Actes des troisièmes journées nationales du

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

Université Lille I
8 au 10 juin 2011

Editeurs : Yves Ledru
Anne-Françoise Le Meur
Olivier Caron

Impression : service de reprographie Polytech, Université Lille I
Photographie : Damien Pollet

à Anne-Françoise,
Présidente du Comité d'Organisation,
décédée accidentellement le 30 avril 2011.

# Table des matières

# Préface

C'est avec plaisir que je vous accueille aux troisièmes Journées Nationales du GDR GPL. Ces journées marquent la dernière année de l'actuel quadriennal du GDR Génie de la Programmation et du Logiciel (GPL), créé en 2008 pour une durée de 4 ans par le CNRS (GDR 3168). Les missions principales du GDR GPL sont l'animation scientifique de la communauté et la promotion de nos disciplines, notamment en direction des jeunes chercheurs. Cette animation scientifique est d'abord le fruit des efforts de nos groupes de travail, actions transverses et de l'Ecole des Jeunes Chercheurs en Programmation.

Le GDR GPL est maintenant dans sa quatrième année d'activité et ces journées seront l'occasion de mesurer le chemin parcouru et de préparer un deuxième quadriennal. Ces dernières années, les journées nationales se sont affirmées comme un temps fort de l'activité de notre GDR, l'occasion pour toute la communauté de se retrouver et se rencontrer. Plusieurs événements scientifiques sont co-localisés avec ces journées nationales : les conférences IDM 2011 et CAL 2011, ainsi qu'une journée consacrée aux "services" et aux "langages et modèles à l'exécution", organisée par le groupe COSMAL.

Ces journées sont une vitrine où chaque groupe de travail ou action transverse donne un aperçu de ses recherches. Une trentaine de présentations ont ainsi été sélectionnées par les responsables des groupes de travail. Parmi eux, nous souhaitons la bienvenue au nouveau groupe "Compilation". Comme les années précédentes, nous avons demandé aux groupes de travail de nous proposer, en règle générale, des présentations qui avaient déjà fait l'objet d'une sélection dans une conférence nationale ou internationale ; ceci nous garantit la qualité du programme.

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Tom Mens (Université de Mons, Belgique) dont la présentation sera commune aux conférences CAL/IDM et GPL, de Roberto Di Cosmo (Laboratoire PPS, Université de Paris VII) et de Cédric Fournet (Microsoft Research). La conférence de Roberto Di Cosmo sera prolongée par une table ronde, animée par Franck Barbier et dédiée au logiciel libre. Nous aurons aussi le plaisir d'accueillir Jean-Pierre Cocquerez, représentant la direction de l'INS2I, qui fera le point sur les actions de notre institut.

Comme les années précédentes, ces journées ont aussi pour objectif de préparer l'avenir en favorisant l'intégration des jeunes chercheurs dans la communauté et leur future mobilité. Dans cet esprit, nous renouvelons l'opération menée l'an dernier en proposant une formule d'inscription pour les jeunes chercheurs. Nous espérons ainsi qu'ils participeront en nombre à cet événement.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de ces journées nationales : les responsables de groupes de travail ou d'actions transverses, les membres du comité de direction du GDR GPL et, tout particulièrement, le comité d'organisation de ces journées nationales mis en place par Anne-Françoise Le Meur qui nous a quitté prématurément. Je remercie chaleureusement l'ensemble des collègues lillois qui n'ont pas ménagé leurs efforts, malgré des circonstances douloureuses.

Ces journées sont dédiées à Jean-Claude Laprie et Michel Sintzoff, qui nous ont quittés en 2010.

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 2011 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
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, INRIA Lille-Nord Europe, Université Lille 1
Sébastien Gérard, CEA-LIST
Jean-Louis Giavitto (Président du jury des posters), IRCAM, CNRS
Laure Gonnord, LIFL, Université de Lille I
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
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
Chouki Tibermacine, LIRMM, Université Montpellier II
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

Anne-Françoise Le Meur (présidente), Université Lille 1
Nicolas Anquetil, Université Lille 1
Olivier Caron, Université Lille 1
Bernard Carre, Université Lille 1
Laurence Duchien, Université Lille 1
Cédric Dumoulin, Université Lille 1
Anne Etien, Université Lille 1
Sébastien Mosser, Université Lille 1
Damien Pollet, Université Lille 1
José Rouillard, Université Lille 1
Romain Rouvoy, Université Lille 1
Lionel Seinturier, Université Lille 1
Xavier Le Pallec, Université Lille 1
Jean-Claude Tarby, Université Lille 1

# Conférenciers invités

## Analyse de l'évolution des aspects sociaux dans les projets logiciels

**Auteur** : Tom Mens, service de Génie Logiciel, Institut d'Informatique, Faculté des Sciences, Université de Mons, Belgique

