Omega)$ peut être interprétée par une politique de confidentialité et/ou une politique d'intégrité. Exprimées en termes de flots d'information, ces politiques sont définies comme suit :

$$\rightsquigarrow^{OS}_{\mathbb{P}[\rho]} = \{o \hookrightarrow^{OS} s \mid \exists \sigma \in \Sigma_{|\Omega}\ (s, o, \mathsf{read}) \in \Lambda(\sigma)\}$$
$$\rightsquigarrow^{SO}_{\mathbb{P}[\rho]} = \{s \hookrightarrow^{SO} o \mid \exists \sigma \in \Sigma_{|\Omega}\ (s, o, \mathsf{write}) \in \Lambda(\sigma)\}$$

Ainsi, $o \rightsquigarrow^{OS}_{\mathbb{P}[\rho]} s$ signifie que pour un état sûr du système, le sujet s accède à l'information contenue dans l'objet o et $s \rightsquigarrow^{SO}_{\mathbb{P}[\rho]} o$ signifie que pour un état sûr du système, le sujet s propage l'information dont il dispose dans l'objet o.

En notant $Exec(I)$ l'ensemble des séquences d'états engendrées par l'implantation I d'un modèle $\mathbb{M}[\rho] = (\mathbb{P}[\rho], \mathcal{R})$, il est à présent possible d'exprimer formellement la propriété de cohérence introduite plus haut comme suit :

$$\hookrightarrow^{OS}_{Exec(I)} \subseteq \rightsquigarrow^{OS}_{\mathbb{P}[\rho]} \wedge \hookrightarrow^{SO}_{Exec(I)} \subseteq \rightsquigarrow^{SO}_{\mathbb{P}[\rho]}$$

Il a été prouvé que cette propriété de cohérence est vérifiée pour les politiques de la Muraille de Chine et de Bell et LaPadula (qui peuvent donc être vues comme de véritables politiques de flots). En revanche, cette propriété n'est pas vérifiée pour les modèles HRU et RBAC (qui sont des politiques libres). Toutefois, pour ces deux modèles, les propriétés permettant de garantir la cohérence ont été caractérisées : elles portent sur les informations de sécurité propres à ces modèles (matrice de droits d'accès, hiérarchie de rôles, permissions).

Mécanismes de détection de flots Lorsque la propriété de cohérence n'est pas satisfaite, on peut adjoindre un mécanisme de détection de flots au mécanisme de contrôle d'accès. Etant donné un ensemble de flots $\rightsquigarrow_{\mathbb{F}}$, détecter les flots appartenant à $\rightsquigarrow_{\mathbb{F}}$ lors de la vie d'un système consiste à observer les séquences d'états du système et à collecter des informations permettant d'identifier celles d'entre elles qui engendrent des flots recherchés. Chaque état σ est donc associé à une information $\psi(\sigma)$ qui rend compte d'une partie du passé de cet état. C'est à partir de cette information que l'on peut définir un prédicat $\mho$ sur Σ permettant de caractériser les états issus de séquences d'états engendrant un flot appartenant à $\rightsquigarrow_{\mathbb{F}}$. Ces états sont appelés des états d'alerte et on note $\Sigma_{|\mho}$ l'ensemble $\{\sigma \in \Sigma \mid \mho(\sigma)\}$. Etant donnés un ensemble E d'états observables et un ensemble $F \subseteq E^\star$ de séquences d'états qui peuvent se produire, un mécanisme de détection de flots est défini par :

$$\mathbb{F}[\rho, E, F, \rightsquigarrow_{\mathbb{F}}] = (\Sigma, \mho)$$

Bien sûr, il faut s'assurer que le prédicat $\mho$ caractérise bien les états issus d'une séquence produisant un flot dans $\rightsquigarrow_{\mathbb{F}}$. Pour cela, nous introduisons les deux propriétés classiques suivantes. $\mathbb{F}[\rho, E, F, \rightsquigarrow_{\mathbb{F}}] = (\Sigma, \mho)$ est :

– *pertinent* ssi tout état d'alerte est issu d'une séquence engendrant un flot dans $\rightsquigarrow_{\mathbb{F}}$:

$$\forall (\sigma_1, \cdots, \sigma_n) \in F \quad \mho(\sigma_n) \Rightarrow \hookrightarrow^{X}_{(\sigma_1, \cdots, \sigma_n)} \cap \rightsquigarrow_{\mathbb{F}} \neq \emptyset$$

– *fiable* ssi tout état issu d'une séquence engendrant un flot dans $\rightsquigarrow_{\mathbb{F}}$ est un état d'alerte :

$$\forall(\sigma_1, \cdots, \sigma_n) \in F \quad \hookrightarrow^{X}_{(\sigma_1,\cdots,\sigma_n)} \cap \rightsquigarrow_{\mathbb{F}} \neq \emptyset \Rightarrow \mho(\sigma_n)$$

où $X \in \{OO, OS, SO\}$.

Ce formalisme a été utilisé avec succès pour formaliser le mécanisme de détection d'intrusions pour le modèle HRU défini dans [11] ainsi que les preuves de fiabilité et de pertinence de ce mécanisme.

4 Conclusion – Perspectives

La sécurité, et plus particulièrement le contrôle d'accès, sont des problématiques actuelles en informatique. En effet, il devient aujourd'hui important de pouvoir contrôler les flots d'information dans les réseaux et dans les systèmes d'information. Il convient donc de développer au sein des systèmes informatiques des mécanismes permettant de filtrer les accès afin de ne laisser passer que ceux autorisés. La conception et le développement de ces mécamismes doivent être menés de manière à garantir leur fiabilité et leur sûreté. L'emploi des méthodes formelles dans le développement d'un moniteur de référence permet de garantir que certaines propriétés de sécurité sont toujours respectées. Dans cet article nous avons rendu compte de manière informelle de quelques expériences formelles que nous avons faites dans cette direction. L'ensemble de ces travaux a permis de définir un cadre formel pour les systèmes de contrôle d'accès. Il fournit un guide méthodologique lors de la conception d'un modèle et peut aider un utilisateur à adopter une démarche rigoureuse lors de son développement. De plus, exprimer des modèles au sein d'un même cadre permet de les comparer et d'obtenir une compréhension plus fine des propriétés souhaitées pour le modèle en cours d'élaboration.

Les perspectives de ces travaux sont multiples. Les modèles étudiés jusqu'à présent expriment des propriétés de sécurité sur les accès autorisés au sein d'un système. Toutefois, ils fournissent un langage de requêtes qui permet l'ajout ou le retrait d'un accès mais qui permet également de modifier les configurations de sécurité du système (requêtes administratives). Un axe de recherche à développer concerne donc l'étude des politiques administratives permettant de régir les modifications des informations de sécurité dynamiques. Cette étude pourrait être le point de départ pour envisager la notion de méta-politiques, permettant de régir les changements d'une politique de sécurité dynamique. De telles politiques pourraient être utilisées avec profit pour maintenir des propriétés de sécurité dans un contexte distribué (chaque site étant régi par une politique dynamique dont l'évolution est régie par une méta-politique).

L'étude de la comparaison de modèles de contrôle d'accès est un premier pas vers l'étude de la composition de ces modèles. Cette problématique mérite aussi d'être développée puisqu'elle correspond à un problème concret très répandu dans les systèmes d'information. En effet, dans la plupart de ces systèmes, un sujet accède généralement à un objet en passant au travers de plusieurs filtres. Par exemple, dans certains systèmes, les séquences d'accès peuvent être

régies par deux politiques : chaque accès est régi par une politique à base de rôles (RBAC), et le séquencement des accès est régi par une politique à base de tâches (TBAC). Il existe évidemment plusieurs procédés pour composer des politiques de sécurité mais il y a peu d'études sur les propriétés de ces compositions. Cette problématique mérite aussi d'être considérée afin de formaliser les concepts liés à la composition pour pouvoir non seulement identifier certains opérateurs génériques de composition, mais aussi pour exprimer les propriétés de sécurité que les mécanismes de composition envisagés peuvent garantir. Les mécanismes de comparaison déjà introduits pourraient s'appliquer pour définir certaines de ces propriétés.

Remerciements. Je remercie vivement Lionel Habib, Thérèse Hardin, Ludovic Mé, Charles Morisset et Valérie Viet Triem Tong avec qui j'ai la chance de travailler sur le domaine du contrôle d'accès et qui ont contribué à ce travail.

References

1. P. Amey. Dear sir, yours faithfully: an everyday story of formality. In *Practical Elements of Safety, Proc. of the Twelfth Safety-critical Systems Symposium*. Spinger-Verlag, 2004.
2. F. Anseaume, J. Baron, P. Berthelin, M. Jacquel, and D. Pipon. Formalisation, spécification et implantation de politiques de contrôle d'accès avec l'atelier Focal. Master's thesis, UPMC, Paris, France, 2008.
3. D. Bell and L. LaPadula. Secure Computer Systems: a Mathematical Model. Technical Report MTR-2547 (Vol. II), MITRE Corp., Bedford, MA, May 1973.
4. J. Blond and C. Morisset. Un moniteur de référence sûr d'une base de données. *Technique et Science Informatiques*, 26(9):1091–1110, 2007.
5. F. Brecht and A.D. Kadja. Implantation d'une politique de contrôle d'accès discrétionnaire avec Focal. Master's thesis, UPMC, Paris, France, 2007.
6. D. F. C. Brewer and M. J. Nash. The chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pages 206–214, 1989.
7. M. Carlier and C. Dubois. Functional testing in the Focal environment. In B.Beckert and R.Hähnle, editors, *Tests and Proofs, Second International Conference, TAP 2008, Prato, Italy, April 9-11, 2008. Proceedings*, volume 4966 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2008.
8. A. Santana de Oliveira. *Réécriture et Modularité pour les Politiques de Sécurité*. Phd thesis, Université Henri Poincaré, 2008.
9. C. Dubois, T. Hardin, and V. Viguié Donzeau Gouge. Building certified components within Focal. In *Symposium on Trends in Functional Programming*, 2004.
10. D. F. Ferraiolo and D. R. Kuhn. Role-based access control. In *Proceedings of the 15th National Computer Security Conference*, 1992.
11. G.Hiet, L.Mé, J.Zimmermann, C.Bidan, B.Morin, and V. Viet Triem Tong. Détection fiable et pertinente de flux d'information illégaux. In *6th Conference on Security and Network Architectures (SARSSI)*, 2007.
12. E. Gureghian, Th. Hardin, and M. Jaume. A full formalisation of the Bell and Lapadula security model. Technical Report 2003-007, Univ. Paris 6, LIP6, 2003.
13. L. Habib. Formalisation, comparaison et implantation d'un modèle de contrôle d'accès à base de rôles. Master's thesis, UPMC, Paris, France, 2007.

14. L. Habib, M. Jaume, and C. Morisset. A formal comparison of the Bell & LaPadula and RBAC models. In *Fourth International Symposium on Information Assurance and Security IAS'08*, pages 3–8. IEEE CS Press, 2008.
15. T. Hardin, M. Jaume, and C. Morisset. Access control and rewrite systems. In *1st International Workshop on Security and Rewriting Techniques, SecRet'06 (Satellite Workshop to ICALP'2006)*, 2006.
16. M. Harrison, W. Ruzzo, and J. Ullman. Protection in operating systems. *Communications of the ACM*, 19:461–471, 1976.
17. M. Jaume. *Descriptions formelles - Application au contrôle d'accès*. Habilitation à diriger des recherches, Université Paris 6, 2008.
18. M. Jaume and C. Morisset. Formalisation and implementation of access control models. In *Information Assurance and Security (IAS'05) International Conference on Information Technology, ITCC*, pages 703–708. IEEE CS Press, 2005.
19. M. Jaume and C. Morisset. A formal approach to implement access control. *Journal of Information Assurance and Security*, 2:137–148, 2006.
20. M. Jaume and C. Morisset. Towards a formal specification of access control. In P. Degano, R. Kusters, L. Vigano, and S. Zdancewic, editors, *Proceedings of the Joint Workshop on Foundations of Computer Security and Automated Reasoning for Security Protocol Analysis, FCS-ARSPA'06*, pages 213–232, 2006.
21. M. Jaume and C. Morisset. Contrôler le contrôle d'accès : Approches formelles. In *Approches Formelles dans l'Assistance au Développement de Logiciels, AFADL'07*, 2007.
22. M. Jaume and C. Morisset. Un cadre sémantique pour le contrôle d'accès. *Technique et Science Informatiques*, 27(8):951–976, 2008.
23. L.J. LaPadula and D.E. Bell. Secure Computer Systems: A Mathematical Model. *Journal of Computer Security*, 4:239–263, 1996.
24. H.M. Levy. *Capability-Based Computer Systems*. Digital Press, Bedford, MA, 1984.
25. McLean. The algebra of security. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7. IEEE Computer Society Press, 1988.
26. C. Morisset. *Sémantique des systèmes de contrôle d'accès*. PhD thesis, Université Pierre et Marie Curie, 2007.
27. C. Morisset and A. Santana de Oliveira. Automated detection of information leakage in access control. In *Proceedings of the 2nd International Workshop on Security and Rewriting Techniques (SecReT'07)*, Paris, France, 2007.
28. Focal project. *Focal, version 0.3.1 Tutorial and reference manual*. LIP6 – INRIA – CNAM, sept 2006. Distribution available at: `http://focal.inria.fr`.
29. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

Une approche incrémentale combinant test et extraction de modèles

Roland Groz[2], Muzammil Shahbaz[1], and Keqin Li[2]

[1] France Telecom R&D
Meylan, France.
muhammad.muzammilshahbaz@orange-ftgroup.com
[2] LIG, Computer Science Lab
Grenoble Universités, France
{Keqin.Li,Roland.Groz}@imag.fr

Un des principaux obstacles à l'utilisation d'approches formelles est l'absence de modèles. Nous nous intéressons ici au problème de l'intégration et du test de composants logiciels. Dans la pratique industrielle actuelle, il est de plus en plus fréquent de travailler en intégrant des composants logiciels externes dont aucun modèle n'est disponible. La documentation fournit en général une spécification insuffisante.

Afin de guider l'intégration, nous proposons d'utiliser des algorithmes d'inférence de machines pour extraire des modèles des composants. L'interaction entre ces modèles dans une composition permet de déduire d'autres tests qui peuvent être confrontés au système. L'ingénieur chargé de l'intégration dispose ainsi d'outils lui permettant de découvrir les interactions effectives entre les composants et de guider son processus d'intégration et de test pour valider les combinaisons de comportements. Nous travaillons sur des modèles d'automates étendus avec des paramètres et des prédicats, mais sans variables internes.

1 Introduction

Alors que le développement logiciel était pendant longtemps une activité intégrée au sein d'une entreprise qui fabriquait l'ensemble de ses produits logiciels, on procède de plus en plus par assemblage de composants provenant de sources externes.

Cette évolution implique de nouveaux défis pour intégrer les approches formelles dans le développement logiciel du côté de l'intégrateur. D'abord, les composants externes sont rarement accompagnés de spécifications formelles. Plus largement, on est confronté à l'absence ou à la non-disponibilité des modèles ou des éléments qui ont permis de construire ces composants. En général, on dispose d'une documentation qui n'est pas forcément suffisamment précise pour répondre aux questions que se pose l'intégrateur, et qui ne peut pas être intégrée directement dans les processus de développement de l'assemblage.

Nous proposons une approche formelle assistant l'intégrateur de composants dans sa tâche d'expérimentation et de test de composants dans une architecture définie a priori. Nous voulons permettre une approche de *test* basée sur

des *modèles* (avec génération automatique de tests), même en l'absence de modèles initiaux pour les composants. Pour cela, nous combinons une approche descendante de génération de tests (pour l'assemblage) à partir de modèles avec une approche ascendante de reconstruction de modèles (de chaque composant) à partir des observations issues du test.

L'objectif est de permettre un test suffisant pour mettre en évidence les interactions des composants dans les divers cas d'usage de l'assemblage. On ne cherche donc pas à dériver un modèle complet des composants, mais une *approximation* en phase avec l'objectif. En général, dans un assemblage, seule une petite partie du fonctionnement d'un composant est sollicitée. Comme le modèle correspondant à cette partie sera dérivé des tests, la génération de tests sur un seul composant ne serait pas pertinente pour découvrir des erreurs (puisque les tests seraient déduits du composant lui-même, donc corrects par construction). C'est pourquoi notre approche prend tout son sens dans un contexte *d'intégration*, où les modèles permettent de déduire de nouvelles interactions à tester. En outre, l'expérience montre que les problèmes d'interfonctionnement entre composants proviennent souvent du traitement de certaines valeurs échangées, alors qu'en général les interfaces sont cohérentes puisqu'elles ont bien été étudiées dans la définition de l'architecture d'intégration. C'est pourquoi nous accordons une importance particulière à l'extraction de modèles paramétrés : chaque interaction entre composants porte également des valeurs.

1.1 Une approche globale

Nous travaillons sur une représentation des composants par des automates communicants. La communication se fait par des symboles d'entrées-sorties, représentant un type d'interaction, enrichis par des paramètres représentant les valeurs échangées lors de l'interaction. Les symboles correspondent par exemple aux primitives décrites dans la documentation du composant.

Nous faisons les hypothèses suivantes.

- Les composants sont des *boîtes noires*. On connaît leurs interfaces, c'est à dire les types des entrées, et accessoirement de leurs sorties. Il est possible qu'on n'en connaisse qu'un sous-ensemble. On connaît par exemple les primitives de services offertes par le composant et le type de leurs paramètres. C'est ce dont disposerait un ingénieur avec la documentation du composant.
- Nous nous plaçons dans une hypothèse de fonctionnement *réactif* du système. Les entrées globales ne seront fournies au système que dans un état stable de celui-ci, où il n'y a plus d'interaction interne possible. On supposera aussi que même si la communication entre composants du système est asynchrone, il n'y a qu'un flot de traitement actif (un seul message en transit dans le système).
- On dispose de *scénarios d'usage* du système permettant d'avoir un ensemble de séquences d'entrées et des valeurs significatives des paramètres associés. L'ingénieur qui a conçu l'architecture pour composer un service à partir des composants aura aussi élaboré des scénarios d'usage typiques,

et il s'agit, maintenant que l'architecture est définie, de voir si les composants interagiront bien selon les attentes qu'on en a. Un apport clé de notre approche va être de permettre l'enrichissement des tests, et la déduction automatique de tests systématiques à partir d'un nombre restreint de scénarios de test.
- Les composants sont intégrés, mais peuvent être aussi testés isolément. Lorsqu'ils sont intégrés, certaines de leurs interfaces sont connectées entre elles (interfaces internes), d'autres restent ouvertes pour communiquer avec l'environnement. On suppose que les interfaces internes restent *observables* (mais pas forcément contrôlables).
- Accessoirement, il se peut qu'on dispose de modélisations très partielles, sous forme d'automates à entrées et sorties (paramétrées), de certains composants, représentant par exemple leur flot de contrôle pour certains scénarios. Notre approche saurait intégrer de tels modèles, mais en leur absence, on peut reconstruire des modèles à partir de rien.

Notre approche va consister à tester d'abord chaque composant selon un *algorithme d'inférence d'automate* qui permettra d'en dériver un premier modèle cohérent avec les observations sur un nombre restreint d'entrée sorties correspondant à celles des scénarios d'usage fournis initialement. Ensuite, nous assemblerons les modèles, et dériverons des tests d'intégration nous permettant de confronter le fonctionnement du système réel aux modèles préliminaires. Ceci nous permettra d'enrichir ces modèles et de déduire de nouveaux test correspondants aux cas "croisés", d'interactions entre composants qui n'avaient pas été testées dans les scénarios antérieurs. Nous avons donc une approche incrémentale et itérative alternant les phases de test avec les phases d'enrichissement des modèles. Le processus peut être arrêté lorsqu'on ne détecte plus d'incohérence entre les modèles et le système ou lorsqu'on a atteint un certain objectif de couverture du système.

Dans la suite de cet article, nous présentons d'abord un exemple simple permettant d'illustrer les algorithmes d'inférence d'automates. Ensuite, nous illustrons l'algorithme lorsqu'on se restreint à des automates sans paramètres (machines de Mealy). Nous étendons le modèle pour prendre en compte les paramètres et expliquons comment l'algorithme peut être étendu. Enfin, nous présentons la méthode d'intégration qui s'appuie sur les algorithmes d'inférence.

2 Un exemple de composant à identifier

Nous donnons un exemple de contrôleur de climatiseur qui régule un système de chauffage et de ventilation. La structure interne du contrôleur et le fonctionnement du système encapsulé par ce composant ne sont pas connus. Il s'agit donc d'un composant considéré comme une boîte noire. Par contre, on peut connaître un ensemble d'entrées en analysant ses interfaces externes. La figure 1 présente un diagramme global du contrôleur.

Le contrôleur de climatiseur accepte des entrées venant de l'environnement pour contrôler le système. Il reçoit les signaux *Marche* et *Arrêt* pour allumer

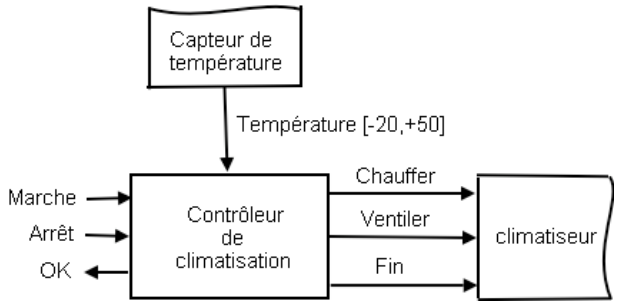


Fig. 1. Un diagramme global de climatiseur

et éteindre le système et Température T qui change le mode du système. Il accepte des valeurs de température entre -20 Celsius et $+50$ Celsius, qui sont des paramètres de l'entrée T. Ensuite, il donne une commande *Chauffer* pour mettre en route le chauffage s'il fait froid (entre -20°Cet 11°C) ou *Ventiler* pour la ventilation s'il fait chaud (entre 16°Cet 50°C). La commande de sortie porte des paramètres pour régler la vitesse de système, par exemple, s'il ne fait pas très froid, le contrôleur envoie une commande pour ralentir le chauffage. De même, il envoie une commande pour relancer la climatisation s'il fait très chaud, sinon, il peut suffire de ventiler.

Cet exemple classique va nous servir à illustrer le fonctionnement des algorithmes d'inférence de modèles. Dans un premier temps, nous montrerons comment on peut en apprendre un modèle d'automate simple, en faisant une abstraction des paramètres. Puis nous étendons l'algorithme pour en apprendre un modèle paramétré.

3 Apprentissage d'automates simples

Le modèle d'automate que nous considèrerons sera celui des automates à entrées et sorties (appelés aussi machines de Mealy). Un automate est un sextuplet $M = (Q, I, O, \delta, \lambda, q_0)$, où Q est l'ensemble des états, $q_0 \in Q$ est l'état initial, I l'alphabet d'entrée, O celui des sorties, $\delta : Q \times I \longrightarrow Q$ est la fonction de transition d'états et $\lambda : Q \times I \longrightarrow O$ est la fonction de sortie. Les ensembles Q, I, O sont bien sûr finis. On s'intéressera à des automates complètement définis, c'est à dire tels que $dom(\delta) = dom(\lambda) = Q \times I$. Au besoin, on complètera pour cela l'automate avec un symbole supplémentaire Ω qui sera une abstraction pour la réponse d'un système à des entrées non valides dans un état donné, sur lequel on rajoutera une boucle étiquetée par Ω en sortie.

Nous cherchons à déduire la structure de contrôle d'un composant qu'on peut tester en boîte noire. Pour cela, nous nous sommes intéressés aux travaux similaires menés soit dans le domaine de l'ingénierie à partir de scénarios [13] [15] [4], soit en vérification [5] [6] ou en test [8] [9]. Dans ce domaine, l'algorithme de base le plus efficace dans ce contexte de test actif est l'algorithme d'Angluin [1]. C'est

un algorithme de complexité polynomiale (en nombre de requêtes au système boîte noire) conçu pour des automates accepteurs déterministes. Notre travail actuel sur l'inférence de machines se situe dans la lignée d'adaptations et d'extensions à l'algorithme d'Angluin pour des modèles comportant des entrées et des sorties avec des paramètres et des prédicats.

Dans la plupart des travaux antérieurs, l'algorithme d'Angluin a été utilisé sans modification avec une simple correspondance entre les symboles d'entrées et de sortie et l'alphabet A d'un automate accepteur. Par exemple, en prenant $A = I \cup O$ [7] [9], ou en prenant des couples (entrée,sortie) $A = I \times O$ [13] comme lettres de l'alphabet. Nous avons proposé une adaptation de l'algorithme dans laquelle on remplace la notion d'acceptation (décision binaire, codée par 0 et 1 dans l'algorithme d'Angluin) par les sorties correspondant aux derniers symboles entrés [11]. Par ailleurs, l'algorithme d'Angluin a été initialement proposé dans le contexte d'un apprentissage dans lequel un oracle connaissant le contenu de la boîte noire peut être interrogé de deux façons.

- par une requête d'appartenance, qui permet de savoir si une séquence d'entrées est un mot accepté (reconnu) par l'automate ; dans notre approche, c'est le composant qui servira d'oracle et fournira les sorties correspondant à la séquence d'entrée
- par une requête d'équivalence, qui permet de savoir si l'automate minimal cohérent avec les observations enregistrées qu'on a pu reconstruire est équivalent à l'automate caché dans la boîte noire ; s'il ne l'est pas, l'oracle fournit un contre-exemple, i.e. une séquence reconnue par la boîte noire mais pas par l'automate construit ; dans notre approche, il n'y a pas de tel oracle omniscient, c'est le test d'intégration qui fournira un éventuel contre-exemple, ou le critère d'arrêt du test qui terminera l'algorithme sur un modèle approché.

Noter qu'on supposera toujours que le composant est réinitialisable ce qui permet de reprendre toutes les séquences de test à partir de l'état initial.

3.1 Algorithme pour machine de Mealy

Nous ne décrivons pas ici le détail de l'algorithme qui a été présenté ailleurs [11], mais nous en donnons les principes et une illustration sur l'exemple du climatiseur. Conformément à l'algorithme d'Angluin, on note les observations faites au cours du test dans une table appelée table d'observation, qui sera étendue progressivement au fur et à mesure des observations. Nous notons cette table (S, E, TO). Les lignes de cette table sont étiquetées par des séquences d'entrées, qui correspondent à des préfixes pour atteindre les états de l'automate. Soit S l'ensemble de ces séquences. On impose à S d'être clos par préfixe. Les colonnes sont elles aussi étiquetées par des séquences, d'un ensemble E auquel on imposera d'être clos par suffixe.

A l'intersection d'une ligne s de S et d'une colonne e de E on portera dans la table la valeur $TO(s, e)$ qui sera la séquence des sorties produites par le composant en réponse à la séquence e, lorsqu'il a été préalablement positionné dans un état par la séquence s.

On commence l'algorithme avec $S = \{\epsilon\}$ (le mot vide) et $E = I$ (l'ensemble des symboles d'entrées du composant). On remplit la première (et seule) ligne avec les résultats des tests consistant à appliquer chacune des entrées dans l'état initial : dans chaque case, on trouve donc la (ou les) sortie(s) correspondant à cette entrée (celle de la colonne de cette case).

On vérifie ensuite la "complétude" de la table en rajoutant toutes les lignes correspondant à $S \cdot I$. A tout moment, dans l'algorithme, il y aura ainsi deux parties dans la table : la partie haute, des lignes étiquetées par S et la partie basse qui étend celles-ci d'une entrée, pour toutes les entrées possibles. Un exemple de table d'observation est donné en figure 2.

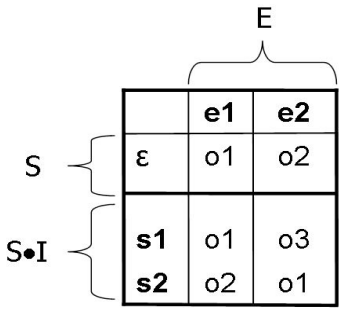


Fig. 2. Structure d'une table d'observation

La table sera dite *close* si toute ligne t de $S \cdot I$ est le doublon d'une ligne de S, i.e ; s'il existe s de S tel que $ligne(s) = ligne(t)$. Elle sera dite *cohérente* si lorsque deux éléments de S, s_1 et s_2 ont des lignes identiques, alors pour toute entrée a de I, $ligne(s_1 \cdot a) = ligne(s_2 \cdot a)$.

Lorsqu'on arrive à un état où la table est *close* et *cohérente* , comme illustré sur la table 1, alors on construit un modèle minimal d'automate (la conjecture) qui permet d'expliquer toutes ces observations de la façon suivante.

- $Q = \{ligne(s) : s \in S\}$
- $q_0 = ligne(\epsilon)$
- $\delta(ligne(s), a) = \{ligne(s \cdot a), a \in I\}$
- $\lambda(ligne(s), a) = \{TO(s \cdot a), a \in I\}$

Chaque séquence préfixe $s \in S$ est un chemin d'accès qu'on associe à un état. Deux séquences peuvent conduire au même état, et la propriété de cohérence garantit qu'alors les observations ultérieures restent cohérentes (pour toutes les entrées et suffixes discriminants déjà recensés). La propriété de clôture garantit qu'on n'atteint pas de nouveaux états, différents des précédents, en prolongeant la séquence. Il est assez facile de montrer que l'automate construit est compatible avec la table d'observation si elle est *close* et *cohérente* , c'est à dire que pour

tout s de $S \cup S \cdot I$ et pour tout e de E, $\lambda(q_0, s \cdot e) = \lambda(q_0, s) \cdot TO(s \cdot e)$ (en ayant étendu λ aux séquences d'entrée). On peut par ailleurs montrer comme dans [1] que l'automate ainsi obtenu est l'automate minimal compatible avec la table d'observation.

Lorsque la table n'est pas *close* , cela signifie qu'on a découvert une nouvelle séquence du type $s' = s \cdot a$ menant à un état inéquivalent aux états précédemment identifiés. On rajoute alors cette séquence s' à S (la ligne correspondante passe de la partie basse à la partie haute de la table), et on complète la table en partie basse par les lignes correspondant à $s' \cdot I$. Lorsque la table n'est pas *cohérente* , on a trouvé deux éléments s_1 et s_2 tels que $ligne(s_1) = ligne(s_2)$ alors qu'il existe a de I et e de E tels que $ligne(s_1 \cdot a \cdot e) \neq ligne(s_2 \cdot a \cdot e)$. Ce qui signifie que $a \cdot e$ est une séquence discriminant les deux états atteints par s_1 et s_2. On rajoute $a \cdot e$ à E et on complète la table (entre autres, pour vérifier si cette séquence pourrait discriminer d'autres préfixes). On continue ainsi jusqu'à obtenir une table *close* et *cohérente* , qui permet de construire un automate conjecturé. Les préfixes présents dans S représentent les chemins d'accès à des états "candidats". Les suffixes présents dans E représentent les séquences dont on a pu établir jusqu'à ce stade qu'elles permettent de distinguer les états 2 à 2.

Une conjecture, qui est compatible avec toutes les observations faites jusqu'à son établissement, peut être remise en cause si des observations ultérieures ne sont pas conformes aux prédictions de l'automate. Dans l'algorithme original d'Angluin, c'est un expert qui fournit un contre-exemple. Dans notre approche, le contre-exemple viendra d'un test ultérieur dans une phase d'intégration. On va alors enrichir la table en étendant S avec le contre-exemple et tous ses préfixes (non déjà présents dans la table). On complète alors la table avec les observations requises jusqu'à aboutir à une nouvelle conjecture.

L'intérêt de l'algorithme est que si le composant a n états, l'algorithme terminera en un temps polynomial en n. Il en sera de même si le composant peut être abstrait en un système à n états en ne distinguant pas les valeurs des paramètres.

On peut noter que l'algorithme est incrémental puisque l'insertion d'un contre-exemple ne fait que rajouter des lignes à la table précédemment construite. Cette caractéristique est également utile si on dispose d'un modèle initial du composant sous forme d'un automate. On peut associer à cet automate une table pour laquelle S est construit à partir des plus courtes séquences d'accès à chaque état et $E = I$. Il faut bien sûr que l'automate fourni initialement soit conforme au fonctionnement du système au moins sur les séquences de $S \cup S \cdot I \cdot I$.

3.2 Apprentissage du contrôleur de climatiseur comme machine de Mealy

Dans une machine de Mealy, il ne peut y avoir de paramètres sur les symboles d'entrée et de sortie. On pourrait certes associer un symbole différent à chaque valeur du paramètre. Par exemple, $T(12)$ pourrait être un symbole d'entrée qu'on pourrait noter $T12$. Mais ceci aurait plusieurs inconvénients.

- On devrait discrétiser l'ensemble du domaine des paramètres d'entrée. Pour les températures de notre exemple (de -20°C à +50°C), il y aurait 71 valeurs,

	M	A	BT	MT	HT
ϵ	OK	Ω	Ω	Ω	Ω
M	Ω	Ω	C	F	V
$M-BT$	Ω	F	C	F	Ω
$M-HT$	Ω	F	Ω	F	V
$M-MT$	Ω	Ω	C	F	V
$M-BT-A$	OK	Ω	Ω	Ω	Ω
$M-BT-BT$	Ω	F	C	F	Ω
$M-BT-MT$	Ω	Ω	C	F	V
$M-HT-A$	OK	Ω	Ω	Ω	Ω
$M-HT-MT$	Ω	Ω	C	F	V
$M-HT-HT$	Ω	F	Ω	F	V

Tab. 1. Une table *close* et *cohérente* pour apprendre le contrôleur de climatiseur

si on s'en tient à des valeurs entières, or il pourrait être souhaitable d'avoir un grain plus fin. Pour des valeurs flottantes, on pourrait donc avoir un grand nombre de valeurs.
- Ceci conduirait à une complexité inutile de l'automate, rédhibitoire pour l'apprentissage et l'utilisation.
- On perdrait la structure de contrôle de l'automate, en la mélangeant avec les données. Or l'intérêt de la modélisation est de pouvoir extraire le contrôle pour en déduire une abstraction représentative et pertinente pour l'utilisation de techniques de génération de tests.

Pour apprendre un modèle de Mealy du contrôleur de climatiseur, nous avons besoin de réarranger son ensemble d'entrée pour que l'algorithme puisse être utilisé commodément. Ceci ne donnera pas un fonctionnement détaillé du contrôleur mais une abstraction de son comportement face à un changement de température.

Par exemple, nous distinguons trois sous-domaines de température, auxquels nous associons donc 3 entrées. Nous remplaçons T par MT (la température moyenne, entre 12°C et 15°C), BT (température basse, entre -20°C et 11°C) et HT (température haute, entre 16°C et 50°C). Donc, nous construisons $I = \{M, A, BT, MT, HT\}$ pour l'algorithme de machine de Mealy. La table 1 est la table d'observation obtenue en fin d'exécution de l'algorithme, quand la table est *close* et *cohérente* . La figure 3 est la conjecture pour le contrôleur comme machine de Mealy déduite de la table. Pour simplifier, les lignes qui ne correspondent à aucune sortie ne sont pas affichées dans la table. Aussi, nous utilisons les abréviations à la place des noms complets des entrées et sorties dans la table et la figure : M (Marche), A (Arrête), C (Chauffer), V (Ventiler), F (Fin).

Il faut noter que plus on raffinera les domaines d'entrée, plus on augmentera la taille de la machine de Mealy calculée, il faut donc éviter une explosion du nombre d'états.

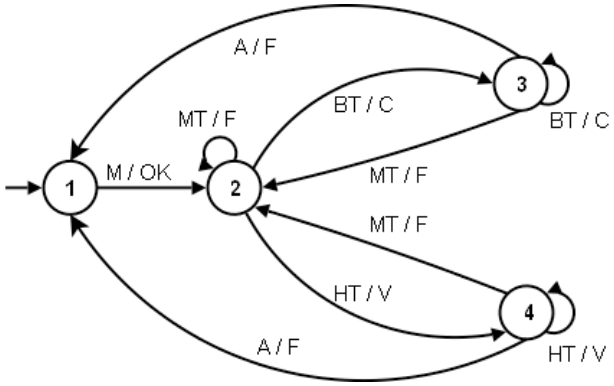


Fig. 3. Conjecture de Mealy

4 Modèle d'automates paramétrés

Comme on l'a noté, le modèle d'automates sans paramètres nécessite des abstractions de modélisation délicates. En outre, dans le contexte du test, il présente l'inconvénient majeur de ne pas instancier les valeurs. Les modèles d'automates obtenus pour les composants ne permettraient donc pas de produire des séquences de test complètes. C'est pourquoi nous proposons d'étendre le modèle pris en compte à des automates dans lesquels les entrées et les sorties sont paramétrés. Afin de prendre en compte les calculs différents auxquels peuvent conduire des valeurs différentes des paramètres pour une même entrée, on considèrera également que les transitions de l'automate peuvent être accompagnées d'une garde (prédicat) portant sur les valeurs des paramètres d'entrée. En cela, nous étendons le modèle paramétré que nous avions proposé dans [12]

En revanche, nous considérons des automates sans variables internes. En effet, autant on peut observer les valeurs des entrées et sorties de la boîte noire, autant on n'a pas accès aux états internes et à la structure de la machine. Il n'est donc pas possible de distinguer entre une mémorisation par des structures de contrôle ou dans des mémoires. Nous discutons cette restriction plus bas.

Le modèle que nous proposons est donc le suivant. Un automate paramétré (PFSM) est un septuplet : $M = (Q, I, O, D_I, D_O, T, q_0)$.

- Q est un ensemble fini d'états
- $q_0 \in Q$ est l'état initial.
- I est un ensemble fini de symboles d'entrée.
- O est un ensemble fini de symboles de sortie.
- D_I est un ensemble de valeurs du paramètre d'entrée.
- D_O est un ensemble de valeurs du paramètre de sortie.
- T est l'ensemble des transitions.

Une transition $t \in T$ est décrite par : $t = (q, q', i, o, p, f)$, où $q \in Q$ est l'état de départ, $q' \in Q$ est l'état d'arrivée, $i \in I$ le symbole d'entrée, $o \in O$ celui de sortie, $p \subseteq D_I$ est un prédicat sur la valeur du paramètre et $f : p \longrightarrow D_O$ est la

fonction de calcul du paramètre de sortie pour cette transition : elle associe la valeur du paramètre de sortie correspondant à celle du paramètre d'entrée.

Noter que les ensembles D_I et D_O ne sont pas nécessairement finis. Cependant, les modèles que nous construirons par apprentissage resteront des approximations finies, dans lesquels les valeurs explorées resteront dans des sous-domaines finis, avec dans tous les cas un ensemble fini de transitions.

Nous imposons néanmoins les restrictions suivantes aux machines que nous considèrerons : nous supposerons qu'elles seront complètes, déterministes, et observables.

Propriété 1 (Complet) *Un automate PFSM est complet si* $\forall q \in Q$, $\forall i \in I$ *et* $\forall x \in D_I$, $\exists t \in T$ *tel que* $t = (q, q', i, o, p, f)$, *avec* $x \in p$.

Comme pour les machines de Mealy simples, on peut compléter un automate paramétré en ajoutant des transitions bouclant sur l'état pour toutes les entrées non acceptées dans cet état. Ces transitions contiennent un symbole de sortie spécial Ω.

Propriété 2 (Déterministe) *Un automate paramétré est déterministe pour les entrées si pour* $t_1, t_2 \in T$ *tels que* $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$ *et* $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ *et* $t_1 \neq t_2$, *si* $q_1 = q_2 \wedge i_1 = i_2$ *alors* $p_1 \cap p_2 = \phi$.

En d'autres termes, un automate est déterministe si pour une entrée donnée et une valeur du paramètre donnée, une seule transition lui est applicable dans un état donné.

Propriété 3 (Observable) *Un automate paramétré est observable si pour* $t_1, t_2 \in T$ *tels que* $t_1 = (q_1, q'_1, i_1, o_1, p_1, f_1)$ *et* $t_2 = (q_2, q'_2, i_2, o_2, p_2, f_2)$ *et* $t_1 \neq t_2$, *alors si* $q_1 = q_2 \wedge i_1 = i_2$ *alors* $o_1 \neq o_2$.

Dans un automate observable, deux transitions différentes partant du même état pour le même symbole d'entrée conduisent à deux symboles de sorties différents, ce qui permet de les différencier et de savoir que l'automate a pris un chemin différent. On peut aussi parler de non-déterminisme (au sens où il y a deux transitions) observable.

Quand M est dans l'état $q \in Q$ et reçoit l'entrée $i \in I$ portant la valeur de paramètre $x \in D_I$, alors l'état d'arrivée q', le symbole de sortie o et la valeur associée par f sont déterminés par les fonctions δ, λ and σ respectivement, définies comme suite :

- $\delta : Q \times I \times D_I \longrightarrow Q$ est la fonction d'état d'arrivée ; i.e. $\delta(q, i, x) = q'$ t.q. $(q, q', i, o, p, f) \in T$ et $x \in p$
- $\lambda : Q \times I \times D_I \longrightarrow O$ est la fonction de sortie
- $\sigma : Q \times I \longrightarrow {D_O}^{D_I}$ est la fonction définissant le paramètre de sortie.

Pour une séquence d'entrée $\gamma = i_1, ..., i_k$ et une séquence de valeurs des paramètres $\alpha = x_1, ..., x_k$, où chaque $i_j \in I, x_j \in D_I, 1 \leq j \leq k$, nous définissons l'association de γ à α comme $\gamma \otimes \alpha = i_1(x_1), ..., i_k(x_k)$, où chaque x_j est associé

à i_j. On procède de même pour associer les valeurs des paramètres de sortie aux symboles de sortie.

Notre modèle présente les extensions suivantes par rapport aux modèles d'automates habituellement pris en compte dans l'inférence de machines.

- Des entrées et sorties paramétrées.
- Des domaines arbitraires (non nécessairement finis) pour les paramètres.
- Des gardes p sur les paramètres d'entrée ; ces gardes sont associées aux transitions.
- Des fonctions arbitraires f pour le calcul des valeurs des paramètres en sortie.
- Ces fonctions peuvent être partielles.

Il y a cependant quelques restrictions dans ce modèle si on le compare aux modèles d'automates étendus du genre EFSM, tels qu'on les trouve dans des formalismes comme Estelle, SDL ou les Statecharts.

- Un paramètre unique pour les entrées et les sorties, et un domaine unique commun à tous les symboles.
- Pas de variables (dite parfois variables d'états). Toute la mémorisation doit être codée dans les états de Q.

Le premier point n'est pas une restriction sévère, c'est une commodité de notation. En effet, comme nous permettons des domaines arbitraires, il suffit d'introduire un codage entre les produits des domaines typés de paramètres des interactions réelles avec le composant et un domaine unique de représentation.

La seconde restriction est plus gênante, mais vient de l'impossibilité d'observer la structure interne de la boîte noire. Si un automate avait à la fois un état et une ou des variables internes, un même comportement pourrait être décrit par un automate à un seul état dans lequel toute la structure de contrôle serait codée dans une manipulation des variables, ou qui pourrait avoir un nombre arbitraire d'états. Il n'y aurait donc pas unicité de la solution calculée, ce qui remettrait en cause toute la démarche algorithmique.

Cette seconde restriction est donc sérieuse, puisqu'elle ne permet d'inférer le modèle que pour des composants ayant des variables dans des domaines finis, et de faible taille, car elle conduit à énumérer les états. Cependant, grâce au modèle paramétré que nous proposons, une forte réduction de la taille des automates inférés est déjà acquise par rapport aux algorithmes existants qui n'infèrent que des automates accepteurs déterministes, puisque nous pouvons factoriser un grand nombre de transitions. Pour aller plus loin et factoriser des états, nous pensons qu'on pourrait recourir à des heuristiques.

Comme nous n'infèrerons qu'une approximation des composants, nous ne prendrons en compte qu'un nombre fini d'états. Nous devrons supposer par ailleurs que les composants étudiés ont un modèle PFSM.

Dans un travail appuyé sur le code source (à des fins de compréhension des programmes), [16] proposent une technique permettant de reconstituer des structures d'automates enrichis avec une mémoire structurée. Un modèle et un algorithme d'inférence d'automates proposant certaines formes de paramètres a été proposé par [2]. Mais ce modèle ne prend en compte que des paramètres

booléens (et donc des domaines finis), et ne comporte pas de sorties (c'est une extension des automates accepteurs traités par Angluin).

5 Apprentissage de systèmes paramétrés

Cette partie explique comment on peut reconstituer un modèle d'automate paramétré de type PFSM présenté en 4. On présente les principes de l'algorithme, sans rentrer dans les détails. Comme le modèle PFSM est assez général pour représenter un composant réactif ayant un nombre fini d'états, on considère qu'on cherche à apprendre (ou identifier) un automate de référence inconnu $M = (Q, I, O, D_I, D_O, T, q_0)$, dont on connaît l'alphabet d'entrée I et le domaine du paramètre d'entrée D_I. Puisque nous pouvons soumettre n'importe quelle séquence d'entrées paramétrées au composant et observer les sorties paramétrées correspondantes, alors pour toute séquence d'entrée $\gamma \otimes \alpha(\gamma \in I^*, \alpha \in D_I^*, |\gamma| = |\alpha|)$, $\lambda(q_0, \gamma, \alpha)$ peut être trouvé par le test. On suppose là encore que tout composant peut être réinitialisé avant chaque test.

5.1 Table d'observation étendue

Dans l'algorithme présenté en 3 pour les automates de Mealy, on a utilisé la structure de table d'observation pour enregistrer les sorties du composant. Pour les PFSM, il faut enregistrer non seulement les symboles de sortie, mais aussi les valeurs des paramètres. Nous notons cette table (S, E, R, TO).

S est un ensemble fini non-vide de *séquences d'accès*, qui permettront effectivement d'accéder aux différents états du modèle. Dans un automate paramétré, l'état d'arrivée n'est pas déterminé seulement par la séquence de symboles d'entrée, mais aussi par la séquence des paramètres d'entrée. Nous appellerons l'association de ces deux séquences une *séquence d'entrée composée* ; ce sont ces séquences composées qui constitueront nos séquences d'accès.

E est un ensemble fini non-vide et clos par suffixe de séquences de symboles d'entrée. Ce sont les *séquences de séparation*, car elles servent à discriminer les états de la conjecture.

R est un sur-ensemble de S. Chaque fois qu'on ajoute une séquence d'accès à S, on obtient un groupe de séquences d'entrées composées en étendant la séquence d'entrée avec tous les $i \in I$ et en choisissant des $x \in D_I$ étendant l'ensemble des valeurs des paramètres des séquences d'entrée ; les séquences d'entrées composées sont ajoutées à R.

La fonction TO est définie sur $R \times E$. Dans le cas des PFSM, partant d'une même séquence d'entrée, on peut observer différentes séquences de sortie selon les valeurs qu'on donne aux paramètres d'entrée. Pour $r \in R$ et $e \in E$, la case $TO(r, e)$ contient l'association entre les paramètres des séquences d'entrée et ceux des séquences de sortie. La table 2 est un exemple d'une telle table d'observation, où la première colonne contient les séquences de R, avec S dans la première partie de la colonne (séparée par la barre horizontale).

5.2 Propriétés des tables d'observation

Dans le cas des automates simples de Mealy, on pouvait construire une conjecture dès que la table d'observation était *close* et *cohérente* . Pour les PFSM, la table doit avoir des propriétés supplémentaires pour construire une machine paramétrée en accord avec les observations des valeurs. Ces propriétés doivent permettre de comparer les lignes des tables, afin que la conjecture puisse être bien définie et minimale.

Les séquences de S représentent des états potentiels. Pour $r_1, r_2 \in R$, si l'on obtient, en partant des états atteints par r_1 et r_2, des séquences de sortie paramétrées différentes lorsqu'on a fourni en entrée les mêmes séquences composées, alors les lignes r_1 et r_2 sont *dissemblables*. Dans ce cas, on sait que r_1 et r_2 correspondent à des états différents.

Si deux lignes ne sont pas dissemblables, il faut, pour pouvoir les comparer, avoir exécuté les mêmes groupes de séquences composées à partir des états correspondants. Une table d'observation dont toutes les paires de lignes satisfont cette propriété sera dite *équilibrée*.

Pour tout $s \in S$ et $e \in E$, si $TO(s \cdot e)$ contient plus d'un symbole de sortie, alors la ligne s sera dite *contestée*. Une ligne s peut être contestée si $s \cdot e$ a été testé avec des valeurs différentes pour les paramètres d'entrée. Dans ce cas, on ajoutera de nouvelles lignes à R, qu'on testera avec les valeurs de paramètres qui sont dans $TO(s \cdot e)$.

Lorsqu'une table est équilibrée, on peut utiliser la relation d'égalité simple "=" pour comparer les lignes. On peut alors utiliser les concepts de table *close* et *cohérente* .

Quand la table d'observation est équilibrée, *close* et *cohérente* , on peut proposer une PFSM de conjecture d'une façon analogue à celle utilisée pour les automates de Mealy, en prenant soin d'associer les correspondances entre valeurs des paramètres en entrée et en sortie.

5.3 Algorithme d'apprentissage pour PFSM

L'algorithme d'apprentissage d'un modèle PFSM est défini par les étapes suivantes.

1. Au départ $R = S = \emptyset$ et $E = I$. Les cellules de la table sont toutes remplies avec l'ensemble vide.
2. Ajouter $\epsilon \otimes \epsilon$ à S, ainsi que dans R.
3. Construire des cas de test paramétrés pour les cellules non encore remplies de la table, et exécuter ces séquences. Pour $r \in R$ et $e \in E$, le cas de test correspondant a pour préfixe r, pour suffixe e et comme séquence de paramètres d'entrée α de ${D_I}^+$ et le test $r \cdot e \otimes \alpha$. [3]
4. Enregistrer le résultat du test $r \cdot e \otimes \alpha$ dans la table. Seule la dernière partie de la séquence de sortie sera enregistrée, celle qui correspond à la longueur

[3] On choisit la séquence de valeurs α parmi les valeurs extraites des scénarios d'usage.

de e (comme dans le cas des machines de Mealy). Le résultat du cas de test $\eta \otimes \beta$, où $\eta \in O^+$ est la séquence de sortie et $\beta \in {D_O}^+$ est la séquence de paramètres de sortie correspondante, est enregistré comme $(\alpha', \eta' \otimes \beta')$, où α' est la partie finale de α, η' celle de η, β' celle de β et $|\alpha'| = |\eta'| = |\beta'|$.

5. Équilibrer la table. Lorsqu'elle ne l'est pas, c'est à dire lorsque pour certaine séquences d'entrée les comportements du système pour différentes valeurs des paramètres d'entrée ne sont pas connus, construire les cas de test correspondants, les exécuter et enregistrer les résultats dans la table.
6. Pour chaque cas de test $r \cdot e \otimes \alpha$, s'il existe $r' \in R, e' \in E$ tel que $r' \cdot e'$ constitue un préfixe de $r \cdot e$, alors le préfixe correspondant des valeurs des paramètres sera rajouté à $TO(r' \cdot e')$. Si r' devient une ligne contestée, alors on ajoute de nouvelles lignes dans R, que l'on teste avec les valeurs de paramètres enregistrées dans $TO(r' \cdot e')$.
7. Fermer la table. Chaque fois qu'elle n'est pas *close* , ajouter la séquence composée correspondante à S et revenir à l'étape 3 pour remplir les cases ajoutées.
8. Rendre la table *cohérente* . Lorsqu'elle ne l'est pas, ajouter la séquence d'entrée correspondante à E et revenir à l'étape 3 pour remplir les cases vides.
9. Lorsque la table est *équilibrée*, *close* et *cohérente* , on peut établir la conjecture M'.

5.4 Apprentissage du contrôleur de climatiseur comme système paramétré

Pour identifier le modèle PFSM du contrôleur de climatiseur, on peut utiliser les entrées indiquées sur la figure 1, sans adaptation particulière, comme dans le cas de l'apprentissage du modèle de Mealy. Ainsi, nous construisons $I = \{M, A, T\}$ avec $D_I = [-20, 50]$ comme domaine pour le paramètre d'entrée. On utilise l'algorithme présenté en 5. La table 2 est *équilibrée*, *close* et *cohérente* . La conjecture est illustrée sur la figure 4 avec les valeurs des paramètres déterminées par l'apprentissage.

Par exemple, les valeurs 4, 12, 20 et 35 correspondraient à des valeurs dans des plages dont la spécification (sous forme de scénarios d'usage) nous apprendrait qu'elles pourraient donner lieu à des comportements distincts. Noter que ces valeurs ne correspondent pas forcément aux valeurs limites de ces plages, et que sauf dans l'état 2 où les 4 valeurs donnent lieu à 4 transitions différentes, pour les autres états, seule une des (plages de) valeur introduit un comportement spécifique.

On voit bien qu'on a appris plus de détails que sur le modèle de Mealy. Grâce à la structure paramétrée de PFSM, nous sommes capables de tester des valeurs différentes pendant l'algorithme d'inférence. Nous venons de comprendre que quand la valeur de la température est 35°C, le contrôleur envoie une commande pour lancer la climatisation CM à moyenne vitesse mv. Par ailleurs, la vitesse de chauffage C est régulée quand la valeur de la température est 4°C. En revanche,

si la température est 20°C, il envoie hv pour mettre en route le ventilateur V à vitesse haute. La conjecture ne recouvre pas tout le domaine du paramètre, mais ne connaît que les valeurs qui ont été effectivement testées. En effet, on ne peut parcourir dans le test tout le domaine des paramètres. Même s'il est fini, la combinaison des valeurs demanderait trop de cas de test, on ferait du test exhaustif.

R	M	A	T
$\epsilon \otimes \epsilon$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M \otimes \perp$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T \otimes \perp - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, F \otimes \perp), (35, F \otimes \perp)$
$M - T \otimes \perp - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, V \otimes hv), (35, F \otimes \perp)$
$M - T \otimes \perp - 35$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, F \otimes \perp), (35, CM \otimes mv)$
$M - T \otimes \perp - 12$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - A \otimes \perp - 4 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - A \otimes \perp - 20 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - A \otimes \perp - 35 - \perp$	$(\perp, OK \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, \Omega \otimes \perp), (12, \Omega \otimes \perp), (20, \Omega \otimes \perp), (35, \Omega \otimes \perp)$
$M - T - T \otimes \perp - 4 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, F \otimes \perp), (35, F \otimes \perp)$
$M - T - T \otimes \perp - 4 - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 20 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 20 - 20$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, V \otimes hv), (35, F \otimes \perp)$
$M - T - T \otimes \perp - 35 - 4$	$(\perp, \Omega \otimes \perp)$	$(\perp, \Omega \otimes \perp)$	$(4, C \otimes mv), (12, F \otimes \perp), (20, V \otimes hv), (35, CM \otimes mv)$
$M - T - T \otimes \perp - 35 - 35$	$(\perp, \Omega \otimes \perp)$	$(\perp, F \otimes \perp)$	$(4, F \otimes \perp), (12, F \otimes \perp), (20, F \otimes \perp), (35, CM \otimes mv)$

Tab. 2. Une table *close* et *cohérente* pour apprendre un modèle PFSM du contrôleur de climatiseur

6 Test d'intégration

Après avoir présenté l'inférence de modèles pour les composants, nous décrivons ici comment exploiter ces algorithmes et ces modèles dans le développement d'un système par intégration de composants. On suppose que l'intégrateur logiciel se situe dans le cadre présenté en introduction, avec les hypothèses que nous y avons faites.

Dans une phase préparatoire de test "unitaire" des composants isolés, on construit pour chaque composant C un premier modèle PFSM $C^{(1)}$ selon l'algorithme décrit en 5. Ensuite, on assemble les composants d'un côté et leurs modèles de l'autre : les sorties d'un composant peuvent devenir les entrées d'un autre.

L'intégration se fait en deux étapes. Dans une première étape, on fournit au système (assemblé) les scénarios d'usage du système global (au besoin, si on ne dispose pas de tels scénarios, on peut en engendrer à partir de la définition d'interfaces). Les scénarios devront comporter normalement une instanciation des séquences d'entrées et de sortie avec des valeurs typiques des paramètres, ou des domaines de valeurs dans lesquelles on pourra choisir des valeurs pour les entrées. On fournit au système et à son modèle les séquences d'entrées paramétrées et on observe les sorties paramétrées. A ce stade, si les sorties ne sont pas celles

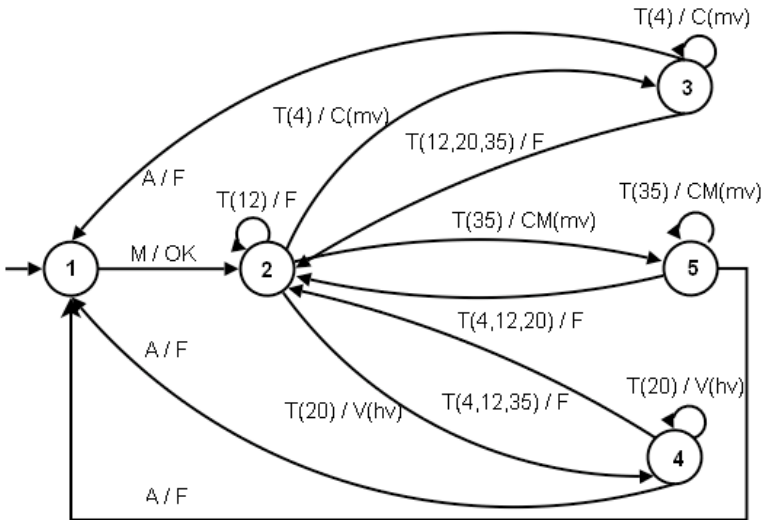


Fig. 4. Conjecture de PFSM

attendues (lorsqu'elles ont été spécifiées), une erreur est détectée, ou si elles diffèrent de celles produites par le modèle, on est en présence d'un contre-exemple qui sera réinjecté dans l'apprentissage pour enrichir les modèles des composants. Une première phase d'enrichissement incrémental des modèles des composants peut donc avoir lieu au cours de cette première étape.

Dans une deuxième étape, on procède au test d'interopérabilité, pour lequel on s'appuie sur les modèles, comme générateurs. Pour cela, on utilise des techniques de génération par exploration de modèles comme celles de [14], [3], [10] etc. Le choix des paramètres associés aux entrées externes du système sera fait en tenant compte des valeurs typiques fournies avec les scénarios d'usage, en essayant d'enrichir les cas déjà testés dans la première étape. Les sorties paramétrées (tant internes qu'externes) observées en testant le système sont confrontées à celles prévues par les modèles. Le test continue jusqu'à ce qu'une sortie soit incompatible avec le modèle (symbole de sortie ou paramètre différent de celui prévu par le modèle) ou qu'un critère de couverture défini pour l'arrêt du test soit atteint. Si une incompatibilité a été détectée, elle fournit un contre-exemple pour raffiner le modèle. L'expert procédant à l'intégration peut également être sollicité pour décider si le contre-exemple est un comportement admissible ou erroné du système intégré.

Les contre-exemples obtenus dans les deux étapes peuvent servir à raffiner les modèles. Ils sont alors injectés dans l'algorithme d'apprentissage comme séquences rajoutées à S pour les composants concernés. Les valeurs des paramètres compatibles avec les modèles sont également prises en compte à ce niveau pour enrichir les tables. On produit alors une nouvelle version du modèle du système avec les des modèles $C^{(i+1)}$ pour chaque composant C.

À la fin du processus d'intégration, on dispose donc d'un modèle paramétré pour chacun des composants qui est compatible avec tous les tests exécutés. Les interactions entre tous les composants auront été systématiquement testés selon le critère de sélection de test et de couverture que l'intégrateur se sera donné. Le processus de test aura également pu être interrompu en cas de découverte de fautes dans le système. Pour plus de détails sur l'algorithme d'intégration, voir [11]

7 Conclusion

Nous avons présenté une approche destinée à assister l'intégrateur de composants logiciels. Prenant acte de l'absence habituelle de modèles formels accompagnant les composants, nous proposons une méthode et des algorithmes permettant d'utiliser les techniques de génération de tests à partir de modèles, en s'appuyant sur des modèles eux-mêmes reconstruits et enrichis au cours du test. Comme les modèles sont eux-mêmes déduits des observations faites au cours des tests, ils ne peuvent servir d'oracles absolus. Mais ils permettent de dériver et de tester systématiquement les interactions entre les composants.

La méthode s'inscrit bien dans une assistance à base formelle au développement logiciel. L'ingénieur dispose d'outils automatiques pour construire les modèles, dériver et exécuter les tests. Il reste maître de l'architecture du système, de la définition des scénarios d'usage, et de l'analyse des fautes et des divergences entre le modèle et le système réel.

Comme les difficultés d'intégration entre composants provenant de sources différentes sont souvent liées à des discordances dans le traitement des valeurs échangées entre les composants, nous accordons un intérêt particulier à la construction de modèles paramétrés. Pour cela, nous avons développé de nouveaux algorithmes d'inférence de machines pour des modèles de ce type.

Nous poursuivons ce travail dans plusieurs directions. D'abord, bien qu'il soit plus expressif que les automates finis étudiés jusqu'ici en inférence de machines (à l'exception de [2]), notre modèle reste limité par l'absence de variables internes, ce qui ne permet de reconnaître que des machines ayant un nombre fini d'états. On pourrait envisager des modèles plus proches d'EFSM, éventuellement en supposant que l'intégrateur est capable de fournir des informations sur la structure de la machine. Le problème d'inférence de telles machines sera cependant difficile. Nous avons développé un outil, RALT, qui met en oeuvre les algorithmes d'inférence pour les automates accepteurs, les machines de Mealy et les automates paramétrés. Nous devrions l'appliquer prochainement à des cas d'étude issus des services de télécommunication développés par France Télécom. Au-delà du travail sur les modèles et l'approche de test incrémental, nous sommes également intéressés à quantifier, ou du moins à qualifier plus précisément, le niveau de confiance à accorder à l'assemblage à l'issue (et au cours) du processus d'intégration. Pour cela, nous travaillons sur la définition d'une notion d'approximation compatible avec notre approche. Enfin, la démarche de test est

paramétrée par les critères de couverture, et nous souhaitons développer la mise en relation formelle de ces critères avec l'approximation réalisée par les modèles.

Références

1. Dana Angluin. Learning regular sets from queries and counterexamples. *Information and Computation*, 2 :87–106, 1987.
2. Therese Berg, Bengt Jonsson, and Harald Raffelt. Regular inference for state machines with parameters. In *FASE*, volume 3922 of *Lecture Notes in Computer Science*, pages 107–121. Springer, 2006.
3. C. Besse, A. Cavalli, M. Kim, and F. Zaidi. Two methods for interoperability tests generation : An application to the tcp/ip protocol. In *Proceedings of TestCom 2002*, Berlin, 2002.
4. Edith Elkind, Blaise Genest, Doron Peled, and Hongyang Qu. Grey-box checking. In *FORTE'06*, 2006.
5. Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Tools and Algorithms for Construction and Analysis of Systems*, 2002.
6. Andreas Hagerer, Hardi Hungar, Oliver Niese, and Bernhard Steffen. Model generation by moderated regular extrapolation. In *Fundamental Approaches to Software Engineering*, pages 80–95, 2002.
7. Hardi Hungar, Oliver Niese, and Bernhard Steffen. Domain-specific optimization in automata learning. In *CAV*, volume 2725 of *Lecture Notes in Computer Science*, pages 315–327. Springer, 2003.
8. O. Koné and R. Castanet. Test Generation for Interworking Systems. *Computer Communications*, 23(7) :642–652, 2000.
9. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of components guided by incremental state machine learning. In *TAIC PART*, pages 59–70. IEEE Computer Society, 2006.
10. Keqin Li, Roland Groz, and Muzammil Shahbaz. Integration testing of distributed components based on learning parameterized i/o models. In *FORTE*, volume 4229 of *Lecture Notes in Computer Science*, pages 436–450. Springer, 2006.
11. Erkki Makinen and Tarja Systa. MAS - an interactive synthesizer to support behavioral modelling in UML. In *ICSE '01 : Proceedings of the 23rd International Conference on Software Engineering*, pages 15–24, Washington, DC, USA, 2001. IEEE Computer Society.
12. Clémentine Nebut, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. Automatic test generation : A use case-driven approach. *IEEE Trans. Softw. Eng.*, 32(3) :140, 2006.
13. D. Peled, M. Y. Vardi, and M. Yannakakis. Black box checking. In *Proceedings of FORTE'99*, Beijing, China, 1999.
14. A. Petrenko and N. Yevtushenko. Solving asynchronous equations. In *FORTE'98*, France, 1998.
15. Stephane S. Somé. Beyond scenarios : generating state models from use cases. In *Proceedings of SCESM*, 2002.
16. Neil Walkinshaw, Kirill Bogdanov, and Mike Holcombe. Identifying state transitions and their functions in source code. In *TAIC PART*, pages 49–58. IEEE Computer Society, 2006.

Développement formel par composants : assemblage et vérification à l'aide de B

Arnaud Lanoix[1,2], Samuel Colin[1] et Jeanine Souquières[1]

[1] LORIA – Nancy Université
Campus Scientifique, BP 239
F-54506 Vandœuvre lès Nancy cedex
{Samuel.Colin,Jeanine.Souquieres}@loria.fr
[2] LINA – Université de Nantes
2, rue de la Houssinière, BP 92208
44322 Nantes Cedex 03
Arnaud.Lanoix@univ-nantes.fr

Résumé Dans une approche composants pour le développement de logiciels, les composants sont considérés comme des boîtes noires. Une application consiste en un assemblage de composants qui communiquent via leurs interfaces. Une description formelle de ces interfaces est nécessaire pour s'assurer de leur compatibilité. En général, les interfaces ne sont pas directement compatibles et un adaptateur doit être introduit. Nous proposons des schémas pour assembler des composants de manière systématique et vérifier leur interopérabilité ; ces schémas sont définis à l'aide de concepts issus de la méthode B. L'assemblage est un raffinement des interfaces requises qui inclut les interfaces fournies ; la correction du processus est validée par les obligations de preuves usuelles.

Mots-clés : composant, adaptateur, assemblage, interface, vérification, construction sûre, raffinement

1 Introduction

L'approche conception de systèmes par assemblage de composants est une approche de développement intéressante et de plus en plus adoptée [1]. Une application à composants consiste en une composition de composants : des composants logiciels "boîte noire" développés par ailleurs sont assemblés les uns avec les autres pour produire le système complet. Le processus d'assemblage sous-jacent est similaire à celui des méthodes de construction et de réutilisation développées dans d'autres disciplines comme le génie mécanique ou le génie électrique.

Les composants sont assemblés via leurs interfaces. Une interface *fournie* par un composant peut être connectée avec une interface *requise* d'un autre composant si la première offre toutes les fonctionnalités permettant d'implanter la seconde : les composants doivent être connectés de manière appropriée. Afin de garantir cette *interopérabilité* entre composants, nous considérons chaque connexion entre interfaces fournie et requise de l'architecture et montrons que les interfaces sont compatibles. Une description appropriée des interfaces est primordiale si l'on veut vérifier que l'assemblage est correct.

Il est bien connu que la correction d'une connexion peut s'exprimer en termes de raffinement : l'interface fournie doit raffiner l'interface requise. La spécification formelle des interfaces et la preuve de leur interopérabilité en utilisant la méthode formelle B a été étudiée dans [2, 3]. Grâce à B, nous prouvons que le modèle de l'interface fournie est un *raffinement* correct de l'interface requise ; en d'autres termes, nous prouvons que l'interface fournie correspond à une implantation correcte de l'interface requise et par conséquent, que les composants peuvent être connectés.

Dans une approche de réutilisation, ceci est insuffisant : les composants existants ont rarement des interfaces fournies qui raffinent directement l'interface requise du composant auquel on veut le connecter. Il faut bien souvent intercaler un adaptateur (ou médiateur) entre les deux composants pour les rendre compatibles ou bien encore développer un nouveau composant par assemblage de plusieurs composants existants pour répondre au besoin. Une première étude de la construction des adaptateurs et de leur preuve est décrite dans [4]. Dans cet article, nous proposons une approche systématique de développement formel par composants basée sur des schémas d'assemblages UML et B.

L'article est structuré de la manière suivante. La section 2 présente notre approche et l'utilisation de la méthode B. La section 3 présente différents types d'assemblage et pour chacun d'eux, un schéma d'architecture et son squelette en B permettant d'exprimer et de vérifier la correction de l'assemblage. Le problème particulier de la mise en correspondance de modèles de données est abordé dans la section 4. La section 5 illustre notre démarche avec le développement d'une étude de cas du contrôle d'accès à un bâtiment. Des travaux connexes sont discutés dans la section 6 et une conclusion avec des perspectives d'évolution termine cet article.

2 Description de l'approche et utilisation de B

Dans notre approche composants, l'architecture du système est modélisée à l'aide de différents diagrammes UML 2.0 [5] :

- les diagrammes de structure composite pour exprimer l'architecture globale du système en termes des composants et des interfaces à connecter ;
- les diagrammes de classes pour exprimer les modèles de données et les signatures des méthodes des interfaces ;
- les diagrammes de séquences pour exprimer les interactions possibles entre composants et décrire des protocoles d'usage complexe.

Le comportement autorisé ou attendu des interfaces est décrit à l'aide de modèles B.

2.1 B et son utilisation dans notre approche

B [6] est une méthode formelle basée sur la théorie des ensembles, permettant un développement incrémental grâce au raffinement. Un développement

commence avec la définition d'une spécification abstraite qui est ensuite raffinée pas à pas jusqu'à l'obtention d'une implantation. Un modèle B est composé de VARIABLES, d'un INVARIANT qui décrit les propriétés de ces variables et d'OPERATIONS qui définissent les possibles évolutions de ces variables. Un modèle B est correct s'il a été vérifié que les opérations préservent l'invariant. Un exemple de modèle B se trouve figure 12.

La méthode B a été appliquée avec succès dans le développement d'applications réelles complexes, comme le projet METEOR [7] ou le métro Val [8]. Elle s'appuie sur des outils robustes. Des obligations de preuves pour la consistance des invariants et la préservation du raffinement sont générées automatiquement par les outils [9, 10].

Dans notre approche, les interfaces des composants sont annotées de modèles B pour exprimer le comportement implanté (dans le cas d'une interface fournie) ou attendu (dans le cas d'une interface requise) par l'interface. Deux notions clés de la méthode B sont utilisées :

- le raffinement qui permet un développement incrémental avec préservation de la correction à chaque étape du développement,
- les mécanismes de composition, permettant un développement modulaire et la vérification de la correction lors de l'appel d'opérations.

Note 1. Dans un processus de développement intégré, les modèles B pourraient être obtenus en appliquant des règles systématiques de transformation de UML vers B [11, 12].
De plus, l'état actuel du développement de la plate-forme Rodin [13] pour le B événementiel ne propose pas encore de mécanismes de (dé)composition, ce qui nous empêche d'évoluer vers cette nouvelle plate-forme.

2.2 Interopérabilité entre composants

Deux composants peuvent être connectés via leurs interfaces respectives si ces interfaces sont *compatibles*. On dit qu'ils sont *interopérables*. En d'autres termes, l'interface fournie implante les fonctionnalités nécessaires à l'interface requise. Nous exprimons la correction de la connexion en termes de raffinement [2].
Définition. Soient deux composants OTS1 [3] et OTS2 tels que OTS1 *requiert* une interface RI_ots1 et OTS2 *fournit* une interface PI_ots2. Ils sont interopérables si et seulement si le modèle B de PI_ots2 est un *raffinement* de celui de RI_ots1.

Remark 1. Notons que l'interface fournie PI_ots2 peut offrir plus de fonctionnalités que n'en nécessite l'interface requise RI_ots1.

Lors d'un développement par composants, les problèmes suivants doivent être étudiés : vérifier que deux composants sont *interopérables*, sinon développer un adaptateur ; développer un nouveau composant à partir de composants existants.

Développer un adaptateur qui assure la bonne connexion entre un composant OTS1 et un composant OTS2, revient à développer un nouveau composant DEV qui fournira l'interface RI_ots1 en utilisant le composant OTS2 via son interface fournie PI_ots2, comme illustré figure 2.

3. Nous nommons les composants existants OTS pour "Off-The-Shelf".

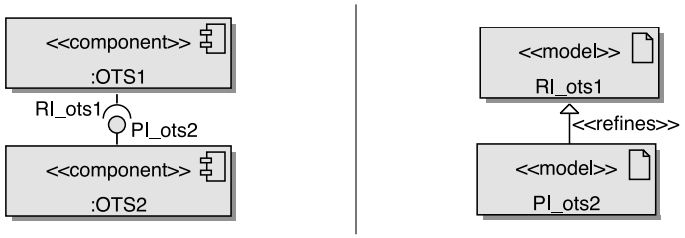


Figure 1. Interopérabilité entre OTS1 et OTS2

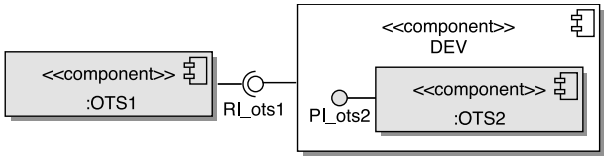


Figure 2. Adaptation vs. développement d'un nouveau composant

3 Différents assemblages de composants

Dans une approche de réutilisation, les composants existants ont rarement des interfaces directement compatibles. Un "nouveau" composant est nécessaire pour les rendre compatibles. Le développement de ce composant peut être plus ou moins complexe, en fonction du nombre et du type de composants existants à assembler. Nous étudions différents cas d'architecture et proposons des schémas pour assembler un ou plusieurs composants et vérifier la correction de l'assemblage.