**Résumé** :
Le génie logiciel empirique s'intéresse aux études empiriques permettant de comprendre et d'améliorer certains aspects du processus logiciel. Nombre d'entre elles sont dédiées à l'évolution des projets logiciels. Elles extraient les données pertinentes venant de dépôts logiciels ou d'autres sources de données couramment utilisées par les développeurs. Nous suggérons d'élargir ce type d'études empiriques en tenant compte de l'information concernant les communautés de développeurs, ainsi que leur façon de travailler, d'interagir et de communiquer. L'hypothèse sous-jacente étant que les aspects sociaux influent significativement la qualité du produit logiciel, ainsi que la manière dont ce produit évolue au fil du temps. Dans cette conférence, nous présenterons un outil permettant d'extraire, de visualiser et d'analyser l'information concernant les communautés gravitant autour d'un projet logiciel. Nous montrerons quelques études empiriques effectuées avec cet outil, et nous ouvrirons des nouvelles pistes de recherche dans ce domaine de recherche combinant l'analyse des réseaux sociaux et le génie logiciel empirique.

## Modèles et outils pour la vérification de programmes utilisant la cryptographie

**Auteur** : Cédric Fournet, Microsoft Research
Avec Tolga Acar, Karthik Bhargavan, Juan Chen, Andy Gordon, Markulf Kohlweiss, Alfredo Pironti, Dan Shumow, Pierre-Yves Strub, et Nikhil Swamy.
**Résumé** :
Malgré les progrès des techniques de vérification, la sécurité des protocoles et autres systèmes cryptographiques restent problématique. Pour permettre aux programmeurs, cryptographes, et formalistes d'y travailler ensemble, nous proposons de vérifier directement les implantations de protocoles, plutôt que leurs spécifications formelles. Nous développons des bibliothèques cryptographiques et des implantations de références en F#, un dialecte de ML, et nous prouvons leurs propriétés en utilisant des vérificateurs de types dépendants (F7, F*) couplés à un solveur SMT (Z3).

Deux approches coexistent pour la vérification cryptographique. Certains outils sont maintenant capables d'analyser automatiquement des protocoles complexes. Cependant, ils reposent sur des modèles symboliques qui idéalisent le comportement des primitives cryptographiques, supposées parfaitement sures. Les cryptographes leur préfèrent des modèles calculatoires, afin de calculer précisément la complexité et la probabilité de succès des attaques. Ces modèles sont plus réalistes, mais aussi plus difficiles à formaliser et automatiser.

Dans cet exposé, je présente ces deux approches et je discute leur intégration avec des techniques de vérification plus généralistes. A l'aide de petits protocoles codés en ML, je montre comment vérifier quelques propriétés de secret et d'authenticité par le typage. Je présente aussi deux applications plus importantes : TLS 1.2, le standard pour HTTPS, et DKM, un composant pour chiffrer les données stockées dans les nuages. Voir `http://research.microsoft.com/~fournet` et `http://msr-inria.inria.fr/projects/sec` pour plus de détails (papiers, prototypes, applications).

## Le défi des distributions de Logiciels Libres

**Auteur** : Roberto Di Cosmo, Laboratoire PPS, Université de Paris VII

**Résumé** :
Les distributions de logiciels libres, comme Debian, RedHat, ou Ubuntu, sont parmi les plus grands systèmes à composants existants, basées sur les paquets, avec leurs métadonnées, avec une série d'outils qui permettent d'ajouter ou enlever des composants selon le besoin des utilisateurs. Faire évoluer les distributions est une tache complexe qui pose des nombreux défis : dans cet exposé, après avoir donné une formalisation simple des distributions de logiciels libres, on passera en revue plusieurs résultats et algorithmes développés ces dernières années pour répondre à des questions comme "quel composant est le plus important parmi les 27.000 de Debian squeeze ?", ou "quel changement de version aura le plus d'impact sur le système ?".

# Analyse de l'évolution des aspects sociaux dans les projets logiciels

Tom Mens[1] et Mathieu Goeminne[1]

Service de Génie Logiciel, Faculté des Sciences, Université de Mons
Place du Parc 20, 7000 Mons, Belgique
{ tom.mens | mathieu.goeminne }@umons.ac.be

**Résumé** Le génie logiciel empirique s'intéresse aux études empiriques permettant de comprendre et d'améliorer certains aspects du processus logiciel. Nombre d'entre elles sont dédiées à l'évolution des projets logiciels. Elles extraient les données pertinentes venant de dépôts logiciels ou d'autres sources de données couramment utilisées par les développeurs. Nous suggérons d'élargir ce type d'études empiriques en tenant compte de l'information concernant les communautés de développeurs, ainsi que leur façon de travailler, d'interagir et de communiquer. L'hypothèse sous-jacente étant que les aspects sociaux influent significativement la qualité du produit logiciel, ainsi que la manière dont ce produit évolue au cours du temps. Dans cette conférence, nous présenterons un outil permettant d'extraire, de visualiser et d'analyser l'information concernant les communautés gravitant autour d'un projet logiciel. Nous montrons quelques études empiriques effectuées, et nous présentons des pistes de recherche dans ce domaine de recherche combinant l'analyse des réseaux sociaux et le génie logiciel empirique.

**Mot-clés :** qualité logicielle, mesure logicielle, évolution logicielle, étude empirique

## 1 Introduction

Depuis le début de ce millénaire, de nombreuses études empiriques ont été menées dans le but d'étudier comment les logiciels open source évoluent au cours du temps. Les deux raisons principales pour cette popularité sont : (i) l'abondance et l'accessibilité de logiciels pour lesquels l'historique entier de tous les artéfacts logiciels peut être librement analysé ; (ii) la popularité toujours croissante du paradigme "open source", même dans les entreprises.

Cependant, la majorité des études empiriques s'est limitée à l'analyse du code source. Celles-ci ignoraient largement l'effet que les communautés d'utilisateurs et de développeurs (et leurs interactions) ont sur l'évolution de la qualité du produit et du processus logiciel. Des changements dans les aspects sociaux (par exemple, la communauté des développeurs et utilisateurs) influencent directement la manière dont le logiciel change au cours du temps. Nous détaillons dans le présent document comment les études relatives à l'évolution des systèmes open source peuvent être étendues afin de prendre en compte les communautés qui les entourent.

En particulier, dans le cadre de l'évolution, nous nous intéressons à comprendre et à analyser l'aspect temporel : nous voulons déterminer comment, et à quelle vitesse les communautés de développeurs et d'utilisateurs influencent le système logiciel et inversement. Notre hypothèse de travail est que les aspects sociaux ont un impact important sur la manière dont la qualité du processus logiciel et du produit logiciel évoluent au cours du temps. Une meilleure compréhension de cet impact nous permettra de proposer de meilleurs modèles permettant d'analyser et de comprendre l'évolution des logiciels, et de meilleurs outils pour guider les développeurs lors de l'évolution du projet logiciel.

## 2 Les aspects sociaux

La *communauté des développeurs* est constituée des personnes qui créent et modifient les artéfacts du produit logiciel. Les programmeurs modifient le code source, étendent les fonctionnalités

du logiciel et en corrigent les erreurs. Les autres artéfacts techniques du produit logiciel sont modifiés par les documentalistes, les architectes, les responsables des relations avec les clients, etc. Pour peu que le projet ait une taille relativement importante, la communauté a tendance à se structurer, ses membres se spécialisant pour répondre au mieux à certains besoins précis du processus de développement.

De l'interaction entre les membres de la communauté naît un processus influençant l'évolution du logiciel et influencé par elle. Jusqu'à présent, les personnes et les aspects sociaux dirigeant leur interaction étaient très largement négligés dans la recherche scientifique. L'évolution de la qualité du logiciel n'était le plus souvent expliquée que par les aspects techniques, voir uniquement par le code source. Cependant, il s'avère que les aspects techniques ne peuvent justifier et ne permettent pas de comprendre toutes les évolutions des logiciels, et certains chercheurs ont commencé à analyser l'écosystème social des projets open source [1,2,3].

La communication est essentielle pour la réussite de projets logiciels [4,5]. Cela est particulièrement vrai pour les projets open source qui ont la particularité d'être développés et utilisés par des personnes *a priori* dispersées géographiquement. Les développeurs peuvent de plus s'intégrer spontanément dans l'équipe de développement, et la quitter quand bon leur semble, ce qui impose des contraintes d'organisation plus souples que pour les systèmes logiciels propriétaires développés par des salariés.

Pour évoluer, les projets logiciels doivent proposer un système de communication formalisé et accessible par tout le monde permettant aux développeurs d'échanger leurs connaissances et de coordonner leurs efforts. En pratique, l'Internet est largement utilisé du fait de l'accessibilité qu'il offre. C'est un élément décisif dans le succès de nombreux projets open source [6]. La communication transite essentiellement par trois vecteurs : l'e-mail, le système de suivi de bogues et le système de contrôle de versions.

## 3 Les outils d'analyse

Afin d'analyser l'évolution de la communauté des développeurs, des outils d'analyse des communications ont été proposés. Dans le passé nous avons développé un outil, Maispion [7], qui se base sur des échanges de mails entre les différents développeurs pour déterminer leur répartition du travail et leur organisation en général. L'analyse conjointe des e-mails échangés et du système de contrôle de versions permet de déterminer quels sont les développeurs principaux du projet, quelles sont les personnes qui communiquent le plus avec les autres, si deux personnes communiquant intensément interviennent dans le développement d'un même module (le cas contraire est le signe manifeste d'un problème d'organisation), etc.