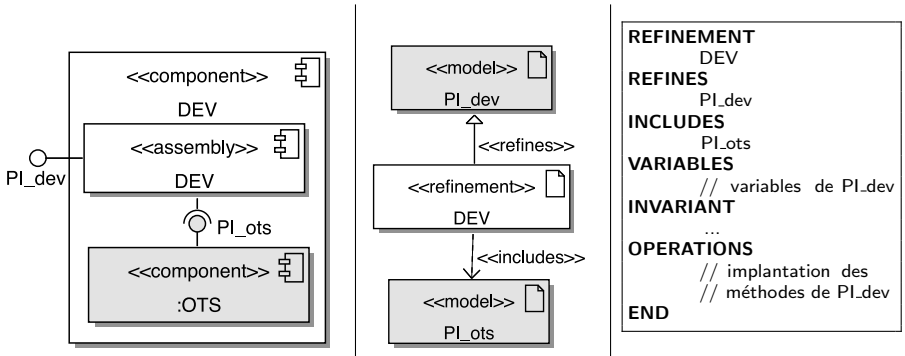
3.1 Cas de base : une seule interface dans l'assemblage

Examinons la construction d'un composant DEV qui doit implanter une interface PI_dev. Il utilisera un composant existant OTS via son interface fournie PI_ots. Cette construction correspond à un *assemblage*, qui exprime comment les attributs et les méthodes de l'interface PI_dev sont implantés à l'aide ceux de PI_ots.

1. chaque attribut de PI_dev est exprimé en termes des attributs de PI_ots ;
2. chaque méthode de PI_dev est exprimée par une combinaison d'appels aux méthodes pertinentes de PI_ots.

DEV est défini à l'aide d'un raffinement B, permettant de prouver la correction de l'assemblage. Nous proposons figure 3 un schéma d'assemblage illustrant l'architecture UML, les relations entre modèles B, ainsi qu'un squelette pour le modèle B de DEV. Les clauses VARIABLES, INVARIANT et OPERATIONS de ce squelette sont à compléter en accord avec les règles *1)* et *2)* énoncées ci-dessus.

La vérification des obligations de preuve assure que le modèle B de DEV



(a) Architecture UML (b) Architecture B (c) Squelette B de DEV

Figure 3. Cas de base

– *raffine* le modèle B associé à l'interface PI_dev (preuve de raffinement)
– en *incluant* correctement le modèle B associé à l'interface PI_ots (preuve d'inclusion)

c.à.d. que l'assemblage est correct au sens ou il réalise les besoins exprimés par PI_dev.

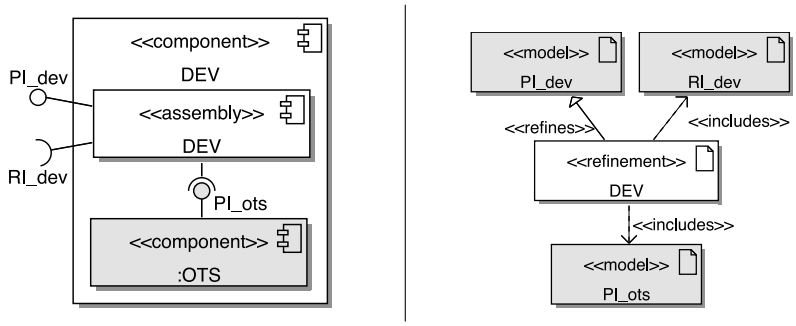


Figure 4. Cas de base étendu

Il est possible que DEV requière une interface RI_dev. Cette interface exprime des besoins qu'il restera à implanter pour utiliser DEV. Néanmoins, DEV peut utiliser les attributs et les méthodes fournis par RI_dev (au même titre que ceux fourni par PI_ots) pour implanter PI_dev. Dans ce cas, le schéma d'assemblage devient celui proposé figure 4.

3.2 Cas de deux interfaces dans l'assemblage

Considérons maintenant le cas où le composant à utiliser requiert aussi une interface RI_ots pour implanter correctement les fonctionnalités fournies par PI_ots.

Dans ce cas, DEV devra implanter l'interface RI_ots (pour répondre à OTS) en plus de l'interface PI_dev.

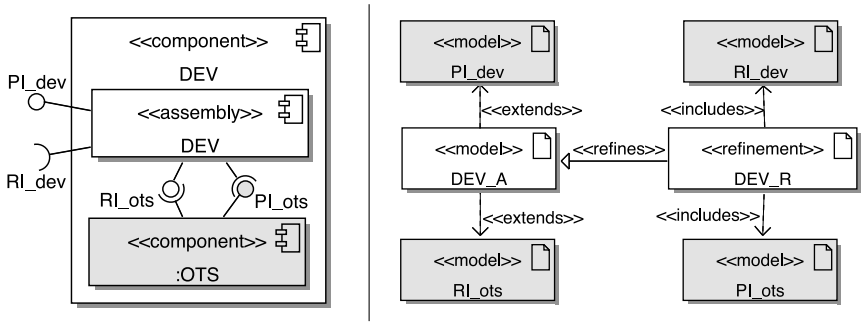


Figure 5. Deux interfaces dans l'assemblage

La construction de DEV nécessite deux étapes de développement comme illustré figure 5 :

- l'introduction d'un modèle abstrait DEV_A qui *étend* les interfaces à implanter PI_dev et RI_ots pour les regrouper dans un seul modèle B [4] ;
- la définition d'un modèle DEV_R qui exprime l'assemblage. Il raffine DEV_A en incluant les modèles B des interfaces à utiliser, ici RI_dev et PI_ots, afin d'assurer que toutes les fonctionnalités à implanter le sont de manière correcte.

3.3 Cas général : assemblage de plusieurs composants

Nous généralisons notre démarche à l'assemblage de plusieurs composants qui fournissent et/ou requièrent des interfaces particulières afin de construire un nouveau composant. DEV doit réaliser l'ensemble des interfaces requises des composants à l'aide de leurs interfaces fournies. La figure 6 donne le schéma d'assemblage dans le cas de deux composants OTS1 et OTS2.

Comme plusieurs composants sont en jeu, le protocole d'appels aux méthodes des interfaces fournies peut être complexe. Il peut être utile de commencer par l'exprimer à l'aide de diagrammes de séquences UML 2.0. À chaque méthode de DEV, on associe l'enchaînement des appels de méthodes nécessaires des différentes interfaces permettant de la réaliser. L'écriture en B des différents diagrammes de séquences est facilitée et pourrait s'automatiser. La preuve de raffinement du modèle B permet de s'assurer que les appels aux méthodes fournies sont valides (preuves d'inclusion) et le raffinement en lui-même assure que l'implantation est correcte.

4. Ce modèle intermédiaire est nécessaire parce que le B classique n'autorise le raffinement que d'un seul modèle à la fois.

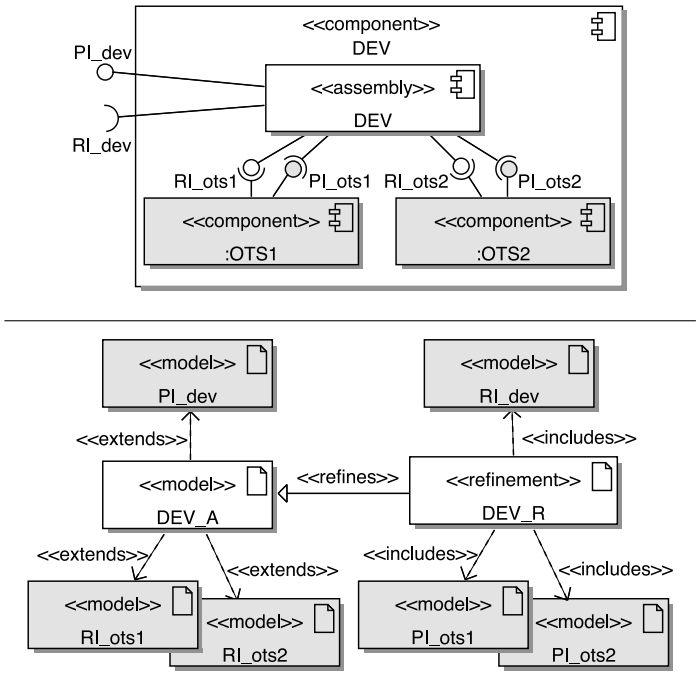


Figure 6. Assemblage de plusieurs composants

Remark 2. B propose un mécanisme de renommage associé à la clause INCLUDES qui permet d'utiliser, dans un assemblage de composants, plusieurs instances d'un même composant via des interfaces fournies identiques.

3.4 Raffinement versus implantation en B

Il est à noter que le REFINEMENT pourrait être une IMPLANTATION. L'avantage principal est de conclure définitivement l'adaptation de manière claire, en proposant un modèle réellement *implanté* en termes des composants utilisés. Les inconvénients sont en revanche plus nombreux :

- L'adaptation est définitive, ce qui signifie qu'il ne sera plus possible de préciser ou d'optimiser l'utilisation qui est faite des composants fournis. En ce sens, se limiter au REFINEMENT B est un avantage qui laisse ouvertes des optimisations futures.
- Il faut prendre en compte les contraintes de construction de B : les variables utilisées dans l'implantation doivent être déclarées comme concrètes, les modèles utilisés doivent être *importés* plutôt qu'inclus.
- Les constantes (fonctionnelles) introduites tout au long du raffinement doivent être valuées, i.e. il est nécessaire de leur donner une valeur concrète. Lorsque ces constantes se basent sur des ensembles abstraits, donc dont les éléments ne sont pas connus, la valuation n'est possible que si ces ensembles sont isomorphes à des sous-ensembles des entiers naturels. Cela signifie donc que les ensembles abstraits doivent pouvoir être remplacés par des sous-ensembles des entiers naturels.
  Cela imposerait d'utiliser les hypothèses de bonne formation des ensembles abstraits pour construire les valeurs concrètes de ces constantes fonctionnelles. Cela est possible en B événementiel du fait de l'obligation d'explicitation des propriétés des ensembles porteurs introduits ; cela n'est pas possible en B classique car celles-ci sont implicites.
- Dans le cas où les éléments des ensembles de base sont énumérés (et donc explicités), les constantes fonctionnelles peuvent être décrites. En revanche, les outils de preuve utilisés ont montré un changement de comportement, en ce sens qu'ils tentaient d'utiliser les valeurs concrètes de ces ensembles et relations plutôt que des tactiques générales de raisonnement. Il en a résulté une preuve des raffinements plus difficile que dans le cas d'ensembles de bases abstraits.

En conclusion, ces inconvénients, qui pour la plupart se situent au niveau pratique plutôt que théorique, ont fait que nous nous sommes limités au raffinement.

4 Mise en correspondance des modèles de données

Il n'est pas toujours facile de réaliser chaque attribut à implanter en termes des attributs fournis, en particulier lorsque les modèles de données des interfaces

PI_dev et PI_ots sont différents (figure 3(a)). Pour exprimer et vérifier cette correspondance, nous procédons par étapes successives, en utilisant le mécanisme de raffinement de B. DEV est développé par une série de raffinements successifs. Il est initialisé, au niveau le plus abstrait, par le modèle B de l'interface requise et se termine avec l'inclusion du modèle B de l'interface fournie. Le processus d'adaptation des modèles B se décompose en trois grandes étapes de raffinement. Une première étude a été menée dans [14]. Le schéma des modèles B de DEV est détaillé Figure 7.

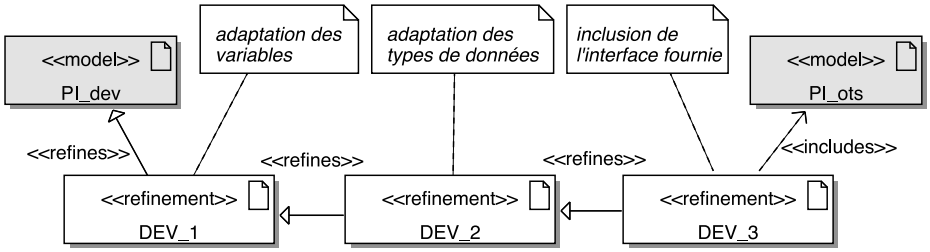


Figure 7. Étapes pour la mise en correspondance de PI_dev avec PI_ots

1) Adaptation des variables. Cette étape prépare la mise en correspondance entre les attributs de PI_dev et ceux de PI_ots :

- de nouvelles variables, qui ré-expriment les variables de PI_dev sont introduites. Elles sont choisies afin de faciliter la mise en correspondance avec celles de PI_ots ;
- le corps de chaque opération de PI_dev est transformé pour prendre en compte ces nouvelles variables.

2) Adaptation des types de données. Cette étape correspond au transtypage des données :

- les variables introduites à l'étape précédente sont toujours exprimées en termes des types de données de PI_dev. Des fonctions de transtypage sont introduites afin de convertir les types de données de PI_dev vers ceux de PI_ots, et réciproquement. De nouvelles variables sont également introduites par l'application des fonctions de transtypage sur les variables introduites à l'étape précédente ;
- le corps de chaque opération de PI_dev est transformé pour tenir compte des modifications introduites sur les variables.

3) Inclusion de l'interface fournie. Les deux étapes précédentes ont servi à préparer cette dernière étape qui consiste à inclure le modèle B de l'interface fournie :

- puisque les variables de PI_dev ont été ré-exprimées et que les types de données ont été transformés, il est maintenant facile de mettre en correspondance les variables (modifiées) de PI_dev avec celles de PI_ots ;

– chaque opération de PI_dev est exprimée en termes d'appels aux opérations de PI_ots.

Ce processus de développement en trois étapes aide à construire l'assemblage, mais aussi à réaliser la preuve de l'adaptation. La preuve complète est facilitée par la décomposition en plusieurs étapes : il est plus facile de démontrer successivement chacune des étapes de l'adaptation plutôt que de démontrer l'ensemble des preuves en une seule étape. Il faut également souligner que les trois étapes ne sont pas toujours toutes nécessaires et qu'il est quelquefois plus facile de subdiviser l'une d'entre elles en plusieurs raffinements, toujours pour faciliter la preuve.

5 Etude de cas : le système de contrôle d'accès

Nous illustrons notre propos à l'aide de l'étude de cas du contrôle d'accès à un ensemble de bâtiments [15]. L'objectif est de développer un système chargé de contrôler l'accès de personnes autorisées à un bâtiment donné d'un lieu de travail. Nous utilisons notre approche et les différents schémas d'assemblage proposés pour développer ce système à l'aide de composants préexistants.

5.1 Le cahier des charges

Le contrôle d'accès s'effectue sur la base de l'autorisation que chaque personne concernée possède. Cette autorisation doit lui permettre, sous le contrôle du système, d'entrer dans le bâtiment. Le nombre de personnes présentes dans le bâtiment doit être connu à tout instant.

Chaque personne autorisée dispose d'une carte d'accès avec un code. Des lecteurs de cartes sont installés à chaque entrée du bâtiment. À proximité de chaque lecteur se trouvent deux voyants, un rouge et un vert, chacun d'eux pouvant être allumé ou éteint. À chaque entrée et sortie du bâtiment se trouve un tourniquet normalement bloqué. Lorsqu'un tourniquet est débloqué par le système, le passage d'une personne est détecté par un capteur. Chaque tourniquet n'est affecté qu'à une seule tâche, entrer ou sortir. L'entrée obéit au protocole suivant :

– si la personne est autorisée à entrer dans le bâtiment (elle est toujours autorisée à sortir), le voyant vert s'allume et le tourniquet se débloque. Le cahier des charges originel fait état d'une contrainte de temps sur la durée de déblocage, contrainte que nous avons préféré abstraire en supposant qu'elle était gérée par le tourniquet lui-même. Dès que la personne franchit le tourniquet, le voyant vert s'éteint et le tourniquet se bloque immédiatement. Si la carte n'a pas été reprise par la personne au bout d'un certain laps de temps, elle est «avalée» par le lecteur ;
– si la personne n'est pas autorisée à entrer dans le bâtiment, le voyant rouge s'allume et le tourniquet reste bloqué. Ici encore, le retrait de la carte est soumis à une durée limite au-delà de laquelle la carte est «avalée» par le lecteur.

5.2 Une architecture composants

Dans une approche développement par composants, le système de contrôle d'accès peut être représenté par AccessControl, présenté figure 8. Les besoins auxquels ce système doit répondre sont exprimés à l'aide des interfaces suivantes :

- RI_Database permettra au contrôleur d'envoyer des requêtes à une base de données contenant les autorisations des usagers et des informations sur les personnes présentes dans les bâtiments ;
- RI_Entry permettra au contrôleur d'accès de commander le blocage/déblocage de l'entrée ; l'interface PI_Entry informera le contrôleur du passage d'une personne ;
- PI_Exit informera le contrôleur lorsqu'une personne sortira du bâtiment ;
- PI_Ident et RI_Ident devront proposer l'ensemble des fonctionnalités liées à l'identification par le contrôleur d'accès. Celui-ci commandera le système d'identification par le biais de l'interface RI_Ident et recevra des informations en retour via PI_Ident.

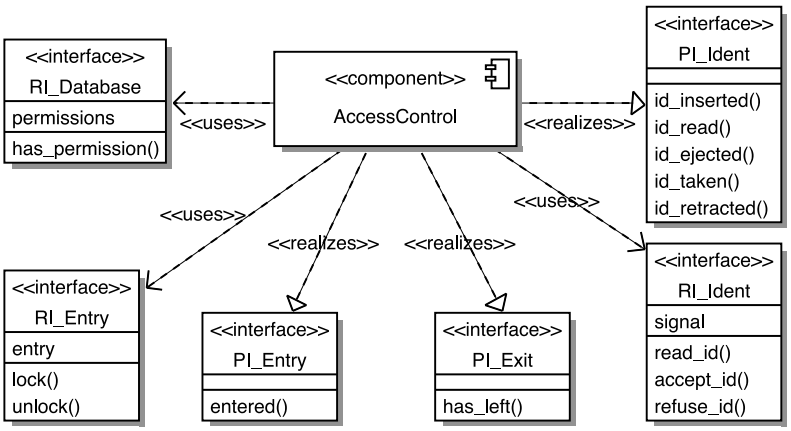


Figure 8. AccessControl et ses interfaces

Un modèle B est associé à chacune de ces interfaces afin d'exprimer de manière précise les comportements et le protocole d'usage requis par le cahier des charges. À titre d'exemple, les modèles B de RI_Entry et RI_Database sont présentés figures 12 et 18.

Composants existants Pour répondre aux besoins exprimés par AccessControl, nous disposons des composants présentés figure 9 :

- DBNetwork décrit un pilote permettant de connecter une base de données via son interface PI_DBNet. Le modèle B associé est proposé figure 18 ;
- Turnstile fournit un pilote chargé de commander un tourniquet. L'interface PI_Turn fournit des méthodes pour commander l'ouverture et la fermeture

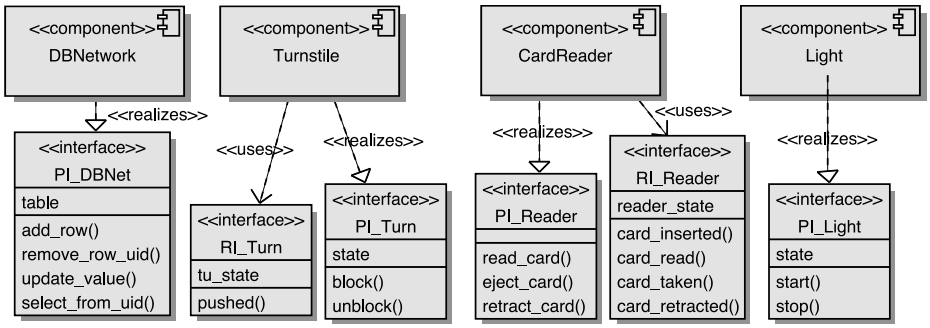


Figure 9. Les composants DBNetwork, Turnstile, CardReader et Light

du tourniquet ; son modèle B est donné figure 12 ; RI_Turn propose une méthode pour informer du passage d'une personne ;

– CardReader fournit un pilote de périphérique à un lecteur de cartes. Ses deux interfaces PI_Reader et RI_Reader correspondent à l'interfaçage entre le lecteur de cartes et son environnement ;
– Light décrit le pilote de commande d'une lampe. Son interface PI_Light permet d'allumer et d'éteindre la lampe.

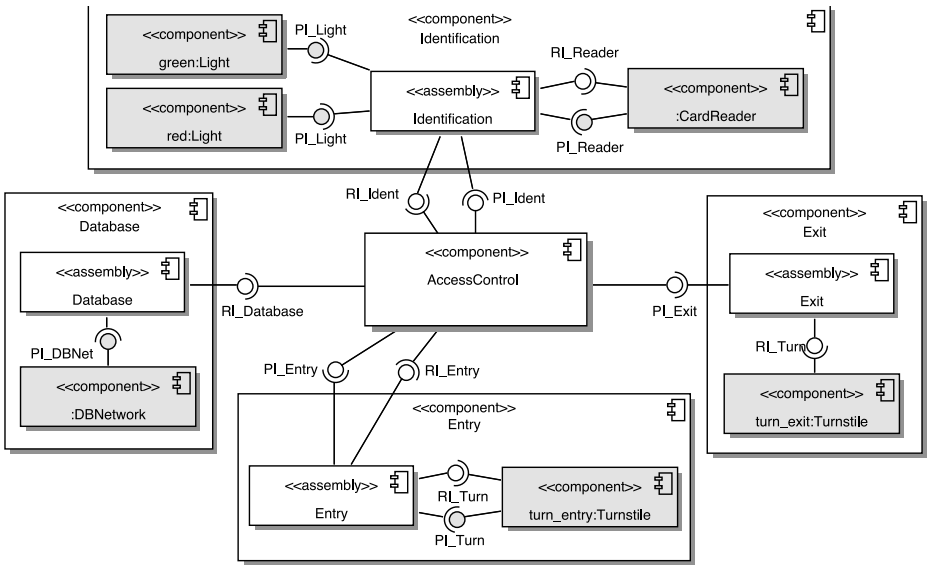


Figure 10. Architecture globale du système de contrôle d'accès

Architecture globale L'architecture globale du système est décrite Figure 10 sous la forme d'un diagramme de structure composite UML. Pour répondre aux besoins exprimés par AccessControl en utilisant les composants existants présentés dans la section 5.2, il est nécessaire de développer des composants intermédiaires, qui assembleront (adapteront) ces composants pour répondre aux besoins : Entry, Exit, Identification et Database. Deux instances de Turnstile, notées turn_entry et turn_exit – pour l'entrée et la sortie du bâtiment – ainsi que deux instances de Light, notées red et green – pour les deux voyants rouge et vert – seront nécessaires.

Dans la suite du papier, nous détaillons le développement des composants Entry, Database et Identification.

5.3 Le composant Entry

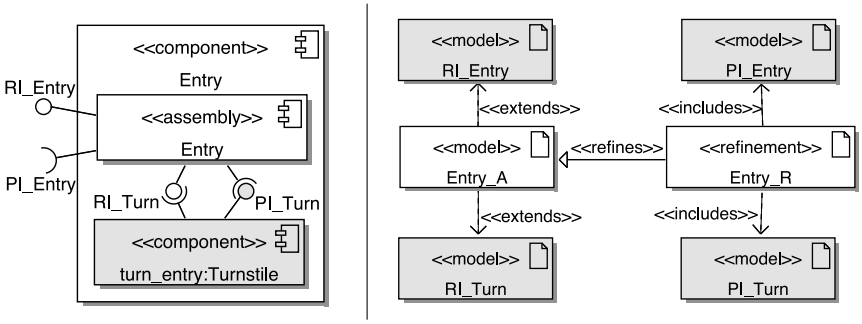


Figure 11. Composant Entry

Un adaptateur est nécessaire pour connecter AccessControl (via RI_Entry et PI_Entry) au composant Turnstile (via RI_Turn et PI_Turn). L'application du schéma correspondant au cas de deux interfaces dans l'assemblage donne l'architecture de ce composant, voir figure 11, dans laquelle :

- Entry_A regroupe les interfaces à implanter ;
- Entry_R réalise l'assemblage.

Le modèle B raffiné Entry_R est complété pour exprimer l'implantation des éléments de RI_Entry et de RI_Turn en utilisant ceux de PI_Turn et PI_Entry : ici, il s'agit principalement d'exprimer des renommages, comme l'indique la figure 12. La preuve du raffinement de RI_Entry et de RI_Turn garantit que le modèle donné pour Entry est correct dans le sens où il réalise bien ce qui est attendu.

5.4 Le composant Identification

Pour répondre aux besoins exprimés par RI_Ident et PI_Ident concernant le processus d'identification d'une personne, plusieurs composants devront être utilisés :

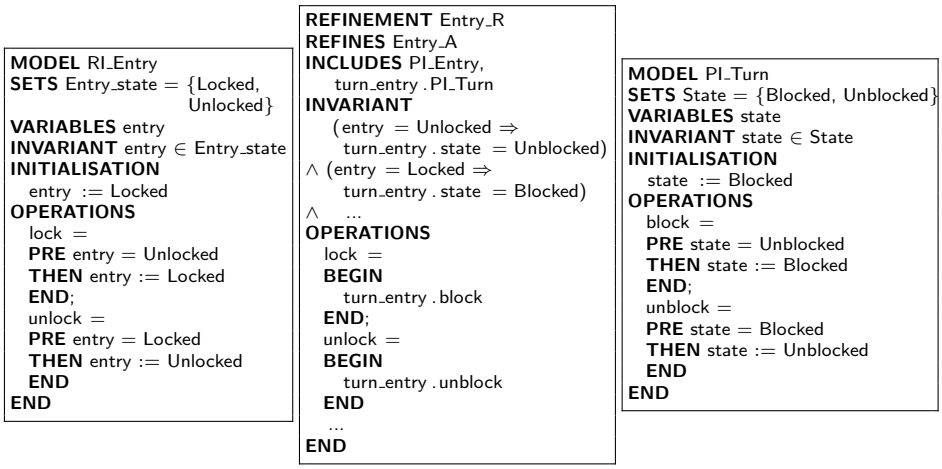


Figure 12. Modèles B de RI_Entry, Entry_R et PI_Turn

– CardReader via ses interfaces RI_Reader et PI_Reader et
– deux instances, green et red, du composant Light via son interface PI_Light.

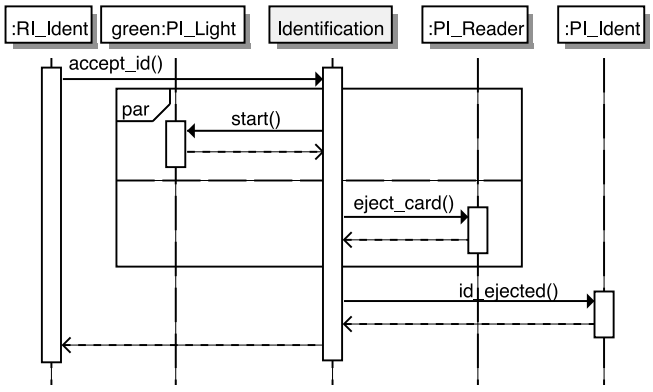


Figure 13. Diagramme de séquences pour accept_id()

Des diagrammes de séquences peuvent être utilisés pour expliciter le protocole d'usage de Identification : à chaque méthode de chaque interface requise, on associe la réaction de l'assemblage, c.à.d. les appels aux méthodes nécessaires des interfaces fournies. Par exemple, la méthode accept_id() de RI_Ident correspond à une notification d'autorisation d'accès d'une personne par le système de contrôle d'accès. Son comportement attendu est décrit à l'aide du diagramme

de séquences de la figure 13. L'adaptateur doit réagir en termes des interfaces fournies :

- en allumant la lampe verte (Green.start()) pour avertir l'usager de son autorisation d'accès tout en éjectant la carte (eject_card()), puis
- en notifiant au contrôleur d'accès l'éjection de la carte (id_ejected()).

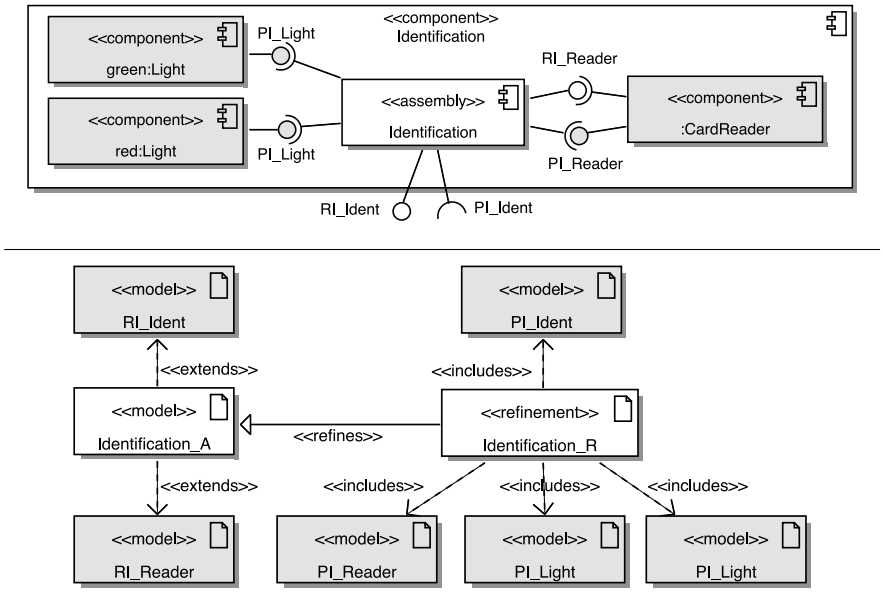


Figure 14. Composant Identification

Le composant Identification est obtenu par application du schéma d'assemblage de plusieurs composants, comme illustré figure 14. Le modèle B abstrait Identification_A regroupe les interfaces à implanter. Le modèle B de l'assemblage, Identification_R, raffine Identification_A afin d'assurer que l'assemblage est correct. Le code B présenté figure 15 complète le squelette du schéma :

- l'invariant établit un lien entre les variables à fournir reader_state et signal et les variables fournies par les interfaces PI_Ident, PI_Reader, green.PI_Light et red.PI_Light ;
- chaque méthode de RI_Reader et de RI_Ident est exprimée en termes d'appels aux méthodes correspondantes des modèles inclus. La méthode accept_id() correspond à une réécriture en B du diagramme de séquences donné figure 13. La méthode card_taken() correspond également à la réécriture du diagramme de séquences correspondant, donné figure 16.

Remark 3. Les méthodes accept_id() et card_taken() illustrent les possibilités d'une traduction "automatique" d'un diagramme de séquences UML 2.0 vers une spécification B. Les concepts manipulés dans les diagrammes de séquences

```
REFINEMENT Identification_R
REFINES Identification_A
INCLUDES
  red . PI_Light , green . PI_Light ,
  PI_Reader,
  PI_Ident
VARIABLES reader_state
INVARIANT
  (green . state = On ⇒
    red . state = Off) ∧
  (red . state = On ⇒
    green . state = Off) ∧
  ( signal = No_signal ⇒
      (green . state = Off ∧
        red . state = Off))
```

```
OPERATIONS
  accept_id =
  BEGIN
    BEGIN
      green . start || eject_card
    END ;
    id_ejected
  END;
  ...
  card_taken =
  BEGIN
    id_taken ;
    SELECT red.state = On
      THEN red.stop
    WHEN green.state = On
      THEN green.stop
    END;
    reader_state := RWaiting
  END;
  ...
END
```

Figure 15. Extraits de Identification_R

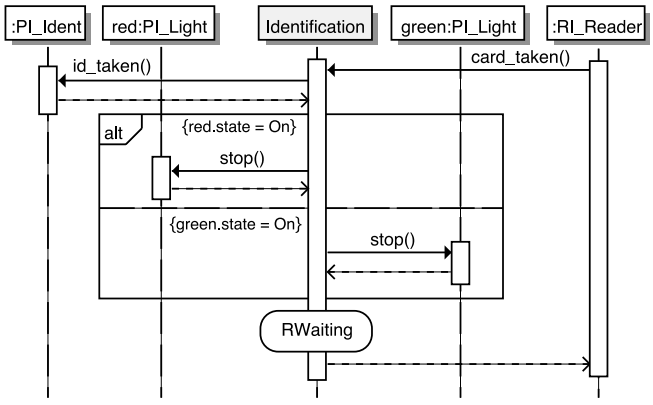


Figure 16. Diagramme de séquences pour card_taken()

(séquences, parallèles et alternatives) sont également des concepts présents dans le langage B.

5.5 Le composant Database

Les interfaces RI_Database et PI_DBNet présentent des modèles de données différents :

- RI_Database permet d'obtenir les portes (bâtiments) autorisées pour une personne donnée ;
- PI_DBNet permet de mémoriser dans une base de données des couples (Uid, Value) d'entiers naturels.

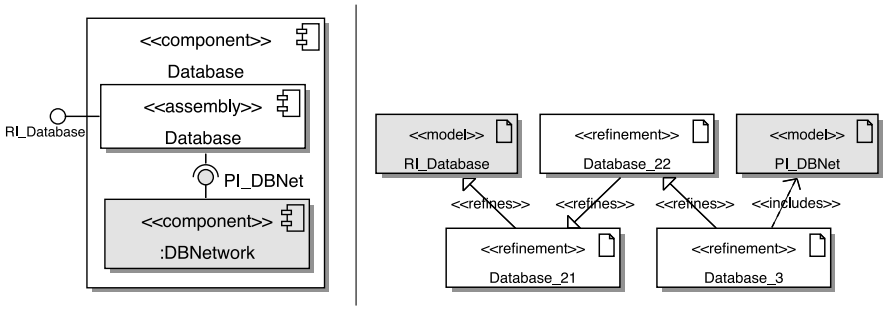


Figure 17. Composant Database

L'adaptation est effectuée en appliquant le schéma de base lorsqu'une seule interface est en jeu dans l'assemblage avec la mise en correspondance des modèles de données. Les schémas de l'architecture UML et B du composant Database sont présentés figure 17. Plusieurs étapes de raffinement sont nécessaires pour développer le modèle B associé :

(1) L'étape d'**adaptation des variables** n'est pas nécessaire puisque celles-ci peuvent être facilement mises en correspondance (voir figure 18) : les utilisateurs sont mis en correspondance avec le champ Uid de la base de données, et les portes avec le champ Value.

(2) L'**adaptation des types de données** est décomposée en deux étapes afin de faciliter la preuve :

- dans Database_21, une fonction de transtypage user_cast est introduite afin de transformer le domaine de la relation permissions en le domaine les entiers naturels ; une nouvelle variable n_permissions est également introduite ;
- il s'agit maintenant de transformer le codomaine de n_permissions en le domaine des entiers naturels. Une fonction de transtypage door_cast, ainsi qu'une nouvelle variable nn_permissions, sont introduites dans Database_22.

(3) La **dernière étape** consiste à associer les attributs Uid et Value de PI_DBNet à nn_permissions, c'est-à-dire à préciser les relations entre les structures de données. C'est également lors de cette étape que le corps des méthodes se réduit à l'appel des méthodes correspondantes du composant fourni (ici, has_permission appelle select_from_uid).

Remark 4. Les états des interfaces mises en œuvre restent disjoints, c.à.d. qu'il n'est pas possible de lier ces états, que ce soit au niveau des préconditions des méthodes ou des modifications effectuées, en utilisant les nouveaux états introduits dans le raffinement. Ce phénomène est induit par l'utilisation de composants indépendants. L'assemblage doit créer des liens qui n'existent pas : ceux-ci imposent un style de programmation défensif, traduit par l'utilisation dans notre exemple de gardes (clause SELECT) plutôt que de préconditions (style offensif).

Modèles B\OPs	évidentes	automatiques	interactives
Types	1	0	0
PI_Turn	5	0	0
RI_Turn	3	0	0
PI_Entry	3	0	0
RI_Entry	5	0	0
PI_Exit	3	0	0
Entry	**12**	**2**	**0**
Exit	**3**	**0**	**0**
PI_Light	5	0	0
PI_Reader	7	0	0
RI_Reader	9	0	0
PI_Ident	11	0	0
RI_Ident	7	0	0
Identification_A	**9**	**0**	**0**
Identification_R	**62**	**6**	**2**
PI_DBNet	12	10	4
RI_Database	3	0	0
Database_21	**6**	**2**	**2**
Database_22	**6**	**2**	**2**
Database_3	**5**	**8**	**2**
TOTAL	**177**	**30**	**12**

Table 1. Obligations de preuves

Les différents assemblages présentés ainsi que les interfaces nécessaires ont tous été validés avec B4free. Cela nous permet d'assurer que les nouveaux composants développés par assemblage de composants existants sont corrects. Le détail des obligations de preuves (OPs) est donné dans le tableau 1.

6 État de l'art

Les travaux de recherche relatifs à l'adaptation de composants sont nombreux et la nécessité de disposer de mécanismes d'assemblage performants pour les réaliser a été reconnue dès les années 1990 [16, 17, 18, 19].

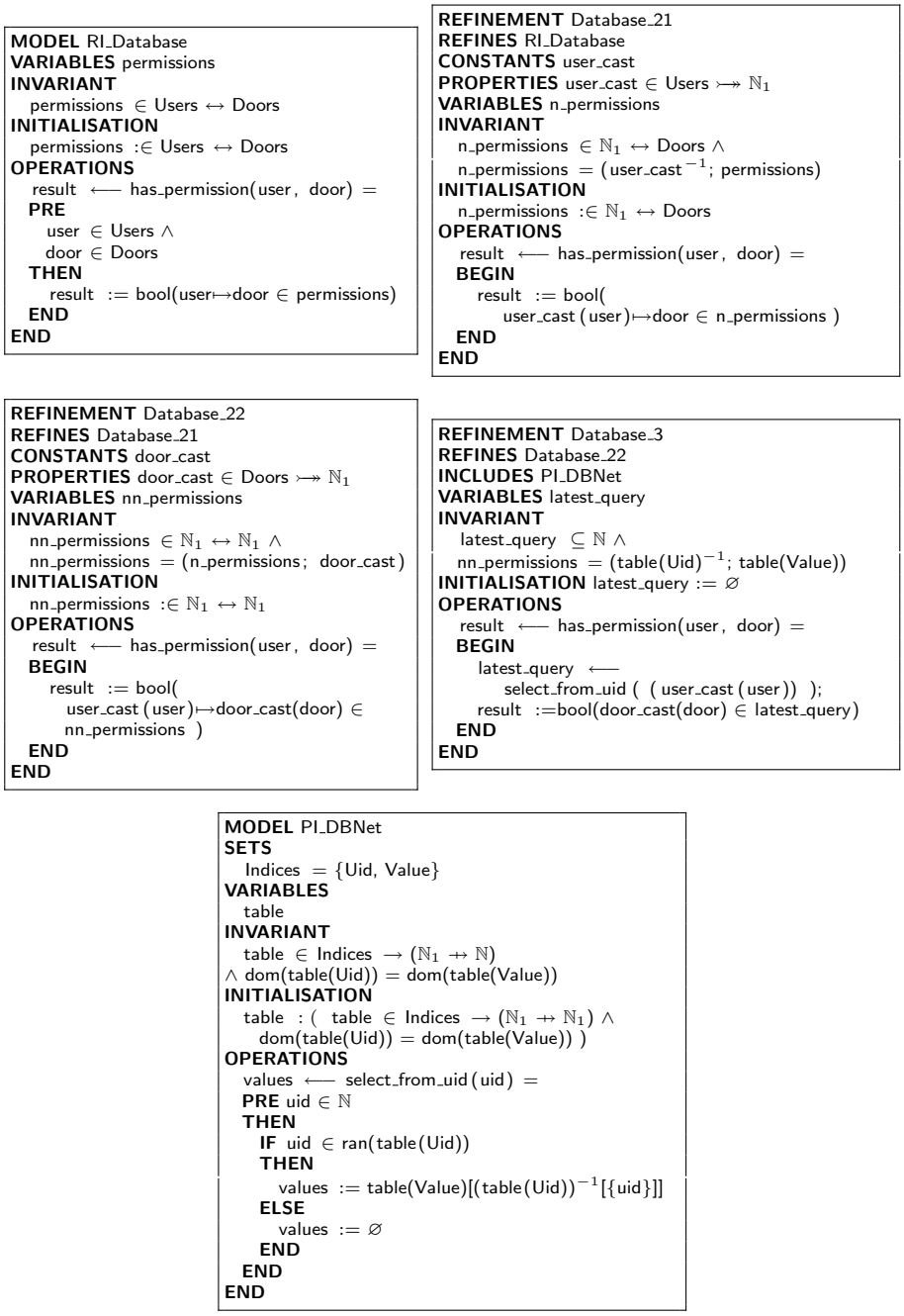


Figure 18. Mise en correspondance des modèles de données

Une des premières approches concernant la réutilisation de modules avec adaptation de leurs interfaces est celle proposée par Purtilo et Atlee [20] : ils proposent un langage dédié, Nimble, où l'adaptation entre interfaces requises et fournies est effectuée par le développeur. Notre approche est assez voisine avec l'utilisation de UML et B comme langages, reposant sur des standards et des outils de vérification.

Des approches pragmatiques ont porté sur l'analyse des problèmes sous-jacents à l'adaptation de composants existants.Une définition formelle de l'interopérabilité et de l'adaptation de composants a été introduite dans [21]. Dans ce cadre, la spécification du comportement d'un composant est décrite à l'aide de machines à états finis pour lesquelles il existe des techniques et des outils efficaces permettant la vérification de la compatibilité des protocoles.

Zaremski et Wing [22] proposent une approche intéressante pour comparer deux composants logiciels, permettant de décider si un composant peut être remplacé par un autre. Ils utilisent les spécifications algébriques pour modéliser le comportement des composants et le prouveur Larch pour prouver la correspondance entre composants.

Reussner et Schmidt considèrent une certaine classe de problèmes dans le contexte des systèmes concurrents [23, 24]. L'incompatibilité des protocoles est résolue par la génération d'adaptateurs en utilisant les interfaces décrites en termes de machines à états finis.

Les travaux présentés dans [25] proposent un processus de génération d'adaptateurs. De nombreux travaux actuels sont dédiés à l'adaptation dynamique [26], qui va plus loin que notre approche : l'adaptation des composants s'effectue lors de l'exécution en recherchant le composant adapté [27, 28]. Ces méthodes se basent sur l'hypothèse de l'existence de relations d'héritages (avec une possible transitivité) entre une interface fournie et une classe qu'on sait pouvoir utiliser. Elles sont fortement basées sur la notion de sous-typage dans un contexte de programmation objet, et donc sont moins flexibles en termes d'expressivité que notre approche, bien qu'elles apportent l'adaptation dans un contexte dynamique.

Ehrig & al [29] présentent un cadre pour modéliser des architectures composants en utilisant des techniques formelles telles que les réseaux de Petri et CSP : les connexions entre interfaces requises et fournies sont représentées par des transformations de graphes utilisant des notions de composition, d'extension et de raffinement. Notre approche est similaire avec l'utilisation de B pour exprimer les transformations comme des raffinements entre interfaces requises et fournies.

Bracciali & al [30] spécifient un adaptateur comme un ensemble de correspondances entre les méthodes et les paramètres des composants requis et fournis. Un adaptateur est formalisé par un ensemble de propriétés exprimées à l'aide du π-calcul.

La génération automatique d'adaptateurs est limitée à une certaine classe de problèmes car la vérification de l'interopérabilité repose sur la décidabilité de l'inclusion des composants. Dans notre approche, nous proposons des schémas

pour construire et vérifier les adaptateurs, en fonction de différents cas de figures de l'architecture, sans aller jusqu'à leur génération automatique.

7 Conclusion

L'approche composants est un paradigme bien connu et utilisé dans le développement de logiciels, aussi bien dans le milieu académique que dans le milieu industriel. Dans cette approche, les composants sont considérés comme des boîtes noires décrites en termes de leur comportement visible et de leurs interfaces, qu'elles soient requises ou fournies. Ils sont assemblés via leurs interfaces. Il est bien connu que la correction d'une connexion peut s'exprimer en termes de raffinement : l'interface fournie doit raffiner l'interface requise.

Dans une approche de réutilisation de composants existants, les composants ont rarement des interfaces fournies qui raffinent directement l'interface requise du composant auquel on veut le connecter. Il est nécessaire d'introduire un adaptateur entre les composants pour les rendre compatibles. Un adaptateur est un programme qui définit comment les interfaces requises sont réalisées en termes des interfaces fournies : il exprime la correspondance entre variables, types et opérations.

Nous avons proposé une approche systématique de développement formel par composants basée sur des schémas d'assemblages UML et B. Ces schémas permettent d'exprimer la notion d'adaptateur et plus généralement le développement d'un nouveau composant par assemblage de composants préexistants , en fonction des différents cas de figures de l'architecture. Nous ne proposons pas de les générer automatiquement.

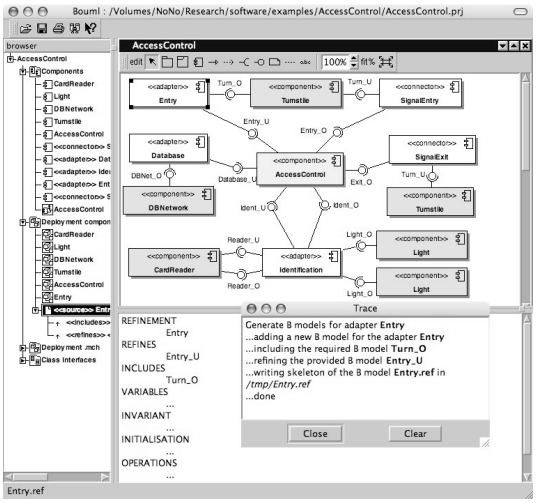


Figure 19. BOUML pour générer un modèle B

Grâce à l'utilisation de la méthode B, et de ses mécanismes d'assemblage et de raffinement pour modéliser les interfaces et les adaptateurs, nous obtenons la preuve de la correction de l'assemblage de composants. Le prouveur B garantit que cet assemblage est une implantation correcte des fonctionnalités attendues en termes des composants existants.

Les points forts de notre approche sont :

- l'utilisation de notations simples et de haut niveau pour exprimer l'architecture du système et ses interfaces ;
- des schémas d'assemblage utilisant les mécanismes classiques de composition et de raffinement ;
- un guide pour développer incrémentalement de nouveau composants ;
- la preuve de la correction de l'assemblage des composants.

L'implantation d'un plugin pour BOUML [5] fondé sur les schémas de développement présentés dans cet article est en cours : la figure 19 montre la génération du squelette du modèle B correspondant à l'assemblage Entry.

L'extension de l'approche avec la prise en compte de propriétés de sécurité dans une architecture composants existante, sans modification de ses fonctionnalités de base [31] est en cours d'étude. Ce travail doit également être complété par un outil d'aide à la détection des incompatibilités.

Remerciements Ce travail a bénéficié d'une aide de l'Agence Nationale de la Recherche dans le cadre du projet TACOS [6], référence ANR-06-SETI-017.

Références

[1] Szyperski, C. : Component Software. ACM Press, Addison-Wesley (1999)

[2] Chouali, S., Heisel, M., Souquières, J. : Proving component interoperability with B refinement. Electronic Notes in Theoretical Computer Science **160** (2006) 157–172

[3] Hatebur, D., Heisel, M., Souquières, J. : A method for component-based software and system development. In : Proceedings of the 32nd Euromicro Conference on Software Engineering And Advanced Applications, IEEE Computer Society (2006) 72–80

[4] Mouakher, I., Lanoix, A., Souquières, J. : Component adaptation : Specification and verification. In : Proc. of the 11th Int. Workshop on Component Oriented Programming, satellite workshop of ECOOP. (2006) 23–30

[5] Object Management Group (OMG) : UML Superstructure Specification. (2005) version 2.0.

[6] Abrial, J.R. : The B Book. Cambridge University Press (1996)

[7] Behm, P., Benoit, P., Meynadier, J.M. : METEOR : A successful application of B in a large project. In : Integrated Formal Methods. Volume 1708 of LNCS., Springer Verlag (1999) 369–387

5. http ://bouml.free.fr
6. http ://tacos.loria.fr

[8] Badeau, F., Amelot, A. : Using B as a high level programming language in an industrial project : Roissy VAL. In : Formal Specification and Development in Z and B. Volume 3455 of LNCS., Springer-Verlag (2005) 334–354

[9] Steria : Obligations de preuve : Manuel de référence, version 3.0. Steria – Technologies de l'information. (1998)

[10] Clearsy : B4free. website, `http://www.clearsy.com/` (2008)

[11] Meyer, E., Souquières, J. : A systematic approach to transform OMT diagrams to a B specification. In : Proceedings of the Formal Method Conference. Number 1708 in LNCS, Springer-Verlag (1999) 875–895

[12] Ledang, H., Souquières, J. : Modeling class operations in B : application to UML behavioral diagrams. In : 16th IEEE International Conference on Automated Software Engineering, IEEE Computer Society (2001) 289–296

[13] RODIN : Rigorous Open Development Environment for Complex Systems. website (August 2007) `http://rodin-b-sharp.sourceforge.net`.

[14] Colin, S., Lanoix, A., Souquières, J. : Trustworthy interface compliancy : data model adaptation. In : Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA), Satellite workshop of ETAPS. (March 2007)

[15] AFADL : Étude de cas : système de contrôle d'accès. In : Journées AFADL, Approches formelles dans l'assistance au développement de logiciels. (2000) actes LSR/IMAG.

[16] Brown, A.W., Wallnan, K.C. : Engineering of component-based systems. In : Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society (1996) 414

[17] Heineman, G., Ohlenbusch, H. : An evaluation of component adaptation techniques. Technical Report WPI-CS-TR-98-20, Department of Computer Science, Worcester Polytechnic Institute (February 1999)

[18] Heisel, M., Santen, T., Souquières, J. : Toward a formal model of software components. In : Proc. 4th International Conference on Formal Engineering Methods - ICFEM'02. Number 2495 in LNCS, Springer-Verlag (2002) 57–68

[19] Canal, C., Murillo, J.M., Poizat, P. : Software adaptation. L'Objet **12**(1) (2006) 9–31

[20] Purtilo, J.M., Atlee, J.M. : Module reuse by interface adaptation. Software - Practice and Experience **21**(6) (1991) 539–556

[21] Yellin, D.D.M., Strom, R.E. : Protocol specifications and component adaptors. ACM Transactions on Programming Languages and Systems **19**(2) (1997) 292–333

[22] Zaremski, A.M., Wing, J.M. : Specification matching of software components. ACM Transaction on Software Engeniering Methodolology **6**(4) (1997) 333–369

[23] Schmidt, H.W., Reussner, R.H. : Generating adapters fo concurrent component protocol synchronisation. In Crnkovic, I., Larsson, S., Stafford, J., eds. : Proceeding of the 5th IFIP International conference on Formal Methods for Open Object-based Distributed Systems. (2002) 213–229

[24] Reussner, R.H., Schmidt, H.W., Poernomo, I.H. : Reasoning on software architectures with contractually specified components. In Cechich, A., Piattini, M., Vallecillo, A., eds. : Component-Based Software Quality : Methods and Techniques. Springer-Verlag, Berlin, Germany (2003) 287–325

.

[25] Poizat, P., Salaün, G., Tivoli, M. : An adaptation-based approach to incrementally build component systems. Electronic Notes in Theoretical Computer Science **182** (2007) 155–170

[26] WCAT2006 : Coordination and adaptation techniques : Bridging the gap between design and implementation. In Becker, S., Canal, C., Diakov, N., Murillo, J.M., Poizat, P., Tivoli, M., eds. : Proceedings of the Third International Workshop on Coordination and Adaptation Techniques for Software Entities. (2006)

[27] Mätzel, K.U., Schnorf, P. : Dynamic component adaptation. Technical report, Ubilab laboratory, Union Bank of Switzerland, Zürich, Switzerland (June 1997)

[28] Kniesel, G. : Type-safe delegation for run-time component adaptation. Lecture Notes in Computer Science **1628** (1999) 351–366

[29] Ehrig, H., Padberg, J., Braatz, B., Klein, M., Orejas, F., Perez, S., Pino, E. : A generic framework for connector architectures based on components and transformation. In : FESCA'04, satellite of ETAPS'04. Volume 108 of ENTCS. (2004) 53–67

[30] Bracciali, A., Brogi, A., Canal, C. : A formal approach to component adaptation. Journal of Systems and Software **74**(1) (2005) 45–54

[31] Lanoix, A., Hatebur, D., Heisel, M., Souquières, J. : Enhancing dependability of component-based systems. In Verlag, S., ed. : Reliable Software Technologies Ada-Europe 2007. Number 4498 in LNCS, Springer-Verlag (2007) 41–54

Session groupe COSMAL

Composants Objets Services :

Modèles, Architectures et Langages

Construction dynamique d'annuaires de composants par classification de services utilisant l'analyse formelle de concepts

Gabriela Arévalo[1], Nicolas Desnos[2], Marianne Huchard[3],
Christelle Urtado[2], Sylvain Vauttier[2]

[1] LIFIA - Facultad de Informática (UNLP), La Plata, Argentina
[2] LGI2P - Ecole des Mines d'Alès, Nîmes, France
[3] LIRMM - UMR 5506 - CNRS and Univ. Montpellier 2, Montpellier, France

garevalo@sol.info.unlp.edu.ar,
{Nicolas.Desnos, Christelle.Urtado, Sylvain.Vauttier}@ema.fr,
huchard@lirmm.fr

Le génie logiciel à base de composants permet de construire des applications par assemblage de composants sur étagère. Pour faciliter ce processus, les composants exposent leur description externe: les interfaces requises et fournies par le composant correspondent à la description syntaxique des services que le composant met à disposition des composants de son environnement ou que le composant s'attend à trouver chez les composants de son environnement pour fonctionner. De précédents travaux sur l'assemblage automatique de composants et sur l'évolution dynamique d'assemblages [7, 6, 5] nous ont fait ressentir le besoin de disposer d'un annuaire de composants performant. En effet, l'étape de recherche, dans un annuaire, de composants disponibles en bibliothèque et compatibles avec ou substituables à un composant donné n'est pas triviale. En tout état de cause, les annuaires de type *pages blanches*, qui sont les plus fréquemment utilisés, ne sont pas adaptés car ils ne sont pas structurés pour permettre le choix de composants compatibles ou substituables.

L'idée de cet article est de proposer les mécanismes de base d'une indexation automatique des composants dans un annuaire de type *pages jaunes* qui soit efficace pour la recherche de composants compatibles avec, ou substituables à, un composant donné. Nous cherchons à exploiter l'information contenue dans la description syntaxique des composants afin d'automatiser entièrement la recherche et la connexion de composants, par opposition aux approches intégrant des informations sémantiques fournies manuellement pour indexer les composants. Notre approche s'appuie sur l'Analyse Formelle de Concepts (AFC) qui permet de pré-calculer un treillis de Galois [4] (ou treillis de concepts [10]) classifiant les interfaces de composants, c'est-à-dire organisant les descriptions de services de telle façon que la recherche en soit naturellement facilitée.

Notre approche se décompose en quatre étapes:

1. Nous posons tout d'abord les fondements d'une extension de la théorie des types des langages orientés-objet au *typage des composants* qui prenne en compte le caractère requis ou fourni des fonctionnalités.
2. Nous utilisons ensuite l'AFC pour construire, à partir de la hiérarchie des types d'objets utilisés par les fonctionnalités comme types de paramètres, un *treillis de signatures de fonctionalités* [2]. Ce dernier prend en compte leur nom, le type de leur paramètre de retour, les types de leurs paramètres d'entrée et leur direction (requise ou fournie). Il permet, pour une fonctionnalité (requise ou fournie) donnée, de trouver toutes les fonctionnalités qui peuvent lui être connectées ou substituées.
3. Nous utilisons encore l'AFC pour construire, à partir de la hiérarchie de signatures de fonctionnalités obtenue à l'étape précédente, un *treillis d'interfaces* [3]. Ce treillis organise les types d'interfaces et permet, pour une interface (requise ou fournie) donnée, de trouver toutes les interfaces qui peuvent lui être connectées ou substituées.

4. Nous utilisons enfin l'AFC pour construire, à partir de la hiérarchie des interfaces obtenue à l'étape précédente, un *treillis de composants* [1]. Ce treillis organise les types de composants et permet, pour un composant donné, de trouver tous les composants qui peuvent lui être connectés ou substitués.

Ce dernier treillis fournit ainsi une classification intelligible pour le développeur ou l'architecte : la représentation graphique du treillis permet de visualiser facilement les composants et leurs relations. De plus, ce treillis sépare le calcul de compatibilité des composants de la recherche qui est réalisée au cours du processus d'assemblage ou d'évolution. Enfin, ce treillis peut être utilisé comme un index, aussi bien pour la recherche d'un composant compatible avec un composant donné (en vue d'assemblage), que pour la recherche d'un composant comparable à un composant donné (en vue d'une substitution).

Plus encore, l'AFC permet la découverte de descriptions externes de composants (nouveaux types de composants) qui n'existent pas dans la biblothèque mais sont plus abstraits et réutilisables que les composants existants. Ces nouvelles abstractions peuvent être un plus pour guider les développeurs dans leur processus d'ingénierie ou de réingénierie ainsi que pour enrichir la bibliothèque.

Nos travaux ont été implémentés comme une extension de l'outil GaLicia [9, 8].

References

1. Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. FCA-based service classification to dynamically build efficient software component directories. *International Journal of General Systems, ISSN 0308-1079, to appear*, Taylor and Francis.
2. Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Precalculating component interface compatibility using FCA. In Jean Diatta, Peter Eklund, and Michel Liquière, editors, *Proceedings of the 5^{th} international conference on Concept Lattices and their Applications (CLA 2007), CEUR Workshop Proceedings Vol. 331, ISSN 1613-0073*, pages 241–252, Montpellier, France, October 2007.
3. Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Construction dynamique d'annuaires de composants par classification de services. In Y. Aït-Ameur, editor, *Actes de la 2ème Conférence francophone sur les Architectures Logicielles 2008 (CAL2008), Revue des Nouvelles Technologies de l'Information (RNTI-L-2), ISSN 1764-1667*, pages 123–138, Montréal, Canada, March 2008. Editions Cépaduès. Cet article a été accepté conjointement à LMO2008.
4. M. Barbut and B. Monjardet. *Ordre et Classification*. Hachette, 1970.
5. Nicolas Desnos, Marianne Huchard, Guy Tremblay, Christelle Urtado, and Sylvain Vauttier. Search-based many-to-one component substitution. *Journal of Software Maintenance and Evolution: Research and Practice. Special Issue on Search-Based Software Engineering*, 20(5):321–344, September/October 2008.
6. Nicolas Desnos, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Guy Tremblay. Automated and unanticipated flexible component substitution. In *Proceedings of the 10^{th} ACM SIGSOFT CBSE*, LNCS 4608, pages 33–48. Springer, 2007.
7. Nicolas Desnos, Sylvain Vauttier, Christelle Urtado, and Marianne Huchard. Automating the building of software component architectures. In *Software Architecture: 3^{rd} European Workshop on Software Architectures, Languages, Styles, Models, Tools, and Applications (EWSA), Revised selected papers*, LNCS 4333, pages 228–235. Springer, 2006.
8. GaLicia. Galois lattice interactive constructor. http://www.iro.umontreal.ca/∼galicia - accessed on Sept. 22, 2008, 2002.
9. Petko Valtchev, David Grosser, Cyril Roume, and Mohamed Rouane Hacene. GaLicia: an open platform for lattices. In Aldo de Moor, Wilfried Lex, and Bernhard Ganter, editors, *Using Conceptual Structures: Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03)*, pages 241–254. Shaker Verlag, July 2003. http://www.iro.umontreal.ca/∼galicia.
10. R. Wille. Restructuring lattice theory: an approach based on hierarchies of concepts. *Ordered Sets*, 83:445–470, September 1982.

Vers la génération de modèles de sûreté de fonctionnement

Xavier Dumas[1], Claire Pagetti[1], Laurent Sagaspe[1], Pierre Bieber[1], Philippe Dhaussy[2]

[1] ONERA-CERT - 2 av. E. Belin 31055 Toulouse
[2] ENSIETA - LISyC - DTN - 2 rue F. Verny 29806 Brest

Contexte. La conception et le développement de systèmes embarqués critiques sont assujettis à la fois à des objectifs économiques, telle la réduction des coûts et du temps de développement, mais également au respect des normes de sécurité. Dans le contexte aéronautique, par exemple, ces contraintes sont amplifiées puisque le processus de développement doit répondre à une certaine fiabilité pour passer l'étape de certification. De ce fait, le cycle de développement est soumis à davantage de validation et de vérification ainsi qu'à une plus grande traçabilité. Dès lors, les activités d'évaluation liées à la *sûreté de fonctionnement* [Lap89], où l'on établit le niveau de confiance justifié qu'il est possible d'attribuer à un système lorsqu'il est utilisé correctement, occupent une place prépondérante. Le cycle de développement d'un système critique est le résultat de l'imbrication d'un cycle de développement que l'on pourrait qualifier de *classique* et d'un cycle de sûreté de fonctionnement.

Généralement, les équipes de développement et de sûreté de fonctionnement sont dissociées, ce qui donne une importance majeure aux moyens d'interaction entre ces équipes. Notre contribution se situe dans l'aide outillée des échanges entre équipes en phase de validation d'architectures préliminaires. Une architecture préliminaire est l'allocation d'une *architecture fonctionnelle*, c'est-à-dire un découpage fonctionnel du système, sur une architecture support préliminaire appelée *architecture matérielle*. L'étape de validation consiste à assurer que les exigences de sécurité issues de l'analyse des risques sont satisfaites. Si l'allocation proposée ne peut répondre aux exigences, les architectures et l'allocation sont rejetées. Les concepteurs doivent alors reconsidérer le raffinement des fonctions et le choix du support afin de proposer une nouvelle solution compatible avec les exigences. Trouver une solution peut donc se faire à la suite d'un certain nombre d'itérations entre les équipes de développement et de sûreté de fonctionnement.

Objectifs et contribution. L'objectif est de proposer un atelier de spécification / modélisation unifié pour les concepteurs et de simplifier les étapes de modélisation de sûreté de fonctionnement. La première étape est la mise en place d'une méthodologie permettant la génération de modèles de sûreté de fonctionnement. L'idée est la suivante : si les spécifications sont exprimées dans un formalisme à base de modèles, ce qui tend à être le cas avec la prise en compte progressive de l'approche "Ingénierie Dirigée par les Modèles" [FEBF06] (IDM), alors il est possible de générer un modèle partiel de sûreté de fonctionnement. Un des

avantages de l'approche est de partir d'un modèle commun à la fois pour le développement -en effet, il existe des générateurs de code à partir de modèles - et pour les analyses -notamment de sûreté de fonctionnement. Sous réserve que les transformations soient correctes, le modèle d'analyse est conforme à la spécification étudiée. Le choix d'une transformation de modèles plutôt qu'une compilation classique repose sur l'idée de mettre en place un atelier unifié ce qui sous tend que plusieurs formalismes seront pris en entrée, comme du Simulink ou du Scade. Les règles de transformation seront relativement proches et l'adaptation du code sera plus simple. De plus, il est prévu de coder des transformations inverses, comme de l'AltaRica vers du AADL. L'IDM permet la réutilisation de briques de base d'une transformation à l'autre.

Pour mettre en œuvre cette méthodologie, nous avons opté pour les langages AADL [FGH06] (pour Architecture Analysis & Design Language) et AltaRica, mais nous aurions aussi bien pu porter notre choix sur d'autres formalismes similaires. Le langage AADL est adapté pour décrire des architectures fonctionnelles et logicielles. Le langage AltaRica [AGPR00] est un langage formel dédié à des modélisations orientées sûreté de fonctionnement. La traduction est implantée sous forme de transformation de modèles [FEBF06] et l'outil utilisé est KerMeta [MFJ05], il aurait également été possible d'utiliser d'autres logiciels de transformation conforme QVT.

Les travaux d'Ana-Elena Rugina [RKK06] sont proches des problématiques décrites dans l'introduction, mais les choix méthodologiques sont différents. Le langage source est le langage AADL étendu avec l'annexe *AADL Error Model Annex* [SA06]. Cette annexe permet de décrire des modèles d'erreur permettant ainsi la modélisation du comportement en présence de fautes. Les modèles sont davantage enrichis et proches des modèles de sûreté de fonctionnement que ceux que nous produisons en utilisant notre méthodologie. Une poursuite du travail est donc de prendre un entrée des modèles AADL étendus avec l'annexe erreur. En revanche, l'avantage de notre approche est la simplicité de l'algorithme : nous produisons un modèle AltaRica en parcourant le modèle AADL une fois ce qui n'est pas le cas de [RKK06] où les transitions déclenchées par des propagations sont identifiées progressivement par itération.

Une version complète de ce travail est présentée dans l'article [DPS+08].

Références

[AGPR00] André Arnold, Alain Griffault, Gérald Point, and Antoine Rauzy. The AltaRica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40 :109–124, 2000.

[DPS+08] Xavier Dumas, Claire Pagetti, Laurent Sagaspe, Pierre Bieber, and Philippe Dhaussy. Vers la génération de modèles de sûreté de fonctionnement. In *conférences LMO'08 et CAL'08*, March 2008.

[FEBF06] Jean-Marie Favre, Jacky Estublier, and Mireille Blay-Fornarino. *L'ingénierie dirigée par les modèles. Au-delà du MDA (Traité IC2, série Informatique et Systèmes d'Information)*. Lavoisier, 2006.

[FGH06] Peter.H. Feiler, David P. Gluch, and John J. Hudak. The Architecture Analysis and Design Language (AADL) : An introduction. Technical report, Society of Automotive Engineers (SAE), February 2006.

[Lap89] J.C. Laprie. Dependability : a unifying concept for reliable computing and fault tolerance. In T. Anderson, editor, *Dependability of Resilient Computers*, chapter 1, pages 1–28. Blackwell Scientific Publications, Oxford, UK, 1989.

[MFJ05] Pierre Alain Muller, Franck Fleurey, and Jean Marc Jézéquel. Weaving executability into object-oriented meta-languages. In *Proceedings of MODELS/UML 2005*, 2005.

[RKK06] Ana-Elena Rugina, Karama Kanoun, and Mohamed Kaâniche. Modélisation de la sûreté de fonctionnement de système à partir du langage aadl. *Rapport LAAS*, 2006.

[SA06] SAE-AS5506/1. Architecture analysis and design language annex volume 1. *Error Model Annex*, 2006.

Composition et expression qualitative de politiques d'adaptation pour les composants Fractal

Franck Chauvel[3], Olivier Barais[1], Noël Plouzeau[1], Isabelle Borne[2], and Jean-Marc Jézéquel[1]

[1] IRISA (INRIA & Université de Rennes 1)
Campus de Beaulieu
F-35042 RENNES (FRANCE)
prenom.nom@irisa.fr
[2] Laboratoire VALORIA (Université de Bretagne Sud)
Centre de Recherche Yves Coppens
Campus de Tohannic
56017 VANNES CEDEX
prenom.nom@univ-ubs.fr
[3] National Laboratory of High Confidence Software Technologies
School of Electronics Engineering and Computer Science,
Peking University, Beijing, 100871, China
franck.chauvel@sei.pku.edu.cn

Abstract. Les plates-formes d'exécution récentes telles que Fractal ou OpenCOM offrent de nombreuses facilités pour assurer la prise en compte de propriétés extra-fonctionnelles (introspection, sondes, chargement dynamique, etc). Cependant, l'intégration de politiques d'adaptation reste délicate car elle nécessite de corréler la configuration du système avec l'évolution de son environnement. Le travail présenté dans cet article étend la plate-forme Fractal avec les mécanismes nécessaires à l'exécution de politiques d'adaptation de haut niveau. L'approche est illustrée à l'aide d'un serveur HTTP qui doit modifier sa configuration (architecturale et locale) en fonction de plusieurs paramètres extra-fonctionels.

1 Introduction

Lors du développement de nouveaux systèmes logiciels, l'aspect fonctionnel est de moins en moins l'unique critère de satisfaction de l'utilisateur : les problématiques extra-fonctionnelles sont au cœur des efforts de recherche actuels. Il est, par exemple, nécessaire d'être capable de maintenir un niveau correct pour des propriétés liées à la qualité de service (QoS pour "Quality of Service") telles que la fiabilité, la disponibilité, le temps de réponse, l'ergonomie, *etc.*

Pour maintenir la qualité de service d'un système, deux approches sont possibles. On peut d'une part,vérifier à priori, (lors de la conception) que les contraintes sur les propriétés de qualité de service ne sont pas violées à l'exécution. L'utilisation de réseaux de Petri permet par exemple d'obtenir des prévisions sur les consommations de ressources (mémoire, batterie, CPU, bande passante, etc). On peut d'autre part, s'assurer, à posteriori, de la qualité de service fournie (à l'exécution) en dotant les systèmes d'une capacité d'auto-adaptation à leur environnement pour assurer au mieux la QoS demandée.

Ainsi, les plates-formes à composant récentes permettant de construire des applications modulaires et reconfigurables telles Fractal ([3]), OpenCOM ([4]) ou Spring ([10]) offrent les mécanismes nécessaires au support des adaptations (introspection, chargement dynamique). Cependant, elles n'intègrent pas directement la notion de politique d'adaptation. Deux raisons principales expliquent cela. D'une part le nombre de propriétés extra-fonctionnelles est potentiellement infini. Il est alors impératif de pouvoir segmenter cet espace pour pouvoir en maîtriser la complexité. D'autre part, il nécessaire de conserver un niveau d'abstraction suffisant pour garder des spécifications exploitables.

La contribution des travaux présentés dans cet article est de proposer un cadre pour la définition de politiques d'adaptation de haut niveau pour les plates-formes à composants. Les mécanismes

nécessaires pour le support de ces politiques d'adaptations sont illustrées au travers d'une extension du modèle de composants Fractal et de son implémentation de référence Julia. Les politiques d'adaptations utilisées sont qualitatives mais leur sémantique, basée sur la logique floue, permet d'inférer des valeurs quantitatives lors de l'exécution pour adapter l'architecture en fonction de son contexte d'exécution. La sémantique du moteur d'inférence présentée dans ce papier permet de préserver la capacité de composer ces politiques indépendanmment de l'ordre dans lequel elles ont été définies.

La suite de cet article est organisée de la façon suivante. La section 2 présente les motivations sur l'exemple d'un serveur HTTP et la section 3 introduit le formalisme utilisé pour modéliser les politiques d'adaptations. La section 4 démontre la sémantique associée aux politiques d'adaptation permettant leur composition. L'intégration dans la plate-forme Fractal est présentée dans la section 5 qui présente à la fois l'algorithme d'adaptation et son implémentation sous forme de contrôleur Fractal. Une validation de l'approche, basée sur l'exemple du serveur HTTP, est présentée dans la section 6. Une sélection des travaux connexes est commentée dans la section 7. La section 8 conclue et présente les perspectives ouvertes par ces travaux.

2 Motivations

Pour démontrer la nécessité des politiques d'adaptation, cette section présente brièvement un serveur HTTP et ainsi que les exigences en termes de qualité et d'adaptation qui lui sont associées.

La figure 1 présente l'architecture proposée pour le serveur HTTP *Cherokee*. Il s'agit d'une variation de l'exemple *Comanche*[4] utilisé dans plusieurs communications ([6]) autour de la plate-forme Fractal. Les requêtes HTTP sont lues sur le réseau par le composant *RequestReceiver* qui les transmet au composant *RequestHandler*. Pour traiter une requête, ce dernier peut soit consulter le cache (composant *CacheHandler*), soit la transmettre au composant *RequestDispatcher* qui interroge alors un ferme de serveurs de fichiers pour résoudre la requête.

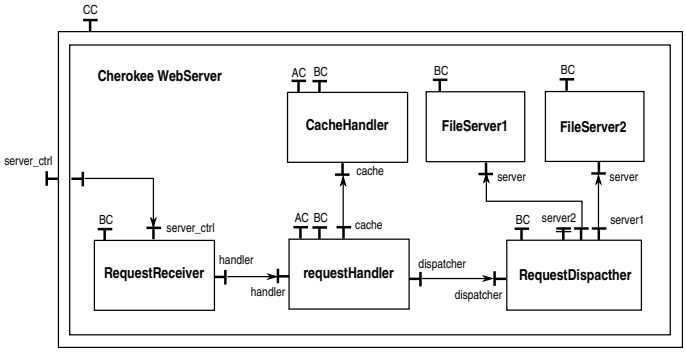


Fig. 1. Architecture du serveur HTTP Cherokee présentée à l'aide de la notation Fractal

Cette architecture est dotée d'un cache (*CacheHandler*) et d'un contrôleur de charge (composant *RequestDispatcher* pour pouvoir maîtriser le temps de réponse et le garder aussi court que possible. Pour pouvoir supporter d'éventuelles montées en charge, les exigences suivantes ont été définies:

1. Le composant *CacheHandler* ne doit être déployé que si le nombre de requêtes HTTP similaires (l'indice de dispersion) est élevé.
2. La quantité mémoire allouée pour le fonctionnement du composant *CacheHandler* doit évoluer en fonction de la charge globale du serveur.