D'autres études ont été menées pour tenter de déterminer les relations entre les propriétés d'un système open source d'une part, et les propriétés de l'équipe de développement d'autre part. Ces propriétés peuvent être la taille de l'équipe [8], la coopération entre les développeurs [1], l'effort mis en oeuvre pour le développement, ainsi que d'autres critères [9].

Un des défis de la recherche en évolution des logiciels consiste à étudier simultanément l'évolution du produit logiciel, du processus suivi ainsi que de l'aspect social en prenant en compte différentes sources de données et en les recoupant afin d'obtenir une vue globale de l'évolution du logiciel. Cette intégration sera basée sur les premiers travaux menés à ce sujet, à savoir le remplissage d'une base de données regroupant toutes les informations collectées.

La Figure 1 présente *Herdsman*, un framework développé en Java par notre équipe [10], permettant d'atteindre cet objectif d'intégration. Il est constitué de plusieurs couches qu'il faut traverser successivement afin de pouvoir analyser l'évolution des logiciels. Le framework prend en entrée plusieurs sources de données (dépôt de versions, liste de diffusion et suivi de bogues), représentant les différents aspects de l'évolution d'un logiciel. La *couche d'extraction* est composée d'outils qui extraient des artéfacts de ces sources de données. Par exemple, une sorte d'artéfacts générée par un outil d'extraction dédié à un gestionnaire de rapports de bogue est un rapport de bogue. La *couche d'analyse* se base sur ces artéfacts pour calculer des métriques logicielles qui sont placées dans une base de données assurant la persistance des valeurs calculées. Cette base de données est

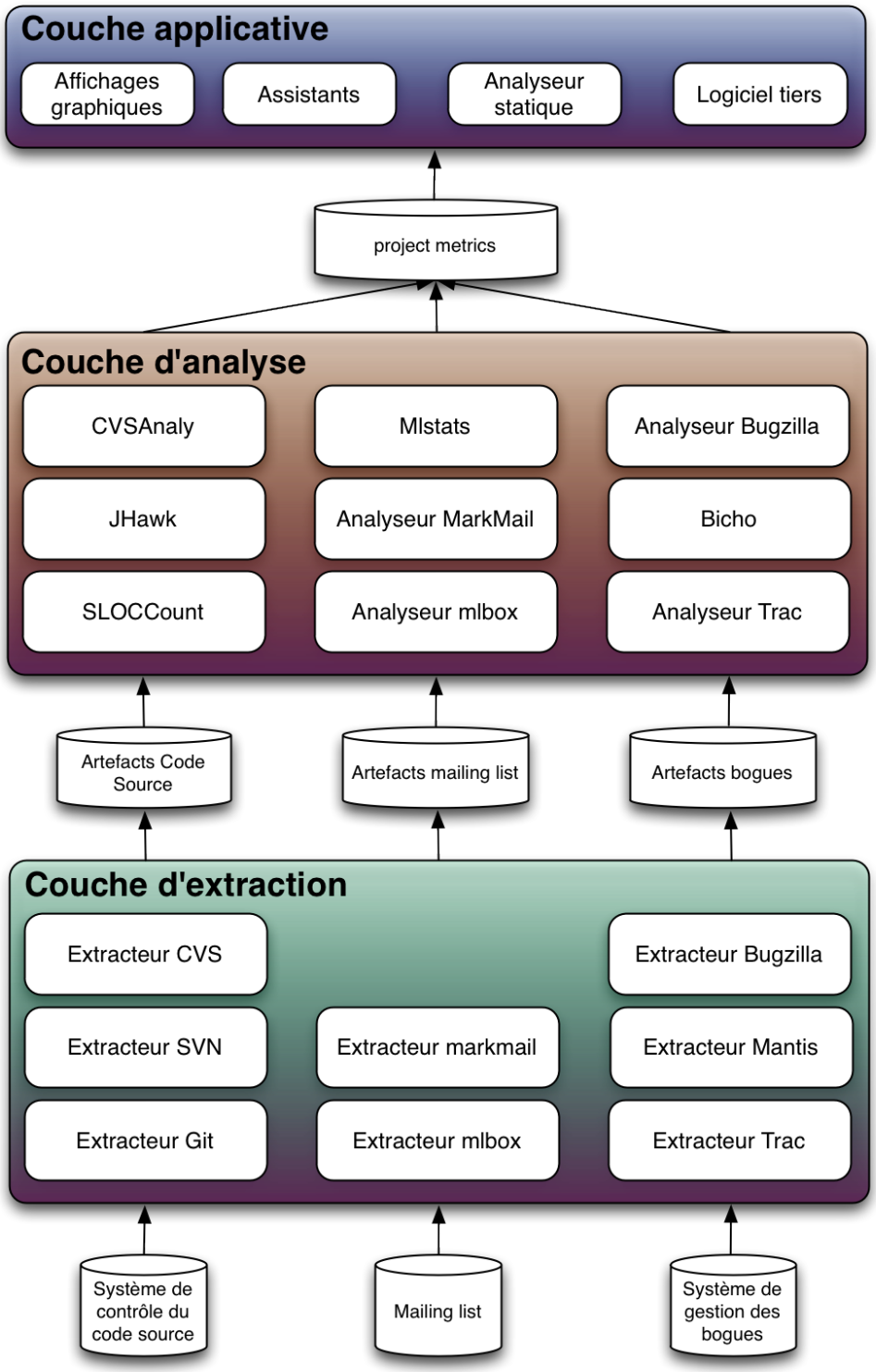



Figure 1: Framework pour l'extraction et l'analyse des données concernant un écosystème logiciel

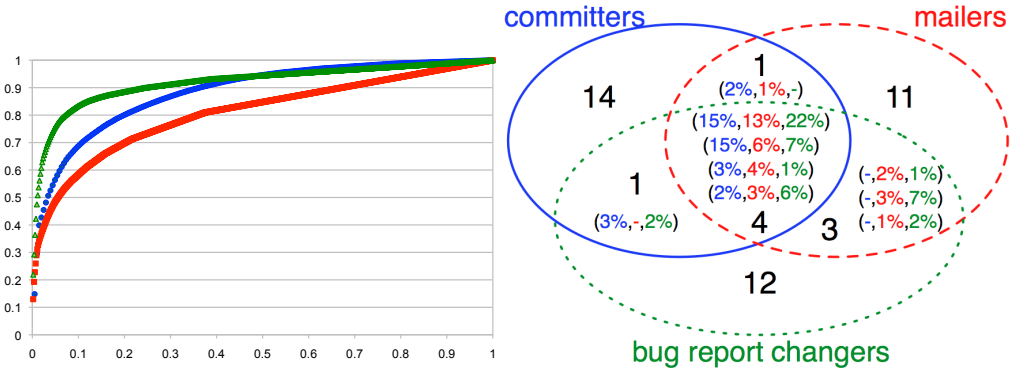
utilisée par la *couche applicative* pour étudier l'évolution du logiciel. La couche applicative est composée entre autre d'outils de visualisation, de générateurs de rapports et d'outils d'analyse statistique. Il est possible de créer de nouvelles applications adaptées aux besoins des utilisateurs.

Une structure de base de données intéressante et utilisable dans la grande majorité des cas est celle proposée par le projet FLOSSMetrics (www.flossmetrics.org), qui permet la collecte des données relatives au code source, aux mails et aux rapports de bogue. Il est toutefois nécessaire de l'adapter selon l'usage qui doit en être fait. Ainsi, il est possible que les métriques supportées par la base de données ne soient pas suffisantes pour l'étude de l'évolution des logiciels.

## 4 Quelques résultats obtenus

Dans cette section nous décrivons quelques résultats obtenus grâce au framework *Herdsman*. Une partie de ces résultats est extrait d'un article précédent [11]. Alors que nous avons analysé plusieurs projets open source, nous montrerons ici seulement les résultats pour le projet *Evince*[1], un lecteur de documents faisant partie de Gnome[2]. *Evince* a été développé principalement dans les langages C et C++, et l'historique de plus de 11 ans de développement est disponible. Il s'agit d'un petit projet, avec en total quatre mille commits, près de deux mille e-mails, et près de mille rapports de bogues.

Nous souhaitons comprendre comment les activités d'un projet logiciel sont distribuées parmi les contributeurs au projet. Pour cela, nous utilisons l'information relative à trois catégories d'activités concernant une personne : les "commits" de fichiers effectués, les mails envoyés, et les rapports de bogue modifiés. La figure 2a montre la distribution cumulative pour ces trois catégories d'activités. La distribution est très déséquilibrée : un petit nombre de personnes s'occupe de la majorité des activités. 20% des "committers" sont responsables de 80% de l'activité totale de commits dans le dépôt de versions ; 20% des "mailers" contribuent pour 70% du nombre de mails ; et 20% de "bug report changers" contribuent à hauteur de 88% de l'activité totale dans le bug tracker.




(a) (b)

FIGURE 2: Analyse de l'activité de la communauté des personnes entourant Evince (novembre 2010). **2a :** La distribution cumulative indiquée par les cercles bleues correspond à l'activité des commits. La distribution cumulative indiquée par les carrés rouges correspond à l'activité de l'envoi des mails. La distribution cumulative indiquée par les triangles verts correspond à l'activié des changements des rapports de bogues. **2b :** Intersection des catégories d'activités pour le top 20 des personnes les plus actives dans chaque catégorie d'activités.