[4] Accessible dans le tutorial Fractal, à l'adresse : `http://fractal.objectweb.org/tutorial/index.html`

3. La durée de validité des informations du cache doit évoluée également en fonction de la charge globale du serveur (nombre de requêtes par unité de temps).
4. Le nombre de serveurs de données déployés doit être corrélé avec la charge globale du serveur

Plusieurs caractéristiques de ces exigences doivent être soulignées. D'une part, ces exigences peuvent être classifiées en deux catégories: les règles locales, et les règles architecturales. Les règles locales concernent la configuration d'un composant particulier. C'est le cas de l'exigence 2 qui ne concerne que la configuration du cache. Les règles architecturales, par opposition, concernent la configuration de la collaboration entre les composants. Les exigences 1 et 4 en sont de bons exemples, puisque qu'elles nécessitent de modifier l'agencement des composants et des connecteurs (appelés *bindings* dans la terminologie Fractal).

D'autre part, ces exigences sont décrites de manière qualitative, c'est-à-dire à l'aide d'un vocabulaire spécifique à chaque type propriété. Dans la première exigence par exemple, il est question d'un indice de dispersion "élevé", ce qui est purement qualitatif car aucune valeur précise n'est fournie pour quantifier le terme "élevé".

3 Modélisation des politiques d'adaptation

Pour prendre en compte les exigences que nous avons présentés dans la section précédente, les politiques d'adaptations sont décrites à l'aide de quatre élements:

- un ensemble de reconfigurations architecturales
- un ensemble de propriétés et les domaines de valeur qui leurs sont associés
- un ensemble de règles pour les configurations locales
- un ensemble de règles pour les reconfigurations architecturales

Reconfiguration architecurales Les différentes reconfigurations architecturales qui sont utilisées dans une politique d'adaptation sont décrites sous la forme d'actions FScript. Chaque reconfiguration est nommée et pointe vers un fichier qui contient la description précise de la reconfiguration sous forme d'actions FScript.

```
policy withCache
is
   reconfiguration addCache is 'addCache.fscript'
   reconfiguration removeCache is 'removeCache.fscript'
```

Propriétés L'environnement du système est capturé par un ensemble de propriétés, chacune relié à un domaine spécifique. Chaque domaine inclut la description du vocabulaire spécifique à utiliser pour qualifier les propriétés qui lui sont associées. L'exemple suivant présente les domaines et les propriétés utilisés pour décrire l'adaptation du cache. La charge moyenne du serveur est décrite par une propriété *load* associée au domaine *AverageLoad* et peut donc être qualifiée en utilisant les termes *low*, *medium* et *high*. Pour plus de concision, la syntaxe que nous proposons offre également la possibilité de décrire une propriété et le domaine qui lui correspond d'une seule traite.

```
domain AverageLoad : Real
   evolves in [0..100] as low medium high

property load : Real
   sensor is getLoad on requestHandler

property requestDeviation : Real
   evolves in [0..100] as low medium large
   sensor is getRequestDeviation on requestHandler

property size : MemorySize
   evolves in [0..2000] as small medium large
   sensor is getMemoryUsed on cache
   actuator is setMemoryUsed on cache

property validityDuration : Duration
   evolves in [0..100] as short normal long
   sensor is getValidityDuration on cache
   actuator is setValidityDuration on cache
```

Pour chaque domaine, la sémantique des termes qui lui sont associés est inférée à partir du nombre de termes et du domaine de définition. Par exemple, la charge du serveur, qui évolue entre 0 et 100 sera 'low' si elle est inférieur à 50, 'medium' si elle est comprise entre 25 et 75 et 'high' si elle est supérieure à 50. (Voir la section 5.1.)
De plus, à chaque propriété sont associés un *sensor* et un *actuator* qui indiquent respectivement comment mesurer et/ou modifier cette propriété dans l'architecture. Pour une architecture Fractal, ces deux élements pointe vers des attributs de configuration d'un des composants impliqués (Fractal Attribute-Controller).

Règles de reconfiguration architecturale Les différentes reconfigurations architecturales possibles au sein d'une architecture sont capturées à l'aide d'actions architecturales. Ces actions peuvent être utilisées dans des règles de reconfiguration architecturale. Chacune de ces règles met en relation le contexte d'exécution du système et l'utilité de déclencher une action architecturale. Dans l'exemple suivant, l'utilité de déployer le composant *Cache* est faible si les requêtes sont toutes différentes (propriété *requestDeviation*).

```
when requestDeviation is 'low'
if count($context::child/*::interface/CacheHandler[server(.)]/component) = 0
   then utility of addCache is very 'high'

when load is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) = 0
   then utility of addCache is 'medium' or 'high'

when load is 'low' and requestDeviation is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then utility of removeCache is 'high'
```

Règles de configuration locale Les reconfigurations locales, c'est-à-dire les reconfigurations qui n'impactent que les propriétés locales d'un composant, sont décrites à l'aide règles qui spécifient l'évolution voulue de la configuration locale. L'exemple suivant présente les règles nécessaires pour corréler la quantité de mémoire allouée pour le composants *Cache* avec la charge moyenne du serveur.

```
when proxy.load is 'low'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'small'

when proxy.load is 'medium'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'medium'

when proxy.load is 'high'
   if count($context::child/*::interface/CacheHandler[server(.)]/component) > 0
   then cache.size is 'large'

end policy
```

Toutes les règles, qu'elles soient locales ou architecturales sont gardées. La contrainte associée est exprimée à l'aide du langage FPath qui permet de naviguer les architectures Fractal comme XPath permet de naviguer les documents XML ([5]). Le contexte d'exécution de la garde, c'est-à-dire le composant composite englobant la collaboration, est dénoté par *$context*.

4 Composition de politiques d'adaptation

Comme nous l'avons expliqué précédemment, la description d'une politique d'adaptation générale à un système est délicate à cause de la complexité de l'environnement et des différentes variations possibles de l'architecture. Pour pouvoir briser cette complexité, il faut pouvoir traiter les problèmes séparément, à l'aide de politiques d'adaptations distinctes. Toutefois, il faut alors être capable de composer facilement les politiques d'adaptation. Si l'on considère la composition comme l'union des règles et des propriétés qui constituent les politiques d'adaptation, l'ordre d'unification, dans les règles notamment, ne doit pas avoir d'impact sur l'interprétation de la politique résultante. La suite de cette section montre cela.

Soit Σ une structure telle que $\Sigma = \langle P, V, T, K \rangle$ où l'on considère les éléments suivants :

- P est un ensemble de propriétés
- V est une fonction partielle qui associe une valeur à chaque propriété telle que :

$$V = \{v : P \nrightarrow R\}$$

- T est l'ensemble des topologies possibles
- K est l'ensemble des intro-actions architecturales qui conservent la valeur des propriétés associées à chaque composant. Plus formellement on obtient :

$$\begin{aligned} K = \{k : P, V, T, K \rightarrow P, V, T, K | \\ \forall\, (t, v) \in T \times V, \forall\, p \in dom(v), \\ \text{let } (t', v') = k(t, v) \\ \text{in } v(p) = v'(p) \end{aligned}$$

Pour pouvoir définir plus formellement l'algorithme, considérons les éléments suivants :

- A comme l'ensemble des politiques d'adaptation où chaque politique d'adaptation est un ensemble de règles d'adaptation tel que :

$$A = \{\, \langle R_P, R_A \rangle \,\}$$

où R_P est l'ensemble des règles configuratives R_A est l'ensemble des règles architecturales. On peut donc définir l'opérateur de composition entre deux politiques d'adaptation comme une union des ensembles de propriétés (en admettant un renommage) et une union des ensembles de règles. Formellement, on obtient :

$$\begin{aligned} \forall\, (f, g) \in A^2 \\ f \otimes g = \langle R_{P_F} \cup R_{P_G}, R_{A_F} \cup R_{A_G} \rangle \end{aligned}$$

- U est une fonction qui associe une utilité aux intro-actions architecturales

$$U = \{\, u : K \rightarrow \mathbb{R} \,\}$$

- *control* est une fonction qui représente l'algorithme de contrôle flou. Elle calcule une nouvelle valeur pour chaque propriété de l'architecture qu'elle a reçue en paramètre et calcule également l'utilité de chaque intro-action architecturale.

$$control : P, V, T, K, A \rightarrow V, U$$

- *select* est une fonction qui sélectionne l'intro-action architecturale la plus utile. Elle retourne l'intro-action la plus utile s'il existe un maximum unique parmi les utilités associées aux intro-actions. Si plusieurs intro-actions ont la même utilité, alors elle utilise une fonction σ (commutative) pour sélectionner l'une des intro-actions ayant une valeur d'utilité maximale.

$$\begin{aligned} select \;:\; U \rightarrow K \\ select(u) = \begin{cases} k \in \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) & \Big| \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| = 1 \\ \sigma\Big(\max\limits_{k_i \in dom(u)} \big(u(k_i)\big)\Big) & \Big| \max\limits_{k_i \in dom(u)} \big(u(k_i)\big) \Big| > 1 \end{cases} \end{aligned}$$

où σ est une fonction commutative qui sélectionne un élément parmi n. Elle rend alors la fonction *select* commutative également.

A l'aide de ces éléments notre algorithme peut se modéliser comme suit :

$$\begin{aligned} apply &: P, V, T, K, A \rightarrow P, V, T, K \\ apply(p, v, t, k, a) &= \text{let } (v', u) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\quad\quad \text{let } k_0 \text{ be } select(u) \\ &\quad\quad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

On peut alors démontrer que l'ordre parmi les règles n'a pas d'importance et que l'opérateur de composition est bien commutatif. Formellement, cela se traduit par l'équation suivante :

$$\begin{aligned} &\forall (p, v, t, k) \in (P \times V \times T \times K), \forall\, (f, g) \in A^2 \\ &\quad apply(p, v, t, k, \langle f, g \rangle) = apply(p, v, t, k, \langle g, f \rangle) \end{aligned}$$

Considérons les trois cas suivants:

– Si $(f, g) \in (R_A)^2$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v', \{u_f, u_g\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\quad\quad \text{let } k_0 \text{ be } select(\{u_f, u_g\}) \\ &\quad\quad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v', \{u_g, u_f\}) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\quad\quad \text{let } k_0 \text{ be } select(\{u_g, u_f\}) \\ &\quad\quad\quad \text{in } k_0(p, v', t, k) \end{aligned}$$

Dans ces deux cas précis, k_0 prend la même valeur puisque l'opération *select* est commutative.

– Si $(f, g) \in (R_P \times R_A)$ alors,

$$\begin{aligned} apply(p, v, t, k, \langle f, g \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\quad\quad \text{let } k_0 \text{ be } select(u_g) \\ &\quad\quad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

$$\begin{aligned} apply(p, v, t, k, \langle g, f \rangle) &= \text{let } (v_f, u_g) \text{ be } control(p, v, t, k, a) \\ &\quad \text{in} \\ &\quad\quad \text{let } k_0 \text{ be } select(u_g) \\ &\quad\quad\quad \text{in } k_0(p, v_f, t, k) \end{aligned}$$

Dans ces deux cas, l'opération *select* est utilisée sur des ensembles à un seul élément et k_0 prend donc la valeur de cet unique élément.

– Si $(f, g) \in (R_P)^2$ alors,

$$apply(p, v, t, k, \langle f, g \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \text{ in let } k_0 \text{ be } select(\varnothing) \text{ in } k_0(p, v', t, k)$$

$$apply(p, v, t, k, \langle g, f \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \text{ in let } k_0 \text{ be } select(\varnothing) \text{ in } k_0(p, v', t, k)$$

Nous considérons ici que f et g ne portent pas sur les mêmes propriétés. Dans ce cas particulier, la fonction *control* produirait une seule valeur pour cette propriété (une tendance moyenne) et l'ordre n'a donc pas non plus d'influence ici.

5 Implémentation pour la plate-forme Fractal

5.1 Du qualitatif vers le quantitatif

Pour pouvoir interpréter les politiques d'adaptations telles que nous les avons modélisées dans la section précédente, il faut pouvoir les interpréter avec l'imprécision qu'elles comportent. Pour cela, nous proposons d'utiliser la théorie des ensembles flous et ses applications en théorie du contrôle pour inférer des valeurs réelles à partir de descriptions qualitatives.

En logique floue ([17]), les variables appartiennent à des ensembles flous. La particularité de ces derniers, est que l'appartenance d'une variable à un ensemble n'est pas stricte (comme pour les ensembles classiques) mais graduelle. Un ensemble flou est donc principalement défini par sa fonction d'appartenance (généralement dénommé $\mu(x)$) et une variable peut donc appartenir à un ensemble à 76% par exemple. La figure 2 propose une modélisation du terme *low* utilisé pour décrire la charge du serveur. Dans notre approche, chaque terme définit dans le vocabulaire d'un domaine particulier est associé à un ensemble flou.

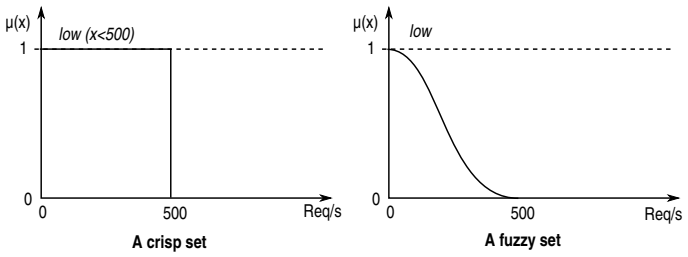


Fig. 2. Défintion d'un ensemble flou et de l'ensemble classique équivalent

En plus des opérateurs de base que sont l'union et l'intersection, la logique floue introduit également des opérateurs unaires (appelés *modificateurs*) qui permettent de construire des expressions plus précises, telles que *load is high or slightly medium*. Les principaux opérateurs sont *not*, *very*, *moderately* et *slightly*.

– Si $(f, g) \in (R_P)^2$ alors,

$$apply(p, v, t, k, \langle f, g \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \text{ in let } k_0 \text{ be } select(\varnothing) \text{ in } k_0(p, v', t, k)$$

$$apply(p, v, t, k, \langle g, f \rangle) = \text{let } (\{v_f, v_g\}, \varnothing) \text{ be } control(p, v, t, k, a) \text{ in let } k_0 \text{ be } select(\varnothing) \text{ in } k_0(p, v', t, k)$$

Nous considérons ici que f et g ne portent pas sur les mêmes propriétés. Dans ce cas particulier, la fonction *control* produirait une seule valeur pour cette propriété (une tendance moyenne) et l'ordre n'a donc pas non plus d'influence ici.

5 Implémentation pour la plate-forme Fractal

5.1 Du qualitatif vers le quantitatif

Pour pouvoir interpréter les politiques d'adaptations telles que nous les avons modélisées dans la section précédente, il faut pouvoir les interpréter avec l'imprécision qu'elles comportent. Pour cela, nous proposons d'utiliser la théorie des ensembles flous et ses applications en théorie du contrôle pour inférer des valeurs réelles à partir de descriptions qualitatives.

En logique floue ([17]), les variables appartiennent à des ensembles flous. La particularité de ces derniers, est que l'appartenance d'une variable à un ensemble n'est pas stricte (comme pour les ensembles classiques) mais graduelle. Un ensemble flou est donc principalement défini par sa fonction d'appartenance (généralement dénommé $\mu(x)$) et une variable peut donc appartenir à un ensemble à 76% par exemple. La figure 2 propose une modélisation du terme *low* utilisé pour décrire la charge du serveur. Dans notre approche, chaque terme définit dans le vocabulaire d'un domaine particulier est associé à un ensemble flou.

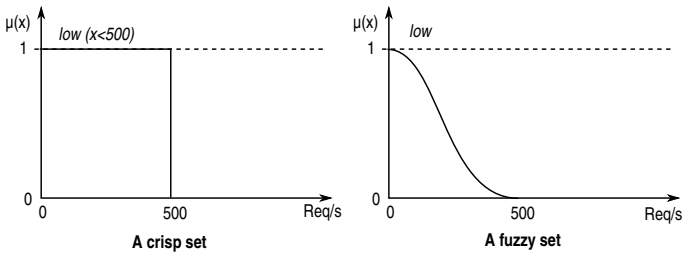


Fig. 2. Défintion d'un ensemble flou et de l'ensemble classique équivalent

En plus des opérateurs de base que sont l'union et l'intersection, la logique floue introduit également des opérateurs unaires (appelés *modificateurs*) qui permettent de construire des expressions plus précises, telles que *load is high or slightly medium*. Les principaux opérateurs sont *not*, *very*, *moderately* et *slightly*.

Cette notion d'ensemble flou a été réutilisée dans le domaine du contrôle flou ([14]) pour permettre de décrire des règles de contrôle de la façon la plus intuitive possible. Les règles peuvent être évaluées en trois étapes distinctes, respectivement: la fuzzification, l'inférence floue, et la défuzzification.

Considérons par exemple les deux règles suivantes pour illustrer l'évaluation de règles qualitatives. Ces deux règles sont également utilisées dans la figure 3.

1. load is medium ⇒ cacheSize is *medium*
2. load is low ⇒ cacheSize is *low*

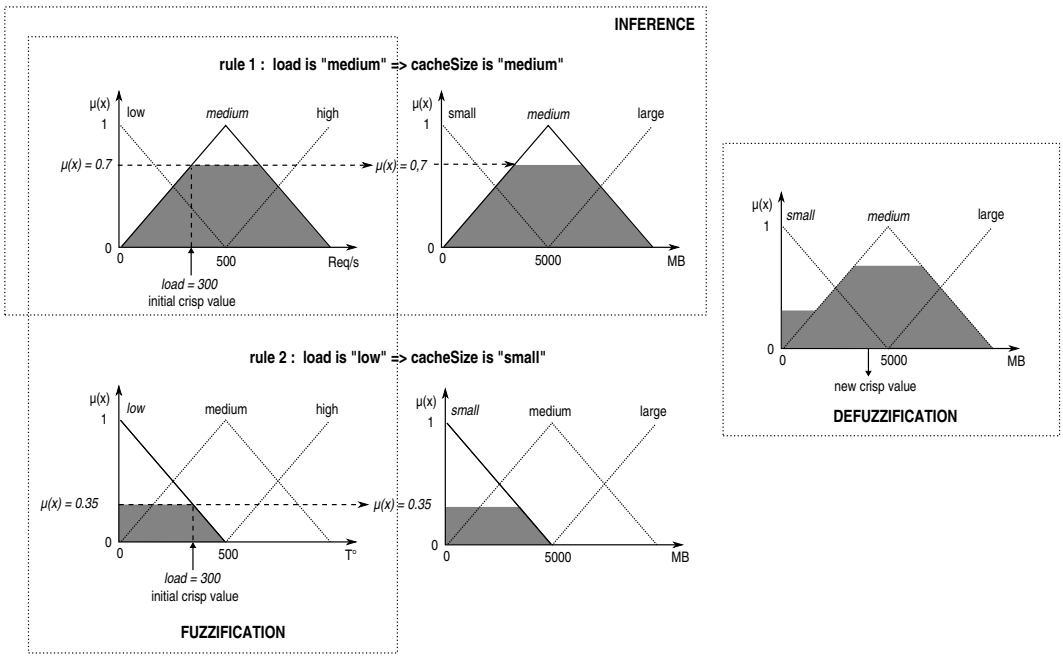


Fig. 3. Procédé d'évaluation des règles floues

La fuzzification Dans le cas des règles floues, il s'agit de mesurer le degré d'appartenance d'une variable à un ensemble flou. On s'intéresse à la partie gauche des règles: pour la règle 1 par exemple, on va calculer le degré d'appartenance de la propriété *load* à l'ensemble représenté par le terme *medium*. La figure 3 explique cela de façon graphique: la charge réelle est mesurée à 300 req/s et la fonction d'appartenance indique que cette charge est donc *medium* à 70%.

L'inférence floue Cette notion consiste à considérer que le degré d'appartenance mesuré dans la partie gauche d'une règle peut être réutilisé tel quel dans sa partie droite. Par exemple, puisque la fuzzification de la règle 1 a établi que la charge est médium à 70%, le principe d'inférence floue permet donc de déduire que la taille du cache sera *medium* à 70% également.

La défuzzification Une fois que la fuzzification et l'inférence floue ont été appliquées sur toutes les règles, chaque propriété peut prendre plusieurs valeurs floues (ou degré d'appartenance). Pour les deux règles utilisées dans la figure 3, la taille du cache est donc *medium* à 70% (règle 1) et *low* à 35% (règle 2). Pour calculer la valeur réelle correspondante (une quantité de mémoire), on agrège les deux aires correspondantes, et on calcule la projection du centre de gravité de l'aire résultante sur l'axe correspondant.

5.2 L'algorithme d'adaptation

L'algorithme utilisé pour appliquer les politiques d'adaptation est basé sur le processus d'évaluation des règles floues présenté dans la section précédente. Appliquer aux différentes règles d'une politique d'adaptation, ce processus permet de calculer à la fois une nouvelle valeur pour chacune des propriétés impactées par les règles mais également l'utilité de chaque action architecturale.

Le principe de l'algorithme est donc le suivant:

1. L'ensemble des règles de configuration locales est évalué, ce qui produit une nouvelle valeur pour chaque propriété impactée par au moins une règle.
2. L'indice d'utilité de chaque action architecturale est calculé à l'aide du même procédé, mais appliqué aux règles de reconfiguration architecturale (l'utilité d'une règle est considérée comme un propriété à part entière dont le domaine, définit par défaut, est une valeur comprise entre 0 et 100).
3. L'action qui a l'indice d'utilité le plus élevé est déclenchée mais elle n'est exécutée cependant que si l'indice dépasse un seuil d'utilité spécifié par l'utilisateur ; ce qui permet de conserver le système dans une certaine stabilité.

Dans la description suivante, le procédé d'évaluation des règles a été encapsulé dans la méthode *control* de la classe *Controller*.

```
operation adaptation(dataRules : Set<Rule>, architecturalRules : Set<Rule>) is
do
  var controller : Controller init Controller.new
  var newValues : Table<FuzzyProperty, Value> init Table<FuzzyProperty, Value>

  controller.control(dataRules, newValues)
  newValues.each{t:Entry | t.getKey().set(t.getValue())}

  var newActionUtilities : Table<ActionUtility, Value> init Table<ActionUtility, Value>

  controller.control(architecturalRules, newActionUtilities);

  var selectedAction : ActionUtility init
        newValues.select{ e:Entry | newValues.notexists{t:Entry | st.getValue() > e.getValue()}
             }.getKey()

  if (maxUtility > UsefulnessBound) then
     rule.action.execute()
  end
end
```

5.3 Un contrôleur dédié à l'adaptation

Les politiques d'adaptations que nous avons présentées jusqu'ici sont relatives à une collaboration entre plusieurs composants, encapsulée dans un composant composite. C'est donc ce composant composite qui est responsable à la fois des connections entre ses sous composants et de leur bonne configuration.

La plate-forme Fractal ([3]) offre un support d'exécution pour les architectures hiérarchiques de composants. Elle offre également un mécanisme d'extension appelé *contrôleur* : les connexions au sein d'un composant composite sont gérées par exemple par le *Binding-Controller* et son contenu par le *Content-Controller*. L'exécution des politiques d'adaptation est donc assurée par un nouveau type de contrôleur : le *Adaptation-Controller*. La figure 4 présente la conception détaillée des principaux éléments constitutifs du contrôleur d'adaptation[5]. Les classes grisées sont celles issues du module de logique floue. Ses fonctionnalités sont les suivantes:

loadAdaptationPolicy permet de charger une nouvelle politique d'adaptation. Le framework offre les outils nécessaires (Parser et Builder) pour instancier et composer des politiques d'adaptation à partir de fichiers textes. A l'aide de la syntaxe présentée dans la section 3, on peut ainsi constuire des politiques d'adaptation, les composer, et les utiliser pour adapter un composant.

[5] Les sources sont disponibles à l'adresse `http://www.irisa.fr/triskell/perso_pro/obarais/pmwiki.php?n=Research.Fuzzy`

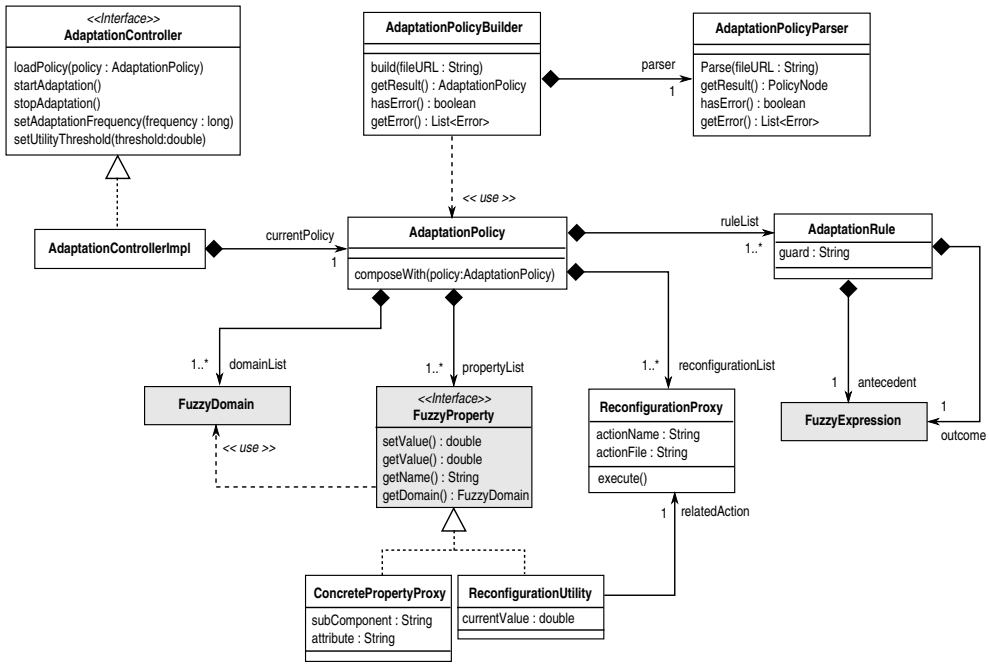


Fig. 4. Conception détaillée du contrôleur d'adaptation pour Fractal

startAdaptation permet de démarrer le processus de contrôle (évaluation des règles d'adaptation) et ainsi le processus d'adaptation. A intervalle de temps régulier, le contrôleur d'adaptation applique l'algorithme présenté dans la section précédente.

stopAdaptation permet de stopper le processus de contrôle. Le composant continu à s'exécuter, mais n'adapate plus sa configuration aux changements de son environnement.

setAdaptationFrequency permet de modifier la fréquence à laquelle l'algorithme d'adaptation est appliqué. Il s'agit d'un délai exprimé milliseconde entre deux appels succesifs à l'algorithme d'adaptation.

setUtilityThreshold permet de fixé le seuil d'utilité au delà duquel une reconfiguration architecturale est déclenchée. Ce paramètre, utilisé directement dans l'algorithme d'adaptation, influe sur la stabilité du système.

Les reconfigurations architecturales qui sont impliquées dans une politique d'adaptation doivent être décrites dans des fichiers séparés à l'aide d'actions FScript ([5]). FScript permet de décrire des procédures de reconfiguration dans les architectures Fractal. Il permet par exemple d'exprimer des connexions d'interfaces, des ajouts ou des suppressions de composants, etc. L'exemple suivant permet d'ajouter le composant cache dans l'architecture. Il s'agit du contenu du fichier *"addCache.fscript"* utilisé dans la politique d'adaptation qui gère le cache.

```
action addCache(server) =
{
  cacheComponent = new("fr.irisa.triskell.cherokee.CacheHandler.CacheHandler");
  requestHandler = $server/child::requestHandler;
  add(server, cache);
  bind($requestHandler/interface::cache, $cache/interface::cache);
  start($cache);
}
```

6 Evaluation et discussions sur l'approche

Cette section présente une première validation de l'algorithme présenté dans les sections précédentes. Cette validation est basée sur une application numérique de l'algorithme car l'utilisation du

contrôleur d'adaptation, dans une application réelle n'est pas encore complétement implémentée. Nous proposons, cependant, d'observer le comportement de l'algorithme au travers d'une simulation, c'est-à-dire dans un environement où les actions architecturales ne sont pas réellement exécutées.

Cette simulation présente le comportement de l'algorithme d'adaptation utilisé dans le contexte du serveur web. Deux politiques d'adaptation sont utilisées: l'une assure les exigences liées à la bonne utilisation du cache et a été présentée dans la section 3 alors que la seconde prend en compte les exigences liées au nombre de serveurs de données déployés. Cette seconde politique est la suivante:

```
policy DataServerManagement
is
   reconfiguration addFileServer is ‘‘addFileServer.fscript’’
   reconfiguration removeFileServer is ‘‘removeFileServer.fscript’’

   property load : AverageLoad
      evolves in [0..100] as low medium high
      sensor is getLoad on requestHandler

   when load is high and requestDeviation is high
      if count($context::child/*::interface/FileServer[server(.)]/component) < 10
      then utility of addFileServer is high

   when load is low and requestDeviation is low
      if count($context::child/*::interface/FileServer[server(.)]/component) > 1
      then utility of removeFileServer  is medium or high

end policy
```

Les résultats obtenus lors de la simulation sont présentés par la figure 5. Les deux graphiques présentent respectivement l'évolution des propriétés impactées par les deux politiques d'adaptation, à savoir, celle relative à la gestion du cache et celle relative à la gestion du nombre de serveurs de données déployés.

La simulation présentée ici décrit le comportement adaptatif de notre serveur web dans un contexte où la charge du serveur (en req/s) augmente jusqu'à une valeur élevée, puis décroit jusqu'à une valeur dite faible. Dans le même temps la dispertion des requêtes évolue plusieurs fois entre des valeurs "faible" et "elévée". La simulation montre ainsi les différentes combinaisons possibles entre la charge du serveur et la dispertion des requêtes.

Pour ce qui concerne la gestion des propriétés relatives à la gestion du cache, on peut noter que le composant cache n'est activé que lorsque la charge est élevée et que la dispertion des requêtes l'est également. De plus, la taille du cache évolue correctement en fonction de ces deux paramètres. Par exemple, entre les étapes 100 et 300 de la simulation la taille du cache augmente en fonction de la dispertion des requêtes. Il en est de même pour la durée de validité des informations qui évolue en fonction de la charge du serveur.

Le comportement de la seconde politique d'adaptation montre que l'ajout de serveurs de données est bien corrélé à l'évolution de la charge du serveur.

7 Travaux connexes

De nombreux ADLs (Architectures Description Languages) ont été décrits dans la littérature ([11]). La plupart de ces travaux ne considère les architectures logicielles que sous un angle statique. Cependant, des travaux récents tels que ([2]) soulignent l'intérêt des architectures dynamiques.

Wright et plus spécialement Dynamic Wright [1] est un autre ADL qui prend en charge les reconfigurations dynamiques. Dans ces travaux, l'objectif est de faire des vérifications sur les reconfigurations architecturales. Dans Wright, le comportement des composants est décrit à l'aide de processus communiquants, ce permet de faire des vérifications de cohérence par exemple. Cependant, Wright ne prend pas en compte les reconfigurations locales comme le permet notre approche

AADL ([15]) est l'un des premiers langages de description d'architectures à avoir pris en compte la qualité de service. AADL permet de décrire différents *modes* d'une architecture, c'est-à-dire les différentes configurations architecturales dans lesquelles un système peut évoluer. Cependant, AADL ne permet de décrire la dynamique qui régit les changements de modes.

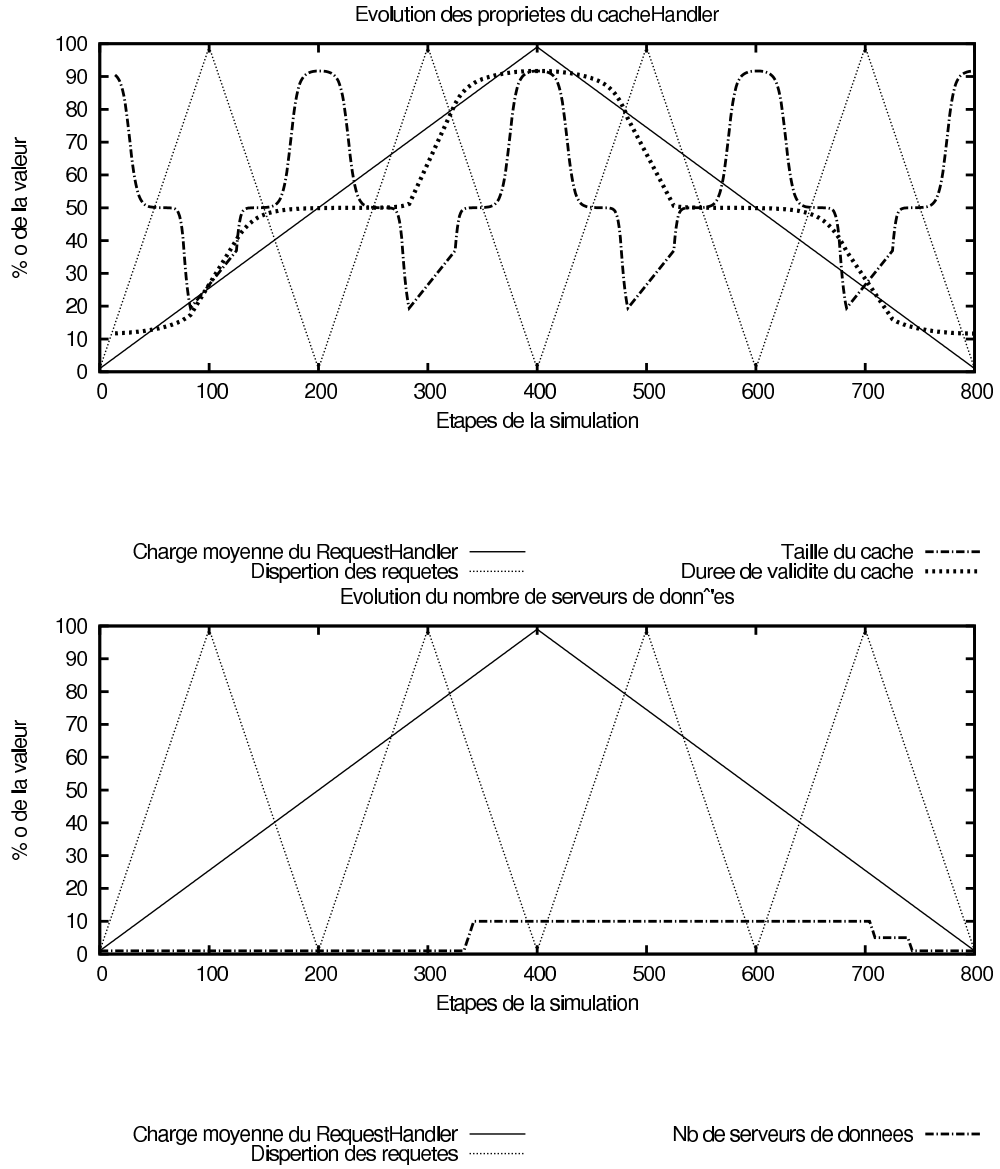


Fig. 5. Evolution des configurations locales des composants impliqués dans la réalisation du serveur web

Dans un cadre plus formel, l'outil π-ADL ([12, 16]) permet de décrire les architectures logicelles à l'aide d'une famille de langages formels. π-ADL permet de faire des vérifications de validité et de conformance sur les architectures, mais n'offre pas une syntaxe de haut niveau proche des exigences, comme le permet notre approche basée sur la logique floue.

Parmi les autres outils existants, Rainbow ([8]) présente un outil qui intégre une gestion des politiques d'adaptation. Il permet au concepteur d'application de concevoir des stratégies d'adaptation qui sont déclenchées par la violation de contraintes. Si l'objectif de Rainbow est similaire au notre, notre approche se caractérise par un niveau d'abstraction élevé dû à l'utilisation de la logique floue qui permet de conserver une description qualitative et non quantitative du contexte du système.

[7] proposent d'enrichir l'ADL de Fractal avec des règles de reconfigurations basée également sur la notion d'évenement-condition-action ou les actions modifie l'architecture et les évènements sont issus de l'environnement. Notre appoche diffère par l'utilisation de la logique floue qui permet d'exploiter une description qualitative de l'environnement.Dans la même veine, [9] propose une approche similaire à la notre dans la mesure ou les adaptations sont représentées sous la forme de règle "evénement-condition-action". Les travaux en question ici sont implémentés également sur la plate-forme Fractal, mais n'offre pas une description qualitative de l'environnement.

8 Conclusion

Dans un environnement hautement dynamique, les systèmes ont besoin de s'adapter aux évolutions de ce dernier pour pouvoir assurer un niveau de qualité maximum. Pour cela, la plupart des plates-formes d'exécution récentes offrent les mécanismes nécessaires à l'exécution d'adaptations dynamiques tels que la reflexivité ou le chargement dynamique. Cependant, ces mêmes plate-formes n'intègrent pas encore une description abstraite de politiques d'adaptations.

La contribution de cet article est de proposer un mécanisme d'exécution de politiques d'adaptation pour la plate-forme Fractal. Les politiques d'adaptation sont interprétées à l'aide d'un moteur qui prend en charge l'interprétation de règles floues. Ces règles floues, par leur imprécision, restent très proche des exigences que pourraient exprimer un concepteur de systèmes adaptatifs.

La solution que nous proposons décrit les politiques d'adaptations sous la forme d'un ensemble de règles qualitatives qui peuvent soit impacter la configuration architecturale (en terme de composants et de connecteurs) soit impacter la configuration locale d'un composant, en modifiant la valeur d'un des paramètres de sa configuration. Le moteur d'adaptation est implémenté sous la forme d'un contrôleur Fractal et peut donc être réutilisé facilement dans d'autres contexte sur la plate-forme Fratal.

Plusieurs extensions peuvent être envisagées à cette approche. La solution proposée permet une conception précoce des politiques d'adaptation. De plus, la logique floue laisse la possibilité d'influer sur plusieurs paramètres liés à l'interprétation des politiques d'adaptation, comme la définition des fonctions d'appartenance associées au vocabulaire des domaines ciblés. Nous envisageons l'utilisation de techniques issues du monde de l'intelligence artificielle pour optimiser ces différents paramètres dans la cadre de simulations de l'évolution de l'architecture et de son environnement. L'utilisation de réseaux de neurones, ou d'un algorithme génétique sont de bons moyens pour mettre au point un ensemble de règles d'adaptation particulièrement efficace dans une situation donnée.

References

1. Robert Allen, Rémi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. *Lecture Notes in Computer Science*, 1382:21–??, 1998.
2. Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33, New York, NY, USA, 2004. ACM Press.

3. 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, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.
4. G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A component model for building systems software, 2004.
5. Philippe David and Thomas Ledoux. Safe dynamic reconfigurations of fractal architectures with fscript. In *Proceeding of Fractal CBSE Workshop, ECOOP'06*, Nantes, France, 2006.
6. Pierre-Charles David and Thomas Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In Welf Löwe and Mario Südholt, editors, *Software Composition, 5th International Symposium, SC 2006, Vienna, Austria, March 25-26, 2006, Revised Papers*, volume 4089 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2006.
7. Jérémy Dubus and Philippe Merle. Vers l'auto-adaptabilité des architectures logicielles dans les environnements ouverts distribués. In Oussalah et al. [13], pages 13–29.
8. David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.
9. Hocine Grine, Thierry Delot, and Sylvain Lecomte. Adaptive query processing in mobile environment. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing*, pages 1–8, New York, NY, USA, 2005. ACM Press.
10. Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, and Dmitriy Kopylenko. *Professional Java Development with the Spring Framework*. Wrox Press Ltd., Birmingham, UK, UK, 2005.
11. N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.
12. Flavio Oquendo. Formally Describing Dynamic Software Architectures with π-ADL. *World Scientific and Engineering Transactions on Systems*, 3(8):673–679, October 2004.
13. Mourad Chabane Oussalah, Flávio Oquendo, Dalila Tamzalit, and Tahar Khammaci, editors. *1er Conférence francophone sur les Architectures Logicielles (CAL 2006), 4-6 September 2006, Nantes, France*. Hermes Science, 2006.
14. W. Pedrycz. *Fuzzy Control and Fuzzy Systems*. Wiley, New York, USA, second edition, 1993.
15. As-2 Embedded Computing Systems Committee SAE. Architecture Analysis & Design Language (AADL). SAE Standards n^o AS5506, November 2004.
16. Hervé Verjus, Sorana Cîmpan, Ilham Alloui, and Flávio Oquendo. Gestion des architectures évolutives dans archware. In Oussalah et al. [13], pages 41–57.
17. L. A. Zadeh. Fuzzy sets. In D. Dubois, H. Prade, and R. R. Yager, editors, *Readings in Fuzzy Sets for Intelligent Systems*, pages 27–64. Kaufmann, San Mateo, CA, 1993.

Session action IDM

Ingénieries Dirigées par les Modèles

Retours sur Models 2008

Xavier Blanc

INRIA Lille-Nord Europe, LIFL CNRS UMR 8022, Université des Sciences et Technologies de Lille

Abstract. La conférence Models est la conférence phare de l'IDM (Ingénierie Dirigée par les Modèles). Sa onzième édition a eu lieu en France, à Toulouse, au mois d'Octobre 2008. Cet article présente les articles Français publiés lors de cet évennement et montre la richesse et la forte implication de la communauté française dans ce domaine.

1 introduction

Models est la conférence phare de l'IDM (Ingénierie Dirigée par les modèles). Créée par les professeurs Pierre-Alain Muller et Jean Bézivin, sa première édition a été organisée en France, à Mulhouse, en 1998. En 2008, sa onzième édition a été organisée pour la deuxième fois en France, à Toulouse.

Historiquement, Models était nommée UML (de 1998 à 2004), car les contributions qui y étaient présentées ciblaient majoritairement le langage UML. Le changement de nom en 2005 a marqué le fait que les contributions scientifiques portent sur toutes les approches de modélisation quelles qu'elles soient et pas seulement sur le langage UML.

Models est une conférence internationale qui a été organisée dans les pays suivants : Fort Collins, USA (UML'99); York, UK (UML'00); Toronto, CA (UML'01); Dresden, Germany (UML'02); San Francisco, USA(UML'03); Lisbon, Portugal (UML'04); Montego Bay, Jamaique (Models'05); Genova, Italie (Models'06), NashVille USA (Models'07).

Le domaine scientifique de Models est celui du génie logiciel et plus particulièrement celui de l'ingénierie dirigée par les modèles. D'année en année, cette conférence s'est imposée comme étant la conférence incontournable de ce domaine. Models est labélisée IEEE et ACM. Elle est classée comme appartenant aux conférences de rang A selon le classement du CORE (classement australien).

271 papiers ont été soumis à Models'08 venant de 40 pays différents. Les trois premiers pays, en comptant le nombre de soumissions, sont la France (40), l'Allemagne (38) et le Canada (24). Sur ces 271 papiers, 58 ont été acceptés, ce qui fait un taux de sélection de 21%. Sur ces 58 papiers, 11 viennent d'équipes françaises. Nous proposons ici de présenter brièvement ces papiers afin de montrer la forte présence de la communauté française dans ce domaine.

2 Les 11 papiers français

2.1 Visualization of Use Cases through Automatically Generated Activity [4]

Dans cet article, les auteurs proposent une génération automatique de diagrammes d'activité UML2 à partir de spécifications détaillées de cas d'utilisation. En effet, les cas d'utilisation sont de plus en plus complétés par des fiches détaillées qui sont exprimées textuellement. Ces descriptions textuelles se doivent de respecter une structuration forte, spécifiée par un template de fiche détaillée de cas d'utilisation. Plusieurs approches proposent de tels templates et sont d'ailleurs référencées dans l'article. Cette structuration forte permet une meilleure précision de la sémantique des fiches détaillées et donc des cas d'utilisation.

En proposant une transformation automatique de ces fiches détaillées vers des diagrammes d'activité, les auteurs répondent au problème de l'exploitation automatique des fiches. En effet, une fois transformées en diagrammes d'activité, il est alors possible d'intégrer les fiches détaillées dans le modèle UML et de les associer directement avec les cas d'utilisation. De plus, la transformation en diagrammes d'activité favorise la lisibilité des fiches.

La transformation proposée par les auteurs prend en entrée une formalisation XML des fiches détaillées et retourne les diagrammes d'activité correspondants. La formalisation XML des fiches détaillées est une des contributions des auteurs et est une généralisation de plusieurs approches de template de fiches détaillées. La transformation est réalisée en Q/V/T et est validée sur un cas d'utilisation fourni par un organisme publique espagnol.

2.2 Behavoiral Modelling and Composition of Object Slices Using Event [10]

Dans cet article, les auteurs adressent le problème bien connu de la séparation des préoccupations en proposant une approche de découpe par tranches (*slice* en anglais). L'objectif étant de découper les cas d'utilisation en tranches, puis de fournir différentes tranches de classes couvrant la découpe des cas d'utilisation et, pour finir, de composer les différentes tranches de classes pour fournir l'application finale.

La particularité de l'approche proposée par les auteurs est de supporter une découpe et une composition des comportements des différentes tranches. La proposition s'appuie sur une communication par évenement pour assurer les interactions entre les différentes tranches de comportement et ainsi assurer une cohérence comportementale de l'assemblage.

Dans cet article, les auteurs fournissent une spécification formelle de leur approche en s'appuyant sur la sémantique des LTS (labeled transition system).

2.3 Metamodel Matching for Automatic Model Transformation Generation [3]

Dans cet article[1] , les auteurs proposent une approche pour développer des transformations de modèles automatiques entre deux méta-modèles. La génération de la transformation s'éffectue en plusieurs étapes :

1. la génération de graphes étiquetés associés à chaque méta-modèle,
2. l'utilisation de l'algorithme Similarity Flooding sur les graphes produits pour faciliter la recherche d'une correspondance,
3. la génération d'un alignement entre les deux méta-modèles.

Lors de la première étape, selon la configuration employée, le graphe produit est différent, ce qui influe énormément sur les résultats de lalgorithme Similarity Flooding. Les auteurs ont présenté une approche automatique de détermination de correspondance entre deux méta-modèles à travers l'utilisation d'un algorithme de recherche. La principale partie du travail s'est effectuée sur la détermination du meilleur passage du méta-modèle vers un graphe étiqueté. Ils ont déterminé en tout six configurations différentes donnant des résultats plus ou moins probants pour les correspondances.

2.4 Implementation of the Conformance Relation for Incremental Development of Behavioural Models [5]

Dans cet article, les auteurs répondent au problème de la vérification incrémentale de la relation de conformité entre systèmes de transition étiquetés et son application aux modèles UML. L'objectif étant de vérifier cette relation sur des machines à états UML2.

En effet, l'élaboration de machines à états UML2 est un processus incrémental complexe dans lequel, à chaque étape du processus, il est important de vérifier la conformité de la machine à états en cours d'élaboration avec les machines à états précédement construites. A chaque étape, il faut contrôler les modifications faites sur les machines à états afin de ne pas effectuer des modifications introduisant des incohérences comportementales.

En ce basant sur la sémantique des LTS (Labeled Transition System), les auteurs fournissent une preuve de leur approche. A l'aide de leur prototype, construit en Java, leur approche est illustrée sur un exemple de modèle UML qui spécifie les comportements d'un téléphone (les modes simple appel et double appel sont alors considérés).

[1] Le résumé de cet article est fourni par M. Acher, de l'Université de Nice Sophia-Antipolis, et V. Aranega, du Laboratoire d'Informatique Fondamentale de Lille, qui ont assisté à la conférence Models grâce au soutien de l'action IDM du CNRS

2.5 A Model-Based Framework for Statically and Dynamically Checking Component Interactions [11]

Dans cet article, les auteurs présentent le framework CALICO qui permet de modéliser une application selon le paradigme composant et qui permet d'effectuer des vérifications aussi bien statiques que dynamiques.

Le framework CALICO permet à ses utilisateurs de modéliser leurs applications en suivant une approche itérative en 6 étapes. La première étape consiste en l'élaboration des modèles décrivant l'application (CALICO offre 4 points de vue). La deuxième étape consiste en la vérification de propriétés statiques. La troisième étape consiste au déploiement de l'application sur un serveur d'applications. La quatrième étape récupère toutes les informations nécessaires aux vérifications dynamiques pendant l'exécution du système. La cinquième étape consiste à effectuer les vérifications dynamiques. La sixième et dernière étape consiste à effectuer les modifications nécessaires du système en réponse aux différentes vérifications effectuées. Ces modifications sont faites directement sur les modèles du système en suivant l'approche model-driven.

2.6 A Model-Driven Measurement Approach [6]

Dans cet article, les auteurs proposent une approche pour construire des systèmes de mesure adaptés à des domaines spécifiques. L'approche permet à un utilisateur d'élaborer le modèle des mesures qu'il souhaite réaliser et de générer automatiquement le système capable d'effectuer de telles mesures. Pour ce faire, les auteurs proposent un méta-modèle de mesure ainsi qu'un générateur de systèmes de mesure.

Les auteurs ont utilisé et validé leur approche pour générer un système de mesures pour les programmes Java (en utilsant les métriques classiques des programmes Java) et un système de mesures adapté aux systèmes de surveillance maritime. Pour chacun des deux cas, les modèles des mesures à réaliser ont été élaborés. Ces modèles ont permi la génération automatique des deux systèmes de mesures.

2.7 A Model-Based Framework for Security Policy Specification, Deployment and Testing [9]

Dans cet article, les auteurs proposent une approche guidée par les modèles pour spécifier, déployer et tester les politiques de sécurité. De telles politiques, même si elles suivent les mêmes paradigmes (filtrage d'accès par rôle par exemple), sont bien souvent spécifiées dans des fichiers propriétaires aux plates-formes d'exécution. Ces formats propriétaires sont d'une part un frein à l'indépendance des applications par rapport à leur plate-forme d'exécution mais d'autre part un véritable obstacle à la génération de tests de sécurité.

L'approche proposée par les auteurs est de modéliser les politiques de sécurité d'une façon indépendante des plates-formes d'exécution. Pour ce faire, les auteurs proposent (1) un méta-modèle de politiques de sécurité complétement

indépendant des plates-formes d'exécution; (2) un méta-modèle de politiques de sécurité par plate-forme d'exécution et (3) les transformations de modèles nécessaires.

A partir des modèles de politiques de sécurité indépendants des plates-formes d'exécution, les auteurs proposent une approche de génération de tests de sécurité par mutation. Cette génération de tests renforce l'intérêt de disposer d'un modèle de politique de sécurité indépendant des plates-formes d'exécution.

2.8 Autonomic Management Policy Specification: From UML to DSML [1]

La conception[2] de systèmes autonomiques est une voie prometteuse pour maîtriser l'augmentation (e.g en coût et en temps) des tâches de gestion. Le cas d'utilisation présenté concerne une application J2EE, qui nécessite d'éditer et de configurer de nombreux fichiers manuellement. Dans ce contexte, il s'agit par exemple de redémarrer automatiquement un server tomcat suite à une erreur ou bien de répliquer des serveurs suite à une charge trop importante. Chaque serveur à gérer est encapsulé dans un composant : l'environnement logiciel est une architecture à composants. L'objectif des auteurs est de faciliter l'implémentation d'encapsulateurs (wrappers) de programmes existants (par exemple pour surveiller le programme), de décrire le déploiement des programmes et enfin de permettre leurs reconfigurations autonomiques.

La première approche a consisté à utiliser le modèle de composants Fractal. Cependant, plusieurs inconvénients ont été rapportés : apprentissage dun nouveau framework (encore un!), écriture des wrappers demandant beaucoup de travail et de temps processus d'ailleurs sujet à des erreurs et difficulté d'exprimer des politiques de reconfiguration. Aussi, les auteurs proposent une adaptation (i.e spécialisation) de la sémantique d'UML à même de fournir des formalismes de haut niveau pour les besoins de leur application. Ensuite, plusieurs DSLs ont été associés pour gérer l'encapsulation, le déploiement et la reconfiguration des composants. La syntaxe concrète et les outils des DSLs ont également été développés. Il est maintenant possible pour un utilisateur final de spécifier des politiques de reconfiguration en utilisant le DSL de reconfiguration des composants.

Les auteurs concluent en détaillant les bénéfices de l'approche (e.g renforcement de la cohérence des modèles, vues de haut niveau, etc.). Les travaux présentés ouvrent de nouvelles perspectives quant à la gestion dirigée par les modèles des systèmes informatiques.

2.9 An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability [7]

Dans cet article, les auteurs proposent une approche permettant de faire face à l'explosion combinatoire des configurations possibles de variation de comportements dans les systèmes adaptables. En effet, un système adaptable est composé

[2] Le résumé de cet article est aussi fourni par M. Acher et V. Aranega

d'un comportement principal et de plusieurs adaptations. Résoudre la composition de ces adaptations dans le comportement principal est un exercice difficile dont la cause est le nombre exponentiel de compositions possibles.

Les auteurs proposent d'utiliser conjointement l'approche par aspect et l'approche guidée par les modèles. Leur proposition résulte en un framework permettant de modéliser les comportements ainsi que les variations et d'utiliser le tissage d'aspects pour identifier les combinaisons possibles des variations dans le comportement principal.

2.10 Managing Variability Complexity in Aspect-Oriented Modeling [8]

Dans cet article, les auteurs adressent le problème de la gestion de la variabilité dans l'approche de modélisation par aspects. En effet, dans cette approche, les différentes variations d'une application sont spécifiées par des modèles. La difficulté devient alors de savoir comment intégrer ces variations dans le comportement principal de l'application et de vérifier si l'intégration d'une variation n'a pas d'impact sur une autre variation.

Pour faire face à ce problème, les auteurs proposent d'ajouter dans les modèles de variation les informations nécessaires à la vérification de l'intégration. La proposition faite par les auteurs permet d'une part de vérifier statiquement l'intégration de plusieurs variations dans le comportement principal d'une application mais aussi d'autre part de fournir le support nécessaire à la réalisation de tests d'intégration.

2.11 Meaningful Composite Structures: On the Semantics of Ports in UML2 [2]

Dans cet article, les auteurs présentent les différentes sémantiques de la propagation des requêtes lors d'assemblage de composants UML2. En effet, en UML2 les composants peuvent être assemblés par leurs ports et il est possible de préciser le traitement réalisé par les ports lors de la propagation des requêtes. Pour autant, le standard UML2 n'est pas très précis sur la sémantique de cette propagation des requêtes ce qui peut être source de problème lourd lors de vérification ou de génératon de code.

3 Synthèse

Les 11 articles que nous venons de présenter sont le reflet de la forte implication qu'a la communauté française dans ce domaine de recherche qu'est l'ingénierie logicielle dirigée par les modèles. Cette forte implication se retrouve dans l'action IDM du CNRS qui rassemble toutes les équipes françaises participant à ce domaine.

La conférence Models'08 a aussi permi de montrer la forte implication du tissus industriel français sur ce domaine (Airbus, CEA, CNES, CS, BluAge,

MIA Software ont sponsorisé la conférence). L'implication des industriels montre d'une part l'importance qu'a ce domaine à leurs yeux mais aussi la crédibilité porté aux académiques travaillant sur ce domaine.

Pour finir, 10 ans après la première édition de cette conférence, la richesse et la diversité des workshops, des keynotes et des tutoriels de sa onzième édition montrent la vitalité de cette communauté.

References

1. B. Combemale, L. Broto, X. Crégut, M. J. Daydé, and D. Hagimont. Autonomic management policy specification: From uml to dsml. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 584–599, 2008.
2. A. Cuccuru, S. Gérard, and A. Radermacher. Meaningful composite structures. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 828–842, 2008.
3. J.-R. Falleri, M. Huchard, M. Lafourcade, and C. Nebut. Metamodel matching for automatic model transformation generation. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 326–340, 2008.
4. J. J. Gutiérrez, C. Nebut, M. J. E. Cuaresma, M. Mejías, and I. M. Ramos. Visualization of use cases through automatically generated activity diagrams. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 83–96, 2008.
5. H.-V. Luong, T. Lambolais, and A.-L. Courbis. Implementation of the conformance relation for incremental development of behavioural models. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 356–370, 2008.
6. M. Monperrus, J.-M. Jézéquel, J. Champeau, and B. Hoeltzener. A model-driven measurement approach. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 505–519, 2008.
7. B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, and G. S. Blair. An aspect-oriented and model-driven approach for managing dynamic variability. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 782–796, 2008.
8. B. Morin, G. Vanwormhoudt, P. Lahire, A. Gaignard, O. Barais, and J.-M. Jézéquel. Managing variability complexity in aspect-oriented modeling. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 797–812, 2008.
9. T. Mouelhi, F. Fleurey, B. Baudry, and Y. L. Traon. A model-based framework for security policy specification, deployment and testing. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 537–552, 2008.
10. I. Ober, B. Coulette, and Y. Lakhrissi. Behavioral modelling and composition of object slices using event observation. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 219–233, 2008.

11. G. Waignier, P. Sriplakich, A.-F. L. Meur, and L. Duchien. A model-based framework for statically and dynamically checking component interactions. In *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3*, pages 371–385, 2008.

Des plate-formes pour l'IDM: « OPEES, OpenEmbeDD, Papyrus, TOPCASED... »

Sébastien Gérard

CEA LIST
Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués (LISE)
Boîte courrier 65, Gif sur Yvette Cedex, F-91191 France.
Sebastien.Gerard@cea.fr

Résumé: L'objectif de cet article court est, d'une part, de rapidement introduire trois initiatives françaises liées à des développements d'outils ou suites d'outils visant à fournir un support à l'ingénierie des modèles, et, d'autre part, de présenter deux initiatives de fédération de ces outils et de leur développement dans un cadre européen et international, au travers de travaux menés avec la fondation Eclipse.

Mots-clés: Eclipse, ingénierie dirigée par les modèles, IDM, modèle, transformation de modèle, open-source, UML.

1 Introduction

L'une des caractéristiques majeures de l'ingénierie dirigée par les modèles (IDM) est la place centrale donnée aux modèles dans le processus de développement d'applications. Dans ce contexte, il est d'usage de dire que le modèle passe d'un état contemplatif à un état productif. Dans le premier cas, il est fait allusion au fait que, usuellement, le modèle occupe une place passive dans les processus de développement, c'est-à-dire. que les modèles ne sont généralement utilisés qu'à des fins informatives dans des documents de spécification ou de conception sans réelle valeur ajoutée sur le développement même des applications. Dans le cadre de l'ingénierie dirigée par les modèles, le modèle est devenu productif, c'est-à-dire. qu'il est concrètement utilisé pour produire des applications. Pour se faire, il est nécessaire de disposer d'un ensemble d'outils permettant de couvrir les deux catégories de besoins suivantes:

- construire les modèles;
- rendre productif les modèles.

Dans ce contexte, au cours des dernières années, plusieurs initiatives nationales ont vu le jour afin de développer en open-source (c'est à dire diffusé, utilisé et modifié librement) des outils, et même des suites d'outils, permettant la construction des modèles et/ou leur mise en production. La suite de cet article dresse les contours de quelques unes de ces principales initiatives, telles que OpenEmbeDD [1], Papyrus [2] et TOPCASED [3]. Ensuite, avant de conclure, l'article présente deux initiatives dîtes

de convergence visant à fédérer les efforts des partenaires français dans le domaine de l'outillage des approches de type IDM et conforter ainsi leur impact au niveau européen et mondiale au travers d'actions communes au sein de la fondation Eclipse.

2 Trois Initiatives Françaises Open-source sous Eclipse

Dans le contexte de l'ingénierie des modèles, et au niveau national, on peut noter particulièrement trois initiatives open-source représentatives de la forte implication des acteurs français, tant académiques que industriels, dans le domaine de l'ingénierie des modèles :

- OpenEmbeDD [1] est un projet de plateforme RNLT qui propose, sur une base Eclipse, trois volets. Le premier volet propose un ensemble d'outils offrants des services de base pour l'ingénierie des modèles permettant d'éditer et de manipuler des modèles et des métamodèles. Les second et troisième volets sont respectivement dédiés à la fourniture de solutions implantant la norme de l'OMG pour le temps-réel embarqué, à savoir MARTE [8], et à la fourniture d'outils permettant d'accomplir des vérifications et des validations formelles dans le cadre d'une ingénierie des modèles [1].
- Papyrus [2] est un projet Open-source visant à fournir un modeleur graphique Eclipse pour UML2 complet et méthodologiquement agnostique. Techniquement, Papyrus s'appuie sur les deux frameworks UML2 [10] et GEF (« Graphical Editing Framework », [9]) fournis par Eclipse. Le framework UML2 propose une implantation du métamodèle de UML2 basé sur le framework de méta-modélisation EMF (« Eclipse Modeling Framework », [11]). GEF fournit lui des facilités pour construire plus rapidement des éditeurs graphiques intégrés à Eclipse. L'ingénierie dirigée par les modèles demande de répondre au plus près aux besoins spécifiques de chacun des domaines techniques ou métiers pour lesquels on veux mettre en œuvre ses paradigmes. Ainsi, être capable de manipuler un langage le plus adapté, le plus proche, que possible aux spécificités des domaines visés devient une préoccupation de première ordre. Cette capacité d'adaptation, le langage de modélisation UML2 la supporte au travers d'une approche dite de « métamodélisation légère », le profil UML2. Cet aspect est l'un des chevaux de bataille de Papyrus et l'une des caractéristiques qui entre autre le différencie des autres modeleurs UML2 existants, qu'ils soient open-sources ou commerciaux. Papyrus propose ainsi une implantation complète du support des profils UML tant pour leur définition, que pour leur application.
- TOPCASED (« Toolkit in Open-source for Critical Applications & SystEms Development ») [3] est un environnement logiciel principalement dédié à la réalisation de systèmes embarqués critiques, et prenant en compte à la fois les aspects matériels et logiciels. Il s'agit également d'un projet en « open source ». L'objectif du projet est de mettre en place des outils pouvant être utilisés pendant au moins 30 ans (durée de vie d'un avion) avec un coût de maintenance minimum, en ayant la possibilité d'intégrer au fur et à mesure

les nouveautés scientifiques et techniques, et en tenant compte des contraintes liées à la certification aéronautique. En pratique, TOPCASED vise à fournir des éditeurs de modèles pour le développement des systèmes depuis leur spécification jusqu'à la réalisation des matériels et logiciels, ainsi que des outils pour la vérification formelle, la production de modèles pour simulation, la génération de codes, de cas-test et de documentation, tous ces outils devant être de haute qualité. Dans le cadre du projet TOPCASED, il s'agit également de définir un ou plusieurs modèles économiques permettant une exploitation industrielle de l'atelier (propriété intellectuelle, licences, services, etc.). Il faut noter que du point de vue des éditeurs, l'un des langages supportés est UML et son extension pour l'ingénierie système, SysML). L'atelier contient donc ses propres éditeurs pour UML2 et SysML. Comme on le verra plus tard, ce dernier point est le sujet d'une action de convergence entre les deux projets TOPCASED et Papyrus.

3 Vers une Convergence des Initiatives

Dans le cadre de l'ingénierie dirigée par les modèles, il est un langage de modélisation qui joue un rôle prépondérant : UML, langage de modélisation unifié normé par l'OMG. Un outillage de qualité et disponible en Open-source pour ce langage de modélisation, considéré maintenant par l'ensemble de la communauté comme incontournable, serait donc un atout essentiel pour l'ingénierie des modèles. Or , au début de l'année 2008, trois des principales communautés développant un outil Open-source, basé sur Eclipse, et permettant la modélisation graphique pour UML2, ont décidé de rassembler leurs efforts pour construire une solution commune plus robuste et plus performante, visant un niveau de qualité et de finition de nature industrielle : MOSKitt [6], Papyrus [2] et TOPCASED [3]. Le premier résultat de cette convergence d'efforts a été une proposition commune d'un nouveau composant pour le projet de modélisation de Eclipse : MDT [7]. Ce nouveau composant, appelé Papyrus (http://wiki.eclipse.org/MDT/Papyrus-Proposal), a été accepté par la fondation Eclipse dans le courant de l'été 2008, et la première fourniture de code a eu lieu en novembre 2008. Au moment de l'écriture de cet article, une nouvelle version de l'outil Papyrus est en cours de développement et devrait être disponible progressivement au cours de cette année 2009 avec un jalon important situé au cours de l'été 2009 correspondant à la nouvelle mise à jour officielle de la plateforme Eclipse. Cette version devrait supporter l'ensemble des 13 diagrammes proposés par UML2, ainsi qu'un support complet du concept de profil UML.

Enfin, il faut noter une autre initiative de projet important dans ce contexte : OPEES pour « Open Platform for the Engineering of Embedded Systems ». OPEES est une proposition de projet européen, inscrite au programme européen Eurêka-ITEA2, et dont la mission est « de construire une communauté capable de pérenniser des technologies d'ingénieries innovatrices dans le domaine des systèmes embarqués à prépondérance logicielle ». Pour cela, OPEES se fixe les objectifs majeurs suivants. Le premier objectif est de mettre en place et pérenniser un écosystème accompagné d'un ou plusieurs modèles économiques efficaces et pertinents. Cela permettra d'une

part d'assurer un développement durable de l'industrie du logiciel embarqué et d'autre part d'aligner les orientations de la plateforme OPEES avec les stratégies industrielles du domaine. Ensuite, le projet vise également à définir et évaluer des méthodes, des processus et des outils afin de sécuriser la disponibilité des composants de la plateforme OPEES. Enfin, dans le cadre de ARTEMIS/EICOSE, OPEES sera un fédérateur et un catalyseur de projets dans le domaine du développement des systèmes embarqués à prépondérance logicielle. OPEES s'affiche ainsi comme une structure qui permettra de pérenniser les développements réalisés auparavant dans le cadre des projets OpenEmbeDD, TOPCASED et Papyrus. En effet, la structure OPEES servira, entre autre, de structure d'accueil et de maturation d'un certain nombre de composants développés dans le cadre de ces projets. Par exemple, Papyrus et ses extensions pour le domaine du temps-réel embarqué, à l'instar de son implémentation de la norme MARTE, feront partis des premiers composants majeurs de cette plateforme d'outils open-source.

4 Conclusions

OpenEmbeDD, Papyrus et TOPCASED sont autant d'initiatives open-sources nationales qui montrent le dynamisme de la communauté académique et industrielle française dans le domaine de l'ingénierie des modèles et en particulier dans son application dans le domaine des systèmes temps-réel embarqués.

Dans ce contexte, il faut noter l'effort de chacun à créer des synergies entre les projets et même des actions de convergence. En particulier, on rappellera l'action commune entre les deux communautés Papyrus et TOPCASED pour créer un projet Eclipse dédié à l'éditeur UML/SysML. Enfin, il faut également noter la structure pérenne appelée OPEES (en cours de préparation dans le cadre du programme européen Eurêka-ITEA2) qui aura pour vocation d'industrialiser entre autres l'atelier TOPCASED et ses composants tel que Papyrus pour UML/SysML.

Références

1. https://openembedd.inria.fr/
2. http://www.topcased.org
3. http://www.papyrusuml.org
4. http://www.eclipse.org/modeling/mdt
5. http://wiki.eclipse.org/MDT/Papyrus.
6. http://www.moskitt.org/eng/
7. OMG: UML Version v2.1.2, formal/07-02-05, http://www.omg.org/spec/UML/2.1.2/
8. OMG: UML Profile for MARTE, Beta 1, ptc/07-08-04, http://www.omg.org/cgi-bin/doc?ptc/2007-08-04.
9. http://www.eclipse.org/gef/
10. http://www.eclipse.org/uml2
11. http://www.eclipse.org/modeling/emf/

Alignement de métamodèles pour la génération automatique de transformation de modèles*

J.-R. Falleri[1], M. Huchard[1], M. Lafourcade[1], and C. Nebut[1]

LIRMM, CNRS et Université de Montpellier 2,
161, rue Ada, 34392 Montpellier cedex 5, France
{falleri, huchard, lafourcade, nebut}@lirmm.fr

Résumé L'application d'une approche dirigée par les modèles entraîne la création de nombreux métamodèles, dans la mesure où l'IDM prône l'usage intensif de modèles définis par des métamodèles. Des métamodèles avec des objectifs similaires sont donc inévitablement créés. Un problème récurrent est alors de rendre compatibles des modèles dont les métamodèles sont similaires, par exemple en vue de les utiliser au sein d'un même outil. Classiquement, ce problème est résolu par le développement de transformations de modèles ad hoc. Dans ce papier, nous proposons une approche détectant automatiquement les correspondances entre deux métamodèles, qui sont ensuite utilisées pour générer un alignement entre les métamodèles. Cet alignement doit ensuite être vérifié manuellement et peut ensuite être utilisé pour générer une transformation de modèle. Notre approche est basée sur l'algorithme *similarity flooding*, utilisé dans les domaines de la mise en correspondance de schémas et de l'alignement d'ontologies. Ce papier fournit également des résultats expérimentaux comparant l'efficacité de plusieurs implémentations de notre approche sur plusieurs métamodèles du monde réel.

1 Introduction

Le succès de l'ingénierie dirigée par les modèles mène à une relative abondance de métamodèles, pour différents domaines, et venant généralement avec des outils permettant de manipuler les modèles, e.g. générateurs de code ou éditeurs graphiques. Fatalement, plusieurs métamodèles poursuivant le même objectif sont créés. On peut raisonnablement souhaiter pouvoir utiliser la batterie d'outils dédiés à un métamodèle pour un modèle dont le métamodèle a un objectif similaire. Pour cela, on transforme en général le modèle par une transformation ad hoc, de manière à le rendre conforme au métamodèle pour lequel on dispose des outils. L'écriture de telles transformations, bien que peu compliquée, peut représenter une lourde et fastidieuse charge de travail. Nous proposons donc d'assister la génération de telles transformations via la génération d'alignement entre métamodèles : l'alignement représente les correspondances entre les éléments du métamodèle source et ceux du métamodèle cible. La mise en correspondance d'éléments est bien connue dans beaucoup de domaines comme l'intégration de schémas ou d'ontologies [1,2]. Elle prend en entrée deux schémas et produit un ensemble de relations (*e.g.*, équivalence et subsomption) entre les entités des schémas d'entrée. Ces relations sont en général appelées alignement entre les schémas. Dans ce papier, nous rapportons des expériences évaluant et adaptant l'utilisation de l'algorithme Similarity Flooding (SF) [3] pour calculer l'alignement entre un métamodèle MM_{source} vers un métamodèle MM_{cible}, dans le contexte de la génération automatique de la transformation d'un modèle conforme à MM_{source} vers un modèle conforme à MM_{cible}. Nous avons choisi l'algorithme SF car tout d'abord il prend en entrée des graphes étiquetés et dirigés représentant les schémas à aligner, et que la conversion entre métamodèles et de tels graphes est simple, et ensuite car sa conception générique permet de l'adapter facilement pour produire de bons résultats en se basant sur le schéma d'entrée. Le point d'entrée de l'algorithme SF étant deux graphes dirigés et étiquetés, une part de notre contribution réside en la proposition et l'évaluation de plusieurs manières d'encoder les métamodèles en de tels graphes. Ces graphes sont utilisés par l'algorithme SF pour un calcul de point fixe dont le résultat indique quels nœuds d'un graphe sont similaires aux nœuds

* France Télécom R&D a soutenu ce travail (CPRE 5326).

du second. Ce calcul est basé sur l'intuition que si un nœud est similaire à un autre nœud, ses voisins doivent être similaires à ceux de l'autre nœud.

L'approche que nous proposons peut se décomposer en trois étapes :

1. Transformation de MM_{source} et MM_{cible} en graphes dirigés et étiquetés G_{source} and G_{cible}. Plusieurs stratégies d'encodage en graphes sont proposées.
2. Application de l'algorithme Similarity Flooding.
3. Génération d'un alignement entre MM_{source} et MM_{cible}.

Nous illustrons ces trois étapes en utilisant les deux métamodèles $exMM_{source}$ et $exMM_{cible}$ de la figure 1. Ces deux métamodèles sont utilisés tous les deux pour représenter des modèles de classes, mais les noms des éléments et leur structure sont un peu différents (l'un est orienté Java et l'autre UML).

2 Des métamodèles aux graphes dirigés étiquetés

Passer des métamodèles aux graphes dirigés étiquetés est une étape cruciale : c'est elle qui extrait des métamodèles les informations qui seront utilisées pour la mise en correspondance, puis qui structure ces informations. La structure obtenue est importante puisque l'algorithme SF construit l'alignement en se basant sur la structure des graphes d'entrée. Le choix des éléments dans le métamodèle et leur structuration en graphe est par la suite nommé configuration.

Noux avons conçu six configurations qui explorent les effets des éléments dérivés (qui peuvent apporter de la redondance et augmenter la complexité de l'algorithme SF), des classes abstraites, et de la clôture transitive des relations basée sur la relation d'héritage. Nous présentons ici ces six configurations. Notre approche utilise Ecore comme méta-métamodèle donc les configurations référencent des éléments Ecore (*e.g.*, `EClass`, `EReference`).

La première configuration est appelée **Minimal** : c'est la plus simple et la plus intuitive. Elle utilise un petit sous-ensemble d'éléments inclus dans le métamodèle, et n'utilise aucun méta-attribut des méta-classes. Un nœud étiqueté est créé pour chaque `EClass`, chaque `EAttribute` non dérivé, chaque `EReference` non-dérivée, chaque `EDataType`, chaque `EEnum` et chaque `EEnumLiteral` contenus dans le métamodèle. Les étiquettes de ces nœuds sont les noms des éléments correspondants. Un arc dirigé et étiqueté *supertype* est créé entre deux nœuds qui représentent une `EClass` quand ces deux `EClasses` entretiennent une relation d'héritage. Un arc étiqueté *own* est créé quand une `EClass` possède un `EAttribute`. Un arc étiqueté *ref* est créé quand une `EClass` possède une `EReference`. Un arc étiqueté *type* est créé quand une `EReference` est typée par une `EClass`. Un arc étiqueté *datatype* est créé quand un `EAttribute` est typé par un `EDataType` ou un `EEnum`. Un

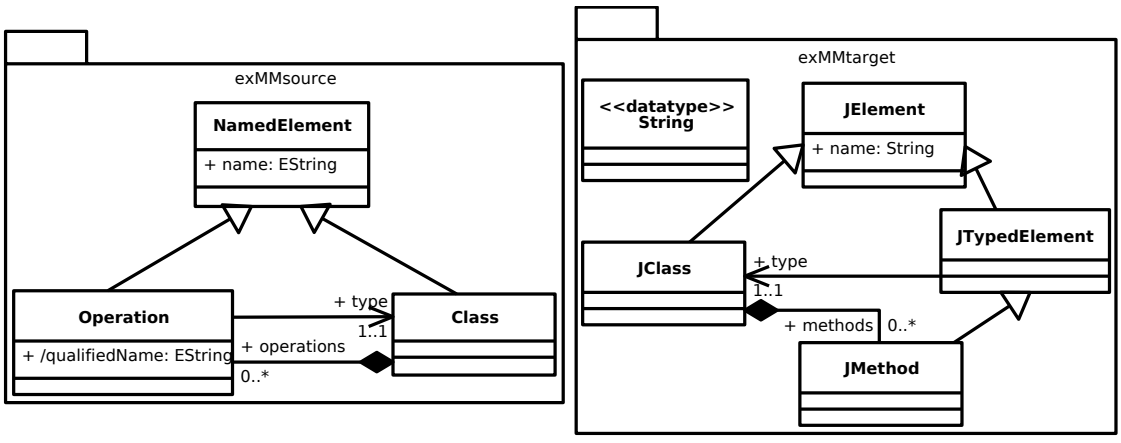


Fig. 1. Métamodèles exemples $exMM_{source}$ (gauche) et $exMM_{cible}$ (droite)

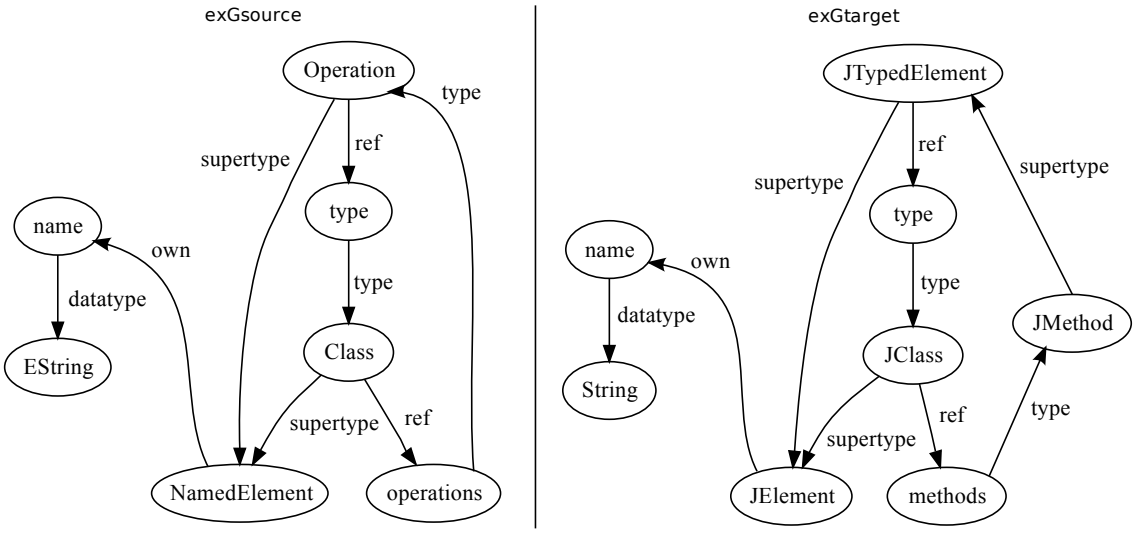


Fig. 2. Graphes générés avec les métamodèles exemple et la configuration Minimal

arc étiqueté *literal* est créé quand un `EEnum` possède un `EEnumLiteral`. La figure 2 montre les graphes exG_{source} and exG_{cible}, correspondant respectivement aux métamodèles $exMM_{source}$ et $exMM_{cible}$, générés avec cette configuration.

La deuxième configuration, appelée *Basic* diffère de la configuration *Minimal* dans la façon dont elle représente les noms des éléments. Avec la configuration *Basic*, un élément E nommé n est représenté par un nœud N étiqueté par un identifiant unique $\#ID$, et lié à un nœud étiqueté n par un arc *label*. La figure 3 montre le graphe exG_{source} généré avec cette configuration. Cette configuration permet d'avoir directement l'information sur la fréquence à laquelle un nom est utilisé (juste en comptant le nombre d'arcs *label* entrant dans un nœud donné). Cette fréquence peut être exploitée par l'algorithme SF.

La troisième configuration, *Standard*, étend la configuration *Basic* pour obtenir des similarités en inspectant non seulement les noms des éléments mais aussi leurs types et principaux attributs (`abstract` pour une `EClass`, `lowerBound` et `upperBound` pour un `EAttribute` et `EReference`, et `containment` pour une `EReference`). Dans cette optique, un nœud représentant un élément E est lié par un arc étiqueté *kind* à un nœud N représentant le type de l'élément E. N est étiqueté selon le type de l'élément E, en enlevant le préfixe "E" pour tous les éléments d'`Ecore`, et en ajoutant

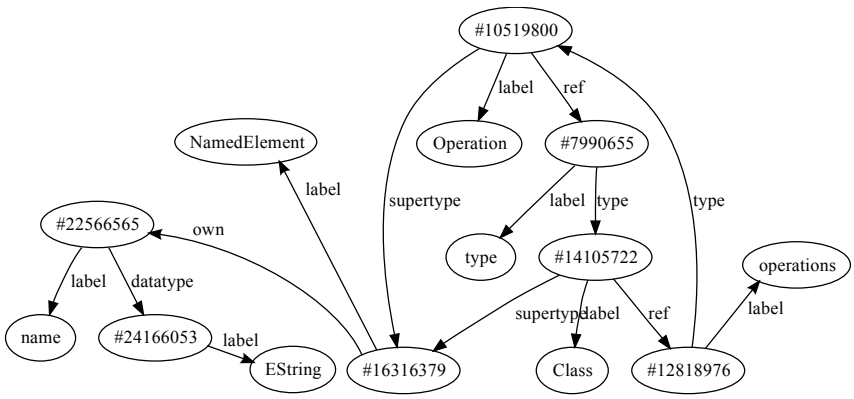


Fig. 3. Graphe exG_{source} correspondant à $exMM_{source}$ en utilisant la configuration Basic (les noms des éléments sont dans un nœud dédié)

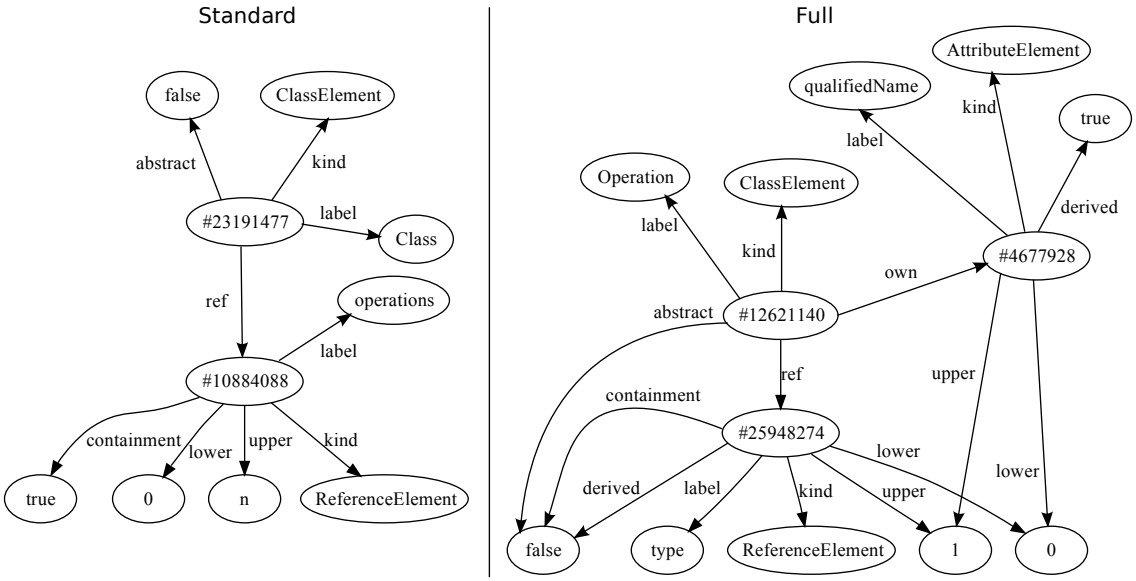


Fig. 4. Extraits du graphe exG_{source} généré en utilisant les configurations Standard et Full

le suffixe "*Element*" (e.g. *EClass* est transformé en *ClassElement*). Pour traiter les principaux attributs, quand un élément E a pour type une méta-classe avec un attribut A, alors le nœud correspondant à E est lié avec un arc étiqueté A à un nœud dont le label est la valeur de l'attribut A pour E. La partie gauche de la figure 4 montre un extrait du graphe exG_{source} généré en utilisant la configuration Standard.

La configuration *Full* étend la configuration Standard en prenant en compte les `EAttributes` et les `EReferences` dérivés. Tous les nœuds représentant un `EAttribute` ou une `EReference` se voient ajouter un arc étiqueté *derived* et menant à un nœud étiqueté *true* ou *false* selon que l'élément est dérivé ou pas. La partie droite de la figure 4 montre un extrait du graphe exG_{source} généré en utilisant cette configuration.

La configuration *Flattened* est basée sur la configuration Standard, mais avec un héritage aplati. Les nœuds représentant les `EClasses` abstraites et les arcs étiquetés *supertype* sont supprimés. À la place, des arcs étiquetés *own* (resp. *ref*) connectent les nœuds représentant une `EClass` *Ecl* à des nœuds représentant le `EAttribute` (resp. la `EReference`) défini par *Ecl* et toutes ses superclasses. De plus, quand une `EReference` *Eref* est typée par une `EClass` abstraite *Ecl*, un arc *type* est créé du nœud correspondant à *Eref* vers chaque sous-classe non abstraite de *Ecl*.

La dernière configuration est appelée **Saturated** et est encore basée sur la Standard. Ici, les relations transitives sont saturées, ainsi que le propose la référence [4]. Les nœuds `EClass` sont maintenant connectés par un arc *supertype* aux nœuds représentant toutes les superclasses de cette `EClass`. Les nœuds `EClass` sont aussi connectés par des arcs *own* (resp. *ref*) aux nœuds représentant les `EAttributes` (resp. les `EReferences`) introduits et hérités par la `EClass`. Enfin, pour un nœud représentant une `EReference`, des arcs *type* sont créés vers le nœud représentant la `EClass` qui type la `EReference` ainsi que vers tous les nœuds représentant les superclasses de la `EClass`.

3 Application de l'algorithme de Similarity flooding et résultats

Nous ne détaillerons pas ici les principes du Similarity Flooding, qui peuvent être trouvés en détail à la référence [3] ou résumés à la référence [5]. L'algorithme Similarity Flooding fournit en sortie des mises en correspondances deux à deux des nœuds des graphes d'entrée, avec pour chaque correspondance une valeur de similarité comprise entre 0 et 1 (plus cette valeur est proche de 1, plus les nœuds sont similaires). La figure 5 montre l'alignement calculé sur nos graphes exemples

issus de la configuration Minimal (figure 2) . L'algorithme Similarity Flooding utilise en interne une valeur de seuil qui est ici de 0.95. Toutes les mises en correspondance calculées sont correctes, à l'exception de celle entre *Operation* et *JTypedElement*.

À partir de cet alignement, nous produisons un modèle d'alignement, conforme au métamodèle donné à la figure 6. Le détail de l'obtention d'un tel modèle est donné à la référence [5].

4 Résultats sur une étude de cas

Notre étude de cas compare les résultats obtenus sur plusieurs métamodèles avec les différentes configurations proposées. Les paramètres utilisés pour l'algorithme similarity flooding sont : $\varepsilon = 0.05$ and $threshold = 0.95$. Les scénarios d'alignement choisis sont : $exMM_{source} \leftrightarrow exMM_{cible}$, $Ecore \leftrightarrow Minjava$, $Ecore \leftrightarrow Kermeta$ et $Ecore \leftrightarrow UML$. Ecore [6] est un méta-métamodèle donc aussi un métamodèle. Minjava [7] est un métamodèle simple pour Java, qui permet de représenter la structure d'un programme Java. Kermeta [8] est une extension d'Ecore qui permet de donner une description comportementale des opérations et des propriétés dérivées d'un modèle de classe. Puisque les éléments du métamodèle ne sont pas les mêmes pour les différentes configurations, nous avons besoin d'un dénominateur commun entre ces configurations pour les comparer. Nous nous basons sur un modèle d'alignement ne montrant que : les classes non abstraites, les attributs et références non dérivées, les types de données et les énumérations. Ces éléments sont inclus par toutes les six configurations étudiées. Nous comparons les modèles d'alignements obtenus via notre méthode avec un modèle d'alignement idéal conçu par un expert (des transformations standard comme Ecore vers Kermeta ont été utilisées pour construire ces alignements idéaux). Nous évaluons les résultats obtenus avec des métriques classiquement utilisées dans le domaine de la fouille de données [9]. Pour chaque alignement, nous calculons la précision : $precision = \frac{\#correct\_found\_mappings}{\#total\_found\_mappings}$, le rappel : $recall = \frac{\#correct\_found\_mappings}{\#total\_correct\_mappings}$ et le score : $f\_score = \frac{2 \times recall \times precision}{recall + precision}$. La précision, le rappel et le score sont compris entre zéro et un. Plus la précision est forte, plus le nombre de mise en correspondance erronées est faible. Plus le rappel est grand, plus le nombre de mises en correspondance non trouvées est faible. Le score est la moyenne harmonique entre la précision et le rappel, il peut être considéré comme une mesure globale de la qualité de l'alignement : un fort score est trouvé quand l'alignement produit est de bonne qualité.

Les résultats de ces expériences sont donnés aux figures 7 et 8. Les résultats montrent que la meilleure configuration est en général Saturated, alors que la configuration Minimal donne de mauvais résultats. Les configurations Full et Standard, bien qu'utilisant plus d'information des métamodèles, produisent dans nos expériences des résultats légèrement plus mauvais que la configuration Basic. Les métamodèles que nous utilisons pour s'aligner sur Ecore sont de taille variable. Minjava a plus ou moins la même taille qu'Ecore, Kermeta est un peu plus grand, et UML est bien plus grand. Nous voyons clairement dans nos expériences que la qualité de

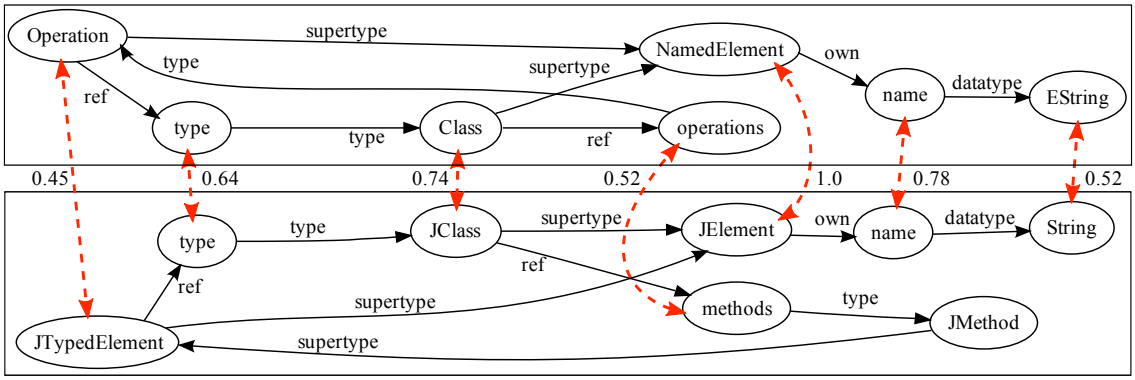


Fig. 5. Alignement calculé entre exG_{source} et exG_{cible}

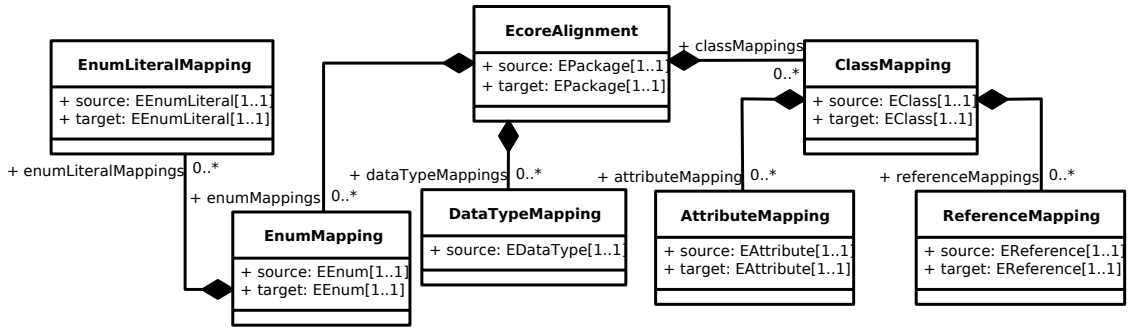


Fig. 6. Métamodèle d'alignement pour Ecore

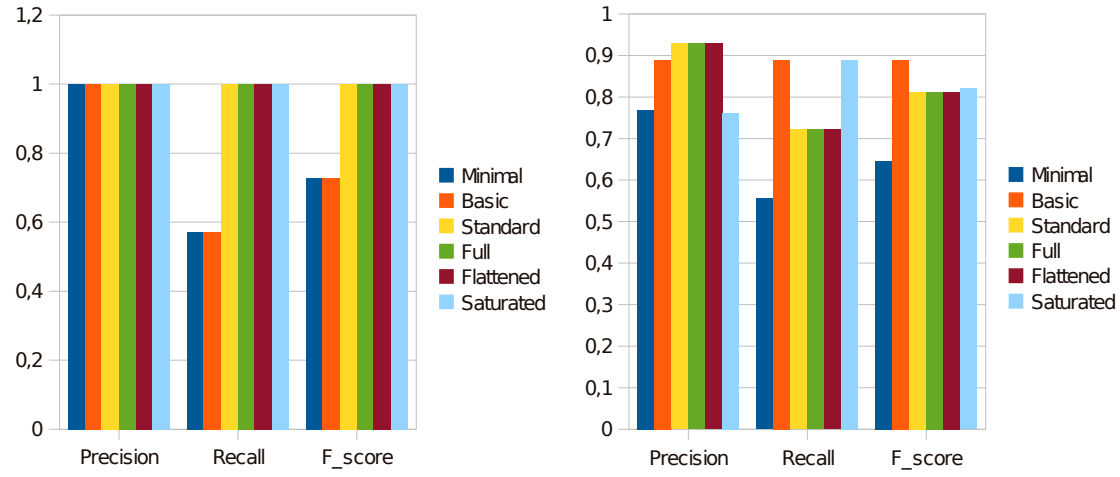


Fig. 7. Résultats de $exMM_{source} \leftrightarrow exMM_{cible}$ (gauche) et $Ecore \leftrightarrow Minjava$ (droite)

l'alignement décroît quand la différence de taille entre les métamodèles alignés augmente. Ceci est partiellement dû au filtre interne à l'algorithme SF, *SelectThreshold*, qui filtre l'alignement produit. Les résultats sur $Ecore \leftrightarrow Minjava$ et $Ecore \leftrightarrow Kermeta$ montrent que des alignements de bonne qualité peuvent être produits avec notre approche.

5 Conclusions et perspectives

Nous avons présenté ici une approche qui produit automatiquement un alignement entre deux métamodèles. Cette approche est basée sur l'application aux métamodèles d'un algorithme de mise en correspondance de schémas bien connu, le Similarity Flooding. Notre principale contribution est l'étude des différents moyens d'encoder un métamodèle donné en un graphe dirigé et étiqueté qui peut être exploité par le Similarity Flooding. Nous avons comparé dans une étude de cas six stratégies en termes de précision et de rappel des mises en correspondance calculées. Cette étude de cas révèle que la stratégie la plus intuitive (Minimal) donne de mauvais résultats, et que la stratégie Saturated est la plus adaptée dans la majorité des cas. La génération d'un modèle d'alignement conforme à un métamodèle dédié et duquel du code de transformation de modèles pourrait être généré est notre deuxième contribution. Nos pistes de recherche consistent en la conception de générateurs de code à la fois pour des langages impératifs et déclaratifs, ainsi qu'en l'étude de l'adaptation de Similarity Flooding pour notre application particulière.

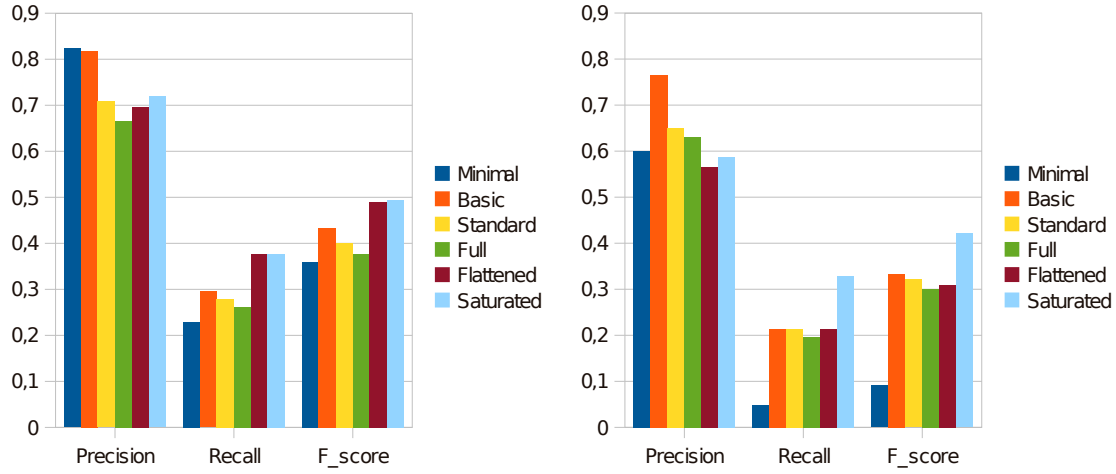


Fig. 8. Résultats de $Ecore \leftrightarrow Kermeta$ (gauche) et $Ecore \leftrightarrow UML$ (droite)

Références

1. Rahm, E., Bernstein, P.A. : A survey of approaches to automatic schema matching. The International Journal on Very Large Data Bases **10**(4) (2001) 334–350
2. Shvaiko, P., Euzenat, J. : A survey of schema-based matching approaches. **3730** (2005) 146–171
3. Melnik, S., Garcia-Molina, H., Rahm, E. : Similarity flooding : A versatile graph matching algorithm and its application to schema matching. In : Proc. of the 18th International Conference on Data Engineering, IEEE Computer Society (2002) 117–128
4. Pottinger, R., Bernstein, P.A. : Merging models based on given correspondences. In : The International Journal on Very Large Data Bases. (2003) 826–873
5. Falleri, J.R., ans Mathieu Lafourcade, M.H., Nebut, C. : Meta-model matching for automatic model transformation generation. In : Proc. of MODELS'08 : 11th International Conference on Model Driven Engineering Languages and Systems. (2008) 326–340
6. Budinsky, F., Brodsky, S., Merks, E. : Eclipse Modeling Framework. (2003)
7. Falleri, J.R. : Minjava. http ://code.google.com/p/minjava/ (2008)
8. Triskell : Kermeta. http ://www.kermeta.org (2008)
9. Do, H.H., Melnik, S., Rahm, E. : Comparison of schema matching evaluations. In Chaudhri, A.B., Jeckle, M., Rahm, E., Unland, R., eds. : Web, Web-Services, and Database Systems. Volume 2593 of Lecture Notes in Computer Science., Springer (2002) 221–237

Session groupe LaMHa

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

Sémantiques formelles d'un mini-langage impératif BSP

Application à la preuve de programmes et à l'optimisation

Jean Fortin and Frédéric Gava

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Université Paris Est

Dans cet exposé basé sur [6,5], nous décrivons plusieurs sémantiques d'un mini-langage impératif muni d'opérations pour le parallélisme BSP (dans un style SPMD). Un intérêt de ce travail est que tout le développement des sémantiques ainsi que les preuves des propriétés (et des programmes) ont été réalisé dans l'assistant de preuve Coq [2,3], ce qui augmente fortement la confiance que l'on peut apporter à notre propos.

1 Modèle BSP

Le modèle BSP [9,7] (*Bulk Synchronous Parallelism*) est un paradigme de calcul parallèle où un nombre fixe de processeurs se coordonnent par le biais d'un réseau (une mémoire partagée peut aisément simuler ce réseau). Un programme BSP s'exécute en super-étapes. À chaque super-étape, des calculs purement locaux sont effectués par les processeurs ainsi que des demandes de communications. Lorsque tous ces calculs sont terminés, les communications sont effectuées et une barrière de synchronisation globale rend ces données accessibles aux processeurs et une nouvelle super-étape peut ainsi commencer.

Ce parallélisme très structuré à l'avantage de la portabilité, de la simplicité (une dizaine de routines est nécessaire par rapport à la centaine de MPI [8]) de la sémantique, une nature relativement déterministe des algorithmes et un modèle de coût fiable permettant de prédire les performances des programmes et de raisonner même dynamiquement sur le meilleur algorithme (suivant les paramètres de la machine parallèle) à choisir.

2 Sémantique naturelle

Nous avons tout d'abord développé une sémantique naturelle dédiée à la preuve de programmes impératifs munis d'opérations de la bibliothèque PUB [4] et qui a été étendue aux programmes divergents. Plusieurs programmes « jouets » ont ainsi été prouvé notamment un algorithme de calcul scientifique : les n-corps avec une boucle systolique et un calcul des préfixes en une étape. La sémantique a été prouvée déterministe et exclusive à la sémantique pour les programmes divergents.

L'expérience qui a ainsi été faite nous a convaincu de l'utilité de développer des outils dédiés à la preuve de programmes BSP : les preuves en n'utilisant que les sémantiques en Coq sont très rébarbatives et l'automatisation de certaines parties de la preuve pourrait être fait. Nous pensons dans le futur utiliser le logiciel WHY pour simuler le parallélisme BSP et avoir automatiquement des obligations de preuves à partir d'assertions logiques dans les programmes BSP.

3 Sémantique à petits-pas

Nous avons ensuite définie une sémantique à petit pas pour le même mini langage et prouvé l'équivalence entre cette sémantique et la sémantique naturelle. Ce travail été étendue pour les sémantiques traitant de la divergence des programmes.

Cette sémantique à petits-pas n'est utilisée que comme intermédiaire vers une dernière sémantique et permet de faire le liens entre la sémantique pour la preuve des programmes et la dernière sémantique dédiées aux optimisations.

4 Sémantique pour les routines de haute-performance

Nous avons enfin donné une sémantique pour les opérations dîtes haute-performance dans le cadre de la programmation BSP. Celle-ci sont bien évidemment non sûres et non-déterministes. Elles sont dîtes haute-performances car les communications ne sont plus mise en tampon et sont asynchrones c'est-à-dire que les valeurs ne sont pas copiées en mémoire avant d'être envoyées et ce à n'importe quel moment au cours de l'exécution de la super-étape.

Les programmes n'utilisant pas ces nouvelles primitives ont la même exécution dans cette sémantique que dans celle à petits-pas et les programmes avec ces routines n'ont pas de sens dans les sémantiques BSP « normales ».

Pour montrer l'utilité d'une telle sémantique, nous avons certifié une optimisation des programmes : transformer des routines BSP normales en leurs pendant haut-performances. Cette fonction d'optimisation est pour l'instant très simple et nous a permis de tester quelles seraient les difficultés pour des optimisations plus ambitieuses : seules les routines de communications qui ne sont ni dans des boucle ni dans des conditionnelles (donc les algorithmes avec un nombre constant de super-étapes et un nombre constant de communications, mais pas forcement en taille) sont traitées. Le tri parallèle pourrait ainsi être optimisé ainsi que certains algorithme sur les graphes.

Prouver cette optimisation est possible en décomposant un programme en blocs (super-étapes) et en prouvant que l'ordre des communications n'affecte pas le bon déroulement du programme : les valeurs envoyées ne sont plus modifiées par le processeur émetteur ou récepteur avant la fin du bloc.

5 Travaux futurs

Dans le cadre de ce travail, nous comptons étendre ces sémantiques à un noyau plus important et qui correspond au langage WHY [1]. Nous comptons aussi développer (et prouver correcte) une simulation de ces programmes BSP par le langage WHY afin de faire de la preuve de programmes BSP. Nous aurions ensuite des sémantiques (par équivalences ou optimisations) vers une exécution haute-performance. Cette exécution (via une implantation MPI) sera ensuite prouvée correcte. Étendre ce travail au standard MPI ou a un sous-ensemble est un objectif possible à long terme.

Références

1. Why : a software verification tool. Web pages at why.lri.fr, 2003.
2. The Coq Proof Assistant (version 8.1pl4). Web pages at coq.inria.fr, 2008.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
4. O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) Library. *Parallel Computing*, 29 :187–207(2), 2003.
5. F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. In *PDCAT 2008*. IEEE Press, 2008. to appear.
6. F. Gava and J.Fortin. Two Formal Semantics of a Subset of the Paderborn University BSPlib. In *PDP 2009*. IEEE Press, 2009. to appear.
7. W. F. McColl. Scalability, portability and predictability : The BSP approach to parallel programming. *Future Generation Computer Systems*, 12 :265–272, 1996.
8. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
9. L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8) :103, 1990.

Functional meta-programming for parallel skeletons

Jocelyn Serot[1] and Joel Falcou[2]

[1] LASMEA, UMR 66O2 CNRS/U. Blaise Pascal, Campus des Cézeaux
F-63177 Aubière, France
[2] LRI, Université Paris-Sud, F-91405 Orsay Cedex, France
Jocelyn.Serot@lasmea.univ-bpclermont.fr, joel.falcou@lri.fr

Abstract. We describe the implementation in MetaOcaml of a small domain specific language for skeleton-based parallel programming. We show how the meta-programming facilities offered by this language make it possible to virtually eliminate the run-time overhead for the resulting programs, compared to a hand-crafted, low-level implementation written in C+MPI.

1 Introduction

The now well-known concept of *skeletons* [1] has been proposed as a solution to the problems raised by low-level parallel programming using message-passing libraries such as MPI. With skeletons, parallel programming boils down to choosing, instantiating and combining high-level, generic constructors taken from a predefined library. Skeletons therefore define a small *domain specific language* (DSL) providing the requested level of abstraction, by which parallel programs can be built without having to deal with low-level implementation details. Several practical realisations of the skeleton concept have been proposed [2, 3, 6, 5, 10]. These realisations essentially differ in the way the related DSL is implemented. In practice, one has to define 1) the syntax and semantics of this DSL 2) the transformation rules used for turning the DSL-level expressions into a low-level implementation (using MPIcalls for instance).

For most cited systems, the DSL is implemented as a library within a classical, sequential host language (C, C++, Caml). This has two advantages. First, this spares the implementor from having to write a dedicated lexical analyser and parser. Second, it greatly eases the interfacing to sequential functions (either because these functions are written within the host language or because its *foreign function interface* can be directly reused). The price to pay, on the other hand, is a significant run-time overhead for the resulting code, compared with a hand-crafted implementation of the same application using low-level MPI calls.

In this paper, we explain how *partial evaluation* and *meta-programming* techniques – which have already been successfully applied in other contexts – can be used to solve the aforementioned problem. More precisely, we describe the implementation, in MetaOcaml, of a small DSL allowing

– high-level specification of parallel programs as a combination of skeletons,
– automatic generation of the underlying, equivalent low-level parallel code as a set of sequential communicating processes,
– execution of this code on a *cluster* architecture supporting the MPIlibrary.

We first briefly recalls what is meta-programming and how it is supported within the MetaOcaml programming language. We then explain why this technique is interesting in the context of message-based parallel programming. We introduce a DSL dedicated to skeleton-based parallel programming, giving its syntax and semantics and describing how this (formal) semantics can be used to derive an implementation in MetaOcaml. Preliminary experimental results are given and related work is discussed. We conclude by summarizing our contributions and giving hints for further work.

References

1. M. Cole. *Research Directions in Parallel Functional Programming*, chapter 13, Algorithmic skeletons. Springer, 1999.
2. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P3L: A Structured High Level Programming Language And Its Structured Support. *Concurrency: Practice and Experience*, pages 225–255, 1995.
3. H. Kuchen. A skeleton library. *Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 620–629, 2002.
4. Christian Lengauer, Don S. Batory, Charles Consel, and Martin Odersky, editors. *Domain-Specific Program Generation, International Seminar, Dagstuhl Castle, Germany, March 23-28, 2003, Revised Papers*, volume 3016 of *Lecture Notes in Computer Science*. Springer, 2004.
5. G. Michaelson N. Scaife and S. Horiguchi. Parallel Standard ML with Skeletons. *Scaleable Computing Practise and Experience*, 6(4), 2006.
6. J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.
7. Tim Sheard. Accomplishments and research challenges in meta-programming. In *SAIG 2001: Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44, London, UK, 2001. Springer-Verlag.
8. J. Falcou, J. Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. Proceedings of the International Conference ParCo 2007, Sep 2007, Aachen. To appear in *Advances in Parallel Computing, Volume 15, IOS Press, February 2008* Available on-line at `http://www.fz-juelich.de/nic-series/volume38/nic-series-volume38.pdf`
9. `http://caml.inria.fr/cgi-bin/hump.en.cgi?contrib=401`
10. `http://ocamlp3l.inria.fr`
11. `http://www.metaocaml.com`

Placement d'applications distribuées hétérogènes sur des architectures de type grappe

Sylvain Jubertie, Emmanuel Melin, Jérémie Vautard, Arnaud Lallouet

Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans B.P. 6759
F-45067 ORLEANS Cedex 2
{prénom.nom}@univ-orleans.fr

Les approches à composants apportent une grande souplesse pour le développement d'applications distribuées et facilitent leur déploiement sur des architectures distribuées telles que les grappes de PC. L'intergiciel FlowVR [1] se place, à l'instar des intergiciels Syzygy [5] et Covise VR [7], dans la catégorie des approches à composants destinées à la création d'applications interactives. Une application FlowVR consiste en un ensemble de composants comportant des ports d'entrées et de sorties reliés par des connexions point à point, formant le graphe de l'application. Le modèle FlowVR définit les schémas de communication et de synchronisation entre les composants de l'application indépendamment de leur code. Cette caractéristique permet de déployer facilement une application sur une nouvelle architecture sans avoir à modifier son code, de tester les performances de différents placements d'une application sur une même architecture, ou encore de pouvoir exploiter de multiples réseaux. Le placement d'une application FlowVR consiste à associer chaque composant de l'application à un noeud de l'architecture, et chaque connexion point à point à des liens réseaux. L'intergiciel FlowVR permet donc de simplifier la conception et le déploiement d'applications distribuées.

Cependant, il reste à évaluer les performances offertes par un placement afin de pouvoir déterminer s'il offre les performances escomptées. Afin de vérifier un placement, une possibilité consiste à l'exécuter sur l'architecture cible et déterminer si l'application se comporte conformément aux attentes de l'utilisateur. Si le déploiement n'est pas satisfaisant, il faut alors en considérer un nouveau. La qualité du placement obtenu repose ainsi sur la connaissance de l'architecture et de l'application du développeur ainsi que sur son expérience. Cependant ce processus s'avère long et laborieux lorsque le nombre de noeuds et réseaux de l'architecture ainsi que le nombre de composants de l'application augmente. De plus, l'exécution de l'application monopolise l'architecture cible. Pour décharger le développeur de cette tâche, nous avons proposé un modèle de performance [3] pour les applications FlowVR permettant, à partir des modélisations de l'architecture et de l'application, d'évaluer les fréquences des composants ainsi que la latence entre eux. L'évaluation des performances d'un placement ne requiert alors plus l'exécution sur l'architecture cible. Cependant, le problème de recherche d'un placement répondant aux exigences de l'utilisateur se pose toujours et nous avons montré que le nombre de placements à étudier subit une explosion combinatoire [2]. Dans le pire cas, il se peut qu'aucun placement de l'application sur

l'architecture cible ne permette d'atteindre les performances attendues et pour le vérifier il faut étudier tous les placements envisageables.

Nous proposons donc de modéliser notre problème de placement sous forme de contraintes et d'utiliser un solveur pour générer et vérifier automatiquement des placements, ceci afin de simplifier et accélérer le déploiement d'applications FlowVR sur des architectures de type grappes de PC. La programmation par contraintes permet d'ajouter ou de modifier simplement des contraintes ce qui la rend plus souple que les techniques d'optimisation telles que les algorithmes génétiques. Notre problème de placement s'exprime naturellement sous forme d'un problème de contraintes sur des domaines finis [4]. Les différents noeuds et liens réseaux de l'architecture sont identifiés par des valeurs entières et possèdent des caractéristiques propres : nombre et type de processeurs, débit et latence des interfaces réseaux. Chaque composant ou connexion entre composants est associé à une variable et possède respectivement pour un composant, une charge processeur ainsi qu'une fréquence propre, et pour une connexion un volume de données. Chaque variable v est associé à un domaine $D(v)$ sur lequel elle est définie. Par exemple, soit une architecture comportant 16 noeuds identifiés de 1 à 16, alors un composant pouvant être placé indifféremment sur un des noeuds de l'architecture est associé une variable C_i définie sur le domaine $D(C_i) = [1..16]$. Différentes contraintes sont ensuite exprimées entre les variables. Des contraintes sont définies à partir du graphe de l'application et du modèle FlowVR. Ainsi, le modèle FlowVR impose que les ports d'entrées connectés d'un composant disposent tous d'un message pour commencer une itération. Cette relation entre composants communiquant permet donc de définir des contraintes d'égalité entre les fréquences de ceux-ci. Des contraintes sont également définies à partir des caractéristiques de l'architecture. Par exemple, la somme des données transitant par seconde sur un lien réseau ne peut dépasser son débit réel. L'utilisateur est également libre de poser ses propres contraintes. Une implantation basée sur le solveur Gecode [6] nous a permis de montrer que cette approche permet de répondre aux nombreuses questions du développeur d'applications distribuées : existe t'il un placement pour mon application sur l'architecture cible ? quels sont les placements possibles ? l'ajout d'une extension matérielle permet-il d'améliorer les performances ? est-il possible d'optimiser l'utilisation de la grappe ? Cet outil permet donc de simplifier et d'accélérer le déploiement d'applications distribuées sur grappes de PC hétérogènes. Nous envisageons à court terme de tester cette approche sur des applications de plus grande taille sur des architectures plus vastes de type grilles de calculs. Afin de permettre ce passage à l'échelle, certaines améliorations de notre solveur sont envisageables telles que l'amélioration de la propagation ou la modification de l'ordre de la recherche.

Références

1. J. Allard, V. Gouranton, L. Lecointre, S. Limet, E. Melin, B. Raffin, and S. Robert. FlowVR : a Middleware for Large Scale Virtual Reality Applications. In *Proceedings of Euro-par 2004*, Pisa, Italia, August 2004.

2. S. Jubertie. *Modèles et outils pour le déploiement d'applications de Réalité Virtuelle sur des architectures distribuées hétérogènes*. Thèse de Doctorat d'Université, Université d'Orléans, december 2007.
3. S. Jubertie and E. Melin. Performance prediction for mappings of distributed applications on PC clusters. In *Proceedings of IFIP International Conference on Network and Parallel Computing, NPC'07*, Dalian, China, september 2007.
4. S. Jubertie, E. Melin, J. Vautard, and A. Lallouet. Mapping heterogeneous distributed applications on clusters. In *Proceedings of EuroPar 2008*, LNCS, Las Palmas de Gran Canaria, Spain, august 2008. Springer.
5. B. Schaeffer and C. Goudeseune. Syzygy : Native PC Cluster VR. In *IEEE VR Conference*, 2003.
6. Christian Schulte and Guido Tack. Views and iterators for generic constraint implementations. In *Recent Advances in Constraints (2005)*, volume 3978 of *Lecture Notes in Artificial Intelligence*, pages 118–132. Springer-Verlag, 2006.
7. A. Wierse, U. Lang, and R. Ruhle. Architectures of distributed visualization systems and their enhancements. In *In Proceedings of 4th Eurographics Workshop on Scientific Visualization*, 1993.

Session groupe LTP

Langages, Types et Preuves

Classes de types de première classe

Matthieu Sozeau[1] et Nicolas Oury[2]

[1] Univ. Paris Sud, CNRS, Laboratoire LRI, UMR 8623, Orsay, F-91405
INRIA Saclay, ProVal, Parc Orsay Université, F-91893
sozeau@lri.fr
[2] Université de Nottingham
npo@cs.nott.ac.uk

Résumé Les classes de types ("*Type Classes*") ont remporté un grand succès dans le langage de programmation fonctionnelle Haskell et l'assistant de preuve Isabelle, comme solution permettant de surcharger des notations et spécifier avec des structures abstraites en quantifiant sur les contextes. Nous présentons un plongement superficiel des classes de types dans une théorie des types dépendants qui fait des classes des objets de première classe et supporte directement les extensions les plus populaires de Haskell. L'implémentation du système est légère et s'appuie sur des constructions existantes du langage qui sont simplement raffinées pour obtenir un ensemble bien intégré à l'environnement Coq. On présente sur des exemples comment ce système peut être utilisé pour la programmation et la preuve.

La surcharge est un concept de haut-niveau orthogonal aux paradigmes de programmation, bien qu'il soit au centre de la plupart des langages à objets. En termes généraux, la surcharge permet de dénoter à l'aide d'un seul nom un ensemble d'objets (méthodes, fonctions, valeurs) et comprend un mécanisme de désambiguation permettant de résoudre la référence vers un objet particulier. Ce mécanisme peut être la résolution au moment de l'exécution comme dans les langages à objets, ou au moment de la compilation comme pour les classes de types. Les classes de types ont été introduites dans le langage de programmation fonctionnel Haskell pour rendre le polymorphisme *ad-hoc* moins *ad hoc* [1].

On oppose cette forme de polymorphisme au polymorphisme *paramétrique* qui permet de définir des fonctions génériques sur tout type mais empêche d'obtenir des comportement différents selon le type effectif utilisé. Prenons par exemple le type polymorphe $\forall\ \alpha, \alpha \to \alpha \to \mathsf{bool}$ des fonctions à deux arguments retournant un booléen. Il est impossible de construire une fonction de ce type qui pour tout α, déciderait de l'égalité de deux objets de type α. Par exemple, il est impossible de décider de l'égalité sur le type $\mathbb{N} \to \mathsf{bool}$ des prédicats sur les naturels. En revanche il est possible de décider de l'égalité sur un grand nombre de types (booléens, naturels, entiers bornés, listes etc...) et l'on aimerait pouvoir dénoter par le même symbole l'ensemble de ces égalités, en utilisant l'information de typage sur les éléments comparés pour désambiguer la surcharge. Les classes de types permettent ceci *via* une définition de *classe* Eq pour l'égalité contenant une *méthode* surchargée == :

```
class Eq a where
  (==) : : a → a → Bool
```

```
instance Eq Bool where
  x == y = if x then y else not y
```

Une fois la classe déclarée on peut construire ses instances, comme l'égalité sur les booléens ci-dessus. Il est dès lors possible d'utiliser le symbole == pour dénoter l'égalité sur les booléens dans notre code. On peut aussi écrire des fonctions polymorphes supposant une instance de classe sur le type paramétrique et de la même façon utiliser les méthodes surchargées. Par exemple, pour définir le prédicat polymorphe in qui teste l'appartenance d'un élément à une liste, on a besoin de pouvoir tester l'égalité sur le type des éléments ce qu'on exprime par une contrainte Eq $a \Rightarrow$:

```
in : Eq a ⇒ a → [a] → Bool
in x [] = False
in x (y  : ys) = x == y || in x ys
```

Dans l'article [2], nous montrons comment traduire cette construction dans Coq, en codant les classes par des structures de première classe : les enregistrements dépendants. La traduction utilise le système d'arguments implicites et une procédure de recherche de preuve qui sont implémentés à l'extérieur du noyau pour permettre d'écrire le code utilisant la surcharge et résoudre les contraintes générées automatiquement.

On montre aussi comment interpréter des constructions plus complexes sur les classes comme les concepts de superclasse et sous-classe qui permettent de créer des hiérarchies de structures. On étend l'interprétation très facilement en utilisant toute la puissance du produit dépendant [3] qui permet de spécifier aisément la paramétrisation donc le partage de structures.

À l'aide des types dépendants et en s'alliant au système de tactiques à la base du "*shell*" interactif de Coq, les classes de types permettent aussi de créer des tactiques génériques et personnalisables par l'utilisateur. On peut par exemple définir une classe surchargeant l'ensemble des preuves de réflexivité du système :

```
Class Reflexive (A : Type) (R : relation A) := reflexive : ∀ x : A, R x x.
```

Cette classe est indexée à la fois par un type A mais aussi une valeur R de type relation A (soit une fonction a deux arguments dans Prop). On peut instancier cette classe sur l'égalité de Leibniz pour n'importe quel type A :

```
Instance eq_refl : Reflexive A (eq A) := reflexive x := refl_equal x.
```

De même, on peut créer une instance pour l'équivalence logique ↔ :

```
Instance iff_refl : Reflexive Prop iff.
```

Il devient maintenant possible d'utiliser la méthode surchargée reflexive là ou l'on attend une preuve de réflexivité de l'égalité, de l'équivalence ou de toute autre relation déclarée réflexive par l'utilisateur. On a démontré l'utilité d'un tel système pour re-développer la tactique de réécriture généralisée du système Coq [4, chapitre 9].

Le système en est encore à ses premiers pas même s'il fait déjà partie intégrante de Coq 8.2 [5]. En particulier, la recherche de preuve est très grossière puisqu'elle ne permet pas de détecter les ambiguïtés lorsque plusieurs instances sont possibles. Ce trait n'est pas gênant lorsqu'on recherche une preuve mais beaucoup plus lorsqu'on programme et qu'on voudrait par exemple savoir qu'une occurrence de + a deux interprétations possibles dans le même contexte.

Conclusion Pour résumer, nous montrons comment interpréter les "*Type Classes*" dans une théorie des types avec types dépendants. Notre interprétation s'étend aux constructions de haut-niveau des "*Type Classes*" comme les superclasses et donne aussi lieu à des usages originaux grâce aux types dépendants. Nous avons implémenté ce système dans Coq et démontré son utilité sur des exemples variés. Les classes de types dépendantes semblent un outil prometteur pour spécifier, prouver et organiser les développements en Coq.

Références

1. Wadler, P., Blott, S. : How To Make *ad-hoc* Polymorphism Less *ad hoc*. In : ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas. (1989) 60–76
2. Sozeau, M., Oury, N. : First-Class Type Classes. In Otmane Ait Mohamed, C.M., Tahar, S., eds. : Theorem Proving in Higher Order Logics, 21th International Conference. Volume 5170 of Lecture Notes in Computer Science., Springer (2008) 278–293
3. Oury, N., Swierstra, W. : The Power of Pi. In : ICFP '08 : Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming, New York, NY, USA, ACM (2008) 39–50
4. Sozeau, M. : Un environnement pour la programmation avec types dépendants. PhD thesis, Université Paris 11, Orsay, France (2008)
5. Sozeau, M. : Type Classes. In : Coq 8.2 Reference Manual. INRIA TypiCal (2008)

The MOBIUS Proof Carrying Code infrastructure

An overview

Gilles Barthe[1], Pierre Crégut[3], Benjamin Grégoire[2], Thomas Jensen[4], and David Pichardie[5]

[1] IMDEA Software, Madrid, Spain
[2] INRIA Sophia-Antipolis Méditerranée, France
[3] France Télécom, France
[4] IRISA/CNRS, France
[5] IRISA/INRIA Rennes Bretagne Atlantique, France

Abstract. The goal of the MOBIUS project is to develop a Proof Carrying Code architecture to secure global computers that consist of Java-enabled mobile devices. In this overview, we present the consumer side of the MOBIUS Proof Carrying Code infrastructure, for which we have developed formally certified, executable checkers. We consider wholesale Proof Carrying Code scenarios, in which a trusted authority verifies the certificate before cryptographically signing the application. We also discuss retail Proof Carrying Code, where the verification is performed on the consumer device.

1 Introduction

MOBIUS [6] is a European integrated project developing basic technologies to ensure reliability and security in global computers formed of a host of Java-enabled devices, such as phones, PDAs, PCs, which provide a common runtime environment for a vast number of mobile applications. Its aim is to give users independent guarantees of the safety and security of mobile applications, using the concept of security through verifiable evidence emphasized by the Proof Carrying Code (PCC) paradigm [Nec97]. The fundamental view behind PCC is that mobile code components come equipped with a certificate that can be checked efficiently and independently by the code consumer to ensure that that downloaded components issued by the producer respects its policy.

PCC complements standard security infrastructures such as PKI, which only guarantee the origin and the integrity of code, and makes an appropriate basis for security of global computers; however, there remain significant challenges to generalize its use in security architectures for global computing, in particular, PCC has mostly been used to enforce safety properties of applications, including type safety, and memory management safety. One goal of the MOBIUS project is to show the adequacy of PCC for enforcing basic security policies such as non-interference and resource control, and for the verification of functional properties of applications.

One other goal of the MOBIUS project is to develop efficient and scalable mechanisms to generate and to check certificates, and to prove formally that the security-critical part of the PCC infrastructure is correct. The soundness of all the MOBIUS Proof Carrying Code infrastructure requires a formal specification of the JVM. This specification is formalised in the Coq proof assistant [Coq04] and is called Bicolano. It is a formal description of the Java Virtual Machine (JVM), giving a rigorous mathematical description of Java bytecode program executions. It closely follows the official description of the JVM [LY99] as provided by Sun. Since the correctness of Bicolano is not formally provable, the close connection with the official specification is essential to gain trust in the specification.

The purpose of this article is to present intermediate achievements of the project with respect to its goal of achieving efficient and trustworthy PCCs tools.

2 Reflective Proof Carrying Code for wholesale checking

In wholesale Proof Carrying Code, mobile code transits through a trusted intermediary, e.g. a mobile phone operator. PCCshort is used by code producers (that are external to the phone operators and untrusted by

[6] http://mobius.inria.fr

them) with proofs which establish that the application is secure. The operator then digitally signs the code before distributing it to the code consumers (the customers, who rely on the operator).

This scenario for "wholesale" verification by a code distributor effectively combines the best of both PCC and trust, and brings important benefits to all participating actors. For the end user in particular, the scenario does not add PCC infrastructure complexity to the device, but still allows effective enforcement of advanced security policies.

Two main instances of the reflective Proof Carrying Code approach have been developed in MOBIUS. First, a lightweight type checker [BPR07] that enforces confidentiality of JVM applications with the same modularity principles as Java bytecode verification. Second, a verification condition generator (VCgen) has been fully certified with respect to the Bicolano semantics. It can be used to ensure functional and non-functional properties of programs.

3 Certified static analysis as lightweight Proof Carrying Code

Rather than using a complete proof assistant as a checker and a logic as the language for proof certificates, it is possible to rely on a lightweight form of PCC, where the code consumer checks the certificate itself with embedded verifiers. These verifiers are part of the *trusted computing base*, and should therefore be validated with respect to Bicolano. The formal framework for this validation is certified abstract interpretation [BJP06] which proposes an unified framework to specify both an analysis to infer program invariant and a checker to verify them. The correctness of the checker is proved formally in Coq. This results in a PCC architecture where both correctness proofs and actual program certificates arise from abstract interpretation. In this architecture, a verified fixpoint checker is combined with an untrusted fixpoint engine that the code producer uses to produce the stackmaps. Stackmaps are further reduced by a (still untrusted) fixpoint compression technique, yielding the final stackmap for the program. The code consumer extracts the fixpoint verifier using Coq's program extraction mechanism and installs it on the device. This extracted verifier can then be used to verify (on-device) the stackmaps accompanying a piece of downloaded software. Our first experiences with Motorola handsets show the feasibility of verifying the results of an analysis on-device with an extracted fixpoint verifier.

4 Conclusion

This article presents two PCC scenarios explored in MOBIUS, and their associated infrastructures for checking certificates. Both approaches rely on proof assistants to ensure the trustworthiness of certificate checkers: we advocate the use of reflective PCC for wholesale scenarios, and the use of certified certificate checkers for retail scenarios. In parallel to developing certified certificate checkers, the MOBIUS project has been actively investigating certificate generation: relevant material is available from the project web page.

Acknowledgments This work is supported by the Integrated Project MOBIUS, within the Global Computing II initiative.

References

[BJP06] F. Besson, T. Jensen, and D. Pichardie. Proof-Carrying Code from Certified Abstract Interpretation and Fixpoint Compression. *Theoretical Computer Science*, 364(3):273–291, 2006. Extended version.

[BPR07] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2007.

[Coq04] Coq development team. The Coq proof assistant reference manual V8.0. Technical Report 255, INRIA, France, March 2004. `http://coq.inria.fr/doc/main.html`.

[LY99] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification. Second Edition*. Sun Microsystems, Inc., 1999. http://java.sun.com/docs/books/vmspec/.

[Nec97] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

Vérification formelle d'un algorithme d'allocation de registres par coloration de graphe

S. Blazy[1], B. Robillard[1] et E. Soutif[1]

1: Laboratoire CEDRIC, 292 rue Saint-Martin, 75141 Paris CEDEX 03
{blazy,robillard}@ensiie.fr,soutif@cnam.fr

Résumé

Le travail présenté dans cet article est à l'interface entre la recherche opérationnelle et les méthodes formelles. Il s'inscrit dans le cadre du projet CompCert ayant pour but le développement et la vérification formelle, utilisant l'assistant de preuve Coq, d'un compilateur du langage C potentiellement utilisable pour la production de logiciels embarqués critiques. Nous nous intéressons dans cet article à l'allocation de registres, qui consiste à optimiser l'utilisation des registres du processeur. Nous proposons d'aborder cette optimisation en la modélisant par un problème dit de coloration avec préférences dont nous vérifions formellement la résolution. Cette vérification prend deux formes : preuve de correction de la spécification Coq pour la première partie de l'algorithme et validation *a posteriori* pour la seconde.

Introduction

Les méthodes de développement formelles permettent de produire du code source certifié. Cependant, une faille du processus de certification demeure au niveau de la compilation de ce code. En effet, un bug dans le compilateur peut introduire des erreurs dans le code assembleur généré, et donc invalider le code source. De là sont nés le besoin de compilateurs certifiés et le projet CompCert. Ce projet a pour but de développer avec l'assistant à la preuve Coq un compilateur réaliste d'un vaste sous-ensemble du langage C vers le langage assembleur du processeur PowerPC, principalement dévolu au domaine des logiciels embarqués [Ler06,BDL06].

Le compilateur CompCert est un compilateur modérément optimisant, qui accomplit de nombreuses passes de transformations de programmes. Celui-ci est certifié, c'est-à-dire qu'il est accompagné d'une preuve Coq de préservation du comportement des programmes (tout au long du processus de compilation). Cette preuve consiste à établir la préservation sémantique de chaque passe du compilateur. Il s'agit d'écrire en Coq une passe de compilation et de prouver ensuite que celle-ci transforme un programme en un programme observationellement équivalent. L'intérêt d'une telle approche est que le compilateur est certifié une fois pour toutes, indépendamment des programmes à compiler.

Le développement du compilateur CompCert est une expérience de conception assistée par preuve. Il ne s'agit pas de prouver un compilateur existant, mais plutôt de définir conjointement les langages intermédiaires, les transformations de programmes et les preuves associées. C'est souvent à l'issue d'une preuve jugée trop difficile qu'il est décidé de modifier un langage intermédiaire, voire de définir un nouveau langage intermédiaire, ce qui nécessite de plus de prouver à nouveau certaines propriétés ayant préalablement été prouvées. Ainsi, le compilateur CompCert actuel dispose de sept langages intermédiaires.

L'allocation de registres est la seule phase du compilateur CompCert qui n'est pas entièrement écrite en Coq. La raison principale est qu'il s'agit d'une transformation de programmes peu adaptée à une écriture fonctionnelle, et que l'effort nécessaire pour spécifier en Coq et prouver ensuite la préservation sémantique est trop important. En effet, classiquement, l'allocation de registres se ramène à un problème de coloration avec préférences de graphe. Ce problème étant $\mathcal{NP}$-difficile, CompCert implante une des heuristiques les plus performantes pour le résoudre.

Dans CompCert, l'allocation de registres est écrite en Caml puis validée *a posteriori* en Coq : pour chaque programme à compiler, il est vérifié formellement que la solution calculée par l'allocation de registres est correcte. L'intérêt de cette approche est que la preuve à effectuer est beaucoup plus facile, puisqu'il s'agit de vérifier que l'allocation de registres a bien renvoyé une coloration du graphe. En outre, cette technique permet de valider une transformation de programme qui n'a pas été écrite en Coq.

Cet article décrit une nouvelle méthode d'optimisation de l'allocation de registres pour le compilateur CompCert. Nous détaillons d'abord une formalisation en Coq d'un algorithme de coloration (sans préférences) adapté à une famille de graphes regroupant la majeure partie des graphes utilisés pour modéliser l'allocation de registres. Il s'agit d'un algorithme exact, dont nous prouvons également l'optimalité en Coq dans la famille de graphes précitée. Nous présentons ensuite la seconde partie de la méthode qui utilise la programmation linéaire en nombres entiers. Cette étape est réalisée par un solveur qui fournit une allocation de registres optimale. Comme il s'agit d'un solveur externe, cette solution est validée *a posteriori* en Coq. Le programme mathématique à résoudre dépend du résultat de la première partie de la méthode, d'où l'importance de l'optimalité du premier algorithme.

L'objectif de cet article est double. D'une part, nous proposons une amélioration de l'allocation de registres actuellement utilisée dans le compilateur certifié CompCert. D'autre part, nous présentons le début d'un travail de formalisation en Coq de structures de données et algorithmes de théorie des graphes et de programmation mathématique. Cet article est organisé comme suit. La première partie introduit l'allocation de registres et présente un état de l'art. Puis, la deuxième partie décrit les notions de théorie des graphes et de programmation mathématique utiles pour l'allocation de registres. Ensuite, la troisième partie explique notre algorithme d'allocation de registres. Enfin, la quatrième partie détaille la spécification Coq ainsi que les principales propriétés que nous avons

prouvées. Le code source complet de ce développement est disponible sur la page `http://www.ensiie.fr/~blazy/register-allocation`.

1 Allocation de registres

1.1 Généralités

Au cours de l'exécution d'un programme, le processeur effectue un grand nombre d'accès à la mémoire afin de lire et écrire les valeurs des variables du programme. Ces accès sont naturellement gourmands en temps. Pour accélérer l'exécution des programmes, le processeur est muni d'un petit nombre de zones de stockage à accès beaucoup plus rapide, les registres. En général, le nombre de registres d'un processeur est nettement inférieur au nombre de variables utilisées dans un programme. Le but de l'allocation de registres est de déterminer où sont stockées les variables d'un programme à tout moment de son exécution : soit en registres si ces derniers sont disponibles, soit en mémoire le cas échéant. La difficulté est de proposer une affectation optimale des registres. Il est par exemple souvent nécessaire de choisir entre conserver une variable v dans un même registre R pendant l'exécution complète d'un programme (ce qui rend R inutilisable pour stocker d'autres variables), et réutiliser R lorsque la valeur de v n'a plus besoin d'être conservée en vue d'utilisations futures (ce qui nécessite de transférer en mémoire la valeur de v).

L'allocation de registres est la passe de compilation la plus étudiée et la plus difficile à mettre en œuvre dans un compilateur. La qualité du code compilé dépend en effet de la qualité de l'allocation de registres. Les deux tâches principales de l'allocation de registres sont le *vidage* de registres en mémoire, et la *fusion* de registres. Le vidage décide quelles variables seront stockées ultérieurement en registres. La fusion tient compte le plus possible des préférences entre variables, afin de minimiser les transferts entre registres. Par exemple, une affectation $x = y$ entraîne une préférence entre les variables x et y correspondant à la condition optimisante, qu'au vu de cette affectation, il serait préférable de stocker x et y dans le même registre.

1.2 Approches heuristiques

L'allocation de registres consiste à minimiser le nombre d'accès à la mémoire, étant donné un nombre fixe de registres. Classiquement, ce problème se ramène à la recherche d'une coloration de graphe. Dans le cas de l'allocation de registres, il s'agit d'un graphe d'interférences (défini dans la section 2.1), obtenu suite à une analyse de vivacité du programme à compiler. Le problème de coloration étant dans le cas général $\mathcal{NP}$-difficile, la quasi-totalité des approches imaginées ont été heuristiques ([Cha82], [BCT94], [GA96], [PP05], ...). Ces heuristiques proposent différentes combinaisons des deux phases de vidage et de fusion. Les plus simples effectuent les deux phases de manière séquentielle tandis que d'autres, plus sophistiquées, les réalisent simultanément. Les heuristiques d'allocation de

registres évoluent aujourd'hui encore, par exemple afin de tenir compte de la rapidité croissante des processeurs ainsi que du coût croissant des accès à la mémoire. Pour un compilateur d'un langage tel que C, l'heuristique la plus efficace à l'heure actuelle est celle d'Appel et George [GA96]. C'est d'ailleurs celle qui a été initialement choisie dans le compilateur CompCert.

Une autre heuristique récente est celle imaginée par Palsberg et Pereira [PP05]. Si son efficacité n'est pas suffisante pour remplacer l'heuristique d'Appel et George, ses fondements sont par contre forts intéressants. En effet, suivant la piste ouverte par Andersson [And03], ceux-ci ont mesuré qu'une large majorité des graphes d'interférences ont la propriété d'être triangulés [1]. Ils affirment en effet que plus de 95% des graphes d'interférences des méthodes de la bibliothèque Java 1.5 et des 27921 graphes de référence publiés par Appel et George ([AG05]) ont cette propriété.

L'allocation de registres est également une transformation de programmes qui :

- renomme les variables du programme,
- insère des instructions de lecture et écriture en mémoire, pour chaque variable à vider en mémoire,
- supprime des affectations lorsqu'elles concernent des variables stockées dans des registres ayant été fusionnés.

Il est donc important de s'assurer que l'allocation de registres préserve le comportement des programmes. De nombreuses approches de validation reposent sur l'utilisation de techniques d'analyse statique. Peu de travaux portent sur la vérification formelle (c'est-à-dire à l'aide d'un assistant à la preuve). [Oho04] propose un système de types dédié à l'allocation de registres, ainsi qu'un algorithme d'allocation de registres correct par construction, mais les instructions considérées sont celles d'un petit langage, et cette démarche semble difficilement applicable pour un compilateur tel que CompCert. [NPP07] propose un langage dédié à l'allocation de registres, ainsi qu'un système de types. La sûreté du typage de ce système de types a récemment été prouvée en Twelf. Le but de ce travail est de fournir un cadre général permettant de comparer différentes stratégies d'allocation de registres.

1.3 Programmation linéaire en nombres entiers appliquée à l'allocation de registres

Les premiers travaux utilisant la programmation linéaire en nombres entiers pour l'allocation de registres sur des problèmes de petite taille furent ceux de Goodwin et Wilken en 1996 [GW96] sur architecture CISC. Les résultats furent encourageants mais pas suffisamment pour supplanter l'approche traditionnelle. Il s'agissait alors d'une formulation incluant phases de vidage et de fusion. En 2001, Appel et George [AG01] ont introduit une formulation de la phase de

[1] La définition d'un graphe triangulé est donnée dans la section 2.2.

vidage pour les processeurs à architecture CISC[2] puis ont appliqué une méthode heuristique pour la phase de fusion (leurs tentatives d'utilisation de la programmation mathématique pour la phase de fusion se sont avérées infructueuses, particulièrement en raison du temps de résolution du problème par le solveur). Les résultats furent meilleurs que ceux de Goodwin et Wilken.

Cette année, Grund et Hack [GH07] ont élaboré un algorithme de coupes (*i.e.* une extension de la résolution par programmation mathématique) pour obtenir un résultat optimal pour la phase de fusion. Leur démarche a permis d'obtenir un résultat optimal pour 471 des 474 graphes de l'*Optimal Coalescing Challenge*[3] dans des temps très raisonnables pour la plupart des cas (430 cas sont traités en moins de 6 secondes).

2 Fondements mathématiques

2.1 Modélisation graphique de l'allocation de registres

Une *coloration* d'un graphe G est une fonction qui à chaque sommet de G associe une couleur de sorte que pour toute arête (i, j) de G les couleurs associées à i et j sont différentes. Si p est un entier strictement positif, une coloration utilisant moins de p couleurs est appelée *p-coloration*. Une coloration est dite *partielle* si certains sommets ne sont pas colorés. Une coloration est dite *optimale* si elle utilise un nombre minimal de couleurs.

Dans le cas d'une allocation de registres, le graphe à colorier est un *graphe d'interférences*, dont les sommets représentent les variables du programme à compiler. Les arêtes sont de deux types. Les arêtes d'*interférence* relient tous les sommets qui représentent des variables qui ne doivent pas occuper les mêmes registres à un instant donné de l'exécution du programme. Les arêtes de *préférence* relient tous les sommets qui représentent des variables telles qu'il existe une instruction d'affectation entre celles-ci (et il n'existe pas d'arête d'interférence entre ces sommets). Un poids est associé à chaque arête afin de tenir compte de la fréquence d'exécution des instructions, ainsi que de la fréquence d'utilisation des variables.

Dans ce qui suit, nous définissons un graphe G comme étant un triplet (S, I, P), où G est le graphe dont l'ensemble des sommets est S, l'ensemble des arêtes d'interférence est I et l'ensemble des arêtes de préférences est P. De plus, les graphes formés par S et I d'une part et S et P d'autre part sont respectivement appelés *interf-graphe* et *pref-graphe* de G.

Soient un entier strictement positif p et un graphe G. Le problème de *p-coloration avec préférences* consiste à trouver une p-coloration partielle de l'interf-graphe de G qui minimise la fonction $f = \sum_{(i,j)\in D} w_{ij} + c|NC|$, où D est

[2] La particularité de l'architecture CISC est que lors d'une opération du processeur, certains opérandes peuvent être en mémoire, d'autres en registres, contrairement à l'architecture RISC dans laquelle tous les opérandes doivent être en registre.

[3] Il s'agit d'une bibliothèque de graphes de référence publiée par Appel pour l'optimisation de la phase de fusion.

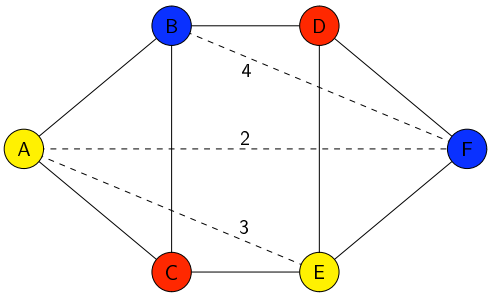


Fig. 1. Instance du problème de 3-coloration avec préférences.

l'ensemble des arêtes de préférences dont les extrémités sont de couleurs différentes, NC est l'ensemble des sommets non colorés, w_{ij} est le poids associé à l'arête (i, j), et c est une constante représentant le coût de vidage en mémoire d'un sommet non coloré. Une p-coloration avec préférences est optimale si elle minimise la fonction f.

L'allocation de registres est finalement modélisée par le problème de k-coloration avec préférences appliqué au graphe d'interférences, où k désigne le nombre de registres utilisables. Chaque couleur représente un registre. La figure 1 est un exemple d'instance résolue du problème de 3-coloration avec préférences. En trait plein sont représentées les arêtes d'interférence et en pointillé les arêtes de préférences. Dans cet exemple la coloration réalisée est optimale puisqu'elle colore tous les sommets et que deux des trois arêtes de préférence ont des couleurs identiques aux deux extrémités. En effet, il est impossible de satisfaire les trois préférences car sinon les sommets A, B, E et F seraient de la même couleur et donc plusieurs contraintes de coloration seraient violées.

Étant donnée cette modélisation, la phase de vidage des registres en mémoire consiste à rechercher un ensemble de sommets k-colorable, c'est-à-dire pouvant être coloré avec k couleurs. La phase de fusion consiste à colorer cet ensemble de sommets.

2.2 Théorie des graphes et programmation mathématique

Les travaux décrits dans cet article nécessitent la définition de notions de théorie des graphes et de programmation mathématique. Nous commençons par présenter la classe des graphes triangulés qui joue un rôle central au sein de notre étude ainsi que les ordres d'élimination simpliciaux qui leur sont intimement liés pour finir par une description de la programmation linéaire en nombres entiers. Dans les définitions suivantes, $G = (S, I, P)$ désigne un graphe, et E désigne un ensemble de sommets de S.

1. Un cycle C est dit *sans corde* si aucune arête ne relie deux sommets non consécutifs de C.
2. Le *graphe induit* par E est la restriction de G aux sommets appartenant à E et aux arêtes reliant deux sommets de E.
3. Un graphe qui ne possède aucun cycle induit sans corde de longueur supérieure ou égale à quatre est dit *triangulé* (*chordal* en anglais).
4. Une *clique* est un graphe dont tous les sommets sont reliés deux à deux.
5. Un sommet s est *simplicial* dans un graphe G si le graphe induit par les voisins de s est une clique.
6. Un *ordre d'élimination simplicial* (noté *oes* ou *peo* pour *perfect elimination order* en anglais) est une permutation $(x_1,\ldots, x_n)$ de S telle que pour tout $i \in \{1,\ldots n\}, x_i$ est simplicial dans le graphe induit par $\{x_i, \ldots x_n\}$.
7. $n(G)$ désigne *l'ordre* de G, c'est-à-dire le nombre de sommets de G.
8. $m(G)$ est la *taille* de G, c'est-à-dire le nombre d'arêtes de G.
9. $\chi(G)$ est le *nombre chromatique* de G, c'est-à-dire le nombre minimal de couleurs nécessaire pour réaliser une coloration de G.
10. $\omega(G)$ est l'ordre de la plus grande clique induite de G.

Une autre méthode utilisée pour la résolution exacte de problèmes d'optimisation est la programmation mathématique. Il s'agit de décrire le problème comme la minimisation (ou la maximisation) d'une fonction (appelée *fonction économique*) sous contraintes de ses variables (égalités et inégalités). Tout programme mathématique (P) peut donc s'écrire sous la forme suivante, où X désigne l'ensemble des solutions admissibles et n désigne le nombre de variables :

$$(P) \begin{cases} Min & f(x) \\ & x \in X \subseteq \mathbb{R}^n \end{cases}$$

Nous parlerons particulièrement de la programmation linéaire en variables {0,1} (PLNE) c'est-à-dire où les variables appartiennent toutes à l'ensemble {0,1}, et où la fonction économique et toutes les contraintes du problème sont linéaires. Notons qu'il s'agit généralement de la résolution exacte de problèmes $\mathcal{NP}$-difficiles par des méthodes énumératives ce qui demande un temps de résolution exponentiel. La résolution est confiée à un solveur commercial (par exemple CPLEX [Ilo02]). L'intérêt de tels solveurs est le recours à des techniques sophistiquées qui permettent de diminuer le temps de résolution du programme mathématique.

3 Description de l'algorithme

3.1 Séparation des phases et non optimalité

L'approche qu'il a été décidé de suivre pour cette étude est analogue à celle suivie par Appel et George [AG01], c'est-à-dire la voie de la programmation

linéaire en nombres entiers et du traitement séquentiel des phases de vidage et de fusion. Il s'agit cependant d'adapter la modélisation, car l'architecture du PowerPC (le langage cible de CompCert) est RISC, et non pas CISC comme dans [AG01]. Concrètement, les contraintes portant sur les variables des programmes linéaires ne sont pas les mêmes. Par exemple, l'architecture RISC oblige toute variable à être en registre au moment de son utilisation tandis que ce n'est pas obligatoire pour une architecture CISC. En contrepartie, un processeur à architecture RISC possède plus de registres dédiés au stockage des variables. Enfin, les instructions à considérer sont plus simples dans le cas d'un processeur RISC, et le nombre de contraintes est donc plus petit que dans le cas d'une architecture CISC.

Par ailleurs, contrairement à [AG01], nous traitons également la phase de fusion par programmation linéaire en nombres entiers. Ce traitement séquentiel pose un problème non négligeable. En effet, la programmation mathématique permet d'obtenir un résultat optimal pour la phase de vidage puis pour la phase de fusion mais ces deux optimisations sont dépendantes l'une de l'autre et peuvent donc ne pas déboucher sur une allocation des registres qui soit globalement optimale.

3.2 Deux processus de résolution

Nous pouvons néanmoins, dans la majorité des cas, éviter ce problème en limitant l'optimisation à la phase de fusion. En effet, la phase de vidage n'est utile que si le graphe d'interférences n'est pas k-colorable. Or, le grand nombre de registres allouables sur architecture RISC rend cette éventualité peu probable. Il suffit donc de tester si le graphe d'interférences est k-colorable pour savoir s'il est nécessaire d'effectuer la phase de vidage. Ce test est possible [4] notamment dans les graphes dont l'interf-graphe est triangulé, puisqu'il existe des algorithmes efficaces réalisant des colorations optimales dans ces graphes. Nous appliquons donc au graphe d'interférences un algorithme de coloration, l'algorithme de coloration gourmande, qui renvoie une coloration optimale si le graphe est triangulé et quelconque sinon. Si le nombre de couleurs utilisé par cette coloration est inférieur ou égal à k, il n'est pas nécessaire de réaliser la phase de vidage et donc le résultat de la phase de fusion correspond à une solution optimale du problème global d'allocation puisque cette dernière phase est traitée par programmation linéaire en nombres entiers.