1. projects.gnome.org/evince
2. www.gnome.org

La figure 2b donne une vue plus détaillée des mêmes données, en mettant en correspondance les personnes contribuant simultanément aux trois catégories d'activités considérées. Pour chaque catégorie, seul le top 20 des personnes les plus actives a été considéré. La figure montre clairement que les personnes les plus actives contribuent à différentes catégories d'activités. Par exemple, les deux committers les plus actifs (15% et 15%) sont également très actifs en ce qui concerne l'envoi de mails (13% et 6%) et les changements des rapports de bogue (22% et 7%).

Le déséquilibre de la distribution d'activité peut être résumé par des indices d'agrégation économétriques, comme les indices de Gini, Theil ou Hoover. Une valeur de 0 pour ces coefficients correspond à une distribution uniforme, c.-à-d. que chaque personne a le même taux d'activité. Une valeur de 1 correspond à une situation où une seule personne fait tout le travail et les autres ne font rien. Nous pouvons calculer les indices pour des dates différentes afin de visualiser l'évolution du déséquilibre de la distribution. La figure 3 montre cette évolution pour Evince, en utilisant l'index de Gini, pour les trois catégories d'activités considérées. Dans les trois cas, après une phase de démarrage où l'index augmente très rapidement, ce dernier tend à se stabiliser à une valeur élevée (autour de 0.8), ce qui correspond à un déséquilibre important dans la distribution de l'activité. Pour la liste de diffusion, ce déséquilibre est moins important car le coefficient se stabilise à 0.6, ce qui signifie que davantage de personnes sont régulièrement impliquées dans l'envoi de mails.

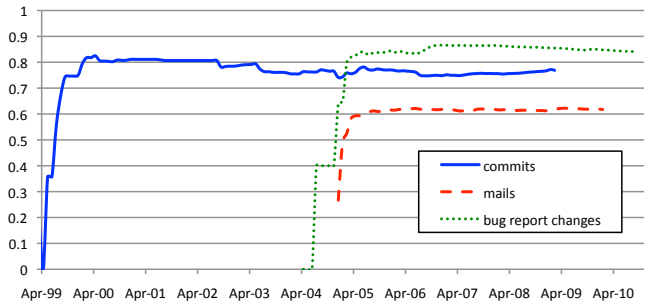
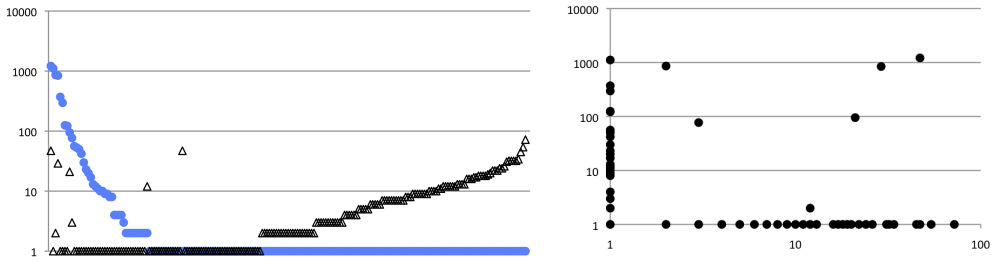



Figure 3: Comparaison de l'évolution de l'index de Gini pour Evince, depuis avril 1999 pour les commits (ligne bleue continue), depuis janvier 2005 pour les mails (ligne rouge discontinue), et depuis août 2004 pour les changements des rapports de bogues (ligne verte en pointillé).

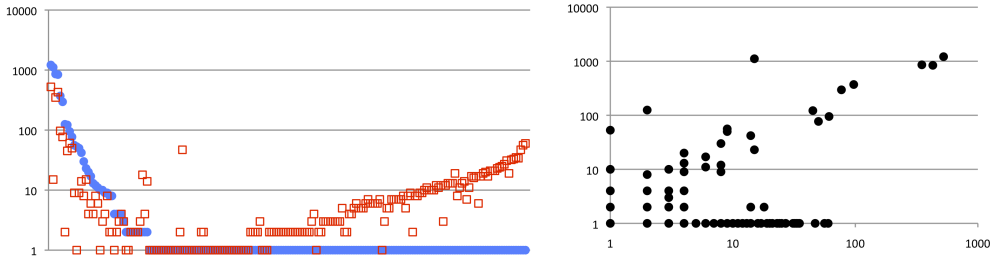
Alors que les résultats précédents nous informent sur la manière dont les membres d'une communauté contribuent à différents types de dépôts, nous pouvons également analyser les patrons d'activités des personnes au sein d'un dépôt donné. Le dépôt de versions contient de nombreux fichiers de différents types : la documentation (p.ex., les fichiers *.html et *.pdf), le code (p.ex., les fichiers *.c, *.h, .java), la traduction (p.ex., les fichiers *.po, *.pot, *.mo, *.charset) et la documentation du développement (p. ex., les fichiers readme*, changelog*, todo*, hacking*). Nous associons à chaque fichier créé, modifié ou supprimé une activité sur base du nom du fichier, et en particulier de son extension.