La figure 2 résume cette approche. La vérification formelle en Coq de l'algorithme est composée de deux parties. La coloration est spécifiée et prouvée en Coq, tandis que les phases de vidage et de fusion sont validées *a posteriori*. Plus précisément, il est vérifié en Coq que les résultats calculés par le solveur externe représentent bien une coloration du graphe d'interférences.

[4] En un temps raisonnable, où le test de k-coloration est dit polynomial.

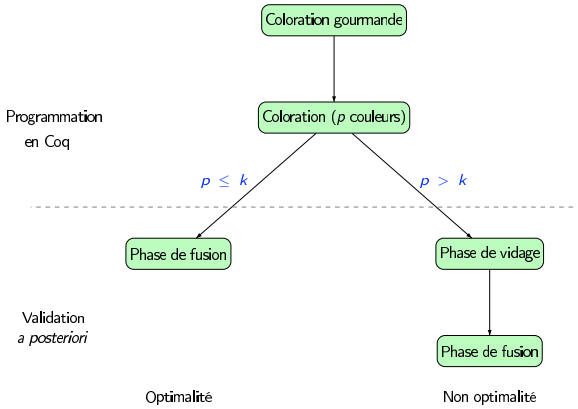


Fig. 2. Principales étapes de l'algorithme.

3.3 Algorithme de coloration gourmande

Le test de k-colorabilité est effectué par l'algorithme de coloration gourmande. Cet algorithme fournit une coloration optimale (*i.e.* utilisant un nombre minimal de couleurs) si l'ordre dans lequel sont coloriés les sommets est l'ordre inverse d'un ordre d'élimination simplicial [Gav72]. Ces résultats ont été prouvés en Coq (*cf.* section 4.5). Il en découle que les graphes triangulés peuvent être colorés de façon optimale grâce à cet algorithme puisque la recherche d'ordre d'élimination simplicial est un problème pour lequel il existe divers algorithmes polynomiaux. Nous avons choisi d'écrire en Coq un algorithme de recherche d'un ordre d'élimination simplicial relativement naïf mais dont la correction est plus simple à montrer que pour les algorithmes les plus efficaces et dont la complexité (en temps) est comparable. L'algorithme est donné ci-dessous. Il consiste informellement à chercher un sommet simplicial s, le retirer du graphe et itérer ce procédé. Cette technique fonctionne car tout graphe induit d'un graphe triangulé est lui même triangulé et possède donc un ordre d'élimination simplicial.

L'appel à la fonction $is\_sv(s, S-T)$ teste si s est un sommet simplicial dans le graphe induit par $S-T$. Plus précisément, il est testé si s et ses voisins de $S-T$ forment une clique. De plus, un graphe est triangulé si et seulement s'il admet un ordre d'élimination simplicial [FG65]. Aussi, l'ordre renvoyé par l'algorithme 1 est un ordre d'élimination simplicial si et seulement si le graphe G est triangulé.

Étant donné un ordre des sommets, l'algorithme 2 de coloration gourmande consiste à colorer les sommets selon cet ordre en affectant à chaque fois la plus petite couleur qui n'est utilisée par aucun des voisins du sommet courant x_i, en commençant la numérotation (coloration) à 1.

Algorithme 1 peo_search (G)

Entrée: Un graphe G=(S, I, P) dont l'interf-graphe est triangulé
Sortie: Un ordre $x = \{x_1, x_2, \ldots x_{n(G)}\}$ d'élimination simplicial de l'interf-graphe

```
1: i := 0, U := ∅
2: tant que i < n(G) faire
3:    trouvé := faux, T := U
4:    tant que trouvé = faux faire
5:       choisir s dans S − T
6:       si is_sv(s, S − T) alors
7:          x_{i+1} := s
8:          trouvé := vrai
9:       sinon
10:         T := T ∪ s
11:      fin si
12:   fin tant que
13:   i := i + 1; U := U ∪ {s}
14: fin tant que
```

Algorithme 2 graph_coloring (G)

Entrée: Un graphe G
Sortie: Une coloration de G, optimale si G est triangulé

```
1: x = peo_search(G), U := ∅
2: pour tout i de n(G) à 1 faire
3:    T := get_nghbs(G, x_i), U := U ∪ x_i
4:    affecter à x_i la plus petite couleur qui n'est affectée à aucun sommet de T ∩ U
5: fin pour
```

3.4 Programmation linéaire en nombres entiers

Nous utilisons pour modéliser la phase de fusion sur un graphe $G = (S, I, P)$ le programme mathématique défini dans [GH07]. Il existe deux types de variables pour modéliser le problème : d'une part, les variables x_{ic} qui valent 1 si et seulement si le sommet i est de couleur c ; d'autre part les variables y_{ij} qui valent 1 si et seulement si (i, j) est une arête et i et j sont de couleurs différentes.

Le programme mathématique est défini dans la figure 3. Il comprend trois séries de contraintes :

- (C_1) à chaque sommet doit être affectée une et une seule couleur,
- (C_2) chaque arête d'interférence doit avoir des extrémités de couleurs différentes,
- (C_3) $y_{i,j}$ doit valoir 1 si i et j sont de couleurs différentes. En effet, si les couleurs sont différentes alors le membre droit de l'inégalité (C_3) vaut 1 lorsque c est la couleur de i.

Pour optimiser la coloration il suffit de minimiser le poids des arêtes de préférence dont les extrémités sont de couleurs différentes, c'est-à-dire à minimiser $f = \sum_{(i,j)\in P} w_{ij} \times y_{ij}$.

$$(P1)\begin{cases} Min & \sum_{(i,j)\in P} w_{ij} \times y_{ij} \\ \text{sous les contraintes} & \\ (C_1)\ \forall i\ \in \{1,\ldots,n(G)\}, & \sum_{c=1}^{k} x_{ic} = 1 \\ (C_2)\ \forall (i,j) \in I,\ \forall c \in \{1,\ldots,k\}, & x_{ic} + x_{jc} \leq 1 \\ (C_3)\ \forall (i,j) \in P,\ \forall c \in \{1,\ldots,k\}, & x_{ic} - x_{jc} \leq y_{ij} \\ (C_4)\ \forall i \in \{1,\ldots,n(G)\},\ \forall c \in \{1,\ldots,k\}, & x_{ic} \in \{0,1\} \end{cases}$$

Fig. 3. Programme mathématique modélisant la fusion de registres.

Nous avons également défini un modèle mathématique adapté au traitement simultané des deux phases. Ce modèle étant assez proche du précédent, il n'est pas présenté dans cet article. Il suffit en effet d'ajouter des variables x_{i0} qui valent 1 si et seulement si le sommet i n'est pas coloré et de modifier les contraintes et la fonction économique du problème en conséquence.

4 Spécification Coq

Cette partie détaille la spécification en Coq de l'algorithme de coloration gourmande, les structures de données utilisées et les théorèmes de correction et d'optimalité de la spécification. Afin de différencier les prédicats et fonctions définis lors du développement de ceux de bibliothèques existantes, les premiers sont en gras et les seconds en italique.

4.1 Définition des graphes

La structure de graphe a été définie en Coq avec la construction `Record`. Deux types de graphes ont été définis : les graphes quelconques et les graphes tels que la liste de leurs sommets est un ordre d'élimination simplicial inverse. La structure générique des graphes est la suivante.

```
Record Graph : Set := mk_Graph{
  vertices : list nat;
  edges : list (nat×nat);
  (P1)  p_is_lex_sorted : is_lex_sorted edges;
  (P2)  p_is_strict_ord : is_strict_ord edges;
  (P3)  p_NoDup : NoDup edges;
  (P4)  p_vertices_edges : vertices = edges_to_vertices edges }.
```

Il a été volontairement choisi de construire tout l'algorithme de coloration à partir uniquement des arêtes du graphe pour s'approcher autant que possible de la structure actuellement utilisée dans CompCert. C'est pourquoi le graphe est modélisé par une liste d'arêtes, une arête étant un couple de sommets. De même, le choix d'utilisation de liste est voulu même s'il conduit à une diminution de complexité d'implantation. En effet, la notion d'ordre d'élimination simplicial

est essentielle et se représente idéalement par une liste, laquelle est par définition une permutation de la liste des sommets du graphe.
La liste *edges* des arêtes du graphe possède quant à elle trois propriétés :

- (P_1) indique que la liste *edges* est triée par ordre lexicographique ;
- (P_2) stipule que la liste *edges* est strictement ordonnée, c'est-à-dire que dans chaque arête, l'identifiant du sommet source est inférieur à celui du sommet destination.
- (P_3) mentionne que la liste *edges* ne contient pas de doublons.

(P_1) et (P_2) servent à améliorer la vitesse des algorithmes. En effet, elles correspondent à des propriétés de tri qui peuvent être exécutées en temps relativement rapide et permettent de diminuer la complexité des algorithmes ou d'y incorporer des conditions d'arrêt. Ces propriétés permettent en outre d'effectuer des parcours parallèles des listes de sommets et d'arêtes du graphe. (P_3) est une propriété de bonne formation du graphe. Toutes les fonctions nécessaires pour transformer la liste initiale ont été implantées et un algorithme de tri rapide a été écrit en Coq à l'aide de la construction `Function` (*cf.* section 4.3).

La liste *vertices* représente le sous-ensemble des sommets du graphe qui possèdent au moins une arête incidente. Le graphe d'interférences étant connexe[5], cet ensemble de sommets est exactement l'ensemble de tous les sommets du graphe. De plus, cette liste est triée par ordre croissant et ne contient pas de doublons, ce qui induit qu'elle est triée par ordre strictement croissant. Ces propriétés découlent de la façon dont a été construite la liste *vertices* à partir de la liste *edges*, c'est-à-dire de la fonction **edges_to_vertices**.

Afin de spécifier ce qu'est un graphe triangulé, il est nécessaire de donner la spécification des ordres d'élimination simpliciaux. Le prédicat (**sv** x v e) signifie que x est un sommet simplicial dans la liste des sommets v par rapport aux arêtes de e. Nous ne donnons pas sa spécification car elle fait elle même appel à d'autres prédicats. La spécification des ordres d'élimination simpliciaux est décomposée en deux définitions **_peo** et **peo** afin d'en alléger l'utilisation.

```
Inductive _peo : list nat → list nat → list (nat×nat) → Prop :=
    peo_cons : ∀ (l vert : list nat) (edg : list (nat×nat)),
            Permutation vert l ⇒
            (∀ (x : nat) (ll rl : list nat), l = ll ++ x : : rl ⇒ sv x (x : : rl) edg) ⇒
            _peo l vert edg.

Definition peo (l : list nat) (g : graph) : Prop := _peo l (vertices g) (edges g).
```

Le second type de graphes représente les graphes triangulés. Le prédicat (**est_clique** g l t) signifie que les sommets de la liste l forment une clique de taille t dans le graphe g. Ce type de graphe est donc celui où l'ordre inverse des sommets est simplicial. Ainsi, pour tout x l'ensemble des sommets qui précèdent x dans la liste *vertices* (autrement dit ceux qui sont colorés avant lui par l'algorithme de coloration gourmande) forme une clique.

[5] Un graphe connexe est un graphe pour lequel il est possible de relier toute paire de sommets par une liste d'arêtes telle que deux arêtes consécutives sont adjacentes.

```
Record Chordal_graph : Set := mk_Chordal_graph{
    gph : Graph ;
    self_peo : ∀ (x : nat), In x (vertices gph) ⇒
    is_clique gph (x : : get_nghbs gph x) (length (get_nghbs gph x) + 1)
}.
```

4.2 Description générale de l'implantation

L'algorithme de coloration gourmande a été écrit en Coq et son optimalité a été prouvée en Coq sur les graphes triangulés. Pour ce faire, il est nécessaire de construire un graphe $my\_chordal\_gph$ afin de créer un enregistrement de type $Chordal\_graph$ car c'est sur ce type de graphe que l'algorithme de coloration permet d'obtenir une coloration optimale. La figure 4 résume les différentes étapes de la coloration de graphe.

Il faut donc tout d'abord, à partir de la liste des arêtes du graphe, construire my_graph de type $Graph$. Ensuite, si my_graph est triangulé alors il admet un ordre d'élimination simplicial. Dans ce cas, il est nécessaire de renommer les sommets de my_graph de sorte que la liste inverse de l'ordre d'élimination simplicial trouvé corresponde à la liste $\{1,\ldots n\}$. Dès lors, il est possible de construire un graphe $my\_chordal\_gph$ permettant de définir un enregistrement de type $Chordal\_graph$, de réaliser la coloration de $my\_chordal\_gph$ (et la prouver optimale), puis de renommer dans le sens inverse les sommets pour prouver que la coloration obtenue (elle-même renommée de la même façon) est une coloration optimale de my_graph. Dans le cas où my_graph n'admet pas d'ordre d'élimination simplicial, my_graph est coloré, mais la coloration obtenue n'est pas optimale.

Nous nous sommes attachés à conférer à notre implantation quelques optimisations afin d'obtenir bonne une complexité algorithmique. En particulier, l'utilisation de divers ordres sur les listes permet un gain de complexité appréciable mais allonge considérablement le développement. Les trois principales propriétés d'ordre que nous utilisons sont :

- l'ordre lexicographique sur des listes de couples,
- l'ordre sur les composantes d'un couple (la première composante doit être plus petite que la seconde),
- la croissance des listes d'entiers.

Il a également été nécessaire de définir d'autres ordres connexes à ceux-ci comme les ordres stricts et les ordres inverses. Il existe plusieurs cas de figure pour lesquels l'usage d'ordre procure un gain significatif. Nous présentons ici un cas très simple : l'élimination des doublons d'une liste.

Si l est une liste de longueur lg, alors un algorithme classique d'élimination de doublons se code en $O(lg^2)$. Par contre, si la liste est triée par ordre croissant, il est possible d'effectuer cette opération en $O(lg)$ puisqu'il suffit de vérifier récursivement que les deux éléments de tête de liste sont différents et d'en supprimer un si ce n'est pas le cas. Si la liste n'est pas triée il est donc préférable d'un point de vue de la complexité algorithmique de la trier puis de supprimer les doublons puisque le tri rapide a une complexité en $O(lg\ log(lg))$.

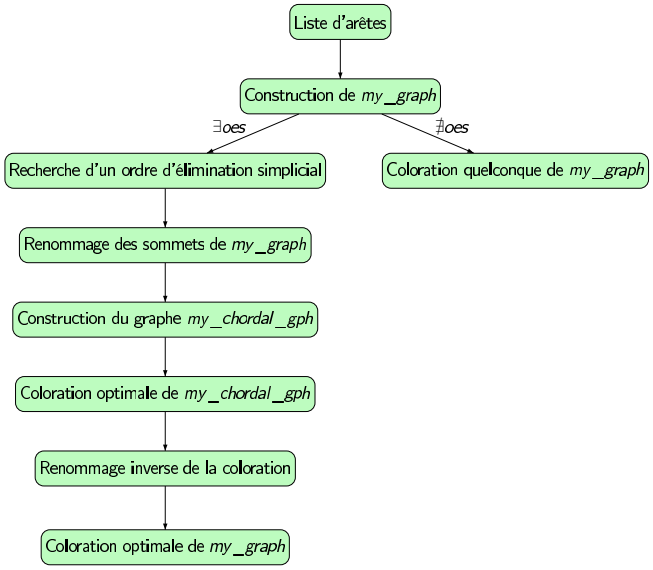


Fig. 4. Démarche générale de la coloration.

4.3 Recherche d'un ordre d'élimination simplicial

Lorsque le graphe est triangulé, connaître un ordre d'élimination simplicial du graphe permet d'obtenir une coloration optimale. Pour le trouver nous utilisons l'algorithme 1 (*cf.* section 3.3). Dans la définition suivante, la fonction (**sv_search_aux** l l') permet de trouver un sommet simplicial étant donnés les sommets de la liste l et les arêtes de la liste l'. La fonction (**remove** x l) permet de supprimer le sommet x de la liste l, et la fonction (**rm** n l) supprime de la liste l tous les couples dont l'une des composantes est n.

La définition de la fonction **peo_search_aux** n'est pas structurelle puisqu'elle repose sur la décroissance de la longueur de la liste. Aussi nous utilisons la construction `Function`. Il suffit ensuite de prouver que la longueur de la liste décroît bien à chaque itération. La correction de l'algorithme de recherche d'ordre d'élimination simplicial consiste à montrer que s'il existe un ordre d'élimination simplicial pour un graphe alors l'algorithme en trouve un.

```
Function peo_search_aux (l : list nat) (l' : list (nat×nat)) {measure length l} :
list nat :=
  match l with
  | nil ⇒ nil
  | _ ⇒ (sv_search_aux l l') : : (peo_search_aux (remove eq_nat_dec
                    (sv_search_aux l l') l) (rm (sv_search_aux l l') l'))
  end.
```

Definition **peo_search** g : list nat :=**peo_search_aux** (vertices g) (edges g).

Lemma **peo_peo_search** : ∀ g, (∃ l, peo l g) ⇒ **peo** (**peo_search** g) g.

4.4 Algorithme de coloration gourmande

Une fois l'ordre d'élimination simplicial trouvé, il n'est pas encore possible de définir une structure de type *Chordal_graph* puisque l'ordre d'élimination trouvé n'est pas croissant. Il faut donc renommer les sommets de façon à ce que l'ordre d'élimination corresponde à une liste croissante d'entiers. La fonction inverse doit aussi être spécifiée afin de pouvoir renommer les sommets après que la coloration du graphe ait été réalisée. Cette phase est périlleuse mais peu intéressante, c'est pourquoi elle n'est pas détaillée ici. Pour la consulter le lecteur pourra se rapporter à la page web du développement complet.

Une coloration est représentée par une liste de couples (s, c) où c représente la couleur (représentée par un entier) affectée au sommet s. L'algorithme de coloration gourmande construit la coloration au fur et à mesure du parcours de la liste des sommets du graphe. Au moment de la coloration d'un sommet, il est nécessaire de connaître les couleurs affectées aux sommets déjà colorés pour pouvoir choisir la couleur du sommet courant. Il faut donc stocker la coloration dans un accumulateur *col* qui est initialisé par la liste vide. La fonction (**get_available_color** *g x col*) permet de rechercher dans le graphe g la plus petite couleur n'étant affectée à aucun voisin du sommet x par la coloration partielle *col*.

```
Fixpoint coloring_aux (g : graph) (l : list nat) (col : list (nat×nat)) {struct l} :
list (nat×nat) :=
  match l with nil ⇒ col
  | x : : l'' ⇒ coloring_aux g l'' ((x, get_available_color g x col) : : col)
  end.

Definition graph_coloring (g : graph) := coloring_aux g (vertices g) nil.
```

4.5 Propriétés prouvées

Nous avons prouvé en Coq deux familles de propriétés concernant d'une part la correction d'une coloration, et d'autre part l'optimalité de l'algorithme de coloration gourmande dans les graphes triangulés.

Le lemme **is_coloring_graph_coloring** est le lemme de correction de la coloration gourmande. Il établit que pour tout graphe g, la coloration calculée par la fonction de coloration gourmande **graph_coloring** est bien une coloration valide de g. Une coloration valide est définie par le prédicat **is_coloring**. Soit une coloration *col* d'un graphe g. Soit k tel que les couleurs de *col* sont numérotées de 1 à k. Alors, *col* est valide si et seulement si tout sommet de g possède une et une seule couleur comprise entre 1 et k (lignes (1) à (3)), et si tout couple de sommets formant une arête d'interférence est coloré par deux couleurs distinctes (ligne (4)).

Inductive **is_coloring** : graph → list (nat×nat) → Prop :=
coloring_cons : ∀ (g : graph) (col : list (nat×nat)),
(1) (∀ (x : nat), *In* x (vertices g) ⇔ ∃ c, *In* (x,c) col)⇒
(2) **no_fst_dup** col ⇒
(3) (∀ (x cx : nat), *In* (x,cx) col ⇒ 1 ≤ cx) ⇒
(4) (∀ (x y cx cy : nat), *In* (x,y) (edges g)⇒ *In* (x,cx) col ⇒
In (y,cy) col ⇒ cy ≠ cx) ⇒
is_coloring g col.

Lemma **is_coloring_graph_coloring** : ∀ g, **is_coloring** g (graph_coloring g).

De plus, nous validons *a posteriori* les colorations calculées par le solveur externe que nous avons utilisé. Étant donné un graphe g, la validation *a posteriori* d'une coloration *col* calculée par un solveur externe consiste à vérifier en Coq le lemme **is_coloring** *g col*. Nous vérifions ainsi la même propriété que celle actuellement vérifiée dans CompCert.

Le lemme suivant est utile pour prouver l'optimalité de la coloration gourmande. Soit *my_peo* l'ordre d'élimination simplicial trouvé, et *my_chordal_gph* le graphe triangulé (de la structure de type Chordal_graph) obtenu à partir de *my_peo* après renommage des sommets. Le lemme **is_coloring_renaming** établit que si *col* est une coloration du graphe triangulé *my_chordal_graph*, alors la coloration obtenue par renommage de la coloration *col* est une coloration du graphe initial *my_graph*. Dans ce lemme, **coloring_renaming** est la fonction qui renomme une coloration afin de rétablir la numérotation des sommets du graphe d'origine.

Lemma **is_coloring_renaming** : ∀ (col : list (nat×nat)),
is_coloring my_chordal_gph col ⇒
is_coloring my_graph (**coloring_renaming** col (*rev* my_peo)).

Le lemme **coloring_optimality** établit l'optimalité de la coloration gourmande. L'optimalité consiste à vérifier que toute coloration valide du graphe utilise au moins autant de couleurs que celle renvoyée par l'algorithme, ou de manière équivalente que la plus grande couleur utilisée par toute coloration est supérieure à la plus grande couleur renvoyée par la coloration de l'algorithme. La fonction **max_color** renvoie le maximum des deuxièmes composantes d'une liste d'entiers (donc ici la couleur maximale de la coloration).

Lemma **coloring_optimality** : ∀ (col : list (nat×nat)),
is_coloring my_graph col→
max_color (**coloring_renaming** (**graph_coloring** my_chordal_gph)
(*rev* my_peo)) ≤ **max_color** col.

La preuve [Wes00] repose sur la propriété suivante : Si $(x_1, \ldots x_{n(G)})$ est un ordre d'élimination simplicial de G, et si l'algorithme de coloration gourmande est appliqué selon l'ordre des sommets $(x_{n(G)}, \ldots x_1)$ alors la coloration est optimale.
Soit G un graphe triangulé et x un ordre d'élimination simplicial de G. Soit i appartenant à $\{1, \ldots n(G)\}$. x étant un ordre d'élimination simplicial, x_i est un

sommet simplicial du graphe induit par $\{x_i, \ldots x_n\}$ c'est-à-dire du graphe induit par les sommets déjà colorés et x_i. Ainsi, le graphe induit par x_i et ses voisins déjà colorés est une clique. Soit t_i la taille de cette clique. La couleur affectée à x_i est donc inférieure ou égale à t_i. Cette inégalité étant valide pour tout i, la plus grande couleur utilisée est inférieure ou égale à la taille de la plus grande clique. Ainsi, nous obtenons l'inégalité $\chi(G) \leq \omega(G)$. De plus, l'inégalité inverse est évidente puisqu'il faut au moins autant de couleurs pour colorer G que pour colorer sa plus grande clique induite, ce qui permet de déduire l'égalité de $\chi(G)$ et $\omega(G)$ et donc l'optimalité de la coloration obtenue.

Notre développement Coq représente environ 10000 lignes de code. Les spécifications et les énoncés des preuves représentent respectivement 4% et 12% du développement. Environ 360 lemmes ont été prouvés. La principale difficulté a été de définir de nombreuses structures de données ainsi que des ordres sur ces structures. Le mécanisme d'extraction automatique de Coq a permis de générer un programme Caml effectuant la coloration gourmande d'un graphe triangulé. Ce programme représente 400 lignes de Caml.

Ce développement Coq a été l'occasion d'utiliser la construction `Function` afin de définir des fonctions récursives non structurelles. Enfin, nous avons été confronté à la lenteur du typeur de Coq. En effet, sur certains lemmes, Coq passait de 30 à 60 minutes à valider la fin de la preuve (la ligne `Qed.`).

Conclusion

Cet article a présenté la vérification formelle en Coq d'un algorithme exact de coloration avec préférences de graphe dédié à l'allocation de registres du compilateur certifié CompCert. L'algorithme est composé de deux phases : 1) la coloration gourmande dont nous avons prouvé la correction dans les graphes quelconques ainsi que l'optimalité dans les graphes triangulés, et 2) une étape de programmation linéaire en nombre entiers qui est résolue par un solveur externe et dont le résultat est validé *a posteriori*.

Afin d'améliorer cette validation *a posteriori*, nous cherchons actuellement à spécifier en Coq la notion de programme linéaire (via une bibliothèque dédiée que nous comptons développer), qui serait toujours résolu par un solveur externe, mais qui serait vérifié par le programme linéaire de Coq. Il s'agit donc de construire la coloration de façon interne à Coq, et de prouver en Coq que toute solution valide du programme linéaire correspond à une coloration valide du graphe d'interférences.

À plus long terme, nous souhaitons vérifier formellement d'autres méthodes de recherche opérationnelle, qui seraient utiles pour améliorer notre allocation de registres et faciliter l'utilisation des méthodes d'optimisation dans les développements de programmes certifiés.

Références

[AG01] Andrew W. Appel and Lal George. Optimal spilling for CISC machines with few registers. In *SIGPLAN Conference on Programming Language Design*

and Implementation, pages 243–253, 2001.

[AG05] Andrew W. Appel and Lal George. 27,921 actual register-interference graphs generated by standard ML of New Jersey, version 1.09 – `http ://www.cs.princeton.edu/~appel/graphdata/`, 2005.

[And03] Christian Andersson. Register allocation by optimal graph coloring. In *Compiler Construction (CC)*, pages 33–45, 2003.

[BCT94] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *Transactions on Programming Languages and Systems (TOPLAS)*, 16(3) :428 – 455, 1994.

[BDL06] Sandrine Blazy, Zaynah Dargaye, and Xavier Leroy. Formal verification of a C compiler front-end. In *FM 2006 : Int. Symp. on Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2006.

[Cha82] G J Chaitin. Register allocation and spilling via graph coloring. *Symposium on Compiler Construction*, 17(6) :98 – 105, 1982.

[FG65] D.R. Fulkerson and O.A. Gross. Incidence matrices and interval graphs. *Pac. J. Math.*, 15 :835–855, 1965.

[GA96] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Trans. Program. Lang. Syst.*, 18(3) :300–324, 1996.

[Gav72] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM J. Comput.*, 1 :180–187, 1972.

[GH07] Daniel Grund and Sebastian Hack. A fast cutting-plane algorithm for optimal coalescing. volume 4420 of *Lecture Notes in Computer Science*, pages 111–125. Springer, 2007.

[GW96] David W. Goodwin and Kent D. Wilken. Optimal and near-optimal global register allocations using 0-1 integer programming. *Softw. Pract. Exper.*, 26(8) :929–965, 1996.

[Ilo02] Ilog. Ilog ampl cplex system, version 8.0, user's guide, 2002.

[Ler06] Xavier Leroy. Formal certification of a compiler back-end or : Programming a compiler with a proof assistant. *33rd symposium Principles of Programming Languages*, pages 42–54, 2006.

[NPP07] V. Krishna Nandivada, Fernando Magno Quintão Pereira, and Jens Palsberg. A framework for end-to-end verification and evaluation of register allocators. In *Static Analysis, 14th Int. Symp., SAS 2007, August, 2007, Proc.*, volume 4634 of *Lecture Notes in Computer Science*, pages 153–169. Springer, 2007.

[Oho04] Atsushi Ohori. Register allocation by proof transformation. *Science Computer Programming*, 50(1-3) :161–187, 2004.

[PP05] Fernando Magno Quintão Pereira and Jens Palsberg. Register allocation via coloring of chordal graphs. *Programming Languages and Systems, 3rd Asian Symp., APLAS 2005, Japan, November, 2005, Proc.*, 3780 :315–329, 2005.

[Wes00] Douglas B. West. *Introduction to Graph Theory (2nd Edition)*. Prentice Hall, August 2000.

Session groupe de travail MTV2

Méthodes de Tests pour la Validation et la Vérification

Constraint-Based Software Testing*

Sebastian Bardin[1], Bernard Botella[1], Frédéric Dadeau[3], Florence Charreteur[2], Arnaud Gotlieb[2], Bruno Marre[1], Claude Michel[4], Michel Rueher[4], and Nicky Williams[1]

[1] CEA LSL, 91191 Gif sur Yvette
[2] INRIA Rennes-Bretagne Atlantique, LANDE Project, 35042 Rennes Cedex
[3] University of Franche-Comté, LIFC / INRIA CASSIS Project, 25030 Besançon
[4] University of Nice-Sophia Antipolis, CeP project, Ecole Polytechnique, 06903 Sophia Antipolis cedex

Abstract. Constraint-Based Testing (CBT) is the process of generating test cases from programs or models by using the Constraint Programming technology. Recently, this method received much attention due to several Research projects launched in France and abroad. This paper aims at presenting the main CBT tools developped by four Research labs: CEA *Laboratoire de Sûreté des Logiciels*, INRIA *Lande research team*, *Laboratoire d'Informatique de Franche-Comté*, and *CeP team* of University of Nice–Sophia Antipolis. The paper concludes by drawing some perspectives on open problems in CBT.

1 Introduction

These last years, much attention has been devoted to the use of Constraint Programming techniques in the automation of program verification. In 2000, Podelski outlined [13] that program verification could be seen as an instanciation of constraint solving while Flanagan proposed theoretic foundations to constrained interpretations of programs [7]. Program verification often reduces to the problem of showing that a constraint system is satisfiable or unsatisfiable. For example, showing a particular property at a given point in a source code leads to solve a constraint system that characterizes a path through a particular state of the program. Recent work focussed on the use of constraint solvers for bounded model checking of source code, but it is in automated software testing that constraint solving for program verification has reached a certain level of maturity.

In this domain, France has been a pioneer since the early work of Marre, Dick and Faivre[11, 6]. Several research projects were launched such as the *RNTL projects INKA (2000-2002) and DANOCOPS (2004-2006)*, that initiated the study of constraint-based testing techniques for embedded C and C++ programs. These projects resulted not only in the development of the INKA tool which was the first to use constraint programming in automatic structural test

* partially granted by the SESUR CAVERN project ref. ANR-07-SESU-003.

data generation for C programs [9, 10] but also to the development of prototypes tools for specification notations such as OCL and JML[3]. The *ACI V3F project (2003-2006)* studied the problems of floating-point computations within various Constraint-Based Testing approaches. Thanks to this project, we started working together around the development of specialized floating-point constraint solvers [2]. Another result of the *ACI V3F* was the first international CSTVA workshop (Constraints in Software Testing, Verification and Analysis), we organized in Nantes in 2006. CSTVA[5] brought together people of distinct communities to discuss fruitful ideas around the use of constraints in program testing. From these discussions emerged the idea of exploiting abstractions to enhance current constraint propagation in solvers dedicated to program testing. The current *SESUR–2007 CAVERN (Constraints and Abstractions for program VERificatioN) project* which started early 2008 aims to explore the combination of Constraint Programming and Abstractions techniques for automated testing of programs. The originality of this project lies in the use of abstractions to develop dedicated propagation-based constraint solvers targeted to handle specific features of imperative programs such as iterative computations, references, dynamic structures and floating-point computations.

Since 2000, several Constraint-Based Testing tools have been developed to investigate constrained models of programs or specification models with the goal of generating test cases against various testing objectives. These tools share some characteristics such as being based on constraint propagation, abstract domains computations and labeling, The purpose of this short paper is to give an overview of some of these tools: PathCrawler and Euclide for C, Cpbpv for Java, Osmose for executable code, Jaut for Bytecode Java,Gatel for Lustre, JMLTT for JML.

2 Tools

- PathCrawler [14] is a tool prototype developed by the LSL laboratory of CEA List. PathCrawler automatically generates test-case inputs guaranteeing full structural coverage of the C function under test: all feasible paths, all reachable branches, all k-paths,... PathCrawler runs an automatically-instrumented version of the function under test on each test-case as soon as it is generated in order to recover the path covered by this test-case and ensure that the next test-case covers a path which is not covered yet. Test-case generation is implemented using constraint logic programming. To ensure that PathCrawler can be used on real-life programs, current work focusses on the treatment of called functions, the treatment of the precondition on the effective input parameters under which the function is to be tested, the treatment of pointer casts, the treatment of integer overflows and floating-point numbers and the early detection of infeasible path prefixes.
- Euclide [8] is a new Constraint-Based Testing tool for verifying safety-critical C programs. By using a mixture of symbolic and numerical analyses

[5] http://www.irisa.fr/manifestations/2006/CSTVA06

(namely static single assignment form, constraint propagation, integer linear relaxation and search-based test data generation), it addresses three distinct applications in a single framework: structural test data generation, counter-example generation and partial program proving. The core algorithm of the tool takes as input a C program and a point to reach in the code. As a result, it outcomes either a test datum that reaches the selected point, or an "unreachable" indication showing that the selected point is unreachable. Optionally, the tool takes as input additional safety properties that can be given under the form of pre/post conditions in ACSL or assertions directly written in the code. In this case, EUCLIDE can either prove that these properties or assertions are verified or find a counter-example when there is one. As these problems are undecidable in the general case, EUCLIDE only provides a semi-correct procedure (when it terminates, it provides the right answer) for them. Current Research works around the tool focus on modular integers and floating-point computations and better constraint solving procedures based on relational abstract domains computations.

– CPBPV [5] is a novel constraint-programming framework for bounded program verification. The CPBPV framework uses constraint stores to represent the specification and the program and explores execution paths non-deterministically. The input program is partially correct if each constraint store so produced implies the post-condition. CPBPV does not explore spurious execution paths as it incrementally prunes execution paths early by detecting that the constraint store is not consistent. CPBPV is parameterized with a list of solvers which are tried in sequence, starting with the least expensive and less general. Experimental results often produce orders of magnitude improvements over earlier approaches, running times being often independent of the variable domains. Moreover, CPBPV was able to detect subtle errors in some programs while other frameworks based on model checking have failed.
– OSMOSE [1] is a tool dedicated to machine code analysis. Potential applications include validation of COTS and mobile codes as well as malware comprehensive analysis. There are two main specific challenges in machine code analysis: low-level semantics of data (bitwise instructions, machine arithmetics with overflows and flags, etc.), and unstructured control-flow (e.g. `goto x`, where `x` is only known at run-time). From a test data generation perspective, OSMOSE follows both the path-based approach and the concolic execution paradigm, mixing concrete execution and symbolic reasoning. The tool is geared toward full coverage of branches or instructions. The main original features of the test data generation technology are the following: (1) path predicates are expressed in the bit-vector theory, and a novel CLP-based approach has been developped to solve such constraints; (2) the concolic paradigm has been adapted to unstructured control-flow, and it appears to be both a very powerful and easy to implement mean of recovering a high-level view (CFG) of the program. (3) specific heuristics have been designed to discard *a priori* paths redundant with the current coverage objective.

- JAUT (Java Automatic Unit Testing) [4] is a test data generator for programs in bytecode Java. Given a bytecode instruction of the method under test, it aims to find an input memory state (values of the parameters and of the instances in the heap) to cover this instruction. Repeating this process, the structural coverage criterion of all-the-statements can be fulfilled. In JAUT, bytecode instructions are modelled with constraints, as well as the conditions that permit to reach the goal. An input memory state to cover the goal instruction can so be found by constraint solving. The tool currently deals with arithmetic operations on integers, heap manipulation and conditional instructions. The implementation of a strategy to avoid enumerating all the paths that lead to the goal until finding an executable one is in progress. Current work also includes dealing with polymorphic function calls.
- GATEL [12] is a test environment for synchronous models of reactive systems described in Lustre/SCADE. The core of the tool is a CLP interpretation of the Lustre language, together with a resolution procedure dedicated to the underlying linear temporal logic. This framework allows to automate a wide range of usual testing activities. Initially designed for generating test sequences according to a test objective, GATEL also addresses symbolic simulation, model debug, conformance checking, coverage analysis/completion, test suite evaluation. Current work is twofold: updating the CLP interpretation and procedure to the latest version of the SCADE Suite (enriched with built-in state machines), upgrading the procedure with powerful abstractions to scale up to real-life industrial models.
- JML-TESTING-TOOLS is an automated animation and test generation tool based on JML annotations [3]. The animation feature is used to simulate the execution of the model, using constraint solving techniques, in order to ensure the conformance of the model behaviors w.r.t. the informal requirements. The test generation part works by first computing boundary test targets, satisfying JML preconditions of the Java methods, according to specific object-oriented data coverage criteria, such as null pointers, aliasings, etc. coupled with a boundary analysis of numerical values. It then builds complete execution sequences, in terms of method invocations, using the symbolic animation of the model, in order to cover these targets, thus producing the test cases.

3 Perspectives

Scalability is the main challenge that the tools presented here have to face to. Dealing with more than hundred of thousands lines of code, with dynamic constructions such as huge dynamic data structures, with non-linear numerical constraints extracted from complex statements are some of the problems we have to deal with. Research works were launched to address these problems in all the research teams mentionned in this paper. Next step will be the dissemination of the *Constraint Programming technology in Program Verification* to Industry.

References

1. Sebastien Bardin and Philippe Herrmann. Structural testing of executables. In *1th Int. Conf. on Software Testing, Verification and Validation (ICST'08)*, pages 22–31, 2008.
2. B. Botella, A. Gotlieb, and C. Michel. Symbolic execution of floating-point computations. *The Software Testing, Verification and Reliability journal*, 16(2):pp 97–121, June 2006.
3. F. Bouquet, F. Dadeau, and B. Legeard. Automated boundary test generation from JML specifications. In *FM'06, 14th Int. Conf. on Formal Methods*, volume 4085 of *LNCS*, pages 428–443, Hamilton, Canada, August 2006. Springer-Verlag.
4. F. Charreteur and A. Gotlieb. Raisonnement contraintes pour le test de bytecode java. In *quatrimes Journes Francophones de Programmation par Contraintes (JFPC'08)*, pages 11–20, Nantes, France, Juin 2008.
5. H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: A constraint-programming framework for bounded program verification. In *Proc. of CP2008*, LNCS 5202, pages 327–341, 2008.
6. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. of the FME'03: Industrial Strength Formal Methods*, LNCS 670, 1993.
7. Cormac Flanagan. Automatic software model checking via constraint logic. *Sci. Comput. Program.*, 50(1-3):253–270, 2004.
8. A. Gotlieb. Euclide: A constraint-based testing platform for critical c programs. In *2th International Conference on Software Testing, Validation and Verification (ICST'09)*, Denver, CO, Apr. 2009.
9. A. Gotlieb, B. Botella, and M. Rueher. A clp framework for computing structural test data. In *Proceedings of Computational Logic (CL'2000)*, LNAI 1891, pages 399–413, London, UK, July 2000.
10. A. Gotlieb, T. Denmat, and B. Botella. Goal-oriented test data generation for pointer programs. *Information and Software Technology*, 49(9-10):1030–1044, Sep. 2007.
11. Bruno Marre. Toward Automatic Test Data Set Selection using Algebraic Specifications and Logic Programming. In Koichi Furukawa, editor, *Proc. of the Eight ICLP*, pages 202–219, Paris, Jun. 1991. MIT Press.
12. Bruno Marre and Benjamin Blanc. Test selection strategies for lustre descriptions in gatel. *Electronic Notes in Theoretical Computer Science*, 111:93 – 111, 2005.
13. Andreas Podelski. Model checking as constraint solving. In Jens Palsberg, editor, *Proceedings of SAS: Static Analysis Symposium*, volume 1824 of *LNCS*, pages 22–37. Springer-Verlag, 2000.
14. N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *In Proc. Dependable Computing - EDCC'05*, pages 281–292, 2005.

Simulated Annealing Applied to Test Generation: Landscape Characterization and Stopping Criteria[1]

Hélène Waeselynck, Pascale Thévenod-Fosse, Olfa Abdellatif-Kaddour

LAAS; CNRS; Université de Toulouse
7 avenue du Colonel Roche, F-31077 Toulouse, France
{Helene.Waeselynck, Pascale.Thevenod}@laas.fr

Abstract. This paper investigates a measurement approach to support the implementation of Simulated Annealing (SA) applied to test generation. SA, like other metaheuristics, is a generic technique that must be tuned to the testing problem under consideration. Finding an adequate setting of SA parameters, that will offer good performance for the target problem, is known to be difficult. Our measurement approach is intended to guide the implementation choices to be made. It builds upon advanced research on how to characterize search problems and the dynamics of metaheuristic techniques applied to them. Central to this research is the concept of landscape. Existing measures of landscape have mainly been applied to combinatorial problems considered in complexity theory. We show that some of these measures can be useful for testing problems as well. The diameter and autocorrelation are retained to study the adequacy of alternative settings of SA parameters. A new measure, the Generation Rate of Better Solutions (GRBS), is introduced to monitor convergence of the search process and implement stopping criteria. The measurement approach is experimented on various case studies, and allows us to successfully revisit a problem issued from our previous work on testing control systems.

Keywords: software testing, metaheuristic search, simulated annealing, measurement.

[1] This paper has been published in: Empirical Software Engineering, Vol. 12, no. 1, pp.35-63, Springer, feb. 2007.

Coverage-biased Random Exploration of Models

Alain Denise[1], Marie-Claude Gaudel[1], Sandrine-Dominique Gouraud[1], Richard Lassaigne[2], Johan Oudinet[1], Sylvain Peyronnet[1]

[1] Université de Paris-Sud 11 & CNRS,
LRI, Bat. 490, F-91405 Orsay Cedex, France
{denise, mcg, oudinet, syp}@lri.fr
[2]Université Paris Diderot 7 & CNRS, Equipe de Logique,
UFR de mathématiques case 7012, site Chevaleret, 75205 Paris Cedex 13
lassaigne@logique.jussieu.fr

Extended Abstract[1]

We describe a set of methods for randomly drawing traces in large models either uniformly among all traces, or with a coverage criterion as target.

Classical random walk methods have some drawbacks. In case of irregular topology of the underlying graph, uniform choice of the next state is far from being optimal from a coverage point of view. For instance, in Figure 1, when considering traces of length 3, trace $b.e.f$ is followed with

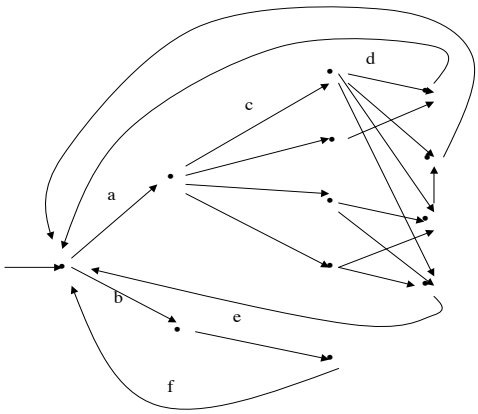


Fig. 1. Some example of irregular topology

probability 0.5 while trace $a.c.d$ has probability 0.03125 only. Moreover, for the same reason, it is generally not practicable to get an estimation of the coverage obtained after one or several random walks: it would require some complex global analysis of the model topology. We present here some methods that give up the uniform choice of the next state. They bias this choice according to the number of traces, or states, or transitions, reachable via each successor. The methods rely upon techniques for counting and drawing uniformly at random words in regular languages as defined by Flajolet et al. [3] and first implemented by Denise et al. [1]. These techniques have, in the

[1] the full paper has been published in the proceedings of the 4th ETAPS Model Based Testing Workshop [4]

considered cases, a linear complexity in the size of the underlying automata [7], thus they allow dealing with rather large models.

First, taking into account the number of traces starting from a state, it is shown how it is possible to ensure a uniform probability on traces of a given length, or below a given length.

However, even linear complexity techniques cannot cope with very large models. But it is possible to exploit the fact that most of them are the result of the concurrent composition of several components, i.e a product, synchronised or not, of several models. We show how it is possible to combine local uniform drawings of traces, to obtain some global uniform random sampling, without constructing the global model.

Considering coverage of other elements of the model than traces, such as states or transitions, is done by maximising the minimum probability to reach such an element (as first suggested by Thévenod-Fosse et al. [6]) thus biasing random exploration toward classical coverage criteria such as state coverage or transition coverage, or less classical ones. Thus the probability of reaching a given coverage criterion after a certain number of drawings can be assessed, leading to a notion of randomised coverage satisfaction.

A common basis of these various pieces of work is that they are based upon powerful techniques on counting and drawing uniformly at random combinatorial structures. It is our belief that these techniques could bring much to simulation, model based testing, structural testing or model-checking.

References

1. Denise A., Isabelle Dutour and Paul Zimmerman. *CS: a MuPAD package for Counting and Randomly Generating Combinatorial Structures*. Proceedings of FPSAC'98, 195-204. (Also in MathPAD **8(1)** 1998.)
2. Denise A., M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne and S. Peyronnet. *Uniform Random Sampling of Traces in very Large Models*. 1st Int. ACM Workshop on Random Testing (2006), 10 -19. http://doi.acm.org/10.1145/1145735.1145738
3. Flajolet Ph., and P. Zimmermann, and B. Van Cutsem. *A Calculus for the Random Generation of Labelled Combinatorial Structures*, Theoretical Computer Science, **132** (1994), 1-35.
4. Gaudel M.-C., A. Denise, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet, *Coverage-biased Random Exploration of Models*, Proceedings of the Fourth Workshop on Model Based Testing (MBT 2008), ENTCS, **220(1)**, 3-14. http://dx.doi.org/10.1016/j.entcs.2008.11.002
5. Oudinet J., *Uniform random walks in very large models*. 2nd Int. ACM workshop on Random Testing (2007) 26-29. http://doi.acm.org/10.1145/1292414.1292422
6. Thévenod-Fosse P. and H. Waeselynck. *An investigation of software statistical testing*. Journal of Software Testing, Verification and Reliability, **1(2)**, (1991), 5-26.
7. van der Hoeven J. *Relax, but dont be too lazy*, Journal of Symbolic Computation, **34(6)** (2002), 479-542.

Session groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

On the Use of Formal Techniques to Support Model Evolution

Mens Tom[1], Van Der Straeten Ragnhild[2]

1 Service de Génie Logiciel, Université de Mons-Hainaut, Belgium

2 Systems and Software Engineering Lab, Vrije Universiteit Brussel, Belgium

Abstract

Model-driven engineering (MDE) is an emerging software engineering discipline that relies on *model transformation*. Model transformations can be very diverse, and encompass, among others, the following techniques: *code generation*, *reverse engineering*, *model refinement* and *model refactoring*. Due to the inherently volatile nature of all kinds of software artefacts, and models in particular, all existing and future MDE approaches should explicitly take into account the inevitable process of *model evolution*. In this paper, we explain why formal support for model evolution is needed. We motivate this by using the formalism of *description logics* to support the activity of *model inconsistency management*, and by using the formalism of *graph transformation* to support the activity of *model refactoring*.

1. Introduction

Model-driven engineering (MDE) is an approach to software development where the primary focus is on models, as opposed to source code. Models are built representing different views on a software system. Models can be reverse engineered, refined, used to generate executable code, and many more. The ultimate goal is to raise the level of abstraction, and to develop more complex software systems by manipulating models only. The manipulation of models is achieved by means of *model transformation*, which is considered to be the heart and soul of model-driven engineering [1].

Any software system that is deployed in the real-world is subject to evolution [2]. Because of this, it is crucial for any software development process to provide support for software evolution. This includes support for version control, traceability management and change impact analysis, change propagation, inconsistency management, All of these activities need to be supported throughout all software development phases, including the modeling phase. This is depicted schematically in Figure 1.

Today, only limited support is available in MDE tools for these evolution activities. There is a need for more sophisticated formalisms, techniques and associated tools supporting model evolution. In this paper, we focus on the activities of model refinement, model refactoring and model inconsistency management in particular. We outline two formally founded approaches that allows us to show how these activities are related, and how tool support for these activities can be achieved.

2. Preliminaries

Two essential techniques to evolve and transform models in a disciplined way are *model refinement* and *model refactoring*.

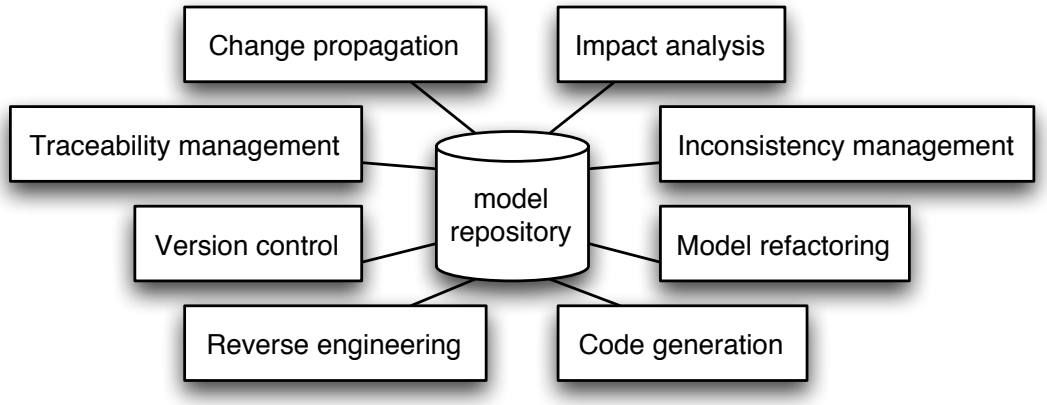


Figure 1: Model evolution activities

- Model refinement is used to transform a model into a refined, more detailed version. A typical example in the model-driven architecture (MDA) approach would be the transformation of a platform-independent model (PIM) into one or more platform-specific models (PSM).
- Model refactorings [3, 4, 5, 6] restructure a model while preserving its behavioural properties. The aim is to improve the models by making them more generic, modular, reusable, and so on.

Another crucial activity, tightly related to model transformation, is the ability to *maintain consistency* of models or, alternatively, to *manage inconsistencies* in a more disciplined way. According to [7], an *inconsistency* is "a state in which two or more overlapping elements of different software models make assertions about aspects of the system they describe which are not jointly satisfiable."

- In the context of model refinement, various notions of consistency have been defined to express the consistency between a model and its refined versions. For example, Ebert and Engels [8] defined a notion of observation consistency and invocation consistency between state machines. Other notions of consistency can be found in [9, 10, 11].
- In the context of model refactoring, it is necessary to ensure that the refactored model preserves the behaviour of the original model.

In this paper, we briefly report on two formal approaches we have experimented with. We believe that such a formal foundation is crucial, as it will result in more precise and less ambiguous tool support.

The first approach, that will be explained in Section 3, relies on the formalism of *description logics* to formally the relation between behaviour consistencies of model refinements and behaviour preservation of model refactorings. To show the practical use of this formalism we developed a plug-in for a commercial UML CASE tool.

The second approach, that will be explained in Section 4, relies on the formalism of *graph transformation* to specify model refactorings in order to analyse dependencies and conflicts between them. Practical experiments have been performed in the state-of-the-art graph transformation tool AGG[1].

[1] `http://tfs.cs.tu-berlin.de/agg`

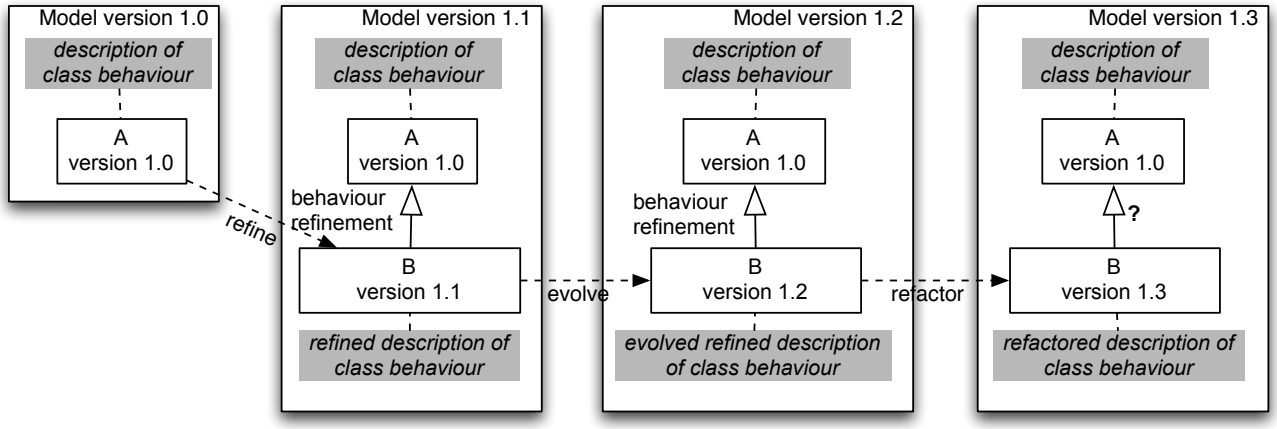


Figure 2: Evolution scenario of our motivating example.

3. First experiment: Description logics

3.1. Motivating example

A motivating example is depicted in Figure 2. A class `A` (version 1.0) is *refined* into a subclass `B` (version 1.1) in a behaviourally consistent way. This means that the behaviour of class `B` (expressed by means of a state machine or sequence diagram, for example) should specialise the behaviour of the class `A` under certain conditions. One such condition is Liskov's well-known substitutability principle. Other possible conditions are the notions of behaviour consistency as defined in [8]. By implementing these formal definitions in a tool, we can automatically detect whether model refinements are behaviourally consistent.

Now suppose that class `B` *evolves* into a new version 1.2, as illustrated in Figure 2. Then we would like to know whether or not the evolved behaviour of class `B` is still behaviourally consistent with class `A` of which it is a refinement.

Similarly, suppose that the behaviour of class `B` is *refactored* into a new version 1.3. Although the refactoring may modify the structure of the behavioural description models (e.g., state machines or interaction diagrams) for class `B`, we would like to guarantee that it does not affect the existing behaviour (by definition of refactoring). In other words, we need to check that the refactored version of class `B` is still behaviourally consistent with the original class `A`. Again, it is possible to formally specify this notion of behaviour preservation, and to verify it in an automated way. A worked out example of non-trivial model refactorings at the level of state machine diagrams has been given in [5, 12]. These model refactorings are similar to those that can be found in the research literature [3, 13].

3.2. Proposed tool chain

The tool setup that we propose is illustrated in Figure 3. It is an enhancement of the tool chain that we have described in [14] for checking inconsistencies between UML models. A plug-in for this tool chain in the UML CASE Tool Poseidon[2] was developed by Jocelyn Simmonds. The tool chain is basically an implementation of a set of logic facts and rules in the RACER Description Logics engine[3]. After extracting UML models from the CASE tool and translating them into facts in the logic repository, logic queries can be executed to detect and resolve particular inconsistencies on UML models.

Currently, we are extending the tool to perform model refactorings and model refinements. This needs

[2] `http://www.gentleware.com`
[3] `http://www.racer-systems.com/`

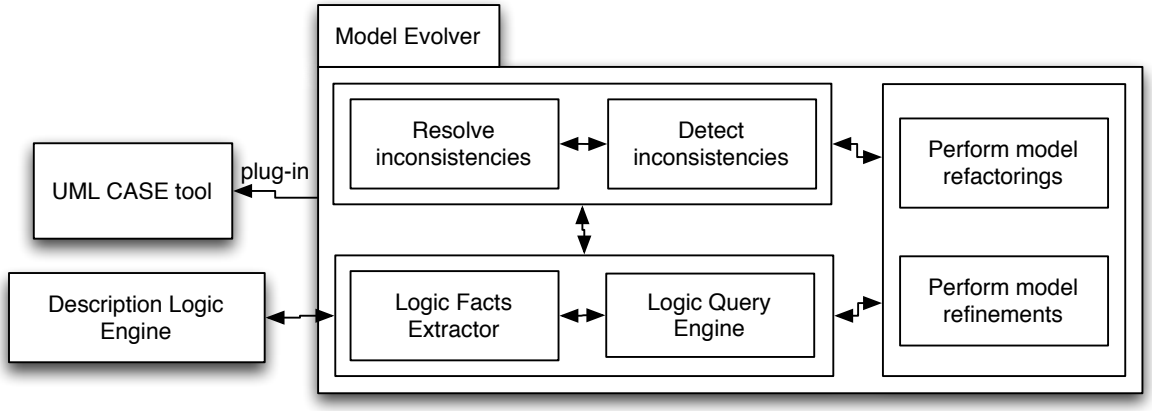


Figure 3: Tool setup

to be tightly integrated with the inconsistency detection mechanism, since we want to be able to ensure that both kinds of model transformation preserve the overal model consistency. In those cases where inconsistencies are encountered, we propose to execute a semi-automatic conflict resolution mechanism. A possible way to present the results of detected inconsistencies to the user is given in Figure 4.

3.3. Description logics

The formalism we have chosen to specify models, model transformations and model consistency is called *Description Logic* (DL). The reasons for choosing this formalism are manifold:

1. The declarative nature of logic is well suited to express models that are most often specified in a declarative way.

2. Logic reasoning algorithms are well understood due to their extensively studied, well-defined and sound semantics. First-order logic and theorem proving have been advocated by several authors for expressing software models and the derivation of inconsistencies from these models (e.g., [15, 16, 17]).

3. DL systems assume an open world semantics, which allows the specification of incomplete knowledge. This is for example useful, for modeling sequence diagrams which typically specify incomplete information about the dynamic behaviour of the system.

4. DL provides a *classification mechanism* that can be used to deduce implicit knowledge from the explicitly represented knowledge in the logic fact base.

5. DL is a decidable fragment of first-order logic.

6. Computationally efficient query languages exist for DL fact bases.

7. DL has specific reasoning capabilities that are implemented by sound and complete reasoning algorithms. These reasoning mechanisms include: subsumption, instance checking, relation checking, and concept satisfiability.

8. DL is well-suited to specify models that express the static structure of a software system. For example, Calí *et al.* [18] showed how to translate UML class diagrams into DL.

9. DL is suited to express, to a certain extent, the behaviour of a software application.

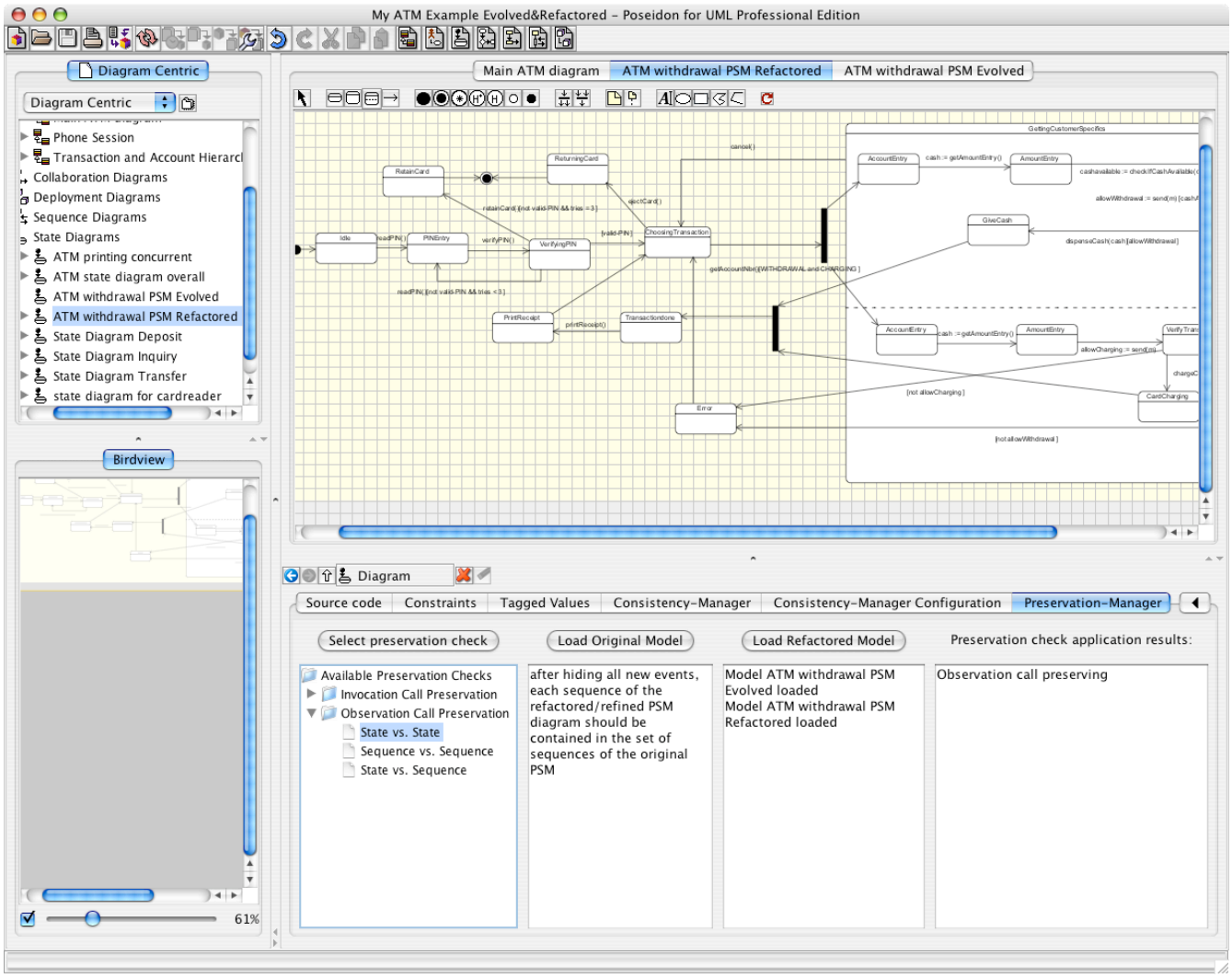


Figure 4: Screenshot of the Model Evolver plug-in.

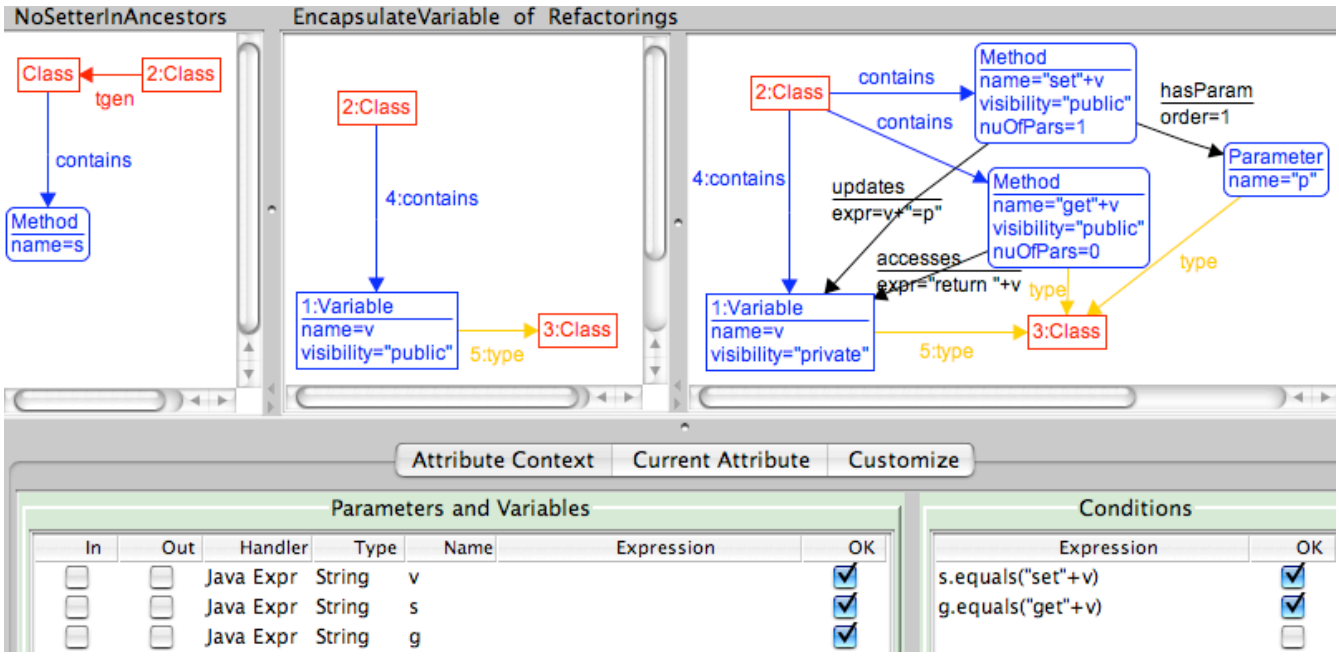


Figure 5: Graph transformation for the *Encapsulate Variable* refactoring.

Observe that some of the advantages of description logics are a direct result of restricting the formalism of first-order logic. As such, the language becomes decidable, but at the same time certain things are not possible to express in description logics. For the experiments we performed, however, this did not turn out to be a problem.

In this paper, we will not further explain the technical details of how UML models can be specified using description logics, or how inconsistencies betweeen different versions of these models can be specified. For the interested reader, we refer to our earlier work explaining all the details [11, 14, 5].

4. Second experiment: Graph transformation

While the description logics experiment explained above mainly focused on managing inconsistencies in and between different UML models, we did not use the formalism to specifiy model transformations themselves. Therefore, in our second experiment, we explored a formalism that allows us to specify model transformations, and model refactorings in particular, using the formalism of graph transformation theory. The goal was to exploit theoretical results to get a better insight, and hopefully also better tool support for, software refactoring.

4.1. Specifying refactorings

As a first step, we formally represented model refactorings as graph transformations. We used the graph transformation tool AGG for this purpose[4]. Figure 5 shows an example of a class diagram refactoring represented as a graph transformation. It consists of a left-hand side (LHS) in the upper middle pane, a right-hand side (RHS) in the upper right pane, and zero or more negative application conditions (NACs) that represent forbidden contexts. One of these NACs is shown in the upper left pane. It specifies that the ancestor classes should not contain a method whose name coincides with the name of the setter method. In addition, the bottom pane contain some Java expressions that impose extra constraints between variables used in the NAC, LHS and RHS.

[4]We have also used other graph transformation tools, such as Fujaba, to specify model refactorings.

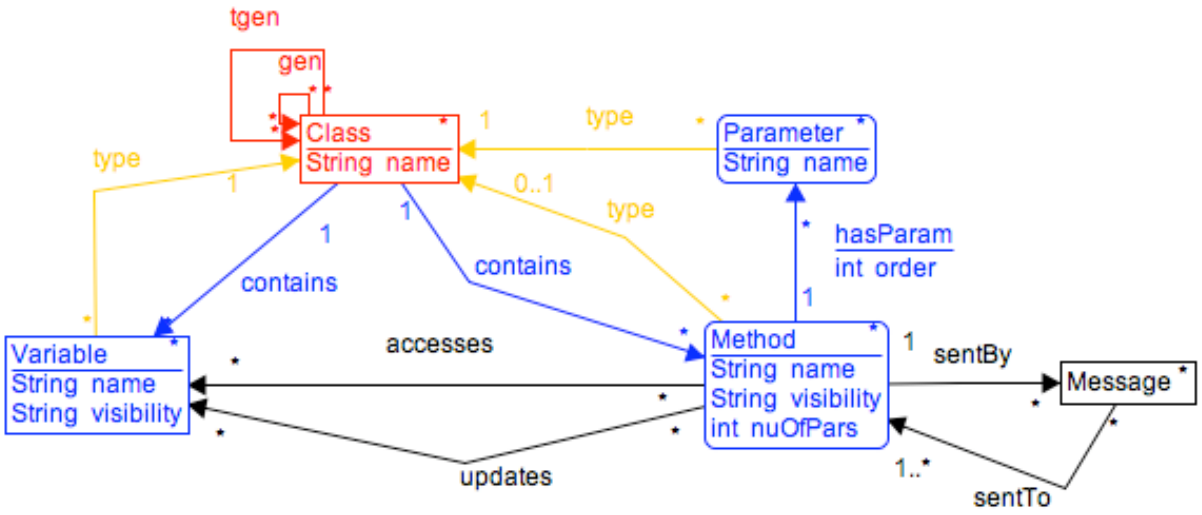


Figure 6: Type graph representing a simplified object-oriented metamodel.

The graphs themselves are directed, attributed and typed. The latter implies that all graphs conform to a type graph, which represents the object-oriented metamodel. An example of such a type graph is shown in Figure 6.

A direct extension of this work that we currently start to explore is the use of graph transformations to specify model refactorings for other types of UML diagrams such as sequence diagrams, activity diagrams and statecharts. The additional challenge here is that we would like to specify the refactoring transformations using a syntax that is close to the one that is used to specify the diagrams themselves. This visual notation that is specific to each type of UML diagram would then be translated automatically to the internal representation of attributed typed graph transformations.

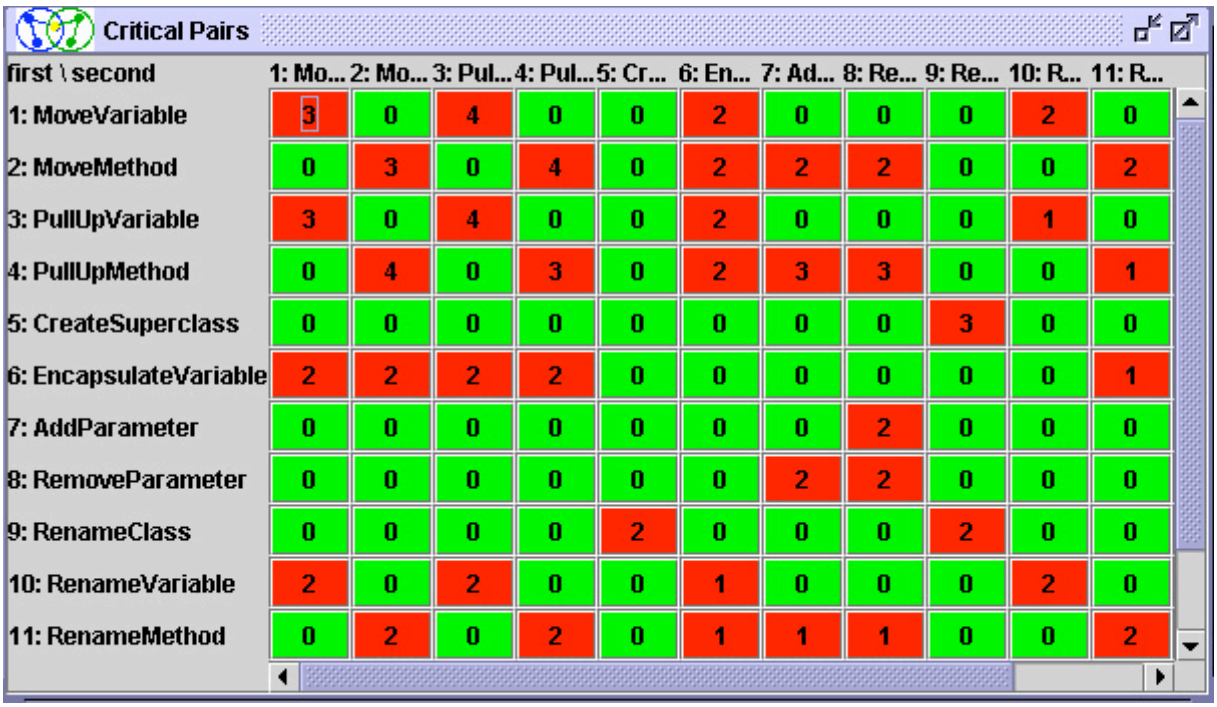
4.2. Detecting refactorings conflicts

In order to detect mutual exclusion relationships between model refactorings applied in parallel, we explored the formal technique of *critical pair analysis* [19, 20] in collaboration with Gabriele Taentzer of the Technical University of Berlin. A *critical pair* is detected between two particular refactorings when it is possible to find a minimal critical context graph to which both graph transformations can be applied in a conflicting (i.e., mutually exclusive) way.

Detecting all such critical pairs in a representative selection of 11 refactorings, resulted in the table of Figure 7. The numbers shown in each field of the table correspond to the actual number of potentially critical situations that can be computed between a given pair of refactorings. This number exceeds one if the two considered refactorings conflict in different ways.

A typical example is the critical pair *Move Method* versus *Pull Up Method* which gives rise to 4 potential conflicts. When the same method in the same source class is moved and pulled up at the same time, we clearly have a potential conflict. The same holds if a method with the same signature is moved and pulled up at the same time to the same target class. In a similar way, two other conflict situations can arise. The same reasoning can be done for all other detected critical pairs, and allows us to get a better insight in the interactions and conflicts between refactorings.

For more information about how we carried out these experiments, and a more detailed analysis of the results, we refer to [21, 22].



first \ second	1: Mo...	2: Mo...	3: Pul...	4: Pul...	5: Cr...	6: En...	7: Ad...	8: Re...	9: Re...	10: R...	11: R...
1: MoveVariable	3	0	4	0	0	2	0	0	0	2	0
2: MoveMethod	0	3	0	4	0	2	2	2	0	0	2
3: PullUpVariable	3	0	4	0	0	2	0	0	0	1	0
4: PullUpMethod	0	4	0	3	0	2	3	3	0	0	1
5: CreateSuperclass	0	0	0	0	0	0	0	0	3	0	0
6: EncapsulateVariable	2	2	2	2	0	0	0	0	0	0	1
7: AddParameter	0	0	0	0	0	0	0	2	0	0	0
8: RemoveParameter	0	0	0	0	0	0	2	2	0	0	0
9: RenameClass	0	0	0	0	2	0	0	0	2	0	0
10: RenameVariable	2	0	2	0	0	1	0	0	0	2	0
11: RenameMethod	0	2	0	2	0	1	1	1	0	0	2

Figure 7: Critical pair analysis of the refactoring specifications.

5. Conclusion

In order to provide integrated tool support for all activities of model evolution, a formal foundation is needed to specify models and their evolution, and to verify whether important properties (such as consistency) are preserved between different model versions.

We have positive experience with the formalism of description logics, and we are currently validating the proposed formalism by means of a UML CASE tool plug-in. The tool can be used to detect inconsistencies in evolving UML design models, and can also be used to check the preservation of behaviour between subsequent model versions.

We also have positive experience with graph transformation theory, which has been used to formally specify model refactorings in order to analyse potential dependencies, conflicts and unexpected interactions between those refactorings. In other work, the same formalism has also been proposed to prove preservation of certain properties (such as call preservation and access preservation [23]).

From our experience, it seems that different model evolution activities may benefit from different formalisations. Description Logic was well-suited to reason about model inconsistencies, while graph transformation was well-suited to reason about model refactorings. How both formalisms relate to each other, and how they might be integrated remains an open question. We also invite researchers to explore other formalisms, and to compare them to our suggested approaches. In particular, it needs to be investigated how other model evolution activities that were not addressed in this paper can be formally supported, and which formalism would be the most suitable choice.

References

[1] S. Sendall and W. Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, September-October 2003. Special Issue on Model-Driven Software Development.

[2] Meir M. Lehman, Juan F. Ramil, P.D. Wernick, Dewayne E. Perry, and W. M. Turski. Metrics and laws of software evolution - the nineties view. In *Proc. Int'l Symposium Software Metrics*, pages 20–32. IEEE Computer Society Press, 1997.

[3] G. Sunyé, D. Pollet, Y. LeTraon, and J.-M. Jézéquel. Refactoring UML models. In *Proc. Int'l Conf. UML 2001*, volume 2185 of *LNCS*, pages 134–138. Springer-Verlag, 2001.

[4] Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *UML 2003 - The Unified Modeling Language*, volume 2863 of *LNCS*, pages 144–158. Springer-Verlag, 2003.

[5] Ragnhild Van Der Straeten, Viviane Jonckers, and Tom Mens. Supporting model refactorings through behaviour inheritance consistencies. In *Proc. Int'l Conf. UML 2004*, volume 3273 of *LNCS*, pages 305–319. Springer-Verlag, 2004.

[6] Jeff Gray Jing Zhang, Yuehua Lin. *Model-driven Software Development - Research and Practice in Software Engineering*, chapter Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. Springer-Verlag, 2005.

[7] G. Spanoudakis and A. Zisman. Inconsistency management in software engineering: Survey and open research issues. In *Handbook of Software Engineering and Knowledge Engineering*, volume 1, pages 329–380. World Scientific Pub. Co., 2001.

[8] J. Ebert and G. Engels. Specialization of object life cycle definitions. Fachbericht Informatik 19/95, Universität Koblenz-Landau, Fachbereich Informatik, Koblenz, 1995.

[9] G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Testing the consistency of dynamic UML diagrams. In *Proc. Sixth Int'l Conf. Integrated Design and Process Technology (IDPT 2002)*, June 2002.

[10] G. Rasch and H. Wehrheim. Checking consistency in UML diagrams: Classes and state machines. In *Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 229–243. Springer-Verlag, 2003.

[11] Ragnhild Van Der Straeten, Tom Mens, Jocelyn Simmonds, and Viviane Jonckers. Using description logic to maintain consistency between UML models. In *Proc. Int'l Conf. UML 2003*, volume 2863 of *LNCS*, pages 326–340. Springer-Verlag, 2003.

[12] Ragnhild Van Der Straeten, Tom Mens, and Viviane Jonckers. A formal approach to model refactoring and model refinement. *Software Systems Modeling*, 2005. Conditionally accepted for publication.

[13] Marko Boger, Thorsten Sturm, and Per Fragemann. Refactoring browser for UML. In *Proc. Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pages 77–81, Alghero, Sardinia, Italy, 2002.

[14] Jocelyn Simmonds, Ragnhild Van Der Straeten, and Viviane Jonckers. Maintaining consistency between UML models using description logic. *Série L'objet - logiciel, base de données, réseaux*, 10(2-3):231–244, 2004.

[15] Bashar Nuseibeh, Jef Kramer, and Anthony Finkelstein. A framework for expressing the relationship between multiple views in requirements specification. *IEEE Transactions on Software Engineering*, 20(10):760–773, 1994.

[16] H. Nissen, M. Jeusfeld, M. Jarke, G. Zemanek, and H. Guber. Managing multiple requirements perspectives with metamodels. *IEEE Software*, pages 37–47, March 1996.

[17] Wolfgang Emmerich, Anthony Finkelstein, Stefano Antonelli, Stephen Armitage, and Richard Stevens. Managing standards compliance. *IEEE Transactions on Software Engineering*, 25(6):836–851, 1999.

[18] Andrea Calí, Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. Reasoning on UML class diagrams in description logics. In *Proc. of IJCAR Workshop on Precise Modelling and Deduction for Object-oriented Software Development (PMD 2001)*, 2001.

[19] Paolo Bottoni, Gabriele Taentzer, and Andy Schürr. Efficient parsing of visual languages based on critical pair analysis and contextual layered graph transformation. In *Proc. IEEE Symp. Visual Languages*, 2000.

[20] Jan Hendrik Hausmann, Reiko Heckel, and Gabriele Taentzer. Detection of conflicting functional requirements in a use case-driven approach. In *Proc. Int'l Conf. Software Engineering*. ACM Press, 2002.

[21] Tom Mens, Gabriele Taentzer, and Olga Runge. Detecting structural refactoring conflicts using critical pair analysis. In *Workshop on software evolution through transformations: model-based versus implementation-level solutions*, volume 3256 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 2004.

[22] Tom Mens, Gabriele Taentzer, and Olga Runge. Analyzing refactoring dependencies using graph transformation. *Software Systems Modeling*, February 2005. Submitted.

[23] Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Proc. First Int'l Conf. Graph Transformation*, pages 286–301. Springer-Verlag, 2002.

Conciliating Property Stability and System Evolution through Software Model Analysis

Ilham ALLOUI

LISTIC - Polytech'Savoie
B.P. 80439
74944 Annecy le Vieux Cedex (France)
ilham.alloui@univ-savoie.fr

Abstract. Model-driven software engineering is founded on the idea of transforming abstract models into more concrete ones. Within this context, software model evolution is a kind of transformation that aims at obtaining a new model of software through the application of change operations on the starting software model. One main issue how to ensure stability of desired properties in the new model? Among such properties, are structural ones (e.g. cardinality of model elements, connectivity, etc.) and behavioural ones (e.g. safety, fairness...). Stability of software system properties refers to the system capability to accommodate changes from its environment while avoiding unexpected effects. A subsequent issue is how to scope model analysis so that only those parts of the model that could be affected will be analysed. To conciliate property stability and system evolution, the paper proposes a model evolution process founded on both property verification and change impact analysis techniques.

Introduction

Model-driven software engineering is founded on the idea of transforming abstract models into more concrete ones [1]. Within this context, software model evolution is a kind of transformation that aims at obtaining a new model of software through the application of change operations on the starting software model. One main issue is then: *how to ensure stability of desired properties in the new model*. Among such properties, are structural ones (e.g. cardinality of model elements, connectivity, etc.) and behavioural ones (e.g. safety, fairness, etc.). Furthermore a subsequent issue is *how to scope model analysis so that only those parts of the model that could be affected will be analysed*.

The paper, through the example of software architecture models, proposes a model evolution process founded on both techniques of change impact analysis and property verification. Indeed on the one hand, those techniques have been proved relevant in some application domains like program analysis and comprehension, distributed

systems thus they could be adapted to be used during a model evolution process as well. On the other hand, support provided for model evolution in model-driven engineering is generally not sufficient to grant the correctness of the new models. Example is the syntactic well-formedness rules in UML.

The rest of the paper is organised as follows. Section 2 briefly introduces to the software architecture domain. Section 3 shows through an example a possible approach to software model evolution founded on both property verification and change impact analysis techniques. Concluding remarks are given in Section 4.

Software architecture domain

As software systems increase in complexity, evolving them becomes difficult too. One way to overcome this problem is to use higher levels of abstraction while reducing the cognitive distance between the initial concept and its final executable implementation [2]. According to this view, software architectures are considered as relevant artefacts in bridging the gap between the initial concept of a software system and its implementation. Software architectures abstract software systems following both structural and behavioural perspectives [3]. An architecture model defines the structure, i.e. elements and their interrelations, in terms of components (computational units), connectors (elements supporting components' interactions) and configurations of interrelated elements. From a behavioural perspective, architectures define elements' actions and relations among them (ordering constraints like sequencing, choice, composition, etc.). Fig. 1 shows a model of a data acquisition software system (configuration) composed of two components a *sensor* and a *data manager* both interacting through a link (connector). Architectural elements are attached to each other via their ports. The state diagram models the behaviour of the data manager[1]. The semantic foundation of this language [4] is the π-calculus and behaviours are considered as processes based on atomic actions like ***via*** *connection-name* ***send*** *data*, ***via*** *connection-name* ***receive*** *data* and operators on processes like composition, sequencing, replication. A data manager behaviour is recursive (this is expressed through naming a sub-state the same way as the super-state i.e. *managingDatabase*) and contains two alternative behaviours, the former starting when receiving a new entry and the second when receiving a key querying for the corresponding data.

[1] The UML-based graphical notation [5] is used here to clarify our purposes.

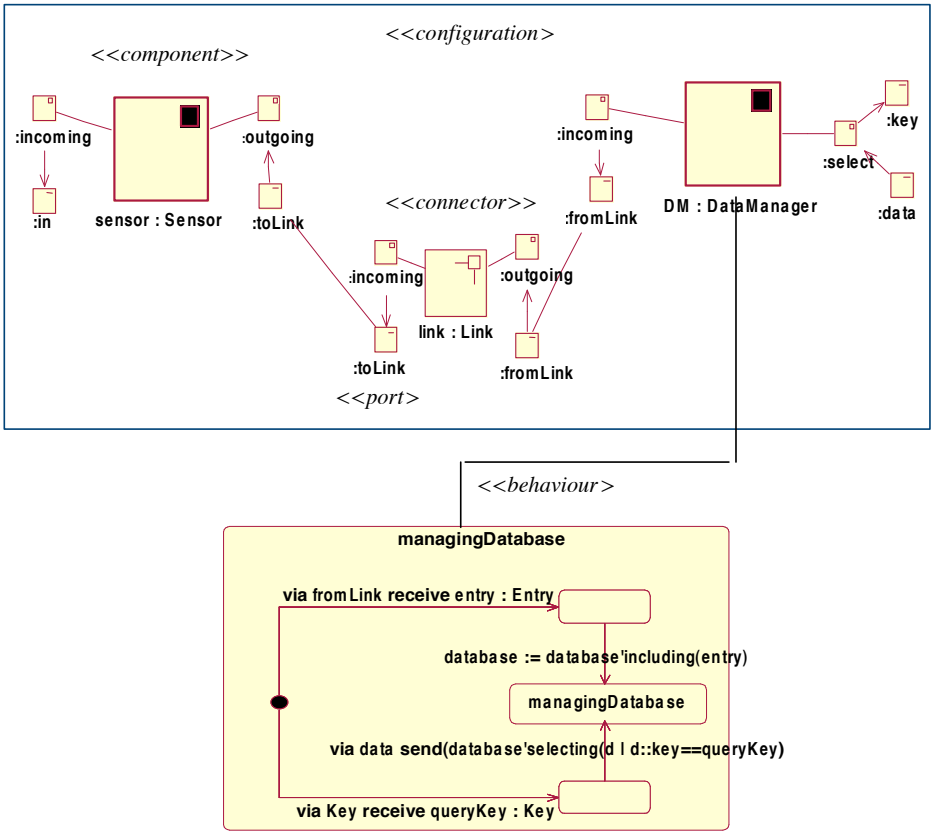


Fig. 1. Example of a simplified software architecture model

Given the architecture above, let us refer to it as *A0*, two desired properties we would like to preserve during model evolution could be the following[2]:

a) *connectivity* that states that every port within architecture *A0* is involved in at least one attachment. This structural property grants there is no detached architectural element in *A0*.

```
connectivity is property {
  on A0.instances apply
    forall {a | on (a.components.ports apply
        union(a.connectors.ports)) apply
          forall {p | on p.attachments apply isNotEmpty}
             }       }
```

[2] Properties are described here using a formal language [6].

b) *firstAction* states that no reply can occur before a call. This is typically a safety behavioural property granting the first action of *DataManager* instances must be an *actionIn* (i.e. *receive*) and not an *actionOut* (i.e. *send*).

```
firstAction is property {
  on A0.instances apply
    forall {cs | on (cs.DataManager.instances
      forall {c | (on c.actions apply isNotEmpty)
          implies
               (on c.actionsIn apply
                exists {call | on c.actionsOut apply
                      forall {reply |
                     every sequence {(not call)*.reply}
                        leads to state {false}
                               }    } )            }
            )  }        }
```

This can be phrased as follows: a sequence of actions starting with an action that is not *send* is not a valid sequence (it leads to a *false* state).

Notice that both example properties must be satisfied at runtime while other properties such as model completeness are related to design time.

Architectural model evolution relies generally on the usual set of primitive change operations such as adding/retracting/replacing an architectural element (port, component, connector, etc.) and other primitives like attaching/detaching ports and imploding/exploding components/connectors. Changes can be applied to both structure and behaviour of architecture. Components can be imploded within a composite one or a composite exploded in several components. Next section presents our approach to architecture model evolution.

Property verification and change impact analysis for model evolution

To meet the requirements cited so far, we propose a model evolution process founded on both techniques of change impact analysis and property verification. The example of software architecture model will be used for illustration purposes. One way to analyse correctness of the modified model is to check again all required properties of the new model. However this would cause an overhead, as most often changes remain local to a certain extent. Change impact can be evaluated before checking properties. Fig. 2 depicts a simplified process for model evolution consisting of three steps: *Analysing* model, *Deciding* for change and *Implementing* change. This evolution process takes as input the model to be changed (Model) and the change one would like to perform on the model (Change).

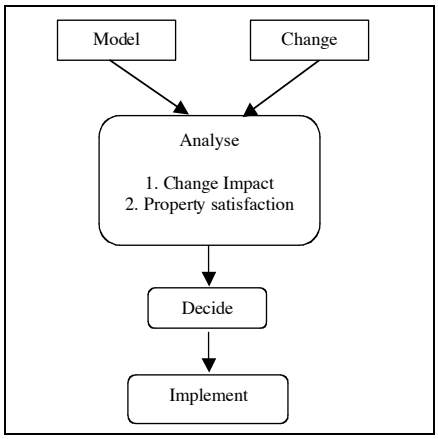


Fig. 2. A software model evolution process

Let us assume that we would like to replace one component *c2* in *A0* by two other components *c21* and *c22* to obtain *A1* (cf. Fig. 3). Component replacement is based on a set of primitive change operations like port attachment/detachment, port adding/retraction and component adding/retraction.

The evolution process would contain at least the following steps:

1) Analyse

- **Impact analysis**

Change impact analysis relies on techniques for assessing the extent of the change, i.e., the software components that will impact the change, or be impacted by the change. Once effects of the changes have been identified, software modellers/architects can use this information to evolve/re-engineer the software system design. A number of impact analysis techniques have been proposed in literature for different programming styles and architectures among them [7, 8] that rely either on program slicing or chaining to extract dependencies related to a point of interest. Change impact analysis approaches focus on both structure (dependencies) and semantics (attached to dependencies) [9] and key points in such techniques are the kinds of dependencies that are considered and relevant metrics to obtain a relevant result. In our example, one dependency could be computed from the behaviour of the element we would like to retract as follows: for each data *a* sent by it, retrieve all components that receive it. This is typically a direct dependency. Indirect dependencies should also be considered, for instance, one component say *C7* directly impacted by *C2* could consequently impact other components that are waiting for data *b* that is sent by *C7* after receiving *a*.

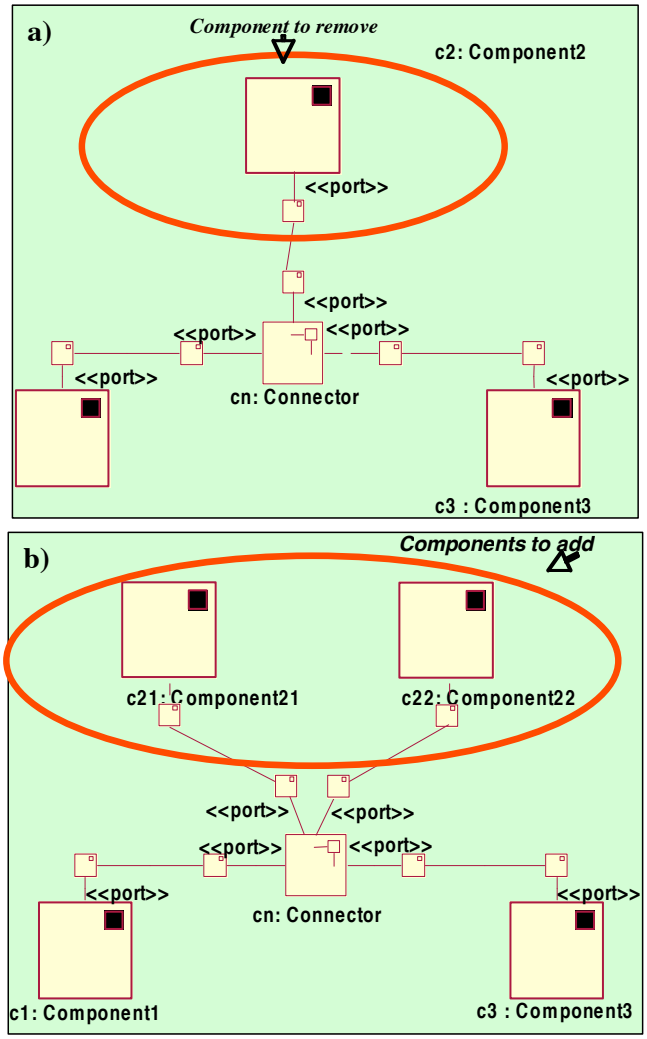


Fig. 3. A0 configuration before and after the change

- **Property verification**

To complement this impact analysis, simulation techniques could be used too to verify at design time if desired properties remain stable after the change. In our example, *connectivity* and *firstAction* properties could be evaluated against the new architecture *A1*. It is worth noticing that while the first property expression does not need to be changed, the second one should "evolve" to one or more property expressions to reflect the change and then to maintain property stability. The interesting issue here is that impact analysis could be applied to architectural properties as well.

Formally *connectivity* property is verified using theorem-proving technique while *firstAction* property is verified according to model checking technique. Both techniques are included in an architecture analysis tool [6].

2) Decide

According to the results of change impact analysis and property verification on the software architecture model, decisions can be made to implement the change or not.

3) Implement

Within this step, changes are implemented on software.

Concluding remarks

Model-level evolution is an important issue as it allows modellers to abstract from implementation details. However, in order to be efficient, "correctness" of model evolution should be ensured at design time using existing techniques and tools like verification and change impact analysis. In this paper we have briefly shown through an architecture example how both property verification and change impact analysis can support model evolution process. The concluding message is two-fold: first, evolution correctness is a semantic issue that should be dealt with at the level of formal methods like model checking and theorem proving that provide a specification language and tool support far analysis. Second, in order to both optimise the evolution process and help in re-engineering the model, change impact analysis techniques should be used. Further work has been done on model-based architecture evolution monitoring [10], architectural impact analysis [11] and evolvable services-oriented architectures [12].

References

[1] A. W. Brown, "An introduction to Model driven Architecture – *Part I", MDA and Today's Systems. The Rational edge*, February 2004.

[2] P. B. Kruchten, "The 4+1 View Model of Architecture", IEEE Software, Nov. 1995.

[3] IEEE Std 1471-2000: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, October 2000.

[4] F. Oquendo, "π-ADL: An Architecture Description Language based on the Higher-Order Typed π-Calculus for Specifying Dynamic and Mobile Software Architectures", *ACM Software Engineering Notes*, Vol. 29, No. 4, May 2004.

[5] Alloui and F. Oquendo, “UML Arch-Ware/Style-based ADL”, Deliverable D1.4b, ArchWare European RTD Project, IST-2001-32360, January 2003.

[6] Alloui, H. Garavel, R. Mateescu, and F. Oquendo, “The ArchWare Architecture Analysis Language”, *Deliverable D3.1b*, December 2002.

[7] Stafford and A. L. Wolf, “Architecture-Level Dependence Analysis for Software Systems”, *International Journal of Software Engineering and Knowledge Engineering*, Vol. 11, No. 4, pp.431-453, 2001.

[8] J. Zhao, “Applying Slicing Technique to Software Architectures", Proc. 4th IEEE lnternational Conference on Engineering of Complex Computer Systems, August 1998.

[9] S. A. Bohner, “Software Change Impacts: An Evolving Perspective Software”, *Proceedings of the International Conference on Software Maintenance (ICSM’02)*, IEEE Computer Society.

[10] S. Cimpan, H. Verjus and I. Alloui, “Gestion de l’évolution dans le cadre d’une approche d’ingénierie logicielle centrée architecture”, *Technique et Science Informatiques (TSI)*, 2009 (à paraître).

[11] I. Alloui, S. Cimpan and H. Verjus, “Towards software architecture physiology: Identifying vital components”, IEEE Computer Society, editor, Proceedings of the 7th Working IEEE/IFIP Conference on Software Architecture (WICSA 2008), pages 293–296, Vancouver, Canada, February 2008.

[12] H. Verjus and F. Pourraz, “A Formal Framework For Building, Checking And Evolving Service Oriented Architectures”, *IEEE European Conference on Web Services (ECOWS 2007)*, Halle, Germany, November, IEEE Computer Society, pp. 245-254, 2007.

Le contrat d'évolution d'architectures : un outil pour le maintien de propriétés non fonctionnelles

Régis Fleurquin[1,2], Chouki Tibermacine[3] et Salah Sadou[1]

[1] VALORIA, Université Bretagne-Sud, Vannes, France
[2] IRISA, INRIA Rennes, France
[3] LIRMM, Université Montpellier II, Montpellier, France
fleurqui@univ-ubs.fr, tibermacin@lirmm.fr and sadou@univ-ubs.fr

Résumé Tout logiciel doit évoluer pour répondre aux exigences changeantes de ses utilisateurs et aux modifications de son environnement. Ces changements, souvent imprévisibles, réalisés par un tiers et dans l'urgence, mènent parfois le logiciel vers un état que ses créateurs n'auraient pas souhaité. Nous présentons dans cet article un cadre pour une évolution contrélée d'applications à base de composants. Ce contrôle garantit la préservation de propriétés architecturales et par là de certaines propriétés non fonctionnelles.

1 Introduction

Une caractéristique intrinsèque d'un logiciel, représentant une activité du monde réel, est la nécessité qu'il a d'évoluer pour satisfaire de nouvelles exigences. La première loi de Lehman, issue de constatations sur le terrain, stipule qu'un logiciel doit nécessairement évoluer faute de quoi il devient progressivement inutile [11]. Bien qu'ancienne cette loi ne s'est jamais démentie. La réactivité, toujours plus grande, exigée des applications informatiques, supports de processus métiers évoluant eux-mêmes de plus en plus vite, a même accru au fil des ans la portée de cette loi. La maintenance est, plus que jamais, une activité incontournable. Cette activité coûte cependant de plus en plus cher. Estimée dans les années 80 et 90 à environ 50 à 60 % [12,14] des coûts associés aux logiciels ; de récentes études évaluent désormais ce coût entre 80 et 90 % [7,16]. La maintenance, trop longtemps perçue comme une activité peu valorisante et de moindre intérêt, doit, à l'instar de l'activité de développement se doter de méthodes, de techniques et d'outils efficaces.

Parmi les activités de maintenance, la compréhension de l'architecture de l'application, avant évolution, et la vérification de sa non régression fonctionnelle et non fonctionnelle, après évolution, sont de loin les plus coûteuses. La première de ces activités, à elle seule, compte pour plus de 50 % du temps de maintenance [4]. Elle est d'autant plus facile que la documentation fournie avec le logiciel est de qualité. Cette documentation doit, en particulier, être complète, à jour et non ambiguë. La seconde de ces activités vérifie d'une part, l'existence après modification de la propriété ou du nouveau service recherché et d'autre part, que les autres propriétés et/ou services n'ont pas été altérés. Cette vérification peut se faire soit a posteriori, une fois la modification entérinée en constatant in vivo ses effets, par exemple au travers de tests de non régression ; soit a priori, au moment où l'on détaille la modification que l'on souhaite entreprendre, par exemple en alertant des conséquences de celle-ci. La non régression a posteriori de traits fonctionnels est de loin la mieux maîtrisée car la plus simple. Bien que complémentaires, il est cependant connu que les techniques préventives, favorisant la détection au plus tôt des problèmes, sont moins coûteuses que les techniques correctives.

Dans cet article, nous nous intéressons aux problèmes posés par les deux activités de maintenance précédentes dans le cadre de la technologie des composants. Nous mettons notamment en valeur l'intérêt de contraindre les évolutions possibles d'une architecture dans le but d'une part, de garantir la présence d'une documentation de conception de qualité et d'autre part, de prévenir la disparition de certaines propriétés non fonctionnelles. Dans la première section, nous soulignons sur un exemple de maintenance, quelques uns des problèmes induits par une documentation de qualité insuffisante et par un processus de mise à jour somme toute trop libéral. Nous décrivons

ensuite l'approche que nous proposons pour remédier à ce type de problème. Dans la quatrième section, nous présentons une implantation de cette approche s'appuyant sur le langage OCL et sur un outil de vérification de contraintes. Dans la dernière section, avant la conclusion, nous discutons des travaux connexes.

2 Illustration de la problématique

La figure 1 présente l'architecture d'un système de contrôle d'accès à un bâtiment (cas d'un musée). Cette architecture est un cas particulier du patron *pipe & filter* : le système reçoit en entrée des données permettant l'authentification d'un usager. Après identification, ces données sont envoyées au composant `Contrôleur d'accès`. Celui-ci ajoute à ce flux d'autres données (l'heure d'entrée, la galerie dans le musée, etc.). Puis, il passe le tout au composant responsable de l'archivage local (composant `Archiveur`). Ces données sont ensuite transmises (composant `Emetteur`), via le réseau externe à un serveur central d'archivage de l'entreprise qui s'occupe de la sécurité du musée. A travers le choix d'une architecture de type *pipe & filter*, les développeurs ont cherché à respecter les exigences de maintenabilité formulées dans le document de spécification de ce système. En effet, ce style architectural garantit un couplage faible entre des composants. Un composant n'est lié qu'à un seul composant par son interface requise et un seul par son interface fournie. Ces développeurs n'ont cependant pas pris la peine de documenter les raisons de ce choix. A ce stade, une information importante pour la compréhension de la structure du système est alors définitivement perdue.

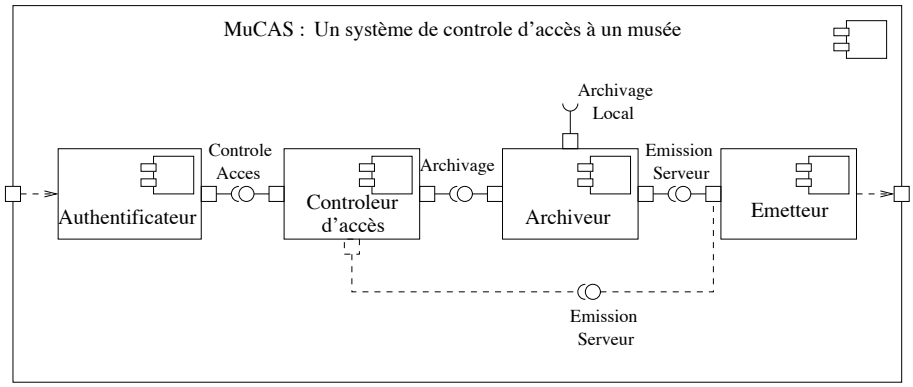


Fig. 1. Architecture simplifiée d'un système de contrôle d'accès à un musée

Supposons maintenant que l'archivage local de certaines informations (voir composant `Archiveur`) est désormais inutile et qu'il faille transmettre ces informations directement au serveur central. Les personnes en charge de cette modification peuvent décider, après consultation des documents de conception, de créer un lien direct entre le composant `Contrôleur d'accès` et le composant `Emetteur`. Le composant `Contrôleur d'accès` se retrouve, alors, avec deux liens : le premier avec le composant `Archiveur`, pour le flux non affecté par la modification ; le second avec le composant `Emetteur` pour les données à transmettre directement (voir les lignes en pointillé sur la figure 1). Cette modification, réalisée en toute bonne foi, fait perdre à l'application les bénéfices d'une structure en *pipe* et, par conséquent, abaisse son niveau de maintenabilité. Une propriété non fonctionnelle, qui ne devait pas être altérée, est mise à mal par le simple fait de remettre en cause un choix d'architecture dont la présence n'était pas innocente.

Le type de problème illustré ici a plusieurs origines. Tout d'abord, les modifications faites à un logiciel sont souvent réalisées longtemps après la version précédente et généralement par

des tiers ; les raisons des choix architecturaux étant rarement explicitées, elles sont alors perdues pour les phases de maintenance suivantes. Même dans le cas où ces raisons sont détaillées, le code est souvent le seul artefact modifié et parcouru sans considération pour les documentations associées. Un réflexe acquis par la plupart des développeurs est donc de n'accorder du crédit, malheureusement avec raison, qu'au seul code source. Quoi qu'il en soit, rien n'empêche au final un développeur de passer outre un choix architectural bien qu'informé des conséquences de cet acte. Résoudre ce type de problème passe nécessairement par deux choses. Premièrement, il doit y avoir obligation de documenter les raisons des choix architecturaux faits lors du développement et de la maintenance. Cette obligation garantit l'existence d'une documentation, remise à jour si nécessaire, utilisable lors de la phase de compréhension d'une architecture. Deuxièmement, le respect de ces choix doit être vérifié lors de chaque mise à jour du code. Cette vérification assure la non régression des propriétés visées par ces choix. L'approche que nous allons décrire respecte ces deux aspects.

3 Notre approche : le contrat d'évolution

Il est admis que ce ne sont pas les fonctions attendues qui déterminent l'architecture d'une application ou d'un composant, mais bien les exigences non fonctionnelles [2]. C'est d'ailleurs le point de vue de certaines méthodes de développement d'architectures, comme la méthode ADD [3] du *Software Engineering Institute*. La connaissance des liens unissant propriétés non fonctionnelles et choix architecturaux est donc du plus grand intérêt pour les personnes en charge de la maintenance et ce à double titre. Premièrement, si l'architecture est bâtie sur la base d'une recherche de certaines propriétés non fonctionnelles, la construction d'une image mentale suffisante de cette architecture, préalable à toute modification, passe nécessairement par la reconstitution de cette connaissance. Faciliter cette reconstitution, c'est diminuer d'autant les efforts à consentir lors de la coûteuse phase de compréhension de la structure existante. Deuxièmement, la mise à disposition de ces informations peut éclairer un développeur lors de l'élaboration d'une stratégie d'évolution. Remettre en cause un choix architectural, c'est en effet se poser la question du devenir des propriétés non fonctionnelles dont ce choix visait l'obtention. Il est dès lors possible, à chaque étape d'un processus d'évolution, non seulement d'identifier les éléments architecturaux concernés par une évolution, mais également, d'identifier les risques potentiels d'altération de certaines propriétés non fonctionnelles. Sur les fils des liens, unissant choix architecturaux et propriétés non fonctionnelles, peut alors s'établir une démarche cyclique allant du besoin vers la stratégie (recherche de la partie de l'architecture à modifier partant du trait de spécification concerné) et de la stratégie vers le besoin (évaluation de l'impact d'une modification de l'architecture sur la spécification).

Nous proposons donc d'expliciter les liens unissant les spécifications non fonctionnelles et les choix architecturaux en usant, d'une part, d'un langage formel à même de décrire des choix architecturaux, et d'autre part, d'un mécanisme d'association à même de lier ces choix à des énoncés de spécifications non fonctionnelles. Le choix d'un langage formel garantira non seulement la non ambiguïté des descriptions, mais permettra également l'automatisation de certaines opérations. Il sera par exemple possible d'alerter, à chaque "pas" d'évolution, le développeur sur les conséquences éventuelles des modifications qu'il applique à une architecture. On se retrouve alors dans un système bouclé. Ce type de système augmente grandement les chances d'aboutir à une solution remplissant les nouvelles exigences, tout en préservant les propriétés non fonctionnelles qui ne devaient pas être altérées.

Dans ce processus, nous ne devons pas interdire un "pas" d'évolution. On signale simplement la tentative de rupture d'un choix architectural, dont on précise les conséquences. A charge pour le développeur, en toute connaissance, de maintenir ou non sa modification. Il se peut, en effet, qu'un choix architectural soit remplacé par un autre sans remise en cause des propriétés non fonctionnelles. Car il existe souvent plusieurs architectures capables de garantir le respect d'une même propriété non fonctionnelle. De plus, pour réaliser une modification, on peut être amené à invalider un choix pour le restaurer plus tard dans un contexte changé. On peut faire ici le parallèle avec un invariant de classe qui peut être non valide dans le corps d'une méthode mais

garanti finalement en sortie de méthode. La seule condition bloquante pour le développeur est d'indiquer, en fin d'évolution, pour tout choix ayant été remis en cause lors du processus de maintenance, le ou les nouveaux choix qui lui sont substitués. Cette condition assure qu'aucune propriété non fonctionnelle ne reste pendante ; c'est-à-dire sans éléments architecturaux ayant pour objet son obtention. Le non respect de cette condition implique, de facto, l'obligation pour le développeur de modifier la spécification non fonctionnelle. Ainsi, lors d'une évolution, de nouveaux choix architecturaux peuvent compléter, amender ou remplacer les anciens choix.

Un choix architectural est perçu dans cette approche comme une contrainte dont on cherche à vérifier la validité à chaque "pas" d'une évolution. L'ensemble de ces contraintes, et les liens qui les associent aux propriétés non fonctionnelles, constituent le **contrat d'évolution** d'un composant. Nous parlons de contrat car il documente les droits et devoirs de deux parties : le développeur de la précédente version du composant qui s'engage à garantir les propriétés non fonctionnelles, sous réserve du respect, par le développeur de la nouvelle version, des contraintes architecturales que le premier avait établies. Le contrat d'évolution est élaboré lors du développement de la première version d'un composant. Des contraintes apparaissent à chaque stade du développement dans lequel un choix architectural motivé est fait. Le contrat est donc construit progressivement et enrichi au fil du projet. Certaines contraintes peuvent même être héritées d'un plan qualité logiciel (contraintes de projet) ou d'un manuel qualité (contraintes d'entreprise) et donc, émerger avant même le démarrage du développement logiciel. Par la suite, dans le respect de la condition de blocage, ce contrat pourra à son tour être modifié. A charge pour les développeurs, informés des conséquences de leurs actes, de garantir à leur tour, l'obtention des propriétés non fonctionnelles sur la base de nouvelles contraintes qu'ils ont pu établir.

Ces contraintes architecturales sont bien plus qu'un autre formalisme pour les commentaires classiques de modèles de conception ou de code. Tout d'abord, à l'instar des pré et post-conditions de langages comme EIFFEL, leur format évaluable autorise de nouveaux usages et modifie le cadre méthodologique de la maintenance. Enfin, certaines contraintes n'ont de sens que dans le cadre évolutif. Il suffit de considérer une contrainte comme : la complexité d'un composant ne doit pas augmenter. Cette contrainte, associée à la propriété de maintenabilité, est une pure condition de maintenance. Elle n'aurait aucun sens dans un commentaire classique. Bien évidemment, à l'extrême, les contraintes peuvent devenir aussi nombreuses que les lignes de code. Il faut donc trouver des niveaux de granularité architecturaux influençant significativement une propriété particulière d'interface. Il est nécessaire d'user avec intelligence et parcimonie de ces contraintes et faire preuve de la même hygiène que lors de l'écriture des commentaires classiques : ni trop, ni trop peu.

4 Une implantation du contrat d'évolution

Notre approche s'appuie sur un langage formel pour décrire le contrat d'évolution et sur un outil pour son évaluation. Nous allons décrire le langage dans une première section et l'outil dans une seconde.

4.1 L'expression du contrat d'évolution

Un contrat d'évolution est composé d'une part, d'un ensemble de contraintes architecturales et d'autre part, de la liste des liens unissant ces contraintes avec des énoncés de spécification non fonctionnelle. Nous allons décrire le langage de description de contraintes. Nous présenterons, ensuite, le mécanisme d'association.

Le langage d'expression de contraintes Il est difficile de prévoir tous les types de contraintes que les développeurs pourraient être amenés à exprimer. Néanmoins, il est certain que des contraintes imposant des styles architecturaux, des patrons de conception ou des conventions de codage doivent pouvoir être écrites. Face à cette diversité, nous avons opté pour une solution à deux niveaux. Le premier niveau s'appuie sur un langage facile à appréhender, largement répandu et standardisé

par l'OMG, à savoir OCL [15]. Le second niveau prend la forme d'un méta-modèle. Ce méta-modèle (voir figure 2) nous permet d'adapter OCL, sans changement de syntaxe, à l'écriture de contraintes sur des architectures à base de composants. Ce second niveau permet également, par simple amendement du méta-modèle, d'étendre comme bon nous semble la liste des opérateurs disponibles sans pour cela avoir besoin de toucher à la syntaxe du langage de premier niveau (donc à la syntaxe OCL). Cette structure à deux niveaux est le gage de l'extensibilité du pouvoir d'expression de notre formalisme.

Pour comprendre comment cette structure à deux niveaux s'articule, il est nécessaire de revenir sur le mode d'expression habituel des contraintes OCL dans un diagramme de classes. Dans ce type de diagramme, OCL s'utilise le plus souvent pour spécifier des invariants de classe, des pré/post conditions d'opération, des contraintes de cycle entre associations, etc. Ces contraintes restreignent le nombre des diagrammes d'objets valides instanciables depuis un diagramme de classes. Elles pallient à un manque d'expressivité de la notation UML graphique qui, employée seule, peut autoriser dans certains cas l'instanciation de diagrammes d'objets non compatibles avec la réalité que l'on souhaitait modéliser. Les contraintes OCL sont décrites relativement à un contexte. Ce contexte est un élément du diagramme de classes, le plus souvent une classe, une opération ou une association présente sur ce diagramme. Voici deux exemples de contrainte OCL.

```
context ArticleLMO inv:
self.taille <= 13
context a: ArticleLMO inv:
a.ecritPar->size() >= 1
```

Dans les deux cas, le contexte est une classe (`ArticleLMO`). Les deux contraintes sont écrites selon le point de vue d'une instance quelconque du contexte (ici un objet instance de la classe `ArticleLMO`). C'est l'approche adoptée pour toute contrainte OCL. La première de ces contraintes référence cette instance en usant du mot clé `self`. A l'opposé, la seconde introduit pour la désigner un identificateur ad-hoc `a`. Ces deux modes de désignation, tolérés par OCL, sont sémantiquement équivalents. Pour toute contrainte OCL, les éléments apparaissant sont des éléments, soit prédéfinis dans le langage OCL (`->`, `size()`, etc.), soit des éléments du diagramme de classes atteignables par navigation depuis le contexte (attribut `taille` et association `ecritPar`).

Il est intéressant de se demander quel peut être le sens de contraintes OCL écrites non pas sur un modèle mais sur un méta-modèle. Un méta-modèle expose les concepts d'un langage et les liens qu'ils entretiennent entre eux. Il décrit une grammaire abstraite. Une contrainte ayant pour contexte une méta-classe va, de ce fait, limiter la puissance d'expression des règles de production de cette grammaire et donc le nombre des phrases (i.e. modèles) dérivables. Certaines structures de phrase sont écartées. Si ce méta-modèle décrit la grammaire d'un langage de description d'architectures, une contrainte exprime que seules certaines architectures (i.e. modèles) sont dérivables (i.e. instanciables) dans ce langage. Le langage est bridé sciemment dans son pouvoir d'expression car on ne tolère pas la description de certains types d'architecture. Par exemple, on peut imposer que, dans tout modèle, les composants aient moins de 10 interfaces requises, en posant cette contrainte dans le contexte de la méta-classe `composant`. Cette contrainte est exactement du type de celle que nous souhaitons pouvoir exprimer. Malheureusement sa portée est globale. Elle s'applique à tout composant et non à un composant particulier comme nous souhaitons le faire. Pour limiter la portée d'une contrainte à un composant particulier, nous proposons de modifier légèrement la syntaxe et la sémantique de la partie contexte d'OCL. Sur le plan syntaxique, nous imposons que tout contexte introduise un identificateur. Cet identificateur doit être, de plus, le nom d'une instance particulière de la méta-classe citée dans le contexte. Sur le plan sémantique, nous interprétons la contrainte avec le sens qu'elle aurait dans le contexte d'une méta-classe mais en limitant sa portée à l'instance citée dans le contexte.

Selon ce principe, la contrainte qui suit, appliquée à notre méta-modèle (voir figure 2), stipule que le composant primitif de nom `ControleurAcces` doit être lié à un et un seul composant par le biais de son interface requise `Archivage` :

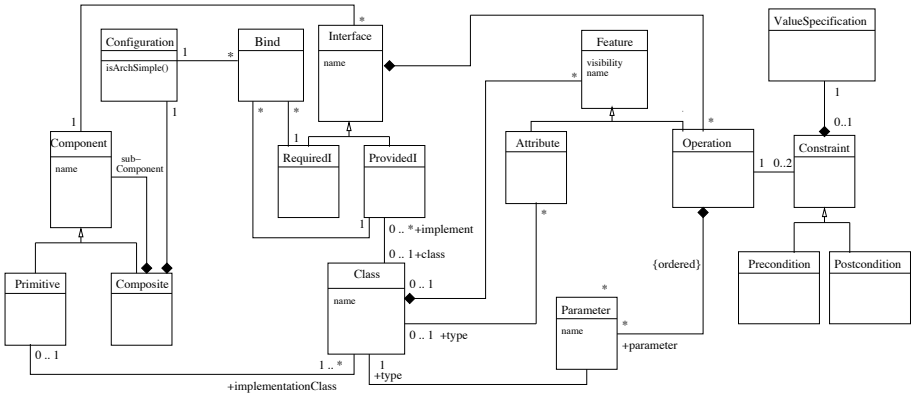


Fig. 2. Méta-modèle pour les architectures d'applications à base de composants.

```
context ControleurAcces:Primitive inv:
self.interface->select(i : Interface | (i.oclIsTypeOf(RequiredI))
and (i.name = 'Archivage')).oclAsType(RequiredI).bind->size() = 1
```

Ainsi, moyennant une modification minime de la syntaxe et de la sémantique d'OCL nous arrivons, en usant d'une structure langagière bicéphale OCL/Méta-modèle, à décrire des contraintes faisant état d'un mécanisme d'introspection. Un composant est à même d'exprimer une contrainte portant sur sa propre structure. A titre d'exemple, la condition de couplage faible que l'on souhaitait voir respectée par l'application présentée à la section 2 s'exprime de la manière suivante :

```
context muCAS: Composite inv:
(self.configuration.bind.requiredI.component=(self.subComponent))
and
(self.configuration.bind.providedI.component=(self.subComponent))
```

La première partie de cette contrainte exprime le fait que tout sous-composant doit avoir un et un seul *bind* via son interface requise. La seconde fait de même pour les interfaces fournies.

En ajoutant des méta-classes ou des opérations dans le méta-modèle on peut étendre le langage d'expression de contraintes. Ainsi, pour faciliter l'écriture de certaines contraintes, nous avons introduit des opérateurs de la théorie des graphes (`isSimple()`, `isConnexe()`, etc.) dans la méta-classe `Configuration`.

Le mécanisme d'association Le mécanisme d'association doit permettre de lier une contrainte architecturale à un énoncé de spécification non fonctionnelle. Une contrainte doit pouvoir être liée à plusieurs énoncés (au moins un) et réciproquement un énoncé peut se voir affecter plusieurs contraintes (au moins une). Dans l'état actuel de nos travaux le mécanisme proposé est encore rudimentaire. Sa formalisation est dépendante du choix d'un langage d'expression de propriétés non fonctionnelles ; en particulier de la structure de ce langage. Or, ce type de langage est un domaine encore ouvert et en devenir, objet de nombreux travaux.

Le mécanisme que nous proposons respecte la structure décrite par la figure 3. Dans son esprit, elle s'inspire de la norme qualité ISO 9126 [9]. Un lien associe une contrainte à une caractéristique qualité externe. Ces caractéristiques sont organisées sous la forme d'une forêt. Cette forêt compte 6 arbres dont les racines sont pour l'instant les 6 caractéristiques externes de plus haut niveau énoncées par la norme : maintenabilité, portabilité, fiabilité, rendement, capacité fonctionnelle, facilité d'utilisation. Le deuxième niveau de chaque arbre contient les sous-caractéristiques externes détaillées par la norme. Chaque contrainte a un nom et, éventuellement, un commentaire. Un lien unit une contrainte à une caractéristique. Chaque lien est typé. Pour l'instant nous n'usons que de deux types de lien : `contribute` et `limit`.

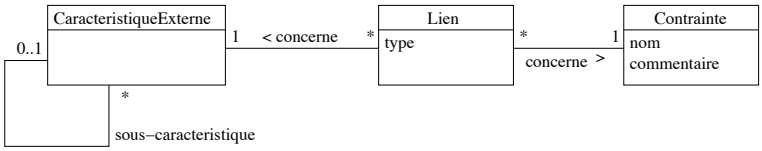


Fig. 3. Structure du mécanisme d'association.

Concrètement, une contrainte annonce dans un en-tête ses liens avec les différentes caractéristiques qualité. Cet en-tête prend la forme d'une suite de commentaires respectant le format suivant :

```
(<caractéristique>,<type de lien>):<description>
```

A titre d'exemple, la contrainte décrite précédemment doit avoir pour en-tête le commentaire structuré suivant :

```
-- (maintainability,contribute) : Architecture en pipe
context muCAS: Composite
...
```

Dans la contrainte ci-dessus, le lien concerne l'attribut maintenabilité. Le type du lien est `contribute`. Il indique que la contrainte favorise l'obtention de la caractéristique concernée.

4.2 Support logiciel pour l'interprétation des contrats d'évolution

Nous avons développé `ACE` (*Architectural Contract Evaluator*), un outil permettant la rédaction puis la vérification d'un contrat d'évolution. Pour cela, il exploite les éléments suivants :

- Le méta-modèle de composants sous la forme d'un fichier XMI ;
- Le composant avant évolution : une archive contenant entre autres la description de l'architecture du composant (sous la forme d'un document XML) et son contrat d'évolution.
- La description de l'architecture du composant après évolution, sous forme d'un document XML.

Le fichier contenant le méta-modèle est utilisé par `ACE` pour contrôler le bien fondé des navigations exprimées dans les contraintes. Fournir notre méta-modèle en paramètre, nous permet de prendre en compte différents méta-modèles ou bien de le faire évoluer sans impact sur l'outil.

`ACE`, dont on peut voir l'interface dans la figure 4, s'utilise de la manière suivante : en phase de rédaction du contrat d'évolution, lors de la création du composant, le fichier représentant son architecture est exploité pour vérifier que les identifiants, apparaissant dans les contraintes, correspondent bien à des éléments de l'architecture. Le contrat est, d'abord, évalué sur l'architecture initiale. Ceci nous permet de s'assurer que la première version du composant valide son propre contrat. Une fois cette version validée, les évaluations suivantes se font sur les versions présentées en tant que des évolutions du composant.

`ACE` s'appuie sur le compilateur `OCLCompiler` [17] pour la génération de l'arbre syntaxique abstrait (AST) des contraintes. Dans sa version actuelle, `ACE` supporte des descripteurs d'architecture décrits avec l'ADL `Fractal` [6]. Les spécificités de l'ADL `Fractal`, et leurs liens avec notre méta-modèle, sont complètement encapsulés dans une classe dite d'adaptation. Le passage à tout autre modèle, conforme à notre méta-modèle, consisterait en l'écriture de la classe d'adaptation correspondante.

5 Travaux connexes

La description de styles architecturaux aussi bien sur le plan statique que dynamique est une problématique abordée par certains ADL comme Aesop [8], Darwin [13] ou Wright [1]. Ces ADL

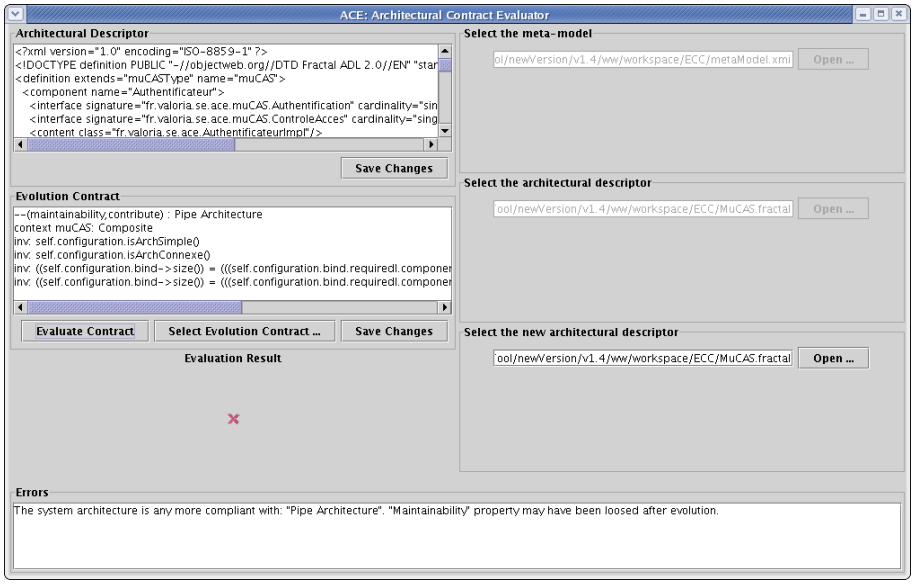


Fig. 4. ACE : outil d'évaluation de contrats d'évolution.

permettent de lister les propriétés inhérentes à tel ou tel style. Il est ensuite possible d'associer à une architecture en cours de modélisation l'un de ces styles. Cette simple déclaration confère à la nouvelle architecture les propriétés énoncées par le style cité. L'objectif premier est ici, à travers la définition de bibliothèques de styles, d'offrir un mécanisme de réutilisation facilitant la définition de nouvelles architectures. Dans notre approche, la définition d'un style procure un moyen permettant de vérifier qu'une architecture au fil de ses évolutions conserve certaines propriétés structurelles. Bien que distinguées dans leur usage, l'expression d'un style côté ADL ou côté contrat d'évolution sont bien sûr de même essence. Sur ce point précis, ces ADL et le langage que nous proposons sont confrontés aux mêmes exigences en terme de pouvoir d'expression. L'étude des ADL offrant des mécanismes de description de styles a influencé de manière non négligeable la structure du méta-modèle que nous proposons. Cette intersection n'est cependant pas la manifestation d'une inclusion dans un sens ou dans l'autre. D'une part, les ADL cités sont à même de modéliser des aspects comportementaux qui sortent de nos objectifs purement structurels. D'autre part, nous devons de notre côté, faire face à une dimension temporelle complètement absente des préoccupations des ADL actuels. Si les ADL doivent pouvoir décrire ce qu'est un système, nous devons, pour notre part, être capable en plus de cela, d'exprimer non seulement ce que peut ou ne peut pas devenir un système mais également les chemins pouvant être pris pour ce faire. Si le respect d'un style est un exemple de contrainte ne réclamant pas pour son expression de faire mention à des aspects temporels. Certaines contraintes, au contraire, ne peuvent s'exprimer que sous la forme d'une confrontation entre nouvelle et ancienne version d'une architecture. A titre d'exemple, une entreprise peut imposer dans son manuel qualité que d'une version à l'autre tout composant ne peut voir son nombre d'interfaces augmenté que d'au plus un (pas plus d'un concept nouveau à la fois). L'expression de ce type de contrainte sort indubitablement du champ d'intérêt des ADL. Si le domaine des ADL recoupe sur certains points notre problématique (en particulier l'expression de styles), nous offrant matière à comparaison et enrichissement, il ne constitue pas pour autant une réponse, ni la seule source possible d'inspiration.

De part les ADL, il existe un certain nombre de travaux ayant affiché clairement comme objectif la préservation de propriétés structurelles d'un système lors de son évolution.

Dans [10], Klarlund et al. ont proposé un nouveau langage de contraintes : CDL (*Category Description Language*). Ce langage, basé sur la logique du premier ordre sur les arbres d'analyse,

permet d'expliciter, de manière formelle, des invariants architecturaux sur un système. Ils appellent une catégorie un ensemble nommé de contraintes et un style CDL, un ensemble de catégories. Une fois ces styles définis par l'architecte du système, un développeur d'application sélectionne les catégories qui l'intéressent et annote les éléments de conception concernés avec les noms de ces catégories. Par la suite, une vérification est faite, par un environnement dédié, pour déterminer si la conception satisfait toutes les catégories. Pour le même objectif, Wuyts propose d'utiliser un dialecte de Prolog, le langage SOUL [18] : une solution basée sur la logique du premier ordre pour décrire les propriétés que doit respecter un programme orienté objets.

Dans les deux systèmes ci-dessus, on retrouve, comme pour notre approche, la notion de contraintes. Toutefois, deux éléments nous séparent : i) leur approche a pour cible les systèmes à base d'objets, alors que la notre vise les systèmes à base de composants ; ii) nous associons à chaque contrainte ses objectifs, c'est-à-dire, un lien avec les propriétés non fonctionnelles cibles alors qu'eux, ne le font pas.

Avec le système `CoffeeStrainer` [5], Bokowski propose une approche plus pragmatique pour définir des contraintes structurelles d'applications Java. Il utilise comme langage, Java lui même. Les contraintes sont définies sous forme de méthodes dont les valeurs de retour sont des booléens. Ces méthodes reçoivent en paramètre des noeuds de l'arbre syntaxique des classes à vérifier (`Field`, `Assignment`, etc). Elles sont insérées dans des commentaires spéciaux à l'intérieur d'interfaces vides de marquage. Une classe doit implanter ces interfaces pour préserver les propriétés représentées par les contraintes. `CoffeeStrainer` se chargera de générer le nécessaire à partir des commentaires.

Contrairement aux approches précédentes, la cible est ici, un langage de programmation particulier. Le moyen d'expression est ce même langage. Cette approche a l'avantage de ne pas créer un langage ad-hoc pour l'expression des contraintes, mais a l'inconvénient de n'être utilisable que pour ce langage. Contrairement à notre approche, dont la cible est l'architecture des applications, donc les relations inter-composants ou inter-classes, l'approche `CoffeeStrainer` vise les contraintes intra-classe. La relation n'est inter-classes que dans les cas d'héritage. Cette approche est complémentaire à la nôtre.

6 Conclusion

Le contrat d'évolution évite la remise en cause, de manière inconsciente, des propriétés importantes d'un composant lors d'une évolution. Toute décision concernant l'architecture du composant, devient explicite et vérifiable. Cela assure une meilleure cohérence entre les différentes versions d'un composant.

Sur le plan conceptuel, nous prévoyons d'étudier l'impact de la composition sur le contrat d'évolution. Pour l'instant, le concepteur doit, de lui-même, déduire cet impact. Nous prévoyons également de formaliser plus avant notre mécanisme d'association.

Sur le plan de l'outillage nous envisageons deux prolongements possibles :

1. Pour l'instant, l'outil `ACE` ne prend en compte que des descripteurs d'architecture définis avec le modèle `Fractal`. Mais, il a été conçu pour intégrer facilement d'autres modèles de composants. Nous confrontons le méta-modèle UML 2.0 avec notre méta-modèle. Nous espérons, à la fin de ce travail, pouvoir proposer un méta-modèle suffisamment générique pour supporter tous les modèles de composants du marché.
2. L'intégration de notre outil de vérification du contrat d'évolution dans un AGL permettrait de guider, de manière continue, l'auteur de l'évolution. Après chaque modification de l'architecture, une évaluation du contrat serait réalisée en arrière plan. L'auteur de l'évolution serait, donc, prévenu de la violation du contrat avant d'aller plus avant dans ses modifications.

Pour plus d'information sur notre travail ou pour télécharger l'outil `ACE`, le lecteur est invité à visiter l'adresse suivante :
http ://www-valoria.univ-ubs.fr/Composants/se/current/Cell.

Références

1. R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 1997.
2. L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice, 2nd Edition*. Addison-Wesley, 2003.
3. L. Bass, M. Klein, and F. Bachmann. Quality attribute design primitives and the attribute driven design method. In *Proceedings of the 4th International Workshop on Product Family Engineering*, Bilbao, Spain, 2001.
4. K. Bennett. Software evolution : past, present and future. *Information and Software Technology*, 38(11) :671–732, 1996.
5. B. Bokowsky. Coffeestrainer : Statically-checked constraints on the definition and use of types in java. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pages 355–374, Toulouse, France, 1999. Springer-Verlag.
6. E. Bruneton. Fractal adl tutorial, version 1.2. http ://fractal.objectweb.org/tutorials/adl/, 2004.
7. L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.
8. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *In Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 175–188, New Orleans, Louisiana, USA, 1994.
9. ISO. Software engineering - product quality - part 1 : Quality model. International Organization for Standardization web site. ISO/IEC 9126-1. http ://www.iso.org, 2001.
10. N. Klarlund, J. Koistinen, and M. I. Schwartzbach. Formal design constraints. In *Proceedings of the 11th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 370–383, San Jose, California, USA, 1996. ACM Press.
11. M. Lehman and L. Belady. *Program Evolution : Process of Software Change*. London : Academic Press, 1985.
12. B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Communiactions of the ACM*, 24(11) :763–769, 1981.
13. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *In Proceedings of the Fifth European Software Engineering Conference*, Barcelona, Spain, September 1995.
14. J. McKee. Maintenance as function of design. In *Proceedings of AFIPS National Computer Conference*, pages 187–193, Reston, Virginia, USA, 1984.
15. OMG. Uml 2.0 ocl final adopted specification. Object Management Group web site : http ://www.omg.org/docs/ptc/03-10-14.pdf, 2003.
16. R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems : Software Technologies, Engineering Processes, and Business Practices*. SEI Series in Software Engineering. Pearson Education, 2003.
17. T. Universität Dresden. Ocl compiler web site. http ://dresden-ocl.sourceforge.net/, 2002.
18. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS-USA)*, pages 112–124, Santa Barbara, California, USA, August 1998. IEEE Computer Society.

Session Posters et Démonstrations

Infrastructure logicielle pour l'auto-management d'unités mobiles. Application à l'autotest pour téléphones portables.

JOURNÉES NATIONALES DU GDR GPL

RIDENE Youssef

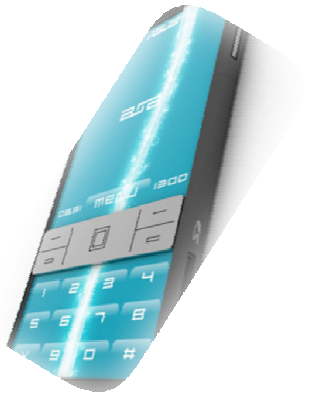
Directeur : Franck Barbier (LIUPPA)

Co-directrice : Nadine Couture (ESTIA)

Keywords:
Self-managing
Self-healing
Built-in test
Self-configurating
Managed code.

Descriptif du travail de recherche

Dans le cadre de cette thèse, nous travaillons sur la notion *d'autonomic computing* appliquée aux logiciels embarqués sur téléphones mobiles. Nous nous intéressons particulièrement aux « built-in tests ». Notre objectif est d'apporter une solution à une problématique industrielle assez concrète liée aux contraintes de test d'applications dans leurs environnements de déploiement. En effet, le test traditionnel en phase de développement ne permet pas d'anticiper les conditions de déploiement où nombre de paramètres de configuration accompagnent le logiciel (logiciel déployé sur unités mobiles, environnement de communication fluctuant dû à la mobilité…). Au-delà des principes d'*auto-adaptation*, *d'auto-configuration*… propres à l'*autonomic computing*, l'idée de la recherche ici proposée est de faire persister du code de test au déploiement pour doter le logiciel de moyens de réagir aux conditions d'usage mal déterminées au moment de développement.



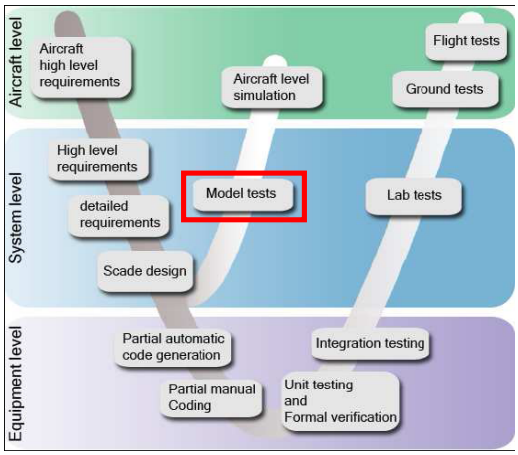
Equipes de travail

NEOMADES — UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR — ESTIA ECOLE D'INGENIEURS CCI BAYONNE PAYS BASQUE

Contact: youssef.ridene@neomades.com

LETO – A Lustre-based test oracle for Airbus critical systems

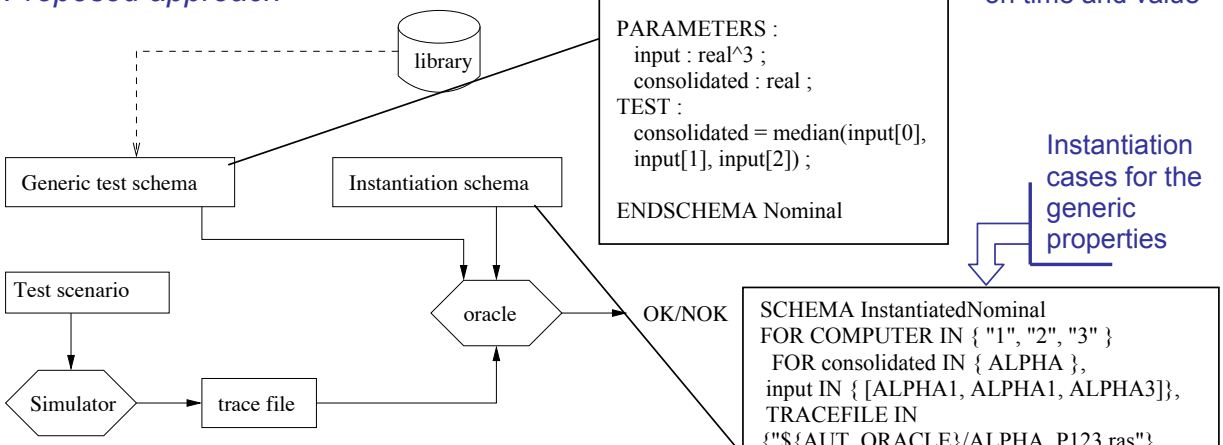
Context – Airbus development process



Objective: Automation of the test oracle procedure

Currently:
- Manual oracle
- Possible automation for regression testing but results have to be strictly identical
- A number of parameterization variants for similar tests

Proposed approach



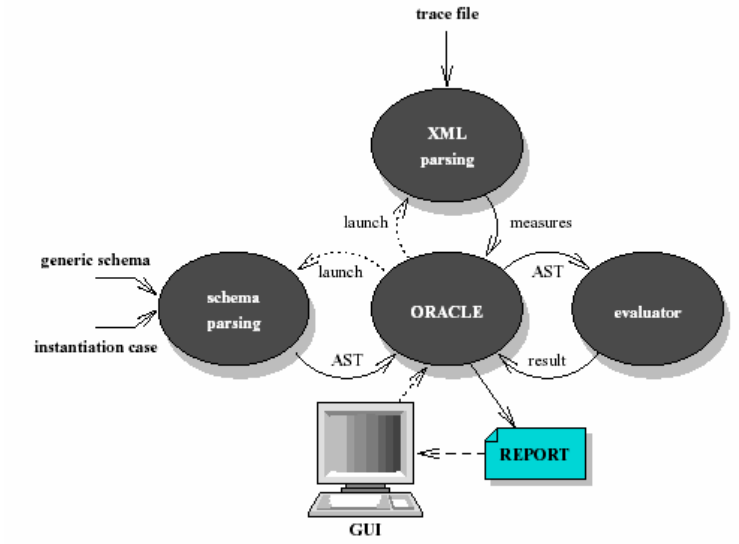
Generic declarative properties
Possible tolerance on time and value

```
SCHEMA Nominal

PARAMETERS :
  input : real^3 ;
  consolidated : real ;
TEST :
  consolidated = median(input[0],
  input[1], input[2]) ;

ENDSCHEMA Nominal
```

Instantiation cases for the generic properties

```
SCHEMA InstantiatedNominal
FOR COMPUTER IN { "1", "2", "3" }
  FOR consolidated IN { ALPHA },
 input IN { [ALPHA1, ALPHA1, ALPHA3]},
 TRACEFILE IN
{"${AUT_ORACLE}/ALPHA_P123.ras"}

      Nominal(input, consolidated)
 ENDFOR
ENDFOR
ENDSCHEMA InstantiatedNominal
```

Implementation



RESULTS
Experiments on Airbus A380 flight control system (ADIRS, Pilot sticks)
- ⇨ Automated analysis of real test sets
- ⇨ Adequate handling of genericity

Practical exercises with Airbus testers

This work was supported by an Airbus contract

MOCAS : un modèle de composant basé état pour l'adaptation dynamique

http://mocasengine.sourceforge.net/

UNIVERSITÉ DE PAU ET DES PAYS DE L'ADOUR

Génie Logiciel

Ingénierie des Modèles

Modèle de Composant

Machine à états

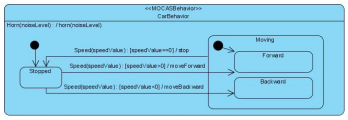
LIUPPA

Problématique : comment modifier le comportement d'un composant logiciel alors qu'il est en cours d'exécution ?

- Comment définir ce comportement et le modéliser ?
- Quand faut-il et quand peut-on intervenir sur le composant ?
- Comment le modifier de manière cohérente ?

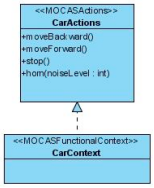
Le comportement d'un composant MOCAS

- Est défini par le concepteur du composant ;
- Est un ensemble d'états :
  - -Discrétisant une (plusieurs) des propriétés du composant,
  - -Établissant la séquence d'appel de ses services (*protocole*).
- Représente alors :
  - -Les valeur de la propriété,
  - -Les étapes dans le protocole du composant.



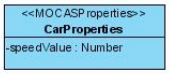
La logique métier d'un composant MOCAS

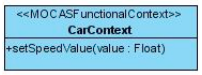
- Concerne l'algorithmique et les structures de données du composant ;
- Est séparée distinctement du comportement grâce à une interface d'actions internes au composant ;
- Est réalisée par le contexte fonctionnel ;
- Est substituable dynamiquement dans le respect de l'interface d'actions.



Les propriétés métier d'un composant MOCAS

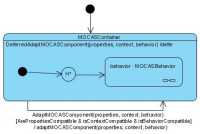
- Sont caractéristiques du « métier » du composant ;
- Sont définies par un type, un nom et une valeur ;
- Sont accessibles aux autres composants en lecture et/ou en écriture suivant les besoins du contexte fonctionnel ;
- Reposent sur des structures de données propres aux besoins du contexte fonctionnel.

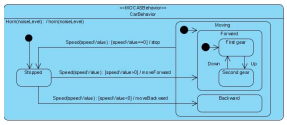




Le conteneur de composant MOCAS

- Réalise l'adaptation dynamique du composant ;
- Vérifie la compatibilité du nouveau comportement avec le comportement courant ;
- Opère lorsque le composant est *quiescent* grâce au mode de fonctionnement *run-to-completion* ;
- Assure le transfert d'états permettant la reprise du fonctionnement après l'adaptation ;

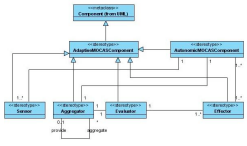




Vers l'Autonomic Computing

Chaque composant MOCAS supporte des composants particuliers formant sa boucle de contrôle :

- Des sondes internes/externes au composant et réactives



(qui attendent un événement particulier) ou proactives (qui sondent en permanence le système);

- Un agrégateur qui centralise les informations reçues des sondes et produit des événements de plus haut niveau;
- Un évaluateur qui reçoit les informations de l'agrégateur et cherche une solution appropriée;
- Des effecteurs qui réalisent chacun une solution particulière.

Coordination de l'adaptation

Un composant fait partie d'un assemblage de composants. Les composants MOCAS communiquent par l'échange asynchrone de signaux structurés. Afin de garantir la cohérence de l'assemblage, ces signaux sont échangés suivant un protocole d'interaction comme FIPA Request. Un composant peut donc émettre un veto contre l'adaptation d'un autre composant si cela va à l'encontre de sa propre stabilité.

Équipe Self-Star : Agents, Composants et Services Autonomiques

http://liuppa.univ-pau.fr/

Cyril Ballagny
Doctorant
cyril.ballagny@univ-pau.fr

Franck Barbier
Directeur de Thèse
franck.barbier@univ-pau.fr

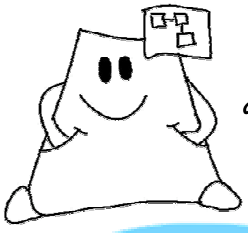
Nabil Hameurlain
Co-encadrant
nabil.hameurlain@univ-pau.fr

GARANTIE DES CONFORMITÉS PAR REJEU DE PATTERNS

PRÉSENTATION DES PERSONNAGES :

LE DÉVELOPPEUR

Définit un modèle devant vérifier plusieurs contraintes.



dont l'application garantit la mise en place des conformités associées

LES PATTERNS CONFORMANTS

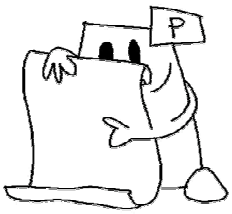
Expression structurelle des requis et contraintes liés à une problématique à travers des modèles et des transformations de modèles.

SCÉNARIO DE REJEU :

L'évolution de ce modèle va prendre en compte :
- L'authentification des connexions au serveur par pattern
- L'ajout d'une relation d'un client au serveur par action utilisateur

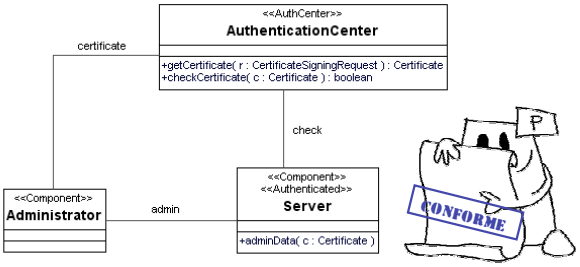
Le développeur applique le pattern P pour l'authentification.

Conformité de P :
- Toutes les méthodes de Server prennent en paramètre d'entrée un Certificate.
- Toutes les classes reliées au Server doivent aussi être reliées au Centre d'Authentification.

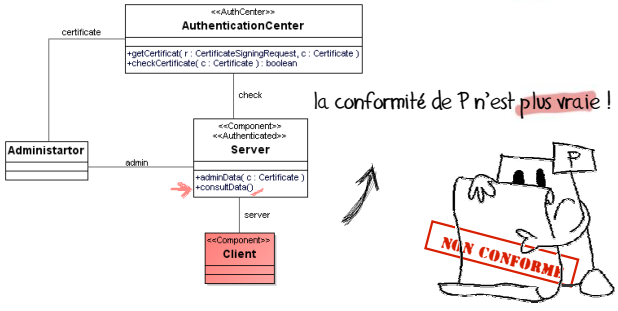


P est défini par des transformations de modèles (Prolog).

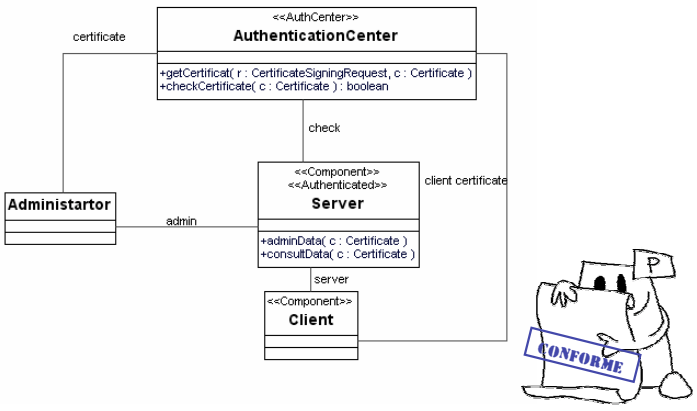
Un nouveau modèle vérifiant la conformité de P est généré par application des transformations.