En analysant le dépôt de versions d'Evince, nous observons qu'une minorité des committers, 57 sur 203 personnes (soit 28,1%) créent, modifient ou suppriment des fichiers de code, alors que le nombre de modifications appliquées à ces fichiers représente 48,4% de toutes les modifications apportées aux fichiers dans le dépôt. Nous pouvons en conclure que les programmeurs sont très actifs, ce qui n'est pas très surprenant puisque les dépôts de versions étaient originalement essentiellement utilisés pour gérer l'évolution du code source d'un projet logiciel. La seconde activité la plus intense est la documentation du développement, avec 26,6% des modifications des fichiers imputables à cette activité. Par rapport aux fichiers de code, beaucoup plus de personnes (189 sur 203, soit 93,1%) s'impliquent dans l'évolution des fichiers associés à cette documentation. La troisième activité la plus intense, avec 12,1% des modifications réalisées sur les fichiers qui lui sont

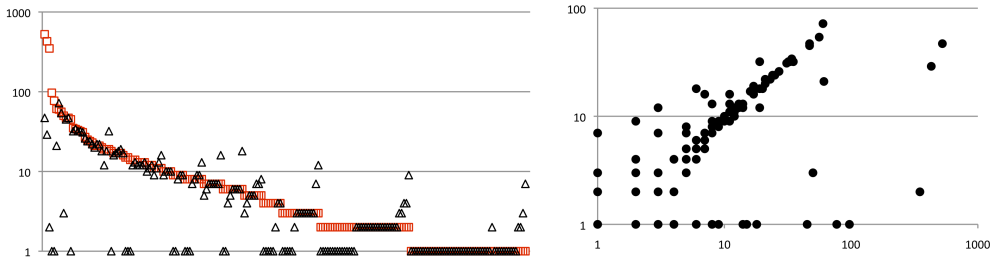
associés, est la traduction de l'application. Les fichiers sont committés par 147 personnes sur 203, soit 72,4%.




(a) programmeurs versus traducteurs




(b) Programmeurs versus documentalistes




(c) Traducteurs versus documentalistes

FIGURE 4: Comparaison des activités des programmeurs (cercles bleus), traducteurs (triangles noirs) et documentalistes (carrés rouges) dans le dépôt de versions de Evince. Dans les figures de **gauche**, l'axe vertical représente, sur une échelle logarithmique, le nombre de modifications des fichiers par personne. Sur l'axe horizontal les personnes sont triés de gauche à droite, d'abord selon le nombre de modifications des fichiers d'un premier type d'activité en ordre décroissant, ensuite selon le nombre de modifications des fichiers d'un deuxième type d'activité en ordre croissant. Les figures de **droite** montrent les nuages de points pour les mêmes données.

La figure 4 détermine, si ces trois types de fichiers ont été gérés par différentes personnes. En particulier, nous souhaitons savoir si les personnes ayant modifié des fichiers de code sont les mêmes que celles ayant modifié les fichiers de traduction ou les fichiers de documentation de développement. La figure 4a montre que très peu de programmeurs (dont l'activité est représentée par des cercles bleus) sont également des *traducteurs* (c.-à-d. personnes contribuant aux fichiers de traduction, dont l'activité est représentée par des triangles noirs). La figure 4b compare les programmeurs et les *documentalistes* (c.-à-d. les personnes modifiant la documentation de développement, dont l'activité est marquée par des carrés rouges). Nous observons une tendance

différente : la plupart des programmeurs sont également impliqués dans la documentation de développement. Le coefficient de corrélation Pearson reste cependant assez faible : 0.396, avec une p-value $< 0.005$. La figure 4c compare les "documentalistes" et les "traducteurs". La corrélation est forte : le coefficient de Pearson est 0.824, avec une p-value $< 0.005$. La plupart des traducteurs sont impliqués dans la documentation de développement et vice versa.

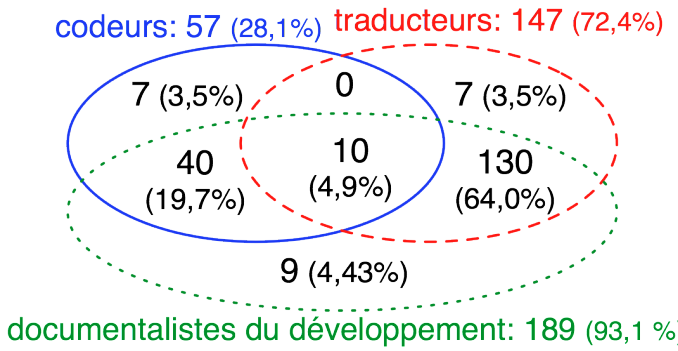



Figure 5: Nombre de personnes modifiant différents types de fichiers dans le dépôt de versions d'Evince.