Le développeur met en place la relation Client/Server, mais ...



Le pattern P est alors rejoué : les transformations sont ré-appliquées et leurs actions réparent la conformité.



CONCLUSION :

- Les patterns conformants sont définis par des métamodèles et des transformations de modèles.
- L'application des transformations établit ou répare les conformités.
- L'application d'un pattern ne fait rien si ses conformités sont vraies (Idempotence).

CLÉMENTINE NEMO
MIREILLE BLAY-FORNARINO
modalis.polytech.unice.fr/~clementine

http://www.lina.sciences.univ-nantes.fr/coloss/

COLOSS Team

Dependable Components and Systems
(Composants et Logiciels Sûrs)

☞ Research domains

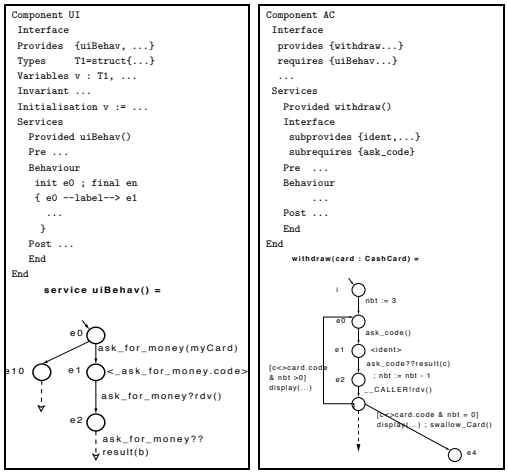
- Models, semantics, languages
- Concepts, methods and mechanized techniques to help designers and developers
- Object and component based modelling
- Formal models to specify and verify system properties
- Multi-facet analysis and specification
- Specification, verification and validation of software (components)

☞ Objectives and Applications

- Techniques for developing dependable components and systems
- Tools for the analysis and development of embedded, concurrent, reactive and mobile systems.

The Kmelia/COSTO Project (1)

Specifying with Kmelia



Expressive and Flexible specifications

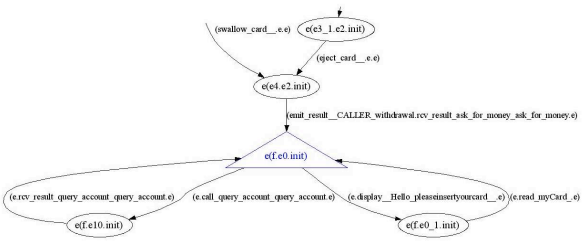
Assemblies	Composition	Interaction
Renaming	Multiprotocol	Subservices

to be verified using COSTO

The Kmelia/COSTO Project (2)

COSTO : Tools, Analysis and Results

- Eclipse Plugins, Graphical facilities
- Bridges with formal languages and tools : Lotos/cadp, Mec
- Exports into various formats
- Document generation (dot, tex, ...)
- Refinement into executable frameworks



Extract from a generated deadlock detection report
(Kmelia → Mec → KmeliaReport)

Products researched or developed

- COSTO (COmponent Study TOolbox) : a design and verification package for studying components based on services
- BOSCO : a template-based generator of repositories from metamodels
- NaBLa (Nantes B Libraries) : formal component libraries specified with the B Method
- ORYX/Atacora : an engineering workbench for the specification and formal multi-facet analysis of integrated software systems

☞ Techniques and Methods Used

Z, B, Process Algebra, PVS, LOTOS, MEC, Petri-Nets, Grafcet, SPIN, etc

☞ Services - Partnerships

- Consulting in modelling systems
- Abstraction and formal analysis of complex systems
- Expertise, courses and training in object and component based modelling (UML, etc)
- Expertise, courses and training in formal methods
- Partnerships for supervising PhD students involved in industrial projects

http://www.lina.sciences.univ-nantes.fr/coloss/

Safecode

Projet ANR ARA-SSIA 2005

Développement de composants sûrs de fonctionnement

Objectif

Élaboration d'un patron de conception permettant l'expression des exigences de sûreté de fonctionnement (SdF) et leur traçabilité tout au long du développement du composant

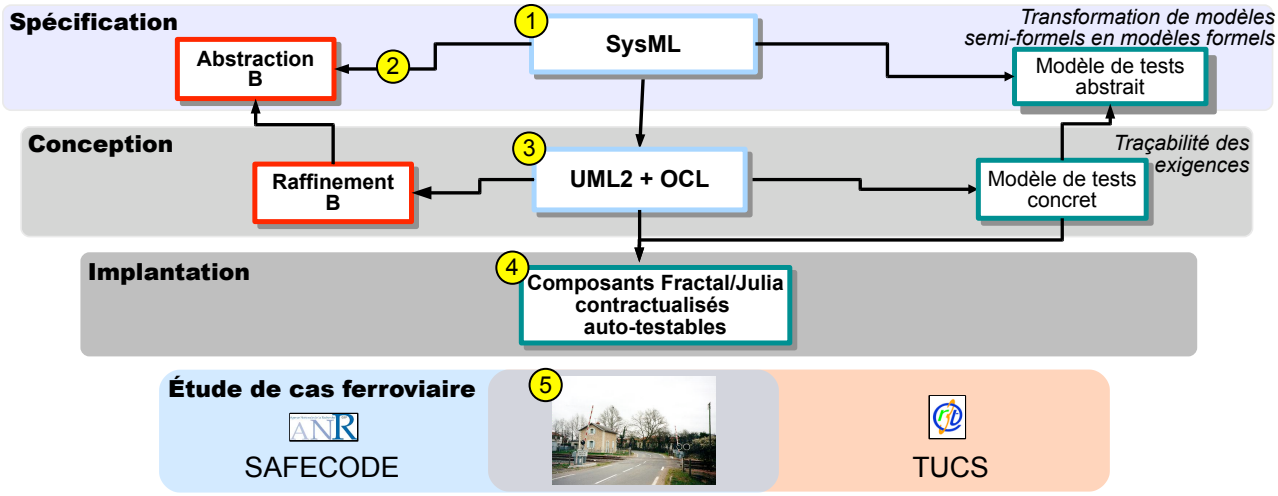
Verrous

- Absence d'exigences de SdF dans les technologies à base de composants
- Processus de certification actuel inadapté aux nouvelles approches par composants

Partenariat

Vérification	*Spécification*	*Tests*
INRETS	HEUDIASYC	VALORIA
Exigences sûreté	**Modélisations**	**Conceptions pour le test**

Démarche



Résultats

1. Sélection de profils UML et guides méthodologiques (SAFECODE-TUCS)
   - *Identification des exigences de sécurité et de sûreté de fonctionnement (ICSEA 2007)*
   - *Mise en oeuvre de SysML pour la conception d'architecture et l'analyse des exigences (ICSOFT'07, rapport WP 3.1)*
2. Traduction UML/B
   - *Couplage de notations UML et B : Aperçu de l'existant (rapport WP 1.3)*
   - *Expérimentation d'un outil de transformation UML vers B pour les systèmes critiques (CAINE 2007)*
3. Composants UML2 avec contrats
   - *Définition d'un méta-modèle de composants hiérarchiques contractualisés compatible avec Fractal (rapport WP 5.2)*
   - *Définition d'un processus de conception à base de composants intégrant les contrats et les tests (rapport WP 3.2)*
4. Fractal, contrats et tests intégrés
   - *Spécification du framework de composants contractualisés auto-testables pour Fractal (CBBT) basé sur ConFract de Ph. Collet (rapport de master, workshop ECOOP'06, soumission ICST 2009)*
5. Étude de cas « passage à niveau »
   - *État de l'art : De UML à la méthode B pour modéliser un passage à niveau (RTS 2007)*
   - *Analyse et modélisation des exigences basées sur une approche SysML (rapport WP 6.1)*
   - *Conception UML2/Composants du passage à niveau (rapport WP 6.2)*

Perspectives

Affinage des règles de transformation UML/B pour SAFECODE

Choisir des règles de transformations favorisant la vérification automatique des spécifications B

Enrichir le prototype de transformation UML/B avec ces nouvelles règles

Précision de la prise en compte des aspects critiques dans la modélisation SysML

Définition de règles de traduction des spécifications SysML vers UML2+OCL

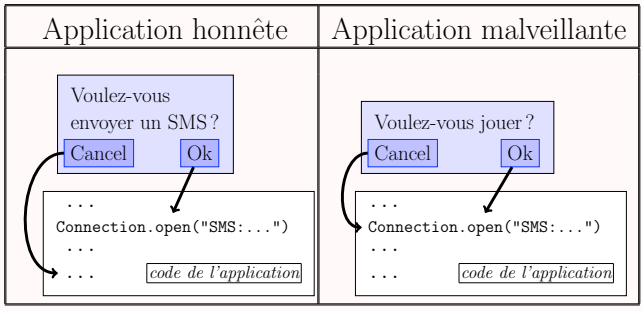
Écriture du prototype du framework de composants CBBT Fractal

Prototype pour l'extraction de tests et de contrats à partir des diagrammes UML

Contact : Walter Schön – UTC/HEUDIASYC UMR 6599 - walter.schon@hds.utc.fr - https://gslt.inrets.fr/projet/safecode

RAVAJ | Réécriture pour la vérification d'applications Java

Quels sont les risques ?



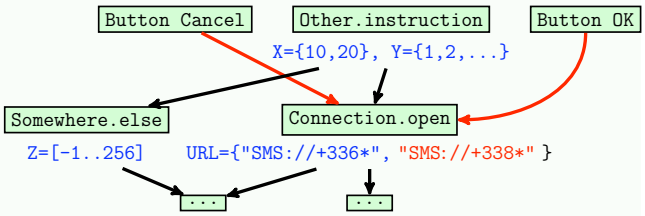
Dans cet exemple simple, l'application « honnête » demande la permission à l'utilisateur avant d'utiliser une ressource critique (ici l'envoi d'un SMS). A l'inverse, une application malveillante peut accéder aux ressources sans en informer l'utilisateur. Dans notre exemple, un SMS sera expédié sans que l'utilisateur en soit avisé.

Pour en savoir plus : Dans le cas d'applications Java MIDP pour téléphone mobile, les applications non certifiées par l'opérateur (la très grande majorité des applications disponibles) peuvent être exécutées dans un *mode sécurisé*. Dans ce mode, quelle que soit l'application, à *chaque fois* que celle-ci accède à une ressource critique, un message demande à l'utilisateur son accord pour l'utilisation de cette ressource. Ce procédé offre un très bon niveau de sécurité. En revanche, il n'est pas utilisable pour des applications utilisant *fréquemment* des ressources critiques (par exemple un navigateur internet), puisqu'il noie l'utilisateur sous des messages de confirmation. De telles applications doivent, préférablement, être certifiées par l'opérateur. Ceci a pour effet de désactiver les messages de sécurité. Cependant, l'opérateur doit avoir des garanties sur l'innocuité de l'application avant de la certifier. Ce qui revient au problème initial.

Comment éviter les mauvaises surprises ?

En vérifiant, à l'aide d'un *analyseur statique*, le code de l'application. Son rôle consiste en particulier à calculer une valeur approchée pour :

1. le **graphe d'appel** de l'application
2. l'ensemble des **valeurs possibles** des paramètres des méthodes



Dans notre exemple, l'analyse statique de l'application malveillante révèle deux problèmes. En premier, les noeuds (`Cancel` et `Ok`) du graphe mènent à la méthode `Connect.open`. En second, l'estimation des valeurs possibles pour l'URL de cette méthode comporte le préfixe de numéro surtaxé (`+338*`).

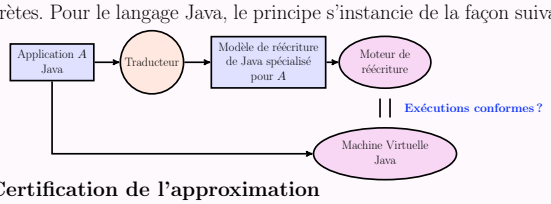
Pour en savoir plus :
La construction du graphe d'appel et l'estimation des valeurs possibles d'un programme sont des problèmes indécidables en général. Les analyseurs statiques ne savent, donc, construire que des sur-approximations. En conséquence, s'ils trouvent des problèmes, comme dans notre exemple, il faut procéder à une vérification plus fine de cette partie afin de déterminer s'il s'agit d'un risque réel ou si cela est dû à une approximation trop forte. A l'inverse si l'analyseur statique dit qu'il n'existe pas de faiblesse, ceci consiste une preuve de sécurité.

Quelles garanties sur l'analyseur ?

D'une part, il faut assurer la conformité entre le modèle de réécriture et l'application initiale. D'autre part, il faut prouver que l'analyse réalisée sur le modèle de réécriture est sûre. Pour ce faire, on réalise une preuve formelle certifiant que l'approximation est bien un sur-ensemble des états accessibles de l'application à analyser.

1 Conformité modèle/application

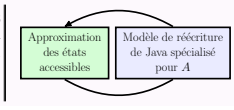
RAVAJ souhaite tirer partie de *l'exécutabilité* des systèmes de réécriture pour tester la conformité des exécutions de réécriture avec les exécutions concrètes. Pour le langage Java, le principe s'instancie de la façon suivante :



2 Certification de l'approximation

Un certificateur d'approximation pour les modèles de réécriture est développé dans l'assistant de preuve Coq. Un tel certificateur sera, comme l'analyseur, *indépendant* du langage et de la propriété cible.

Pour en savoir plus : Le certificateur prouve qu'en appliquant le modèle de réécriture aux états accessibles de l'approximation, on obtient *uniquement* des états déjà représentés dans l'approximation. En d'autres termes, l'approximation est *complète* pour (le modèle de) l'application A.

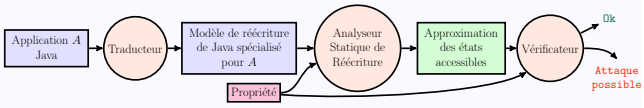


Comment garder l'analyseur à jour ?

Un analyseur est *très dépendant* du langage et des librairies utilisées pour la conception de l'application à vérifier. Le projet ANR **RAVAJ** souhaite dissocier les deux en traduisant l'application et les librairies dans un format intermédiaire, *un système de réécriture*, et en réalisant l'analyse sur ce dernier. L'intérêt est multiple :

- un traducteur est plus simple à réaliser et à mettre à jour qu'un analyseur
- pour des analyses simples, les analyseurs de réécriture sont plus facilement adaptables à la propriété à vérifier que les analyseurs classiques
- il est plus facile de maintenir et d'optimiser *un seul analyseur* dont le format d'entrée n'évolue pas qu'un exemplaire d'analyseur par type ou version d'un langage.

Par exemple, dans le cas du langage Java, le schéma d'analyse est le suivant :



Pour en savoir plus : A partir du modèle de réécriture et de la propriété, l'analyseur de réécriture calcule un sur-ensemble de tous les états accessibles de l'application. Ensuite le vérificateur, s'assure qu'aucun des états du sur-ensemble ne viole la propriété. Comme on ne connaît pas a priori les entrées de l'application, le sur-ensemble des états accessibles peut être non borné. Afin de représenter de façon finie cet ensemble, on utilise des automates d'arbres qui permettent de décrire finiment des ensembles infinis d'états accessibles. Les partenaires IRISA et LIFC de **RAVAJ** ont déjà l'expérience de ce type d'outils formels et de leur utilisation en analyse de programmes (protocoles cryptographiques, byte code Java).

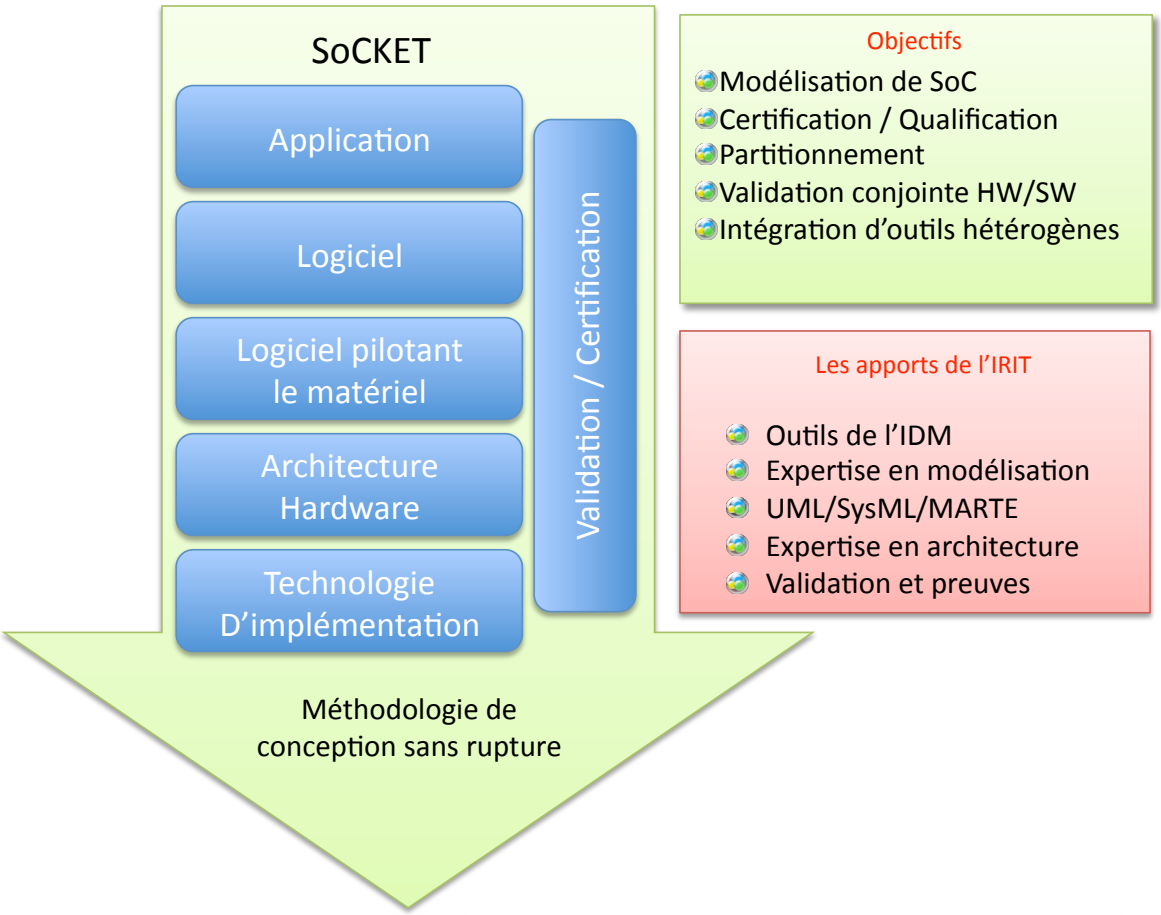
Partenaires **RAVAJ**	Équipe	Laboratoire
Benoît Boyer, Thomas Genet, Thomas Jensen, Vlad Rusu	Lande et Vertecs	IRISA (Rennes)
Pierre-Cyrille Héam, Olga Kouchnarenko	Cassis	LIFC (Besançon)
Emilie Balland, Pierre-Étienne Moreau	Paréo	Loria (Nancy)

Site web : `http ://www.irisa.fr/lande/genet/RAVAJ`
Contact (coordonnateur) : `genet@irisa.fr`
Membres extérieurs : Yohan Boichut LIFO Orléans
Observateur extérieur : Pierre Crégut, France Telecom R&D, AMS/SLE

SoCKET

Projet DGE
11 partenaires
3,7 M€
2008 - 2011
3 équipes IRIT
Double pôles mondiaux de compétitivité

SoC toolKit for critical Embedded Systems



Objectifs

- Modélisation de SoC
- Certification / Qualification
- Partitionnement
- Validation conjointe HW/SW
- Intégration d'outils hétérogènes

Les apports de l'IRIT

- Outils de l'IDM
- Expertise en modélisation
- UML/SysML/MARTE
- Expertise en architecture
- Validation et preuves

Pôles de compétitivité

Partenaires Industriels

Université de Toulouse

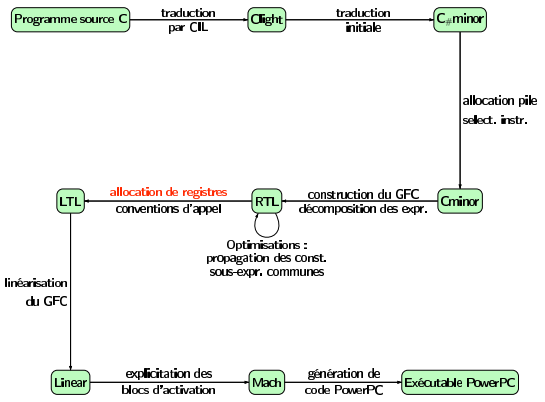
Laboratoire CNRS IRIT
Equipes ACADIE/MACAO/TRACES

Poster GdR – GPL - 2009

Vérification formelle et optimisation de l'allocation de registres

Benoît ROBILLARD (http://www.ensiie.fr/~robillard)
sous la direction de Sandrine BLAZY et Éric SOUTIF
Laboratoire CEDRIC, CNAM-ENSIIE
GDR GPL, Groupe de travail LTP

L'allocation de registres du compilateur CompCert



Allocation de registres

- Une des phases les plus complexes du processus de compilation
- Variables stockées en registres (rapides, limités) et mémoire sinon
- But : choisir où placer les variables pour optimiser les accès
- Règles :
  - Placer les variables qui interfèrent dans des registres différents
  - Placer les variables liées par une instruction d'affectation dans un même registre (optimisation du code généré)
- Trois sous-problèmes
  1. Splitting : copier des variables afin de limiter les interférences
  2. Spilling : placer en mémoire des variables pour libérer des registres
  3. Coalescing : éviter des transferts (affectations) inutiles

Le projet CompCert (ANR-05-SSIA-0019)

- Compilateur de *C*, spécifié, écrit et vérifié avec Coq
- Compilateur traduit en Caml par le mécanisme d'extraction de Coq
- Preuve d'équivalence observationnelle entre source et cible
- Allocation de registres écrite en Caml vérifiée *a posteriori*

Un problème de coloration à différents niveaux d'optimisation

Modélisation par un problème de coloration (du graphe d'interférence)

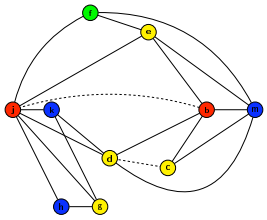
Splitting plus précis = plus de copies ⇒ allocation de registres plus précise mais plus difficile (combinatoire élevée)
Graphe d'interférence :

- les sommets sont les variables du programme
- deux variables qui interfèrent sont liées par une arête d'interférence
- deux variables d'une affectation sont liées par une arête de préférence

Une couleur est un registre ⇒ une coloration (respectant les interférences) est une allocation de registres
Spilling = rendre le graphe colorable
Coalescing = choisir la coloration qui attribue la même couleur aux extrémités d'un maximum d'arêtes de préférence

Programme sans splitting (usuel mais peu précis)

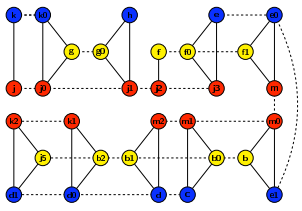
```
Live-in : k j
g := mem[j+12]
h := k-1
f := g*h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e+8
d := c
k := m+4
j := b
Live-out : d k
```



Graphe triangulés en majorité
Heuristique d'Appel et George (CompCert)
Algorithme de coloration gourmande vérifié formellement

Programme avec splitting extrême (le plus précis)

```
Live-in : k j
k0 := k||j0 := j||g := mem[j0+12]
j1 := j0||g0 := g||h := k0-1
j2 := j1||f := g0*h
f0 := f||j3 := j2||e := mem[j3+8]
e0 := e||f1 := f0||m :=mem[j2+16]
m0 := m||e1 := e0||b :=mem[f0]
b0 := b||m1 := m0||c :=e0+8
b1 := b0||m2 := m1||d := c
d0 := d||b2 := b1||k := m+4
k2 := k1||d1 := d0||j5 := b2
Live-out : d k
```



Spilling optimal par PL
Coalescing optimal par PL + coupes mais explosion combinatoire
Algorithme de réduction : résolution 300 fois plus rapide du coalescing (testé sur l'optimal coalescing challenge)

Allocation de registres par programmation linéaire

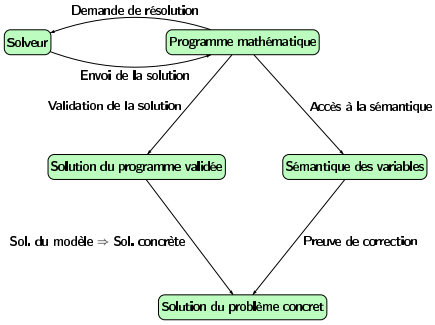
Variables

$x_{ic} = 1 \Leftrightarrow i$ est de couleur c
$y_{ij} = 1 \Leftrightarrow i$ et j sont de couleurs différentes

Modèle

$$(P)\begin{cases} Min & \sum_{e \in E_{pref}} w_e y_e \\ s.c. & \\ \forall i \in \{1,\dots,n(G)\}, & \sum_{c=1}^{K} x_{ic} = 1 \\ \forall (i,j) \in E_{int},\ \forall c \in \{1,\dots,K\}, & x_{ic} + x_{jc} \leq 1 \\ \forall (i,j) \in E_{pref},\ \forall c \in \{1,\dots,K\}, & x_{ic} - x_{jc} \leq y_{ij} \\ \forall i \in \{1,\dots,n(G)\},\ \forall c \in \{1,\dots,K\}, & x_{ic} \in \{0,1\} \end{cases}$$

Schéma de preuve



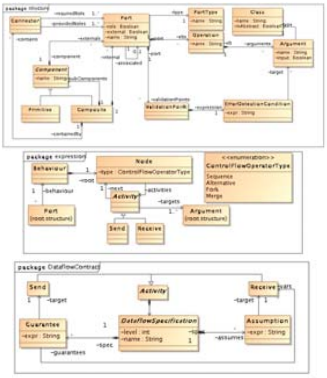
A Component AssemblY Interaction Control framewOrk

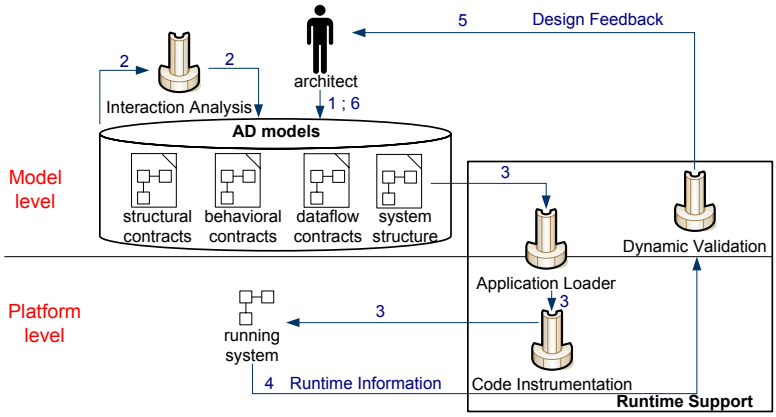
CALICO is a model-based approach to statically and dynamically verify component interactions. Models are used from design time to testing time for a given execution context :

- To specify the application structure and properties, independently of any component platform
- To statically determine incompatible or partially compatible component interactions
- To instantiate the application for a target platform and instrument it to detect any potential incompatible interaction at execution time
- To specify modifications to apply to the running application in order to fix the detected problems

1 : Design

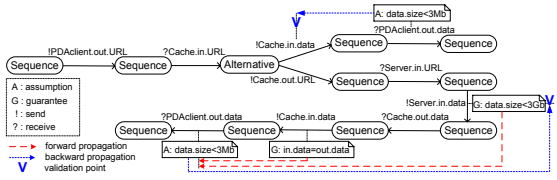
The architect defines his/her application through several views : structural, behavioral and dataflow models.





2 : Static interaction analysis

Compatibility of component interactions is checked across the entire component assembly. Partially compatible interactions are identified with *validation points* and their check will be resumed at execution time.



3 : Instanciation and instrumentation

At the platform level, CALICO instruments the application code using aspects to enable the capture of the needed runtime information independently of the underlying platform, *e.g.,* tracing information to check the application dataflow properties. The application is loaded on the target platform with respect to the structural model.

4 : Dynamic interaction verification

Needed runtime information is intercepted in the executing application to complete the pending check of partially compatible interactions.

5 : Design feedback

The coupling between the model level and runtime level enables architects to easily locate model elements that led to runtime application errors.

6 : Design update

Accordingly to design feedback, architects can correct his/her design. An *update model* is generated through the comparison with the initial model. The update model contains a sequence of operations for modifying the system structure, such as, adding or removing components and connectors to/from the system.

Status

CALICO is generic and its model approach enables interaction compatibility checks even if the underlying platform does offer any verification tools.

The current implementation of CALICO supports different target component platforms, such as Fractal, OpenCCM and OpenCom, and different kinds of properties, such as structural constraints expressed with OCL or behavioral constraints.

CALICO is implemented as an Eclipse plugin, with EMF and Spoon.

http://calico.gforge.inria.fr

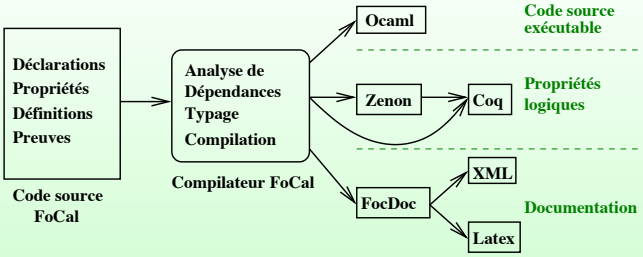
Guillaume.Waignier@inria.fr
Anne-Francoise.Le_Meur@inria.fr
Laurence.Duchien@inria.fr

FoCalTest : Un outil de test automatique pour FoCal

Matthieu CARLIER
carlier@ensiie.fr
sous la direction de
Catherine DUBOIS
dubois@ensiie.fr
Laboratoire CÉDRIC, CNAM-ENSIIE
GDR GPL, groupe de travail MTV²

FoCal : Environnement de développement de logiciel certifiable

- Un programme FoCal ≡ spécification ∪ implantation
  - spécification : propriétés du premier ordre
  - implantation : langage de programmation à la ML
- Raffinement : héritage, redéfinition
- Preuves : Zenon (prouveur automatique), Coq
- Compilé vers OCaml (implantation) et Coq (preuves/spécification)



Pourquoi tester ?

Propriétés non prouvées/prouvables

- Tester les axiomes
  Ai-je bien implanté mes fonctions ?
- Tester à partir de spécifications externes
  Mon implantation correspond-elle à cette spécification ?
- Tester les propriétés sur du code importé de OCaml
  Elles ne sont pas prouvables mais sont-elles vérifiées ?

Propriétés en cours de preuve

- Tester avant de faire une preuve
  Y a-t-il des contre-exemples ?
- Tester pendant une preuve
  Pourquoi n'arrivé-je pas à prouver mon lemme ?
  Mes lemmes intermédiaires sont-ils vérifiés ?

FoCalTest : Test de propriétés

Propriétés sous test

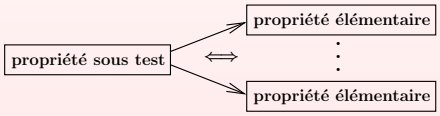
$$\forall X_1 \cdots X_n,\ \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n \Rightarrow (A_1^1 \vee \cdots \vee A_{n_1}^1) \wedge \cdots \wedge (A_1^m \vee \cdots \vee A_{n_m}^m)$$

$$\alpha ::= \alpha \vee \alpha \mid \alpha \wedge \alpha \mid A$$

$$A ::= \mathrm{f}(A, \ldots, A) \mid X$$

Propriétés élémentaires

- Propriété sous test réécrite en un ensemble de propriétés élémentaires



$$\forall X_1 \ldots X_n,\ \underbrace{A_1 \Rightarrow \ldots \Rightarrow A_m}_{\text{Précondition}} \Rightarrow \underbrace{B_1 \vee \ldots \vee B_m}_{\text{Conclusion}}$$

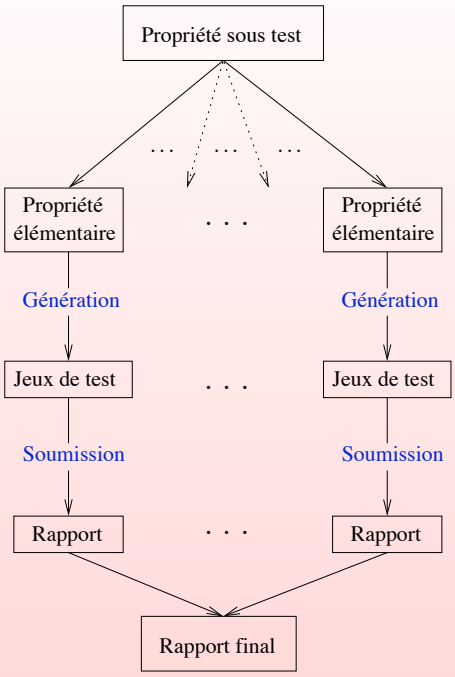
- Les propriétés élémentaires sont testées séparément

Jeux de test

- 1 jeu de test = 1 valuation des X_i qui satisfait la précondition
- Résultat du test : évaluation de la conclusion
- Deux stratégies de synthèse des jeux de test :
  1. Génération aléatoire : évaluation de la précondition
  2. Approche par contraintes :
     précondition traduite en 1 ensemble de contraintes
     1 solution du système ≡ 1 jeu de test

Rapport

- Calcul de la couverture MC/DC sur la conclusion
- Résultat dans un rapport de test au format XML

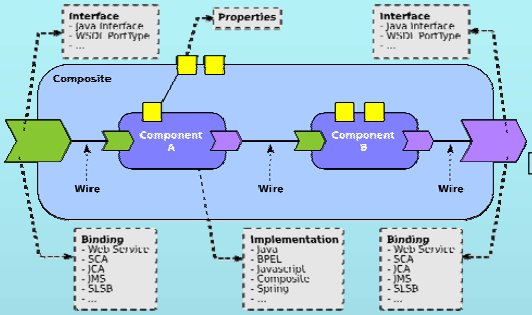


FraSCAti – An Open SCA Platform

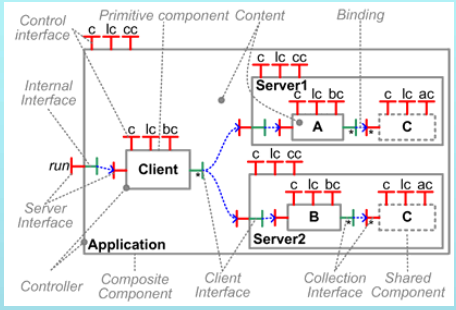
A Marriage of SOA & Components

http://frascati.ow2.org

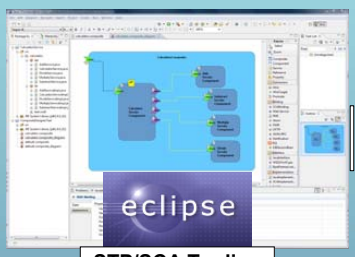
Development Team
Pierre Carton, Christophe Demarey, Nicolas Dolet, Damien Fournier, Philippe Merle,
Valerio Schiavoni, Lionel Seinturier (contact author)
FirstName.LastName@inria.fr



SCA - The standard component model for SOA



Fractal - The modular and reflexive component model



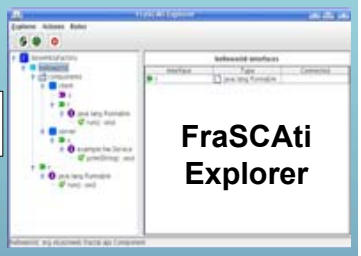
STP/SCA Tooling

deploy

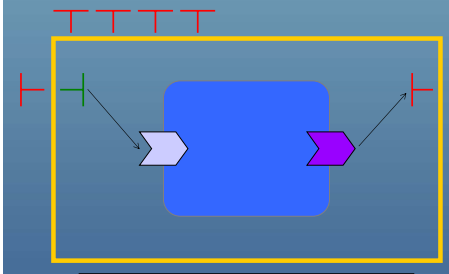
FraSCAti

An open SCA runtime platform built on top of OW2 Fractal

manage

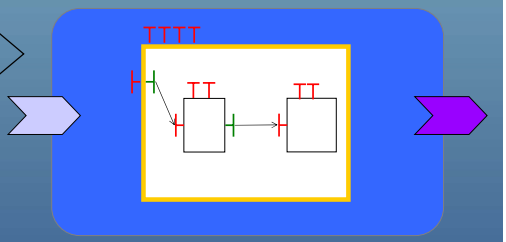


FraSCAti Explorer



Reconfigurable SCA Applications

run



SOA for Fractal

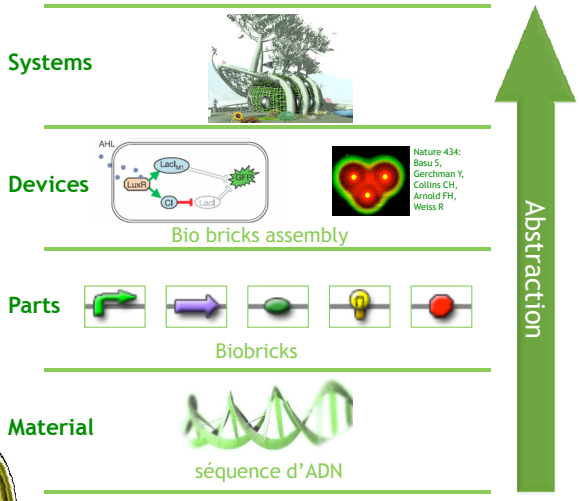
Distributed Enterprise Service Bus

With some fundings by

Programmer des Fonctions Biologiques Synthétiques

F. Delaplace – DYNAMIC, J.-L. Giavitto – LIS (IBISC, Univ. Evry), O. Michel, A. Spicher - LACL (PXII)
[delaplace, giavitto]@ibisc.univ-evry.fr, [michel,spicher]@univ-paris12.fr

La biologie synthétique a pour objectif de concevoir des fonctions biologiques nouvelles. Elle est actuellement en quête de principes de conception permettant une réalisation fiable et sécurisée. Ces principes sont similaires à ceux qui guident la conception des systèmes informatiques à travers une organisation hiérarchique par couche. Chaque couche correspond à un niveau d'intégration de fonctions biologiques de plus en plus complexes. Le composant fondamental correspond à la **biobrick** qui se définit à la fois comme une séquence d'ADN particulière et comme une fonction biologique élémentaire. Les unités fonctionnelles (devices) résultent de l'assemblage des biobricks et forment les parties d'un système.



Une hiérarchie de langages

[Physica D] Topological Rewriting and the Geometrization of Programming , J.-L. Giavitto and A. Spicher, Physica D 237 (2008).
[Unconv. Prog.] Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems , J.-L. Giavitto, O. Michel and A. Spicher LNCS 5380, Springer (2008).

Programmer les génomes synthétiques.

L'objectif est de concevoir et développer les outils permettant de **compiler** le *comportement global d'une population* (de bactéries) en des *processus cellulaires locaux* à chaque entité (bactérie).

L'approche se fonde sur une « tour » de langages de programmation dont le plus abstrait définit un modèle computationnel pour une population cellulaire et le plus fondamental correspond à un agencement de séquences d'ADN.

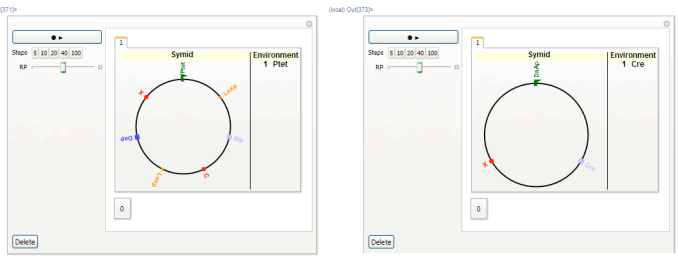
Dans cette approche, un programme ne définit plus une fonction qui associe une sortie à une entrée, mais un *système dynamique distribué* qui essaie de maintenir des invariants en dépit de perturbations et des changements de l'environnement.

Source Djyn

$$\{\sigma(\mathrm{Ptet} \longrightarrow \mathrm{Rfp} \Rightarrow \langle \mathrm{DaAp} \rangle \Rightarrow \nabla \mathrm{Dap})\}$$

Structure intermédiaire SYMID

```
{Symid[Sx[Ptet,Plus], Symid[Symid[Co[Loxp,Cut], Symid[Tag[Rfp,None]],
Ts[C,Stop], Co[Loxp,Cut]], Tag[Dap,Send]], Ts[X,Stop],
Symid[Sx[DaAp,Plus], Symid[Tag[Cre,None]],Ts[X,Stop]]}
```



Programme correspondant au projet SMB , IGEM Paris 2007

DJYN

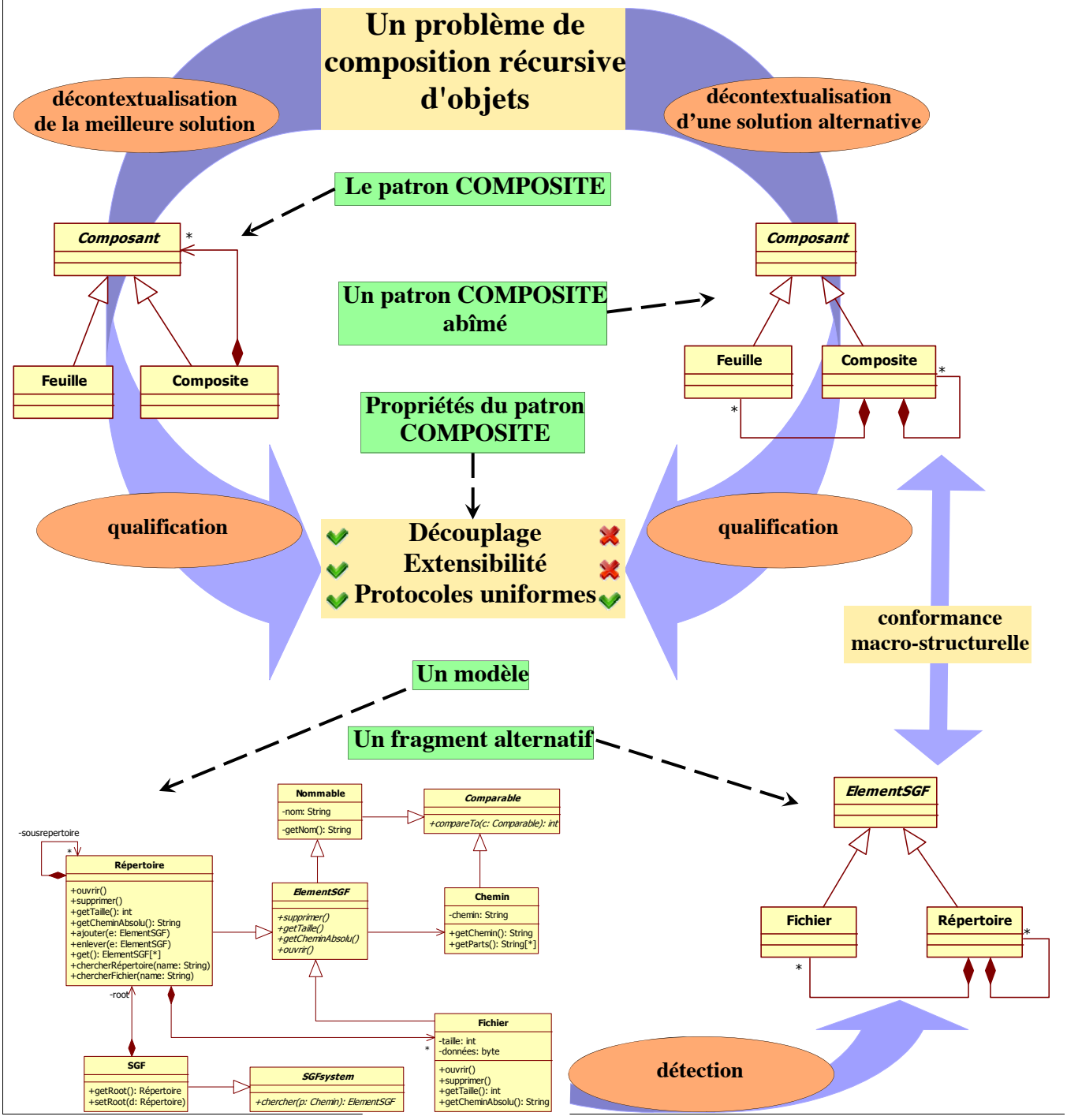
Djyn est un langage de programmation intermédiaire décrivant des processus cellulaires. Langage déclaratif réactif s'appuyant sur le paradigme senseur/actuateur, un programme Djyn spécifie un ensemble de règles. La partie gauche des règles correspond à une expression logique et la partie droite à une liste de « tags ». Ce langage se compile d'abord en une structure intermédiaire SYMID, abstraction d'un plasmide, traduit à son tour en un ou plusieurs plasmides. Une règle correspond à l'association d'un promoteur et d'une séquence codante.

« Bad smells » de conception et « leçons apprises » en conception objets

Cédric BOUHOURS, Hervé LEBLANC, Christian PERCEBOIS, {bouhours, leblanc, perceboi}@irit.fr - Université de Toulouse - UPS, IRIT, équipe MACAO

Une activité de revue de conception dirigée par les patrons de conception :

1) Détection automatique de bad smells de conception par la vue macro-structurelle d'un patron abîmé.
2) Vérification semi-automatique de l'intention et de l'intérêt de la substitution des bad smells détectés par interrogation d'une base de connaissances dédiée aux leçons apprises.
3) Substitution des bad smells par la contextualisation d'un patron de conception.



Extraction d'architecture guidée par les modèles

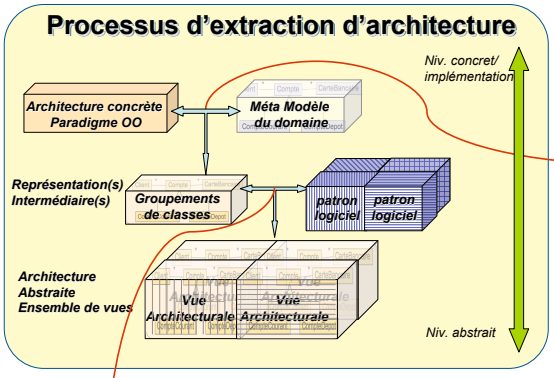
Aide pour la compréhension du code dans le contexte de l'évolution du logiciel

Projet ANR Cook
(partenaires: INRIA Lille, Université de Savoie)

Problèmes - Questions

- (Re)construction de vues fonctionnelles d'un système logiciel à partir du code source
- Mise en correspondance entre le paradigme objet (au niveau du code) et les paradigmes utilisés dans les langages de description d'architectures logicielles
- Les travaux existants pour l'extraction utilisent soit des techniques exploratoires (groupement des classes par *clustering* linguistique, cohésion, etc.) soit une connaissance a priori de l'architecture (techniques de reconnaissance)

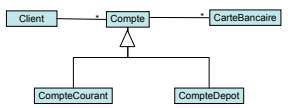
Processus d'extraction d'architecture



Clustering linguistique utilisant le méta-modèle du domaine

- **Entrée** : représentation (modèle) de l'architecture concrète du système avec l' environnement MOOSE
- **Sortie** : groupes des classes
- **Comment** : les classes contenant le même terme sont mis dans le même groupe
  - Le termes peuvent correspondre à des **concepts du domaine** ou à des relations entre les concepts de ce domaine
  - Les classes ne contenant aucun terme du méta modèle du domaine sont mises dans un groupe supplémentaire « hors domaine »
  - Le groupement peut être raffiné pour prendre en compte des *traits* de classes
    - La recherche des termes du méta modèle ne se limite plus aux simples noms de classes, mais se poursuit au niveau des noms de méthodes
    - Une classes peut être coupée en *traits* qui se trouvent dans des éléments architecturaux différents

Exemple d'une application bancaire



Trois groupes de classes:
- *Client*
- *Compte*
- *CarteBancaire*

Tissage avec des patrons logiciels

- **Entrée** : groupements de classes obtenus par techniques de *clustering* linguistique
- **Sortie** : architecture
  - Ensemble d'éléments architecturaux interconnectés
  - Les éléments architecturaux sont définis/donnés par le patron logiciel
  - Cette étape raffine chaque élément architectural avec les classes ou *traits* qui y sont attachés
- **Comment** :
  - Plusieurs patrons logiciels sont modélisés (MVC, Client Serveur, etc.)
  - Pour chaque patron considéré, une vue du système est générée
    - Des règles indiquent la façon dont les classes sont distribuées dans les différents éléments (architecturaux) du patron
    - Les classes (ou *traits*, avec leurs classes de provenance) sont redistribuées dans les différents éléments architecturaux
    - Une classe peut appartenir à plusieurs éléments architecturaux
    - Des sous ensembles de l'architecture issue de l'étape 1 se retrouvent comme sous structure de chaque élément du patron
    - Observation : les classes du groupe hors domaine sont évaluées pour être distribuées dans les différents éléments architecturaux
    - Les classes non distribuées sont mises dans un autre élément architectural « hors patron »

Exemple de règles pour MVC
1. *Chaque classe dont le nom correspond à un terme du méta modèle du domaine va dans* ***l'élément M*** *(Compte, Client, CarteBancaire)*
2. *Chaque classe dont le nom contient un terme du domaine va dans* ***l'élément M*** *(CompteCourant, CompteDepot)*
3. *Les classes contenant des termes comme 'display', 'userinterface', 'frame', 'dialog', etc. vont dans* ***l'élément V***
4. *Les classes contenant des termes comme 'store', 'restore', 'data', 'bd',etc. vont dans* ***l'élément M***
5. *Les classes (ou traits) contenant des méthodes qui d'une part sont invoquées par des méthodes contenues dans des classes (ou traits) appartenant à l'élément M (respectivement l'élément V) et qui d'autre part invoquent des méthodes contenues dans des classes (ou traits) appartenant à l'élément V (respectivement l'élément M), vont dans* ***l'élément C***

Analyse des vues architecturales

- Plusieurs analyses sont effectuées sur les vues:
  - Adéquation du système par rapport à un patron considéré
    - En fonction du pourcentage des classes qui se trouvent dans les différents éléments architecturaux
  - Evolution dans le temps de l'architecture
    - En pratiquant l'extraction sur des versions successives du système logiciel et en comparant les résultats obtenus

Implémentation

- SmallTalk
  - Outil d'analyse
  - Outil de clustering sémantique
  - Outil de tissage avec les patrons
- Outils adossés à l'environnement de ré-ingénierie Moose

Laboratoire d'Informatique, Systèmes, Traitement de l'Information et de la Connaissance
4, chemin de Bellevue - BP 806 - 74016 ANNECY CEDEX
Tel: +33 (0)4 50 09 65 80 - Fax: +33 (0)4 50 09 65 90
Contacts: Sorana CIMPAN, Hervé VERJUS & Azadeh RAZAVIZADEH {sorana.cimpan,herve.verjus,azadeh.razavizadeh@univ-savoie.fr}
Équipe ***Logiciels et Systèmes***

π-ADL.NET

Concretising Software Architectures based on the π-Architecture Description Language

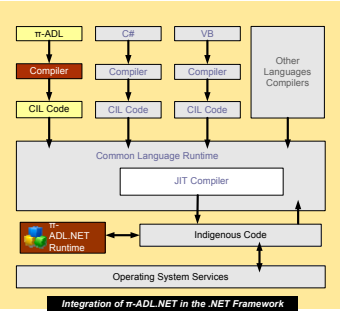
Author: Zawar Qayyum
Research Supervisor: Flavio Oquendo
ArchLog Group at VALORIA, Univ. Bretagne-Sud

http://www-valoria.univ-ubs.fr/ARCHLOG/

Motivation: Traditionally architecture description languages confine themselves to high-level software design, and only support implementation through transformation into an implementation level language. The purpose of this work is to broaden the scope of application of the architectural approach to software development by implementing an architecture-centric implementation language.

Research Question:
Is it feasible to develop an architecture description language (ADL) that supports the phases from architecture description to implementation, in order to:

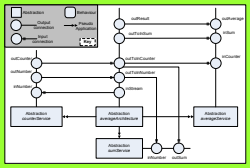
- *Preserve the architectural integrity of the system at the implementation level;*
- *Support analysis of the concrete architecture;*
- *Support evolution of the implementation while enforcing its architecture integrity;*
- *And directly use the implementation mechanisms of the hosting platform?*



Integration of π-ADL.NET in the .NET Framework

```
value counterService is abstraction () {
  inNumber : Connection(Integer);
  outNumber : Connection(Integer);
  outCounter : Connection(Integer);
  x : Integer;
  counter = location(0);
  replicate { via inNumber receive x; }
  via outNumber send x;
  if (x != 0) then {via counter send 'counter + 1;}
  else {via outCounter send 'counter;}
}

value sumService is abstraction () {
  inNumber : Connection(Integer);
  outSum : Connection(Integer);
  sum : location(0);  x : Integer;
  replicate {via inNumber receive x; }
  if (x <> 0) then {via sum send 'sum + x;}
  else {via outSum send 'sum;}
}
```

```
value averageService is abstraction (){
  inSum : Connection(Integer);
  inCounter : Connection(Integer);
  outAverage : Connection(Real); counter : Integer;
  replicate {via inSum receive sum; }
  via inCounter receive Counter;
  if (counter <> 0) then {via outAverage send sum / counter ;}
  else {via outAverage send 0 ;}
}
```



```
value averageArchitecture is abstraction () {
  inStream : Connection(Integer);
  outResult : Connection(Real);
  outToInNumber : Connection(Integer);
  outToInCounter : Connection(Integer);
  outToInSum : Connection(Integer);
  compose {
    via counterService send Void where { inStream unifies
      inNumber and outToInNumber unifies outNumber and
      outToInCounter unifies outCounter } ;
  and
    via sumService send Void where {outToInNumber unifies
      inNumber and outToInSum unifies outSum };
  and
    via averageService send Void where { outToInSum unifies
      inSum and outToInCounter unifies inCounter and outResult
      unifies outAverage };
  } //end compose
} //end averageArchitecture
```

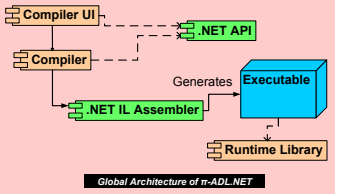
A Service Architecture in π-ADL for Averaging a Stream of Integers

Approach: Transform **π-ADL** into an implementation language by:

- Providing an implementation level compiler and analysis tool for the π-ADL;
- Adapting the π-ADL syntax in order for it to fulfil the role of a deterministic implementation language while retaining its formal ADL comportment; and
- Proposing formally founded platform specific extensions to the ADL in order to enable it to access reusable software components available on the implementation platform. The resulting compiler environment is called π-ADL.NET.

Advantages of .NET:

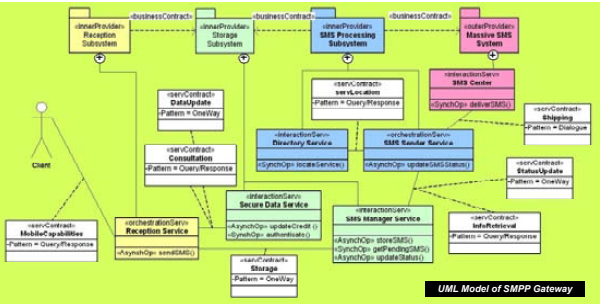
- A robust software development platform with a very large collection of reusable software components and tools, and a variety of middleware support;
- Provides a library of components and tools for developing compilers for new languages.;
- Supports multi-lingual development, allowing different parts of the system to be developed in appropriately specialized languages.



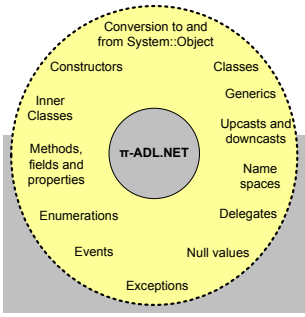
Global Architecture of π-ADL.NET

Case Study – SMPP Gateway: π-ADL.NET has been applied to the modeling of a service oriented architecture (SOA). This architecture conforms to the MIDAS framework, based on OMG's MDA standard. In this case study, this MIDAS based architectural style was applied to an SMS mass distribution system. Using π-ADL.NET, the following SOA concepts were modelled:

- Services
- Operations
- Dynamic discovery
- Service contracts



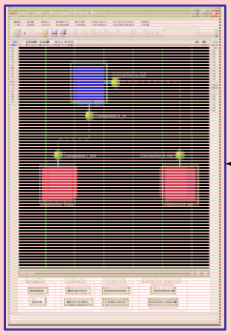
UML Model of SMPP Gateway



.NET Extensions for π-ADL.NET: The .NET extensions for π-ADL.NET are designed in order to allow the language to access the software components available on the .NET platform. At the same time the extensions allow the development of multi-lingual systems using an architectural approach; the architectural description in π-ADL.NET can be connected with components developed in detail-oriented languages (e.g. C#, Visual Basic) in order to construct a complete system.

Future Work

- Develop a visual modelling environment while exploiting the latest graphic technologies, in order to enable rapid development of executable architectures using π-ADL. The environment will facilitate integration of component implementations developed using programming languages such as C#,Visual Basic, C++ etc.
- Support the construction of hierarchical models and provide drill down functionality at the behaviour and abstraction level.
- Develop multiple graphical notations for different architectural styles which can be used for applications in their domain, while conforming to the π-ADL base syntax.
- Develop an IntelliSense system for assistance in describing software architectures, in order to design well structured, error free software architectures.
- Develop a system for simulating the execution of the architecture in order to visualise system behaviour at the architectural level.
- Support a library of architectures which can be used as base for developing newer, more specialized software architectures.
- Integrate the modeller into Microsoft Visual Studio .NET as a plug-in in order to complement and benefit from the functionality of the IDE.



```
behaviour
{
    i : sequence[Integer];
    ii : sequence[Integer];
    ci : Connection
[sequence[Integer]];

    i = sequence{1,1,2,3};
    compose
    {
        via ci send i;
    and
        via ci receive ii;
        via out send ii;
        via out send "\n";
        via seqAbs send i;
    }
}

value seqAbs is abstraction(seq :
sequence[Integer])
{
    ci : Connection
[sequence[Integer]];
...
```

```
ldarg.0
        ldfld class
[PiAdlLib]PiADL.NET.ConnectionTe
mplate`1<class
[PiAdlLib]PiADL.NET.Sequence>
piProg.seqAbs::c$_ci
        ldarg.0
        ldflda class
[PiAdlLib]PiADL.NET.Sequence
piProg.seqAbs::seq
        callvirt instance
void class
[PiAdlLib]PiADL.NET.ConnectionTe
mplate`1<class
[PiAdlLib]PiADL.NET.Sequence>::r
eceive(!0&)
        .locals init (int64
seq$index134)
        ldc.i8 1
        stloc seq$index134
        ldarg.0
        ldfld class
[PiAdlLib]PiADL.NET.Sequence
piProg.seqAbs::seq
...
```

Proposed graphical environnement that facilitates round-trip engineering between π-ADL text and the graphical representation

European University of Brittany
VALORIA Computer Science Research Laboratory
Scholarship funded by the Higher Education Comission of Pakistan
UBS UNIVERSITÉ DE BRETAGNE-SUD

SmPL: SimPLe SamPLes to Update Device Drivers

(supported by the ANR (FR) and the FTP (DK))

ECOLE DES MINES DE NANTES

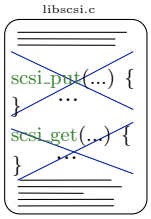
Gilles Muller
Ecole des Mines de Nantes
Gilles.Muller@emn.fr

Yoann Padioleau
Julia Lawall
University of Copenhagen
julia@diku.dk

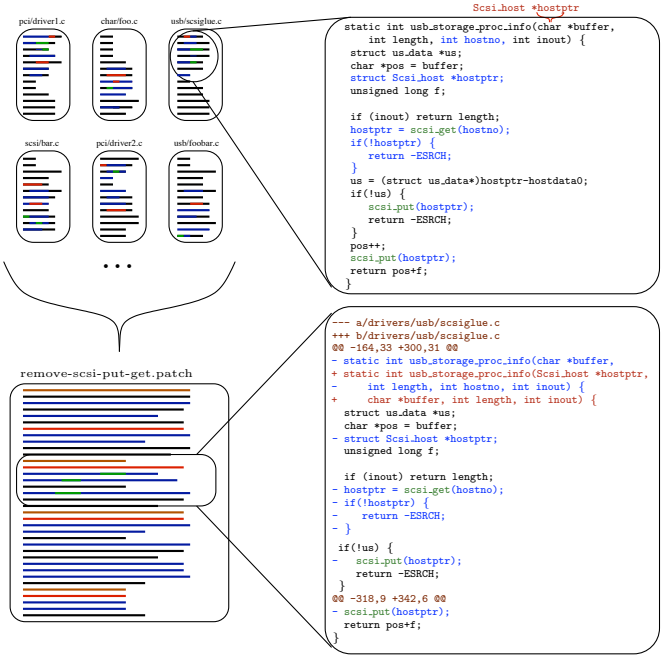
DET NATURVIDENSKABELIGE FAKULTET
KØBENHAVNS UNIVERSITET

THE PROBLEM

Evolution in API of a generic library



⇒ Lots of Collateral Evolutions in clients of this library 😵 [Eurosys'06]



Collateral evolutions are mostly done manually, because hard to script.

The program transformations require working on a high level representation of the program (syntatic and semantic, as in a compiler). Not just sed.

→ time consuming
(may involve 100 files, 1000 code sites)
→ error prone

Then, the modifications are transmitted to other Linux programmers via patch files.

Legend:
- scsi get/put function calls to delete
- dependent code to delete
- code to add

OUR SOLUTION: A declarative easy-to-use transformation language to specify collateral evolutions [Eurosys'08].

Linux programmers exchange, read, and manipulate program modifications in terms of patches.
→ Our language is based around the idea and syntax of a patch, extending patches to Semantic Patches.

A single small Semantic Patch can modify hundreds of files, at thousands of code sites. 🙂

Semantic Patch Language (SmPL) by example

```
@@
struct SHT sht;
local function proc_info_func;
@@
    sht.proc_info = &proc_info_func;

@@
identifier hostptr, hostno, buffer, length, inout;
@@
  proc_info_func (
+     struct Scsi_Host *hostptr,
      char *buffer, int length,
-     int hostno,
      int inout) {
   ...
-   struct Scsi_Host *hostptr;
   ...
-   hostptr = scsi_get(hostno);
   ...
-   if(!hostptr) { ... }
   ...
-   scsi_put(hostptr);
   ...
  }
```

1. looks like real code, looks like a real patch
   A developer can construct a semantic patch by copy pasting existing driver code and then modifying and generalizing it to generate the semantic patch.
2. abstracts away differences in spacing, indentation, comments
3. abstracts away specific names given to variables and expresses constraints between code sites by declaring and using metavariables
4. declares arbitrary intervening code sequences, including straight-line code and arbitrary branching, with the '...' operator [POPL'09]
   Semantic patches work at the control-flow level.
5. abstracts away other variations using isomorphisms (e.g. `if(!hostptr)` ≡ `if(hostptr==NULL)`)

Features of SmPL that make semantic patches Generic to accomodate the many variations in device driver coding style.

→ Semantic patches developed for over 60 collateral evolutions
→ Over 180 Coccinelle-based patches integrated into Linux

http://www.emn.fr/x-info/coccinelle/

Soleil
A Component Framework for Java-based Real-time Embedded Systems

Aleš Plšek, Frédéric Loiret, Michal Malohlava, Philippe Merle, Lionel Seinturier
{ales.plsek|frederic.loiret|philippe.merle|lionel.seinturier}@inria.fr

Goal

Methods reducing a complexity of developing real-time systems:
- Introducing general-purpose languages (Java)
- Applying software engineering paradigms **CBSE** (Component-based Software Engineering)

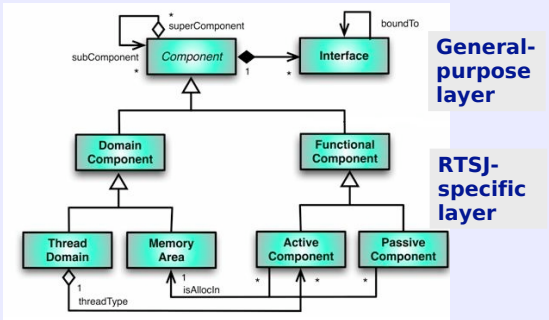
Real-time Specification for Java (RTSJ)
- Determinism in Java is achieved by introducing
- Memory areas (scoped, immortal, heap)
- Schedulable entities (real-time threads, events)

Goals
- **Application of CBSE paradigm into RTSJ world**:
  - **Enhancement** of development with RTSJ
  - **Efficiency** through *CBSE* and *Generative Programming*
  - **Safety** through *Formal Verification*

Component Metamodel for RTSJ

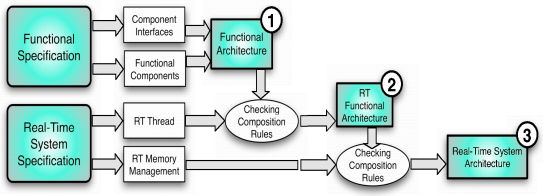
Domain Components
- **RTSJ defined as first-class entities**
- **Explicit separation of functional and real-time components**

Metamodel formally specified
- **Formal verification of architectures towards RTSJ**
- **Reasoning about the application at the design time**



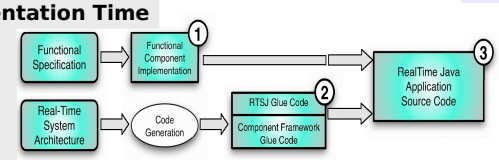
Framework Methodology

Design Time
- Separation of Concerns



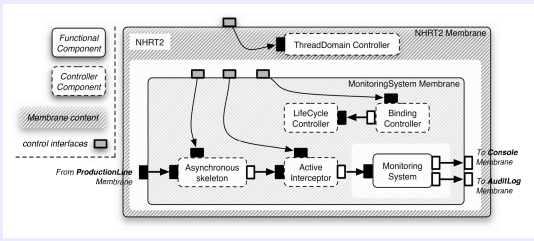
Implementation Time

- Automatic generation of RTSJ-related and framework related code.



Execution Infrastructure

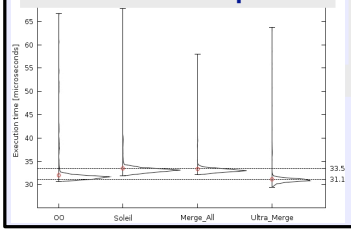
Component Membranes
- Component-oriented Container
- Extensive non-functional support at Runtime
- Runtime Reconfiguration



Experimental Evaluation

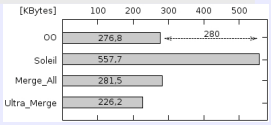
- Achieving **predictability**
- Reducing overhead

Performance Comparison



Legend
OO – object-oriented impl.
Soleil – Soleil implementation
Merge_All & ***Ultra_Merge*** Soleil with different optimization levels

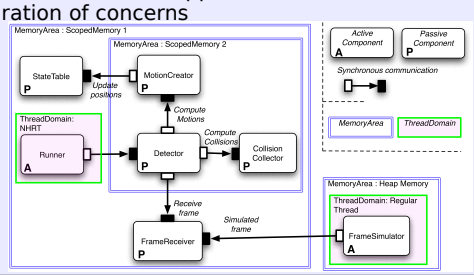
Footprint Reduction



Case Study

Real-time **Collision Detector**
- 10Hz Period, detecting collision courses of aircrafts

Prove of Concept
- Componentization of the application
- Full separation of concerns



Bibliography

[1] A. Plšek, F. Loiret, P. Merle, and L. Seinturier. ***A Component Framework for Java-based Real-time Embedded Systems***. In Proceedings of 9th International Middleware Conference, Leuven, Belgium, December 2008.

[2] Aleš Plšek, Philippe Merle, Lionel Seinturier. ***A Real-Time Java Component Model***. In Proceedings of the 11th International Symposium on Object/Component/Service-oriented Real-Time Distributed Computing (ISORC'08), May 2008, Orlando, Florida, USA.

[3] Michal Malohlava, Aleš Plšek, Frédéric Loiret, Philippe Merle and Lionel Seinturier. ***Introducing Distribution into a RTSJ-based Component Framework.*** In RNTS 2008: JRWRTC'08, October 2008, Rennes France

[4] Aleš Plšek, Jiří Adámek. ***Carmen: Software Component Model Checker***. QoSA'08, October 2008, Karlsruhe, Germany

MAY : Make Agents Yourself

Un générateur d'API agent à base de composants logiciels

Victor NOËL, Sylvain ROUGEMAILLE, Jean-Paul ARCANGELI, Jean-Pierre GEORGÉ, Frédéric MIGEON et Stéphane DUDOUIT

Université de Toulouse - UPS - IRIT
Equipe Systèmes Multi-Agents Coopératifs (SMAC)
118, route de Narbonne F-31062 Toulouse Cedex 9

Résumé MAY (Make Agents Yourself) permet la conception de « modèles » d'agents adaptatifs par assemblage de composants logiciels (micro-composants notés μ-composants) et vérifie la cohérence des assemblages. Chaque μ-composant définit un mécanisme d'interaction de l'agent avec son environnement (capteur ou effecteur) ou un mécanisme interne régissant le fonctionnement de l'agent. Le concepteur définit les possibilités d'adaptation en spécifiant quels μ-composants sont changeables dynamiquement. A partir d'un modèle, MAY génère une architecture logicielle à base d'objets Java utilisable dans le cadre d'une API qui permet la programmation des comportements des agents. Pour l'utilisateur, MAY se présente sous forme d'un *plug-in* Eclipse.

1 Principes généraux

Dans le cadre de nos travaux dans le domaine de l'aide au développement des systèmes multi-agents, nous proposons l'outil MAY (Make Agents Yourself) qui vise deux objectifs relatifs à l'adaptation :

- les systèmes artificiels du futur seront de plus en plus complexes, décentralisés, ouverts et instables ; ils devront être composés d'entités autonomes capables de s'adapter dynamiquement ;
- afin de limiter l'effort de développement, il est souhaitable de disposer d'une plateforme spécialisée pour le besoin applicatif plutôt qu'une plateforme « agent » plus ou moins généraliste.

Notre réponse réside dans la génération d'une API (Application Programming Interface) dédiée, à partir de la définition « à la carte », par le concepteur, de modèles d'agents en adéquation avec ses besoins et dotés de capacités d'auto-adaptation.

1.1 Le modèle d'agent

Un modèle d'agent définit l'ensemble des mécanismes élémentaires qui régissent le fonctionnement de l'agent à savoir :

- les mécanismes d'interaction de l'agent avec son environnement (capteurs, effecteurs) y compris les mécanismes de communication (envoi ou réception de message par exemple) ;
- les mécanismes internes tels le cycle de vie de l'agent, des gestionnaires de connaissance, un mécanisme de déplacement si l'agent est mobile, des mécanismes d'évolution ou de reproduction...

Ces différents mécanismes sont définis sous la forme de composants logiciels appelés micro-composants (notés μ-composants) car chacun fournit un service spécifique correspondant à une petite partie de l'agent [1]. Un modèle d'agent est un assemblage valide de μ-composants (les dépendances entre composants sont satisfaites). L'ensemble des μ-composants constitue le niveau opératoire de l'agent que l'on distingue du niveau comportemental dans lequel le développeur définit l'activité de l'agent au sein de l'application. En complément, la portée de chaque service est fixée par le concepteur qui indique s'il est accessible ou pas depuis l'environnement (externe), depuis le niveau opératoire, depuis le niveau comportemental.

Par ailleurs, le concepteur spécifie les composants qui sont dynamiquement changeables donnant ainsi le cadre d'une auto-adaptation sûre : à l'exécution, il est possible pour l'agent de remplacer un μ-composant par un autre offrant les mêmes interfaces en transférant l'état au besoin.

Le développement d'un agent s'inscrit donc dans une démarche de séparation des préoccupations et des niveaux. Le modèle d'agent peut être assimilé à un conteneur (adaptable) pour le composant de comportement. C'est une sorte de machine abstraite qui définit la sémantique opérationnelle de l'agent. Programmer le comportement applicatif revient à programmer cette machine abstraite.

1.2 L'architecture

Dans l'implantation actuelle, les μ-composants sont des objets Java connectés à un médiateur pour faciliter l'adaptation dynamique [2]. Chaque classe de μ-composant implante l'interface fournie et étend une classe abstraite qui implante l'interface requise (conformément à la portée des services) et qui cache le lien avec le médiateur. Par ailleurs, un *proxy* regroupe les services externes et sert de référence à l'agent dans le système. L'architecture (le médiateur, les classes abstraites et le *proxy*) est générée automatiquement.

2 MAY en pratique

MAY se présente sous forme d'un *plug-in* Eclipse (qui s'appuie sur les *plug-in* Ecore GMF, EMF et GEF). Lors du développement (cf. Fig. 1), l'utilisateur de MAY spécifie un modèle d'agent abstrait au moyen du langage graphique de description d'architecture μADL [3] (cf. Fig. 2), puis il implante les μ-composants du modèle par des classes Java (cf. Fig. 3) ; enfin, l'API dédiée est générée à partir du modèle et intégrée à une archive jar.

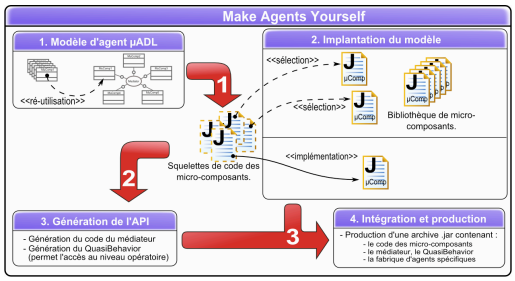


FIG. 1. Le processus de développement

MAY s'intègre dans un processus d'Ingénierie Dirigée par les Modèles dans lequel on utilise conjointement des langages généralistes (UML) et spécifiques au domaine des systèmes multi-agents (AMAS-ML [3]) qui permettent, par transformation de modèles, de produire la description de l'architecture en μADL (ainsi que des aspects comportementaux).

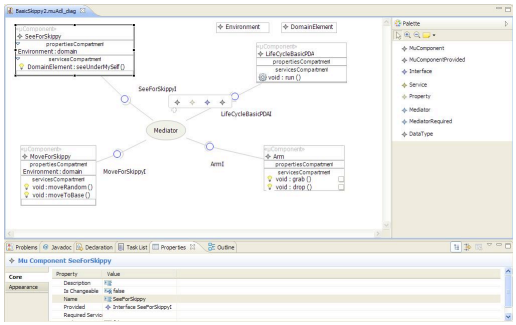


FIG. 2. L'éditeur de μADL

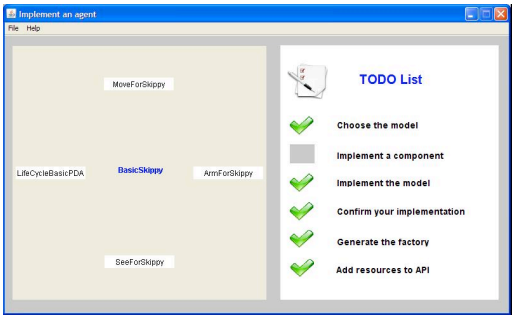


FIG. 3. Implémentation d'un modèle d'agent

La première version de MAY[1] a fait l'objet d'expériences en intelligence ambiante et en simulation distribuée. Maintenant, nous prévoyons d'utiliser MAY pour regénérer le *middleware* JAVACT[2], et d'autre part pour « agentifier » des composants logiciels et les rendre ainsi autonomes et adaptatifs.

Références

1. Leriche, S. : Architectures à composants et agents pour la conception d'applications réparties adaptables. Thèse de doctorat, Université Paul Sabatier, Toulouse, France (2006)
2. Leriche, S., Arcangeli, J.P. : Flexible Architectures of Adaptive Agents : the Agent-φ Approach. Rapport de recherche RR-2008-11-FR, IRIT, Université Paul Sabatier, Toulouse (2008)
3. Rougemaille, S. : Ingénierie des systèmes multi-agents adaptatifs dirigée par les modèles. Thèse de doctorat, Université Paul Sabatier, Toulouse, France (2008)

[1] http://www.irit.fr/MAY
[2] http://www.irit.fr/PERSONNEL/SMAC/arcangeli/JavAct_fr.html

AMPLE Traceability Framework: Un outil pour la traçabilité dans les lignes de produits logiciels

N. Anquetil[1] and J.-C. Royer[1⋆]

Ecole des Mines de Nantes
La Chantrerie
4, rue Alfred Kastler. B.P. 20722
F-44307 NANTES Cedex 3
{Nicolas.Anquetil,Jean-Claude.Royer}@emn.fr

Résumé La traçabilité logicielle a pour but de garantir la capacité de remonter du produit final (le code) jusqu'au producteur (les besoins). Si c'est un problème pour le développement « traditionnel », ce l'est encore plus pour les lignes de produits logiciels avec leur modèle plus complexe (deux processus de développement, plusieurs applications, variabilité) Dans le cadre du projet AMPLE, les membres de ce projet ont définit une infrastructure de traçabilité extensible et ouverte que nous présentons ici.

1 Introduction

La traçabilité a pour objectif de garantir que l'on peut facilement aller d'un besoin spécifique d'un système jusqu'au code qui l'implémente et inversement. On doit pouvoir aussi retrouver facilement des artefacts similaires entre eux mais qui ne sont pas forcément reliés par le processus de développement. Dans le cadre du projet européen AMPLE[1] une infrastructure de traçabilité a été définie qui permette de répondre aux besoins d'extensibilité et de flexibilité du projet. Nous introduisons ici les problèmes liés à la traçabilité pour les lignes de produits (§2), avant de présenter l'infrastructure définie (§3) et les extensions qui ont été développées à l'heure actuelle (§4).

2 Traçabilité dans les lignes de produits logiciels

Les lignes de produits sont une approche de développement logiciel où l'on définit une architecture générale pour plusieurs applications dans un domaine donné [1]. L'architecture générale prévoit des points de variation et quelles peuvent être les variations qui permettent de générer différentes applications. Cette conception différente du développement amène un accroissement (quantitatif et qualitatif) de la complexité de la traçabilité [2] : (i) il y a plus de types de produits (ex. : modèle de variabilité) ; (ii) il y a deux processus (dévelopement de l'architecture générale et développement des applications) ; et, (iii) il y a de nouveaux liens de traçabilité (ex. : entre une application et l'architecture générale).

⋆ Les résultats présentés ici sont le fruit du travail de tous les membres du projet AMPLE et non uniquement des auteurs

[1] Aspect oriented, Model driven, Product Line Engineering, http ://ample.holos.pt/

3 L'infrastructure : ATF

Pour répondre à ces difficultés le projet AMPLE a défini les besoins d'un outil de gestion de la traçabilité pour les lignes de produits logiciels [3] : permettre de stocker et gérer les liens de traçabilité ; permettre des requètes complexes ; être extensible. Ses besoins se sont traduits par une infrastructure implémentée par une série de plugins Eclipse : *AMPLE Traceability Framework* (ATF). ATF s'organise autour d'un premier plugin, ATF-core, qui se charge du stockage des artefacts et leurs liens de traceabilité. ATF-core permet de définir des types d'artefact (ex. : besoins, cas d'utilisation, paquet, classe, ...), des artefacts, des types de liens (ex. : implémente, contient, ...) et des liens. Il offre quelques requêtes de base ainsi qu'un point d'extension Eclipse pour créer des « cadastreurs », des outils qui sont chargés d'enregistrer les données dans la base de traçabilité. Les membres du projet ont enveloppé ATF-core dans une interface graphique, ATF-frontend, qui offre deux nouveaux points d'extension : la possibilité de définir des vues (par exemples visualisation d'une liste de liens de traçabilité comme un graphe ou un arbre) et celle d'ajouter de nouvelles requêtes plus élaborées. ATF-frontend se charge de faire le lien entre le résultat d'une requête et une visualisation et offre la possibilité de définir des flux de travail. De plus ATF-frontend donne toujours accès à la création de cadastreurs dans ATF-core.

4 Instanciations de ATF

À partir de cette structure de base, différents membres du projet AMPLE ont put développer de manière individuellle et très rapidement une vingtaine d'extensions : Des cadastreurs variés qui extraient les artefacts et/ou les liens de traçabilité de fichiers (ou projets) Java, Rational Rose, Enterprise Architect, FMP[2], MoPLine, ou XML (format « propriétaire » retraçant la dérivation d'une application à partir d'un modèle en MDD). Il existe aussi un cadastreur manuel où les liens sont créés entre des artefacts déjà présents dans la base. Deux requêtes élaborées ont été créées, par exemple pour faire de l'analyse d'impact d'une modification. Diverses visualisations sont disponibles : en arbres (vue hiérarchique), comme des graphes (avec plusieurs algorithmes de positionnement), comme diagramme de Venn, ou plus simplement comme une liste textuelle (« X <lien> Y »). Finalement des vues spéciales ont aussi été créées qui produisent une séries de métriques sur le résultat d'une requête (nombre d'artefacts, nombre de liens, etc.), ou qui exportent une matrice de traçabilité au format Excel.

Références

1. Pohl, K., Böckle, G., van der Linden, F. : Software Product Line Engineering - Foundations, Principles, and Techniques. Springer Verlag, Heidelberg (2005)
2. Anquetil, N., Grammel, B., Galvao, I., Noppen, J., Khan, S.S., Arboleda, H., Rashid, A., Garcia, A. : Traceability for Model Driven, Software Product Line Engineering. (2008) 77–86
3. Sousa, A., Kulesza, U., Rummler, A., Anquetil, N., Mitschke, R., Moreira, A., Amaral, V., Araújo, J. : A Model-Driven Traceability Framework to Software Product Line Development. (2008) 97–109

[2] Feature Modeling Plugin de Eclipse

Conception de systèmes par applications de modèles paramétrés

Olivier Caron[1], Bernard Carré[1], Areski Flissi[1], Alexis Muller[1], and Gilles Vanwormhoudt[1,2]

[1] Laboratoire d'Informatique Fondamentale de Lille
[2] Telecom Lille I

Résumé Nous présentons nos travaux sur les composants de modèle et leur opérateur associé "apply" qui permet la conception de systèmes par assemblage de composants de modèles (rangés dans des bibliothèques). Nous illustrons ces travaux à l'aide d'un outil démonstrateur "CocoaModeler". Cet atelier UML 2 étendu aux composants de modèles supporte des chaînes de production flexibles selon différentes stratégies de ciblage et différentes plates-formes technologiques.

1 Les composants de modèles et leur assemblage

Notre approche vise la spécification de composants métiers réutilisables et composables dans des contextes (domaines) applicatifs différents. Nous avons défini la notion de composant de modèle paramétré par un "modèle requis" et fournissant un modèle enrichi [1]. On dépasse ainsi la notion de contrat d'assemblage de composants souvent réduite à une interface de services unitaires. La conception d'un système

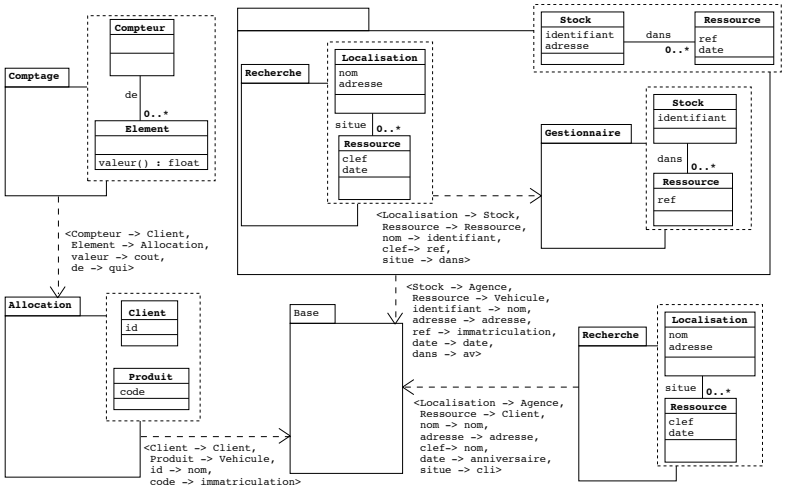


FIG. 1. Construction d'un système par assemblage logique de composants de modèles

revient alors à assembler de tels composants par les modèles (voir exemple d'un assemblage de la figure 1). Nous proposons pour cela un opérateur d'application de modèles paramétrés (opérateur dénommé "apply"). Cet opérateur permet de relier le modèle requis à un modèle **conforme** (sous-modèle de la cible). La conformité de modèles est vérifiée par un jeu de contraintes [1]. La cible est soit un modèle du système, soit un composant de modèle. On obtient ainsi une démarche hiérarchique et homogène permettant de spécifier des assemblages complexes. Des propriétés d'ordre permettent de garantir la cohérence des alternatives de composition. Des règles et contraintes au niveau des modèles permettent de faire des vérifications au plus tôt sur la cohérence des systèmes ainsi construits [2].

2 Chaîne de production flexible

La figure 2 illustre différentes chaînes de production d'un système à partir des modèles (niveau 1) jusqu'à une plate-forme d'exécution (niveau 4) [3]. Le premier niveau permet d'exprimer des assemblages de

modèles paramétrés. Ces modèles paramétrés sont stockés dans des bibliothèques de modèles afin de faciliter leur réutilisation. A partir d'un modèle d'assemblage (niveau 1), il est possible d'obtenir un modèle du système résultant selon différentes représentations (fusionnée ou éclatée en vues) . A un niveau archi-

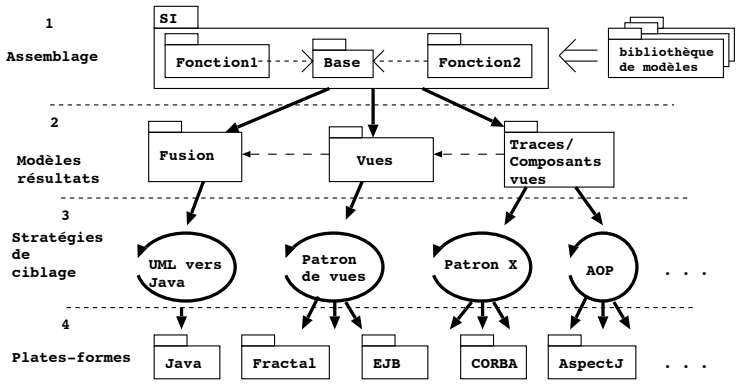


FIG. 2. Chaînes de production

tectural (niveau 3), nous proposons différentes stratégies de mise en oeuvre, sous la forme de patrons de conception, permettant d'exploiter les capacités de modularisation et d'adaptation des assemblages [4,5]. Des projections ont été expérimentées sur différentes plates-formes (EJB, CORBA, FRACTAL,. . .) [4,1,6].

3 L'outil démonstrateur : CocoaModeler

L'approche est formulée par une extension du méta-modèle UML2 et un ensemble de contraintes [1,7]. Elle est mise en œuvre dans l'atelier CocoaModeler[3] en utilisant les plugins Eclipse EMF et EMF-UML 2. Les fonctionnalités de cet atelier sont :

- la spécification de composants de modèles via des templates UML 2, la vérification structurelle de ces composants, le chargement et la sauvegarde dans des bibliothèques de modèles
- la conception d'un système par assemblage de composants de modèles
- L'opérationalisation de l'opérateur apply (mode fusion)
- La génération de code d'un assemblage selon la stratégie de ciblage (fusionné, éclaté en vues) et la plate-forme

Références

1. Muller, A. : Construction de systèmes par application de modèles. PhD thesis, Université de Lille (2006)
2. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G. : On Some Properties of Parameterized Model Applications. In : Proc. of ECMDA'05 :European Conference on Model Driven Architecture. Number 3748 in LNCS (2005)
3. Muller, A., Caron, O., Carré, B., Vanwormhoudt, G., Bouzitouna, S. : Ingénierie multi-modèles : Projection flexible d'assemblages de modèles. In : Conf. francophone Langages et Modèles à Objets (LMO'07), Toulouse (mars 2007)
4. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G. : A Framework for Supporting Views in Component Oriented Information Systems. In : International Conference on Object-Oriented Information Systems. Volume 2817 of Lecture Notes in Computer Sciences., Geneva - Switzerland, Springer Verlag (September 2003) 164–178
5. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G. : Mise en oeuvre d'aspects fonctionnels réutilisables par adaptation. Numéro spécial de la revue l'Objet : Programmation par aspects **11**(3) (2005)
6. Barais, O., Muller, A., Pessemier, N. : Vers une séparation entités/fonctions au sein d'une architecture logicielle à base de composants. Numéro spécial L'Objet : Ingénierie des composants et systèmes d'information **11**(4) (2005)
7. Caron, O., Carré, B., Muller, A., Vanwormhoudt, G. : Formulation of UML 2 Template Binding in OCL. In : 7th International Conference on UML (UML 2004). Number 3273 in LNCS, Lisbon - Portugal (October 2004)

[3] http ://www.lifl.fr/GOAL/cocoa/pmwiki.php ?n=Main.Outils

Traçabilité d'exigences temporelles dans l'outil UML/SysML TTool

Pierre de Saqui-Sannes[1] and Ludovic Apvrille[2]

[1] Université de Toulouse, LAAS-CNRS, ISAE
10 av. Edouard Belin, B.P. 54032, 31055 TOULOUSE Cedex 4, France
[2] Institut TELECOM / TELECOM ParisTech, LTCI CNRS,
2229 route des crêtes, B.P. 193, 06904 Sophia-Antipolis Cedex, France
pdss@isae.fr,ludovic.apvrille@telecom-paristech.fr

Résumé La démonstration proposée concerne la traçabilité d'exigences temporelles tout au long du cycle de développement d'un système temps-réel, potentielement distribué. L'outil TTool, basé sur un profil UML2, permet de saisir les exigences au format SysML, puis de confronter, par utilisation de techniques de vérification formelle, ces exigences temporelles aux diagrammes UML du système.

Key words: UML, exigences, systèmes temps-réel, analyse, conception, matrice de traçabilité.

1 Introduction

L'outil TTool [1], ou "TURTLE Toolkit" in extenso, est un outil de modélisation de systèmes temps réel et distribués. Il se démarque d'autres outils UML [2] et SysML [3] par les solutions qu'il apporte en termes de traçabilité d'exigences temporelles. TTool supporte le profil UML temps réel TURTLE (Timed UML and RT-LOTOS Environment) [4], respecte sa syntaxe compatible avec le méta-modèle d'UML 2.1 et implante sa sémantique formelle exprimée par traduction vers l'algèbre de processus temporisée RT-LOTOS [5][6].

La méthode associée au profil UML TURTLE comprend 7 étapes (dont une de prototypage sur laquelle nous ne recherchons pour l'instant pas la traçabilité temporelle) :

1. Recueil des exigences au sein de diagrammes d'exigences SysML renforcés par une formalisation des exigences temporelles [7].
2. Analyse à base de cas d'utilisation documentés par des scénarios pertinents (diagrammes de séquences).
3. Vérification formelle des diagrammes d'analyse par confrontation aux exigences temporelles.
4. Synthèse de squelettes de diagrammes de conception (diagramme de classes/objets pour l'architecture et diagrammes d'activités pour les comportements) à partir des diagrammes d'analyse.
5. Conception orientée objet à partir de la synthèse précédente par enrichissement du diagramme de classes/objets et des diagrammes d'activités.
6. Vérification formelle des diagrammes de conception par rapport aux exigences temporelles [7].
7. Prototypage exploitant les générateurs de code Java et SystemC de TTool.

Pour la saisie d'exigences, TURTLE s'appuie sur un diagramme d'exigences SysML étendu. Les exigences non temporelles demeurent écrites en langage naturel. On leur associe un type (fonctionnel, non fonctionnel) et un niveau de risque (bas, élévé). Le stétérotype «FormalRequirement» et les diagrammes temporels étendus appelés "Temporal Requirement Description Diagrams" (TRDDs) permettent d'exprimer les exigences temporelles sans ambiguïté. Celles-ci servent de point de départ à la génération d'observateurs en charge de piloter les vérifications formelles des étapes méthodologiques suivantes (analyse, conception). Enfin, un attribut des exigences appelé *Violated_Action* permet en phase de vérification formelle d'idenfier les exigences respectées et violées.

L'outil TTool facilite et automatise l'accès aux outils de vérification formelle CADP [8], RTL [9] et UPPAAL [10]. Ces trois outils implémentent la génération de graphes d'accessibilité et des algorithmes de model-checking. TTool s'appuie sur ces outils pour la construction de matrices de traçabilité. Ainsi, depuis un diagramme donné (analyse, conception), la phase de vérification formelle consiste à générer automatiquement du code formel (RT-LOTOS, UPPAAL) depuis le diagramme considéré, puis à adjoindre automatiquement à ce code formel des observateurs représentant les exigences temporelles, et enfin à générer, par utilisation des outils précités, un graphe d'accessibilité depuis la spécification formelle constituée du système et des observateurs. Sur ce graphe, la satisfaction des exigences peut-être étudiée par la recherche de *labels* particuliers sur les transitions. En effet, la violation d'une propriété s'accompagne en vérification formelle par la génération d'une transition dont le *label* correspond à celui qui est défini au niveau de l'exigence SysML par *Violated_Action*. L'analyse de ce graphe par TTool permet de construire automatiquement une matrice de traçabilité qui liste les exigences temporelles satisfaites et non satisfaites.

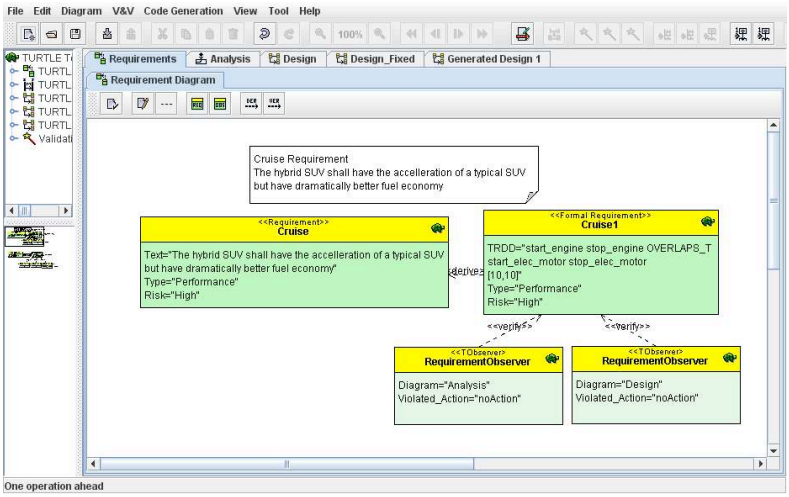


Fig. 1. TTool : capture d'écran d'un diagramme d'exigences SysML

Références

1. TTool, the TURTLE toolkit, http ://labsoc.comelec.enst.fr/turtle/ttoolindex.html
2. Object Management Group, Unified Modeling Language Specification, Version 2.1.1, http ://www.omg.org/cgi-bin/apps/doc ?ptc/08-06-08.pdf
3. Object Management Group, "UML Profile for Systems Engineering, SysML", Version 1.0, http ://www.omg.org/cgi-bin/apps/doc ?formal/07-09-01.pdf, September 2007.
4. Apvrille, L., Courtiat, J.-P., Lohr, C., de Saqui-Sannes, P. : TURTLE : A Real-Time UML Profile Supported by a Formal Validation Toolkit. IEEE Transactions on Software Engineering, Vol. 30, No. 7, pp. 473-487, July 2004
5. ISO Standard 8807 : LOTOS, a formal description technique based on temporal ordering of observational behaviour, 1988
6. J.-P. Courtiat and C.A.S Santos and C. Lohr and B. Outtaj : Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Descr iption Technique, Computer Communications, Elsevier, Vol. 23, n 12, pages 1104-1123, 2000
7. B. Fontan, Méthodologie de conception de systèmes temps réel et distribués en contexte UML/SysML, thèse de doctorat de l'Université de Toulouse, Janv. 2008
8. CADP toolkit : http ://www.inrialpes.fr/vasy/cadp.html
9. RTL toolkit : http ://www.laas.fr/RT-LOTOS/
10. UPPAAL toolkit : http ://www.uppaal.com/

Une première formalisation du modèle des exigences

Abderrahman Matoussi, Frédéric Gervais, and Régine Laleau

LACL, Université Paris-Est
{abderrahman.matoussi,frederic.gervais,laleau}@univ-paris12.fr

1 Motivation

Les méthodes formelles et les méthodes d'analyse des exigences 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 exigences et la phase de spécification est en train de s'élargir et la réconciliation paraît de plus en plus difficile. Quelques travaux [3,4] ont essayé de combiner ces deux phases en utilisant KAOS [2] comme technique d'analyse des exigences. Néanmoins, cette réconciliation demeure verticale puisque ces travaux génèrent directement des spécifications formelles à partir de la dernière étape de la méthodologie KAOS. Pour remédier à ce problème, l'objectif de cet article est d'inclure la phase d'analyse des exigences dans le développement logiciel associé aux méthodes formelles tout en restant au même niveau d'abstraction. Ainsi, notre approche vise à déduire la spécification Event-B [1] à partir du modèle de buts KAOS et cela passe dans un premier temps par prouver formellement ce dernier modèle. Nous justifions le choix d'Event-B par sa grande similarité et complémentarité avec la méthodologie KAOS : (i) Event-B est basé sur la logique des prédicats du premier ordre ce qui facilite son intégration avec KAOS qui est basé sur la logique temporelle du premier ordre ; (ii) Event-B et KAOS sont basés sur la notion d'observation qui permet de modéliser à la fois le système et son environnement ; (iii) la notion de raffinement (l'approche constructive) caractérise à la fois KAOS et Event-B.

2 Un aperçu général de l'approche

L'approche proposée a pour objectif d'introduire dans la méthode Event-B la phase d'expression des exigences. Il pourra alors être possible de prouver le modèle des exigences et d'établir des liens formels entre ce modèle et la spécification d'un système. Un aperçu général de l'approche est illustré dans la figure 1.

Première étape : Expression des buts en Event-B

Les buts jouent un rôle très important dans l'ingénierie des exigences et par conséquent dans le développement logiciel. Tandis que les spécifications répondent à la question « que fait le système ? », les buts résolvent quand à eux les questions « qui ? quand ? comment ? ». C'est pour cette raison que la première

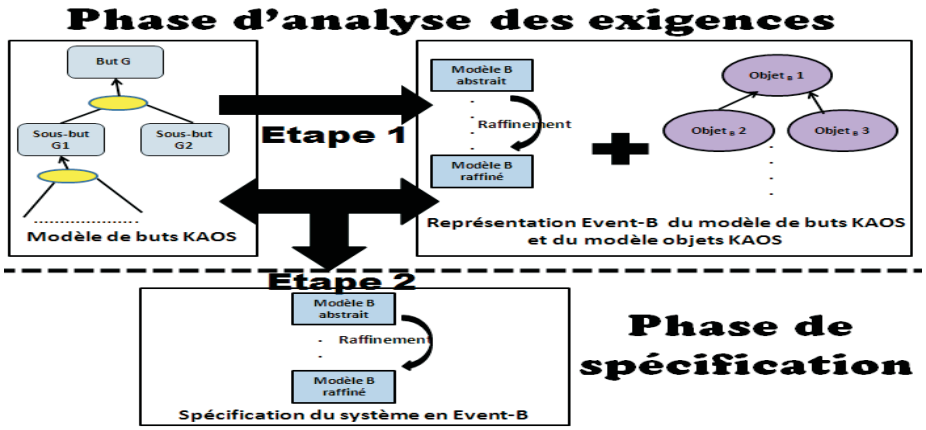


Fig.1. Un aperçu général de notre approche

étape de notre approche consiste à exprimer le modèle de buts KAOS avec Event-B. Cette étape traite les différents types de raffinement de buts (ET, OU exclusif, OU inclusif) en exploitant aussi les patrons de raffinement de buts [2]. Étant donné qu' un but KAOS exprime une propriété qui doit être établie, l'idée principale de notre approche est de représenter chaque but KAOS comme un événement B et la propriété comme la post-condition de cet événement. Chaque niveau dans la hiérarchie de buts KAOS est représenté comme un niveau de raffinement dans le modèle équivalent en Event-B.

Seconde étape : Déduction des opérations

La deuxième étape de l'approche consiste à déduire formellement les opérations (qui opérationnalisent les buts) à partir du modèle de buts KAOS et de sa représentation B en utilisant aussi la représentation B du modèle objets. De cette façon, cette étape assure que les opérations préservent toutes les propriétés du modèle de buts KAOS. L'idée de base pour construire l'opération opérationnalisant un but est la suivante : cette opération peut être exécutée tant que le but associé n'est pas satisfait, c'est à-dire que sa post-condition n'est pas vérifiée.

Références

1. J.R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B'98*, volume 1393 of *LNCS*, pages 83–128, Montpellier, France, April 1998. Springer-Verlag.
2. E. Letier. Reasoning About Agents in Goal-Oriented Requirements Engineering. Ph.D. Thesis, *ftp ://ftp.info.ucl.ac.be/pub/thesis/letier.pdf*, 2001.
3. 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. IEEE.
4. C. Ponsard and E. Dieul. From Requirements Models to Formal Specifications in B. In *REMO2V'2006*, Luxembourg, June 2006.

On Modelling Constraint Satisfaction Problems with Logical Rules

François Fages, Julien Martin

Projet Contraintes, INRIA Rocquencourt,
BP105, 78153 Le Chesnay Cedex, France.
http://contraintes.inria.fr

Abstract. In this paper, we present a rule-based modelling language for constraint programming, called Rules2CP. Unlike other modelling languages, Rules2CP adopts a single knowledge representation paradigm based on 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. We describe the compilation of Rules2CP statements to constraint programs over finite domains, by a term rewriting system and partial evaluation. We prove the confluence of these transformations and provide a complexity bound on the size of the generated programs. The expressiveness of Rules2CP is illustrated first with simple examples, and then with a complete library for packing problems, called PKML, which, in addition to pure bin packing and bin design problems, can deal with common sense rules about weights, stability, as well as specific packing business rules. The performances of both the compiler and the generated code are evaluated on Korf's benchmarks of optimal rectangle packing problems.

From a programming language standpoint, one striking feature of constraint programming is its declarativity for stating combinatorial problems, describing only the "what" and not the "how", and yet its efficiency for solving large size problem instances in many practical cases. From a non-programmer application expert standpoint however, constraint programming is not as declarative as one would wish, and constraint programming systems are in fact very difficult to use by non-programmers outside the range of already treated examples. This well recognized difficulty has been presented as a main challenge for the constraint programming community, and has motivated the search for more declarative front-end problem modelling languages, such as for instance OPL [1,2], Zinc [3,4] and Essence [5].

In the industry, the business rules approach to knowledge representation has a wide audience because of the declarativity and granularity of rules which can be introduced, checked, and modified one by one, and independently of any particular procedural interpretation by a rule engine [6]. This provides an attractive knowledge representation scheme for fastly evolving requirements, and for maintaining systems with up to date information.

In this poster, we show that such a rule-based knowledge representation paradigm can be developped as a front-end modelling language for constraint programming. We present a general purpose rule-based modelling language for constraint programming, called Rules2CP. Unlike multi-headed condition-action rules, also called production rules, Rules2CP rules are restricted to *logical rules*, with one head and no imperative actions, and where bounded quantifiers are used to represent complex conditions. Such rules comply to the principle of independence from a procedural interpretation by a rule engine [6], which is concretely demonstrated in Rules2CP by their compilation to constraint programs using a completely different representation.

Unlike the other modelling languages proposed for constraint programming, Rules2CP adopts a restricted set of data structures based on records and enumerated lists, given with iterators. We show that this is sufficient to express constraint satisfaction problems, together with search strategies where the search tree is expressed by logical formulae, and complex heuristic choice criteria are defined as preference orderings on variables and formulae.

Rules2CP has a simple syntax for declaring records, rules and goals. Search trees can also be specified declaratively in Rules2CP with logical formulae, and search heuristics can be defined as preference orderings on variables, values, conjunctive and disjunctive formulae, using pattern matching on rule names. This is in contrast with other modelling languages for which search strategies still need be programmed. For instance the following Rules2CP model deals with five tasks, four precedence rules and one goal for solving the disjunctive scheduling problem:

```
t1 = {start=_, duration=2}. t2 = {start=_, duration=5}.
t3 = {start=_, duration=4}. t4 = {start=_, duration=3}.
t5 = {start=_, duration=1}. cost = start(t5).
prec(T1, T2) --> start(T1) + duration(T1) =< start(T2).
disj(T1, T2) --> prec(T1,T2) or prec(T2,T1).
precedences --> prec(t1,t2) and prec(t2,t5) and prec(t1,t3) and
                prec(t3,t5) and prec(t1,t3) and prec(t3,t5).
disjunctives --> disj(t2,t3) and disj(t2,t4) and disj(t3,t4).
? domain([t1,t2,t3,t4,t5], 0, 20) and
  precedences and search(disjunctives) and minimize(start(t6)) and
  conjunct_ordering([greatest(duration(A)+duration(B) if ^ is disj(A,B))]).
```

The compilation process of Rules2CP models into constraint programs over finite domains with reified constraints, is described with a term rewriting system and partial evaluation. We prove the confluence of these transformations which shows that the generated constraint program does not depend on the order of application of the rewritings, and we provide a complexity bound on the size of the generated program. In the previous example, the generated code is as follows:

```
? domain([T1,T2,T3,T4,T5],0,20),
  T1+2#=<T2, T2+5#=<T5, T1+2#=<T3, T3+4#=<T5, T1+2#=<T3, T3+4#=<T5,
  minimize((((T2+5#=<T3;T3+4#=<T2),(T2+5#=<T4;T4+3#=<T2),
             (T3+4#=<T4;T4+3#=<T3)), labeling([up],[T5])),T5).
```

The expressive power of Rules2CP is illustrated with a particular Rules2CP library, called the Packing Knowledge Modelling Library (PKML), developed in the EU project Net-WMS (`http://net-wms.ercim.org`) for dealing with real-size non-pure bin packing problems of the automotive industry. The performances of both the compiler and the generated code are evaluated in this section on Korf's benchmarks of optimal rectangle packing [7]. We show that search strategies for scheduling can be easily expressed in Rules2CP in this manner, as well as the search strategies of Simonis and O'Sullivan [8] for solving Korf's optimal rectangle packing problem [7], with a constant overhead factor in the generated code.

References

1. Van Hentenryck, P.: The OPL Optimization programming Language. MIT Press (1999)
2. Hentenryck, P.V., Perron, L., Puget, J.F.: Search and strategies in opl. ACM Transactions on Compututational Logic **1** (2000) 285–320
3. Rafeh, R., de la Banda, M.G., Marriott, K., Wallace, M.: From Zinc to design model. In: Proceedings of PADL'07, Springer-Verlag (2007) 215–229
4. de la Banda, M.G., Marriott, K., Rafeh, R., Wallace, M.: The modelling language Zinc. In: Proceedings of the International Conference on Principles and Practice of Constraint Programming CP'06), Springer-Verlag (2006) 700–705
5. Frisch, A.M., Harvey, W., Jefferson, C., Martinez-Hernandez, B., Miguel, I.: Essence: A constraint language for specifying combinatorial problems. Constraints **13** (2008) 268–306
6. Group, B.R.: The business rules manifesto (2003) Business Rules Group `http://www.businessrulesgroup.org/brmanifesto.htm`.
7. Korf, R.E.: Optimal rectangle packing: New results. In: ICAPS. (2004) 142–149
8. 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)

Moose: an Agile Reengineering Environment

Alexandre Bergel, Stéphane Ducasse

RMoD research group, INRIA Lille, France

1 Reverse Engineering

Software systems need to evolve continuously if they are to be effective . The reengineering process comprises various activities, including model capture and analysis (*i.e.,* reverse engineering), assessment of problems to be repaired, and migration from the legacy software towards the reengineered system. In practice, this is realized as a set of transformation through various abstraction layers from legacy code towards a new system . This involve various documents. In addition to the code base, there may be documentation, bug reports, tests and test data, database schemas, and especially the version history of the code base. Other important sources of information include the various stakeholders (*e.g.,* users, developers, maintainers), and the running system itself. The reengineer will neither rely on a single source of information, nor on a single technique for extracting and analyzing that information.

Reengineering is a complex task, and it usually involves several techniques. The more data we have at hand, the more techniques we require to apply to understand this data. These techniques range from data mining, to data presentation and to data manipulation. Different techniques are implemented in different tools, by different people. An infrastructure is needed for integrating all these tools.

2 The Moose Environment

Moose[1] is a reengineering environment that offers a common infrastructure for various reverse- and reengineering tools . At the core of Moose is a common meta-model for representing software systems in a language-independent way. Around this core are provided various services that are available to the different tools. These services includes metrics evaluation and visualization, a repository for storing multiple models, a meta-meta model for tailoring the Moose meta-model, a generic GUI for browsing, querying and grouping, an information visualization engine.

In essence, Moose functions as a repository for software models, providing numerous services for importing, viewing, querying and manipulating these models. Its meta-model determines how software systems are modeled within Moose.

The first important requirement for Moose is that *it does not impose a fixed meta-model*, but rather provide the infrastructure to extend the meta-model according to the needs of the analysis. Moose supports the various needs of tools, rather than confining them to a fixed view of the world. This first crucial point makes Moose a flexible and innovative environment.

The second important point of Moose, is to adopt a *data-driven reengineering stance* rather an analysis-driven one. This means that a new analysis may be easily integrated into Moose and different analyzes may be conducted on different parts of the system under analysis. Traditional reverse engineering environments (*e.g.,* iPlasma[2], ArchView[3]) usually provide a fixed set of globally applicable analyses. In that respect, Moose has an effective flexibility compared to these environment.

3 Demonstration

This demonstration is structured in different parts.

[1] http://moose.unibe.ch
[2] http://loose.upt.ro/iplasma/
[3] http://seal.ifi.uzh.ch/archview/

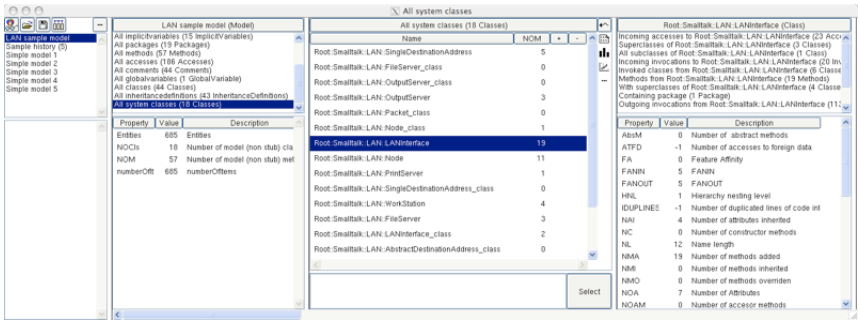


Fig. 1. Browsing the system with MooseFinder

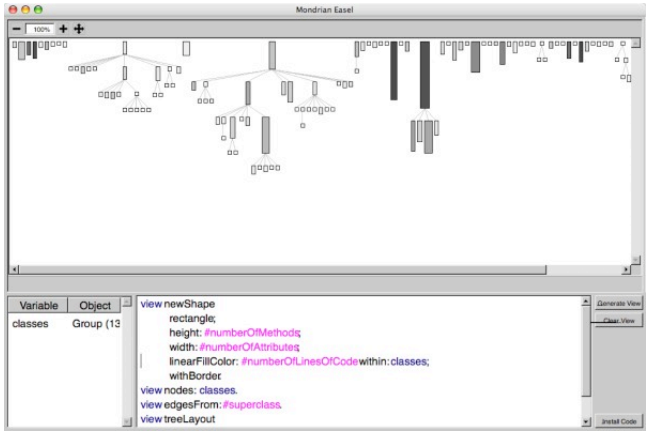


Fig. 2. Building visualization scripts

Firstly, a brief review on the essential tools will be provided. MooseFinder[4] is used to browse a model. A large set of metrics are automatically computed for selected elements. Different pre-defined visualizations are available and switching between these different views is realized through 1 or 2 moose clicks (Figure 1).

The second part will essentially use Mondrian[5] , an information visualization engine, to "visualize" a software system. Particular views will be incrementally defined through some scripts (Figure 2).

The third and final part of these demonstrations will review Package DNA [6] and EDSM , two tools being developed at INRIA.

[4] http://moose.unibe.ch/tools/moosefinder
[5] http://moose.unibe.ch/tools/mondrian
[6] http://moose.unibe.ch/tools/packagedna/packagefingerprint, http://moose.unibe.ch/tools/packagedna/packageblueprint

MotOrBAC: a tool to specify, administrate and deploy security policies

Fabien Autrel, Frédéric Cuppens, Nora Cuppens-Boulahia, and Céline Coma
{fabien.autrel, frederic.cuppens, nora.cuppens, celine.coma}@telecom-bretagne.eu

IT-TELECOM Bretagne, 35576 Cesson Sévigné (France)

1 Introduction

Nowadays private and public organizations face several problems when they try to specify and enforce the security policy of their information systems. Among those problems we can highlight the fact that several administrators are in charge of information systems, a large number of components which implement security mechanisms must be configured, a large number of subjects, actions and objects must be introduced in the security policy, the security policy consistency should be ensured, etc. The use of a single tool to manage the security policy would solve many problems. However such a tool would be useless if it does not implement a formal model which allows the expression of security requirements and administrative needs. After reviewing several security models, we conclude that the *OrBAC* model fits the specifications of the aforementioned tool. *OrBAC* (Organization based access control) is a security model based on Datalog which provides means to formally specify a security policy independently from its implementation.

MotOrBAC is a tool implemented to administrate, analyze and deploy security policies specified using the *OrBAC* model. The application is entirely written in Java and uses the Jena library. The Jena library is used to process RDF data, the format in which *OrBAC* policies are represented.

2 Architecture

MotOrBAC is built on top of the OrBAC application programming interface (figure 1). The *OrBAC* API manages a set of plug-ins implemented as OSGi bundles, which offer an easy and flexible way to extend its functionalities. MotOrBAC can use those plug-ins and update the plug-in list on the fly. Currently several plug-ins exist among which an OrBAC to XACML translation plug-in, an OrBAC to IPtables translation plug-in and a plug-in which can instantiate an OrBAC policy from a prolog-like policy specification or translate an OrBAC policy to such a notation. The scripts generated by these different plug-ins can be used to automatically provision the security configuration of security components (for instance firewalls) using the Netconf protocol.

Fig. 1. The MotOrBAC tool architecture which is based on the java *OrBAC* API

3 Main functionalities

MotOrBAC can be used to do several tasks on *OrBAC* security policies:

- Edit policies: the administrator can create the abstract entities as defined in the OrBAC model (organizations, sub-organizations, roles, activities, views, contexts) as well as abstract permissions, prohibitions and obligations.
- Policy simulation: after having specified concrete entities (subjects, actions and objects) and assigned them to abstract entities, the concrete policy can be inferred.
- Policy consistency verification: abstract conflicts between abstract rules can be detected.
- Once abstract conflicts have been detected, MotOrBAC is able to suggest the administrator some solutions to solve them. Conflicts are solved at the abstract level since it has been proved that if no abstract conflicts exist then no concrete conflicts can appear.
- Administrative rights management: the administrative rights of a subject or a role can be specified in order to decentralize the policy administration.

Most of those tasks can also be done by interacting with the *OrBAC* API as explained in the next section.

4 Integration of the *OrBAC* API

The *OrBAC* API can be used to programmatically create and interpret *OrBAC* policies. A solution to enforce an *OrBAC* security policy is to use the *OrBAC* Java API, on top of which MotOrBAC is built. It can be used to load MotOrBAC RDF policies and interpret them, i.e access control requests can be made on a loaded policy. *Jena* features an inference engine which is used by the *OrBAC* API to infer the concrete policies and detect possible conflicts. When an *OrBAC* RDF policy is loaded by the API, the concrete policy can be inferred and stored in memory. An instance of the *OrbacPolicy* Java class which encapsulates an *OrBAC* policy uses a cache of concrete security rules to enhance the performances when the policy is queried. Contexts are evaluated upon a query, this feature is actually used in the MotOrBAC simulation tool to show the contexts state. The contexts implementation can be easily extended in order to interface the API with other applications and add new types of contexts.

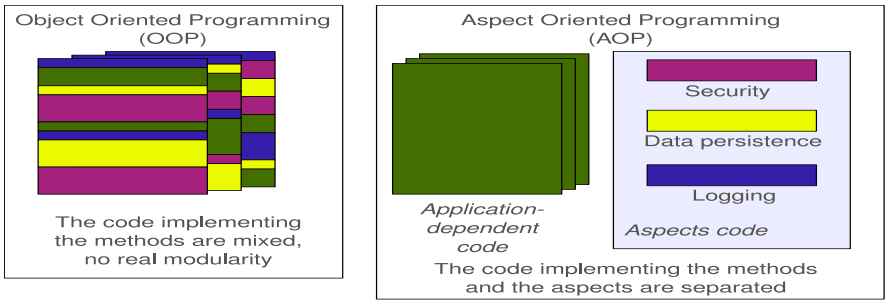


Fig. 2. Benefits of the aspect oriented programming model

Integrating the *OrBAC* Java API into a Java application can be done without modifying the application source code. Aspect Oriented Programming (AOP) can be used to separate security concerns from other concerns relative to the application (figure 2). We have developed a use-case which simulates a medical form where entries must be filed by subjects having specific roles and rights. This medical use-case does not include any security related code. Using *AspectJ*, the source code of this use-case is then secured by weaving the security concerns with the *OrBAC* API.

For more information. See the OrBAC web site for more information and papers about the OrBAC model, contextual security policy management, conflict management, policy deployment, etc. The MotOrBAC prototype is available on sourgeforge.net.

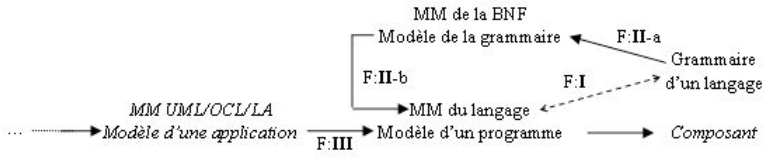
Modélisation des langages de programmation et Processus de développement de logiciels

T.T. Le Thi[1], T. Millan[1], E. Poupart[2], L.Sabatier[3], J.C. Dabin[4]

[1] IRIT-MACAO UPS Toulouse
[2] CNES Toulouse
[3] SOFT-MAINT/SODIFRANCE Toulouse
[4] AIRBUS France - Avionics Embedded Software

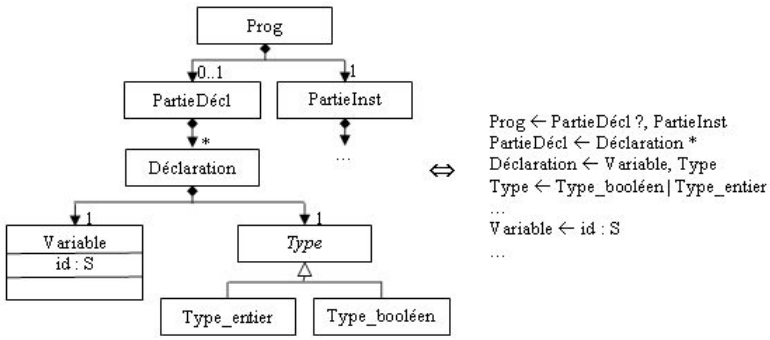
1 Langage, Modèle et Processus

Cette étude a pour objectif de montrer comment exploiter les descriptions formelles des grammaires des langages de programmation au niveau des phases terminales d'un processus de développement de composants logiciels. Un langage est décrit par une grammaire, elle-même décrite selon la BNF. La figure F suivante montre que le Méta-Modèle (MM) d'un langage (F:II-b) peut être obtenu en passant par le MM de la BNF (F:II-a) où l'on pourra vérifier toutes les propriétés du langage:



2 Méta-Modélisation de la grammaire d'un langage

- Harmonisation du formalisme d'une grammaire abstraite d'un langage et de son MM (F:I):



- Exemple de propriétés statiques : Toute variable ne peut être déclarée qu'une seule fois.

```
context Prog   inv declVar :
if          self.partieDecl <> oclUndefined( PartieDecl )
then        self.partieDecl.declaration.variable->forAll( v1, v2 | v1 <> v2 implies v1.id <> v2.id )
else        true
endif
```

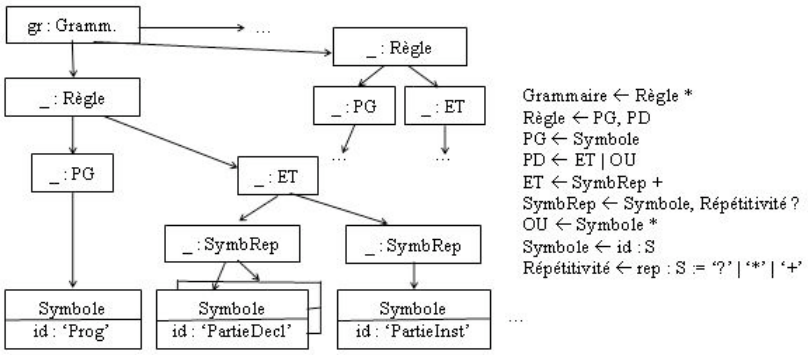
- Exemple de comportement des constructions syntaxiques, "injecté" dans la classe PartieDecl:
– langage d'actions (LA) inspiré de la plate forme USE (Université de Brême)

```
PartieDecl ←    exec( e : Environnement )
                begin       var i : Integer := 1
                  while     i <= self.declaration->size()
                  do        self.declaration->at( i ).exec( e )
                            set i := i + 1
                  wend
                end
```

. Application d'OCL et d'un LA directement sur la grammaire du langage.
. Etude du comportement, en totalité ou partiel, des modèles des composants avant les activités de génération et de déploiement des codes des composants logiciels.
. Intégration au niveau de MM du langage de spécificités et d'exigences du domaine métier.

3 Méta-Modélisation de la BNF

- Toute grammaire est décrite selon la BNF : La figure suivante montre les règles de la BNF et un fragment du diagramme d'objets modélisant le langage de l'exemple précédent, vu comme une instance du MM de la BNF



A ce niveau de MM, on peut vérifier un certain nombre de propriétés. En particulier : l'ensemble des règles peut se représenter sous la forme d'un graphe orienté dans lequel il existe un chemin liant tout symbole au point d'entrée.

4 Discussion, Bibliographie

- Insertion dans un processus de développement des modèles de composants dans le but d'effectuer un dernier niveau de contrôle avant les activités de génération des codes (figure F), d'où une meilleure continuité entre les phases d'élaboration des modèles d'une application et les phases plus techniques liées à la génération et au déploiement des composants logiciels.

- Possibilité de rajout au niveau du MM de la BNF de propriétés, de 'patrons' de codes, ... spécifiques au domaine métier (évolution des langages vers des DSL).

- Avec UML/OCL/LA (USE, KerMeta, pOCL...), contribution à l'unification des différentes approches de description des propriétés des langages de programmation et de la mise en oeuvre des comportements des différentes constructions syntaxiques des langages. En ce qui concerne les invariants et la mise à jour des modèles, une solution consiste à utiliser une plateforme B pour prouver les post-conditions (sémantique axiomatique).

- Remarque : Le composant de transformation MM UML vers MM du langage devrait prendre en compte la vérification des propriétés du langage.

- Application au projet ANR DOMINO'2006 : Prise en compte de modèles UML, élaboration (F:III) des modèles des composants, et génération de codes. Deux cas d'étude : le premier sur "l'interopérabilité pour les procédures opérationnelles spatiales au sol ou embarquées " (CNES), le deuxième sur " la transformation de UML vers C embarqué dit de confiance " (Airbus).

- Références bibliographiques : Langages, Transformations de modèles, Processus, ...

Composition de modèles par points de vue dans le profil VUML

Younes Lakhrissi, Adil Anwar, Bernard Coulette, Sophie Ebersold, Iulian Ober
{prénom.nom}@irit.fr
MACAO-IRIT, Université de TOULOUSE

Contexte : L'intérêt majeur de la modélisation multivue est de permettre l'analyse/conception des systèmes complexes à travers la séparation des points de vue des acteurs du système. Cette séparation facilite la production et la modification éventuelle de modèles spécifiques aux acteurs. Par contre, garantir la cohérence du modèle composé reste un défi à relever. C'est dans cette perspective que notre équipe a développé le profil VUML (View based UML) [3].

VUML offre un formalisme pour modéliser un système logiciel par une approche combinant objets et points de vue. Le principal ajout à UML est celui du concept de **classe multivue** que nous définissons comme une unité d'abstraction et d'encapsulation permettant de stocker et de restituer l'information en fonction du profil de l'utilisateur.

Démarche de développement VUML : Sur le plan méthodologique, VUML propose une démarche centrée utilisateur qui permet d'intégrer la notion de point de vue dans le processus de développement des systèmes. Dans cette approche, plusieurs modèles de conception modélisant les besoins de chaque acteur interagissant avec le système sont développés par différents concepteurs. Ces modèles doivent être ensuite composés (fusionnés) pour produire un modèle VUML unique représentant le modèle de conception global du système.

La démarche que nous proposons est constituée de trois phases (cf. figure 1) : la première est la phase d'analyse des exigences, la deuxième est la phase d'analyse/conception par point de vue, et la troisième est la phase de composition des modèles. Chacune de ces phases est constituée de deux volets : le premier guide les développeurs dans la structuration structurelle d'une application en VUML, le deuxième complète le premier volet en aidant à exprimer le comportement des objets multivues présents dans l'application.

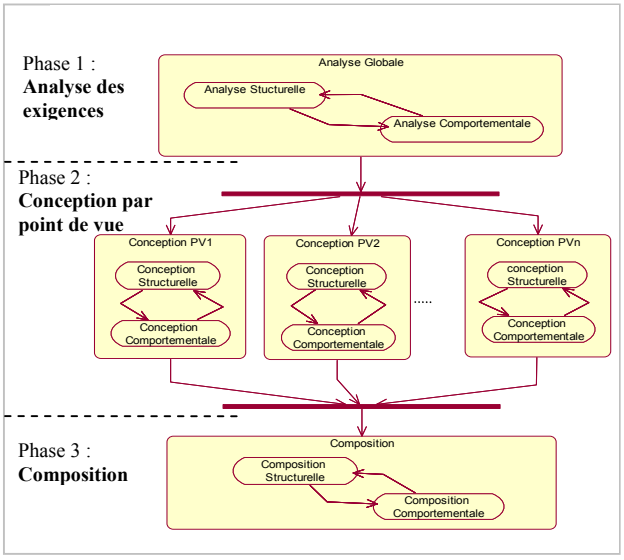


Figure 1. *Vision générale de la démarche VUML*

Composition structurelle : Face à la problématique de la composition de modèles, l'IDM apparaît comme une solution prometteuse puisque l'on peut considérer certaines étapes de la composition de modèles comme des transformations de modèles. En effet, la composition des modèles PIM peut être vue comme une transformation exogène du même niveau d'abstraction (horizontale) car elle prend en entrée des PIM en UML et génère en sortie un PIM conforme à VUML. Nous avons appliqué cette approche pour implanter la fusion de diagrammes de classes UML.

Cette opération est composée de deux étapes, la première étape, manuelle, consiste à éliminer les différents conflits (noms, structurels, etc.) entre les modèles source. La deuxième étape, automatique, est réalisée par deux transformations. La première transformation crée un modèle de correspondances dans lequel sont stockées les différentes relations de correspondance entre les modèles. La deuxième transformation crée le modèle composé VUML en utilisant le modèle de correspondances, les règles de composition, et les règles de translation [1].

Composition comportementale : Ce second type de composition consiste à composer le comportement des objets-vue et à assurer la cohérence du comportement des objets multivue. En fait, un objet multivue est composé d'un ensemble d'objet-vue ayant chacun un comportement décrit par une machine-vue développée lors de la phase de conception par point de vue. Dans l'objectif de garantir un maximum d'indépendance dans le développement du comportement des modèles-vue tout en assurant la cohérence du modèle global, nous proposons un concept - nouveau dans la terminologie UML -, qui est le mécanisme d'observation.

L'utilisation des observations dans la modélisation comportemntale en VUML se fait de la manière suivante : au niveau d'un modèle-vue, quand les conditions de déclenchement d'un comportement ne peuvent être précisées à cause de la dépendance avec d'autres points de vue, le développeur insère une déclaration d'observation abstraite et l'utilise dans son modèle comportemental. Lors de la phase de fusion, le lien entre les modèles-vue se fait en précisant la définition de l'observation, ce qui évite de devoir modifier les modèles-vue en profondeur [2].

Ce travail est mené dans le cadre du réseau franco-marocain STIC et du PAI Volubilis COMPUS.

Références :

1. Anwar A., Ebersold S., Coulette B., Nassar M., Kriouile A., «Vers une approche à base de règles pour la composition de modèles : application au profil VUML». *Revue RSTI-L'Objet*. Hermès Publications, Vol. 13, N. 4/2007, p. 73-103, 2007.
2. Ober I, Coulette B, Lakhrissi Y, «Behavioral Modelling and Composition of Object Slices Using Event Observation». ACM/IEEE International Conference on Model Driven Engineering Languages and Systems (MODELS 2008), Toulouse, Springer, LNCS 5301, p. 219-233, septembre 2008.
3. Nassar M., «Analyse/conception par points de vue : le profil VUML», thèse INPT, Toulouse, septembre 2005.

Manifestations associées

Action IDM
Journée thématique Modèles Orientés Aspects*
Organisée dans le cadre des journées du GDR CNRS GPL
26 janvier 2009
Toulouse

Dans la lignée de l'ingénierie dirigée par les modèles (IDM), un certain nombre de travaux actuels portent sur les modèles orientés aspects en s'inspirant des langages à aspects. Dans les langages à aspect, un aspect représente une préoccupation transversale à une décomposition initiale. Un aspect se définit alors en deux parties qui sont le point de coupe (*pointcut*), c'est-à-dire l'expression de l'endroit où l'aspect doit être appliqué sur la décomposition initiale, et le conseil (*advice*), c'est-à-dire la modification à appliquer. Un aspect peut s'appliquer à plusieurs endroits dans la décomposition initiale. On parlera de coupe. La décomposition initiale est nommée le plus souvent programme de base et est vue également comme l'épine dorsale du logiciel. Il s'agit le plus souvent d'une décomposition fonctionnelle du programme. Le programme complet est obtenu par tissage des aspects (*aspect weaving*) sur le programme de base. La notion d'aspect dans le monde de la programmation présente plusieurs avantages parmi lesquels la séparation des différents points de vue d'un logiciel et la définition d'un processus de composition de ces points de vue.

Dans le cadre de l'ingénierie dirigée par les modèles, le processus de conception d'un système logiciel passe par l'écriture de modèles de différentes natures. Il s'agit le plus souvent de modèles statiques ou dynamiques, fonctionnels ou extra-fonctionnels. Ces divers modèles représentent une vue du système logiciel selon une orientation spatiale ou temporelle. L'objectif est alors de pouvoir produire automatiquement le système logiciel final à partir de ces différents modèles. Chacun de ces modèles pris séparément représente alors un aspect du logiciel et leur assemblage doit permettre une représentation la plus complète possible du logiciel final en vue de la génération d'un système logiciel conforme à ces modèles. Le processus de modélisation peut alors être vu comme une activité de description de séparation de préoccupations et de leur composition.

Modèles et composition ont ici un sens plus large que la notion d'aspect et de tissage dans les langages à aspects. Ils permettent de définir des outils de descriptions et de traitement de ces processus de modélisation. Par la notion de *tissage de modèles*, on souhaite arriver dans l'ingénierie dirigée par les modèles à une meilleure compréhension de la composition. De plus, dans un contexte où la notion de transformation est à la base de l'IDM, cette notion joue à la fois un rôle de simplification et de complémentarité qui reste à déterminer.

L'objectif de cette journée est de rassembler les communautés scientifiques travaillant à la définition de processus de séparation des préoccupations et de composition de modèles. La journée débutera par un état de l'art par Benoît Baudry de l'équipe Triskell, INRIA Rennes - Bretagne Atlantique. Divers travaux en cours seront ensuite présentés par les équipes du laboratoire I3S de l'Université de Nice-Sophia Antipolis, de laboratoire IRIT de l'Université de Toulouse, de l'INRIA Rennes - Bretagne Atlantique et de l'INRIA Lille - Nord Europe. Une table ronde clôturera la journée de façon à ce que les communautés puissent échanger leurs points de vue sur l'utilisation et les apports des approches à aspects à l'ingénierie dirigée par les modèles.

*http://www2.lifl.fr/~duchien/MOA2009/MOA2009.html

Programme de la journée :

- 10h00 Accueil avec café
- 10h30 - 12h00 Un tour d'horizon des approches Modèles Orientés Aspects, par Benoît Baudry, INRIA Rennes - Bretagne Atlantique
- 12h00 - 13h30 Pause déjeuner
- 13h30 - 16h Présentations
  - Composition de Modèles dans le profil VUML par Adil Anwar, Sophie Ebersold, Bernard Coulette, IRIT-MACAO, Université de Toulouse
  - Sensibilité au contexte dans une chaîne de production dynamique pour équipements mobiles par Carlos Andrés Parra, Rafael Leaño, Xavier Blanc, Laurence Duchien, Equipe ADAM, INRIA Lille - Nord Europe
  - Support flexible de dérivation de produits par composition de modèles par Gilles Perrouin, Equipe Triskell, INRIA Rennes - Bretagne Atlantique
  - Méta-composition d'activités par Mireille Blay-Fornarino, Equipe Rainbow, Laboratoire I3S
  - Composition de modèles avec Kermeta par Mikael Clavreul, Equipe Triskell, INRIA Rennes - Bretagne Atlantique
- 16h-16h30 Pause café
- 16h30-18h00 :
  - Présentation du challenge MOA, Jean-Michel Bruel, IRIT, Université de Toulouse
  - Discussions sur les orientations de la modélisation orientée aspect
- 18h00 Conclusion de la journée

Atelier "Outils pour l'IDM" – Mardi 27 janvier 2009

Hubert Dubois[1] & Mireille Blay-Fornarino[2]

[1] CEA LIST, CEA Saclay, 91191 Gif-sur-Yvette Cedex, France, Hubert.Dubois@cea.fr
[2] I3S/Université de Nice, 06903 Sophia-Antipolis Cedex, France
Mireille.Blay@polytech.unice.fr

1 Introduction

La communauté française de recherche en informatique est l'une des communautés les plus actives dans le domaine de l'ingénierie dirigée par les modèles (IDM) avec un nombre important d'équipes de recherches travaillant dans ce domaine. Cette activité est notamment fortement soutenue par le secteur industriel au travers de nombreux projets liés à l'IDM.

Cela se traduit par exemple par un record de papiers soumis et acceptés à la conférence référence de ce domaine, MoDELS (*International Conference on Model Driven Engineering Languages and Systems*[1]), qui, en 2008 s'est tenue à Toulouse du 28 septembre au 03 octobre. Cela se traduit également par un grand nombre d'outils liés à l'ingénierie des modèles, comme des environnements de transformations de modèle ou des modeleurs/méta modeleurs. L'offre actuelle en termes d'environnement de développement dédié, d'outils ou encore de briques logicielles est telle qu'il est parfois difficile de situer les outils les uns par rapports aux autres.

Ainsi, un atelier dédié aux « **Outils pour l'IDM** » a été mis en place en association avec les journées du GDR GPL à Toulouse ; cet atelier a été programmé le **mardi 27 janvier 2009** et se déroule dans les locaux de l'ENSEEIHT.

2 Objectifs

L'objectif de cet atelier, organisé par l'action transverse *Action IDM*[2] commune aux GDR GPL[3] (Génie de la Programmation et du Logiciel), ASR[4] (Architecture, Système et Réseau) et I3[5] (Information, Interaction, Intelligence), est de permettre à tout un chacun de présenter son travail en termes d'IDM et cela au travers de présentations et de démonstrations des outils développés dans le contexte de leur travail (laboratoire, équipe, projet de recherche, etc…). Il pourra s'agir d'outils commercialisés comme d'outils réalisés dans le cadre de projets de recherche de type

[1] Site internet de MoDELS : http://www.modelsconference.org
[2] Site internet de Action IDM : http://www.actionidm.org
[3] Site internet du GDR GPL : http://gdr-gpl.imag.fr
[4] Site internet du GDR ASR : http://asr.cnrs.fr
[5] Site internet du GDR I3 : http://www.irit.fr/GDR-I3

ANR, européen ou autre mais aussi d'outils réalisés dans le cadre de travaux de recherches, de thèses. Les outils présentés pourront ainsi avoir des niveaux de maturité différents. L'idée est également de faire un point sur l'état des travaux relativement à l'IDM au travers des outils développés dans nos laboratoires ou entreprises.

3 Programme de l'atelier

09h30 – 09h45 : **Session plénière – Introduction et présentation de la journée**
09h45 – 11h00 : **Session plénière - Présentations des outils**

09h45 – 10h00	**Platypus, un méta-environnement orienté donnée** (*Alain Plantec* / Université de Brest)
10h00 – 10h15	**MDE MDWorkbench** (*Philippe Soulard* / SODIUS)
10h15 – 10h30	**Sintaks, outil de manipulation de syntaxe textuelle** (*Michel Hassenforder* / UHA-ENSISA)
10h30 – 10h45	**Kermeta, composition de modèles** (*Mickael Clavreul* / IRISA)
10h45 – 11h00	**MAGILLEM, environnement de contrôle de flot pour la conception ESL (*Electronic System Level*)** (*Emmanuel Vaumorin* / *Magillem*)

11h00 – 11h15 : *Pause* et installation des ateliers démonstrations
11h15 – 12h30 : **Ateliers - Démonstrations des outils**

12h30 – 14h : *Déjeuner*

14h- 15h15 : **Session plénière - Présentation des outils**

14h00 – 14h15	**TopCased, environnement pour modéliser des systèmes complexes** (*Marc Pantel* / IRIT-ENSEEIHT)
14h15 – 14h30	**OBP (Observer-Based Prover) développé dans le cadre des projets TopCased et DOMINO** (*Philippe Dhaussy* / ENSIETA)
14h30 – 14h45	**Papyrus, plateforme Eclipse de modélisation et de définition de profils UML2** (*Hubert Dubois* / CEA-LIST)
14h45 – 15h00	**OpenEmbedded, une plateforme en matière d'IDM** (*Vincent Mahé* / IRISA)

15h00 – 15h30 : *Pause* et installation des ateliers démonstrations
15h30 – 17h00 : **Ateliers - Démonstrations des outils (suite)**
17h00 – 17h30 : Session plénière - Bilan de l'atelier « Outils pour l'IDM »

Journée thématique sur la composition d'objets, de composants et de services

Groupe de travail COSMAL : Composants Objets Services : Modèles, Architectures et Langages

Organisée dans le cadre des journées du GDR GPL, le 27 janvier 2009 à Toulouse

Présentation

Dans la continuité des travaux menés par les équipes du groupe COSMAL, de nombreux travaux s'intéressent aux approches par objets, composants et services pour la réalisation de logiciels. Les besoins dans ce domaine sont très variés et une des problématiques est d'offrir les paradigmes appropriés à un développement basé sur la réutilisation. Dans ce cadre il est essentiel de proposer le bon niveau d'abstraction et de séparation des préoccupations.

L'encapsulation des propriétés fonctionnelles et extra-fonctionnelles et du comportement du logiciel dans différentes entités (objets, aspects, points de vue, composants et services) nécessite ensuite de proposer des mécanismes pour composer ces différentes entités ensembles dans le but de constituer le logiciel. Le thème de la journée est justement de s'intéresser à cette étape dans la construction du logiciel.

Notre objectif est de rassembler les communautés scientifiques travaillant à la définition de processus de composition au niveau des langages et des architectures. L'idée est de mettre en exergue les différents défis de la composition dans l'élaboration du logiciel. La journée débutera par une présentation de Philippe Collet qui donnera un premier point de vue sur l'état de l'art du domaine. Cette présentation sera l'occasion de proposer une manière de positionner les travaux des équipes participant au groupe de travail COSMAL. Le reste de la journée prendra la forme de présentations de travaux aboutis ou en cours, le tout ponctués de discussions. Les communautés pourront échanger leurs points de vue sur les apports des différents paradigmes et les problèmes liés à la composition. De nouveaux thèmes de recherche et de nouveaux partenariats pourront ainsi émerger.

Organisation de la journée

Une présentation destinée à expliquer le contexte, les problèmes et les enjeux de la composition d'objets, de composants et de services sera faite par Philippe Collet. Ensuite, nous organiserons la journée sous forme d'une longue discussion qui sera ponctuée par les présentations inscrites au programme au fur et à mesure que les thématiques qu'elles traitent seront abordées.

Liste de thématiques qui pourront être abordées :

Orchestration et chorégraphie de services,
– Composition hiérarchique de composants,

- Composition des propriétés fonctionnelles des composants
- Composition autonomique
- Compatibilité et conformité dans la composition
- Influence des propriétés extra-fonctionnelles sur la composition
- Influence de l'expression de la généricité
- Programmation par sujets,
- Programmation par aspects,
- Programmation par composants
- Contrôle de la composition
- Composition dans le cadre de systèmes autonomiques
- Composition dans les approches à objets
- Composition dans les architectures logicielles
- Composition dynamique ou statique
- Composition invasive vs non-invasive
- Composition et fiabilité, adaptabilité, évolutivité et portabilité ?
- ...

Programme

voici la liste des présentations qui pontueront la journée :

- O.Caron, B.Carré, A.Muller, G.Vanwormhoudt. *Adaptation fonctionnelle de composants gros-grain avec JBOSS/AOP.*
- Ismael Bpuassida Rodriguez. *Gestion des architectures dynamiques pour l'adaptabilité des applications distribuées coopératives. Application au provisionnement de la qualité de service.*
- Younes Lakhrissi, Iulian Ober, Bernard Coulette. *Spécification et composition du comportement dans le profil VUML.*
- Zawar Qayyum, Flavio Oquendo. *Supporting Architectural Composition at Runtime with Π-ADL.NET.*
- Luc Fabresse. *Des modèles aux programmes à base de composants : Besoin de langages à composants.*
- Francoise Baude, Virginie Legrand-Contes. *Exécution distribuée et agile de compositions de services.*
- Pascal André, Gilles Ardourel, Christian Attiogbé, Arnaud Lanoix, Mohamed Messabihi. *Prise en compte d'assertions pour la correction d'assemblages de composants Kmelia.*
- Anthony HOCK-KOON, Mourad OUSSALAH. *Composition Dynamique de Services dans les Environnements d'Intelligence Ambiante.*
- Riadh BEN HALIMA. *Spécification, conception et réalisation d'applications coopératives distribuées autoréparables à base de Services Web.*

Organisateurs

- Mourad Oussalah, LINA
- Philippe Lahire, I3S

Journée de l'action AFSEC

Programme

9h30 : Accueil

9h45 : "Pola: un langage pour la spécification et la vérification des systèmes temps réel", F. Peres, PE Hladik, F. Vernadat, LAAS-CNRS

Cet exposé présente le D.S.L. (Domain specific language) Pola pour la spécification des systèmes temps réels. Pola est un langage déclaratif permettant d'exprimer par des termes "métier" les types d'informations nécessaires à la spécification d'un système de tâches: des tâches et leurs caractéristiques, des politiques d'ordonnancement, des ressources et la distribution des ressources aux tâches. Pola permet également une description fine des tâches en proposant un mécanisme de composition avec un langage de comportement. La vérification du système se fait par model checking via une traduction en réseaux de Petri temporels (et leurs extensions). L'intérêt du model checking est de proposer une technique de vérification automatique dont nous verrons une application très concrète sur une étude de cas pour laquelle la vérification a pu s'effectuer de manière entièrement automatique.

10h15 : "FIACRE, un langage formel pour la spécification de systèmes temps-réels en vue de leur vérification", B. Berthomieu, F. Vernadat, R. Saad, S. Dal Zilio, LAAS-CNRS

FIACRE (Format Intermédiaire pour les Architectures de Composants Répartis Embarqués) est un langage formel pour la description de systèmes temps réels, embarqués ou distribués, en vue de leur vérification ou simulation. Les descriptions FIACRE sont structurées en processus et composants: Les processus décrivent des comportements séquentiels par un ensemble fini d'états de contrôle, chacun associé à une description symbolique des transitions possibles depuis cet état. Les composants permettent d'exprimer de manière hiérarchique et compositionnelle les interactions entre les composants élémentaires qui constituent un système. Ces interactions, résultant de synchronisations, communications par messages, ou communications par variables partagées, peuvent être soumises à des contraintes temporelles et/ou des contraintes de priorités. Les descriptions FIACRE peuvent être vérifiées dans les environnements CADP (INRIA-Grenoble) et TINA (LAAS-Toulouse) grâce à deux compilateurs traduisant ces descriptions dans les formats attendus par ces deux boites à outils. FIACRE a été développé dans le cadre des projets TOPCASED et OpenEmbeDD, en collaboration avec INRIA/VASY et IRIT/ACADIE, dans lesquels il est utilisé comme formalisme pivot entre les langages "métier" et les outils de model-checking.
Liens: Forge FIACRE, Outils TINA, Projet OpenEmbeDD (ANR TechLog), Projet Topcased (Pôle AE/SE).

11h : Pause café

11h15 : "Abstraction d'horloges dans les systèmes synchrones", Marc Pouzet

12h : Repas

13h50 : "présentation du groupe WEED", Marc Boyer, ONERA

14h : "Approche par trajectoire : calcul des délais pire cas dans les réseaux", Steven Martin

Le temps mis par un paquet pour traverser un réseau dépend essentiellement du temps passé dans les files d'attente des différents nœuds visités. Des bornes mathématiquement calculables peuvent être établies sur les temps de réponse et les gigues de bout-en-bout des flux considérés. Pour cela, nous appliquons l'approche dite « par trajectoire », qui ne considère que des scénarios possibles.

14h50 "Convolutions et déconvolution de courbes affines par morceaux: contribution à une modélisation plus fine en calcul réseau", Eric Thierry, LIAFA et Anne Bouillard, IRISA

Le calcul réseau est une théorie permettant de calculer des majorants sur les bornes de délai et d'occupation mémoire dans réseaux. Elle se plonge dans le diode (min,+), et en utilise principalement trois opérateurs: convolution, déconvolution et clôture sous additive. Pour pouvoir modéliser finement des flux de communication, il faut pouvoir calculer effectivement ces opérations sur les classes de courbe considérées. Nous montrons dans cet exposé comment les calculer pour des fonctions affines par morceaux.

15h40 : Pause café

16h : Table ronde sur les projets à venir de l'AFSEC et discussions des sous-groupes