La figure 5 donne une autre vue des mêmes données, confirmant nos observations : pour chaque type de fichier (fichier de code, fichier de traduction et fichier de documentation du développement), nous comptons combien de personnes ont été actifs. Nous observons que la plupart des programmeurs (50 sur 57) sont également des documentalistes du développement et que la plupart des traducteurs (140 sur 147) sont également des documentalistes du développement. Nous remarquons également que peu de programmeurs (10 sur 57) s'occupent de la traduction et *vice versa* (10 sur 147). Cette disparité démontre l'importance de la prise en compte de l'activité des individus lorsqu'on souhaite analyser les données historiques d'un dépôt logiciel.

## 5 Pistes de recherche future

Dans la section précédente nous avons présenté quelques résultats préliminaires. Nous proposons plusieurs pistes pour étendre nos recherches dans le futur.

Nous étudierons à la fois la communauté des développeurs et la communauté des utilisateurs d'un projet logiciel. La communauté des utilisateurs est essentielle, *a fortiori* pour un projet open source, car nous soupçonnons une relation forte entre la popularité d'un projet logiciel et sa qualité, ou entre le nombre de demandes de changements et la qualité.

Pour la communauté entourant un projet logiciel donné, nous désirons étudier quels sont les individus et groupes de personnes importants, quelles sont leurs interactions, et la manière dont ces interactions *évoluent* au fil du temps. De même, nous désirons comprendre comment la structure de ce réseau social influence la qualité du projet et du produit logiciel.

Nous comparerons, au sein d'un même projet, les profils d'activités des individus et des groupes d'individus, dans le but d'identifier des patrons d'activités récurrents et d'analyser la manière dont ces patrons évoluent au fil du temps. Nous comparerons ces patrons au travers de nombreux projets open source, afin d'identifier les différences et similarités dans leurs communautés, ainsi que l'impact de ces comportements sur le projet logiciel.

Nous utiliserons des outils et méthodes statistiques, des techniques provenant du data mining (comme le clustering) et du domaine de l'analyse des réseaux sociaux.

## 6 Conclusion

Les aspects sociaux ont un impact important sur la manière dont la qualité du processus logiciel et du produit logiciel évoluent au cours du temps. Les études empiriques de l'évolution des systèmes logiciels doivent tenir compte de la communauté entourant le logiciel ainsi que de la façon dont cette communauté influence l'évolution du logiciel.

Nous avons développé un outil pour effectuer des analyses de l'historique des projet logiciels open source et de leurs communautés, en utilisant et combinant différents sources de données. Cet outil nous permet de mieux comprendre les aspects sociaux d'un project logiciel, ainsi que l'interaction entre les communautés et le produit logiciel. A terme, cet outil sera étendu pour développer un support automatisé permettant de guider la communauté à améliorer le processus logiciel ainsi que la qualité du produit logiciel.

**Remerciements**

La recherche présentée dans cet article est effectuée dans le cadre de deux projets de recherche. L'action de recherche concertée AUWB-08/12-UMH19 *"Évolution logicielle dirigée par les modèles"* est financée par le Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgique. Le projet *"Centre d'Expertise en Ingénierie et Qualité des Systèmes (CEIQS)"* de la portefeuille TIC est co-financé par le Fonds Européen de Développement Régional (FEDER) et Wallonia, Belgique.

## Références

1. Greg Madey, Vincent Freeh et Renee Tynan : The open source software development phenomenon : An analysis based on social network theory. *In Americas Conference on Information Systems*, 2002.
2. Audris Mockus, Roy T. Fielding et James D. Herbsleb : Two case studies of open source software development : Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346, 2002.
3. Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida et Yunwen Ye : Evolution patterns of open-source software systems and communities. *In Proc. Int'l Workshop on Principles of Software Evolution*, pages 76–85, New York, NY, USA, 2002. ACM.
4. Jr. Brooks, Frederick P. : *The Mythical Man-Month : Essays on Software Engineering.* Addison-Wesley, 1975.
5. Tom DeMarco et Timothy Lister : *Peopleware : productive projects and teams.* Dorset House Publishing, 1987.
6. Eric S. Raymond : *The cathedral and the bazaar : musings on Linux and open source by an accidental revolutionary.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.
7. François Stephany, Tom Mens et Tudor Gîrba : The open source software development phenomenon : An analysis based on social network theory. *In International Workshop on Smalltalk Tools (IWST '09)*, Brest, France, 2010. ACM Press.
8. Juan Fernández-Ramil, Daniel Izquierdo-Cortazar et Tom Mens : What does it take to develop a million lines of open source code ? *In* Cornelia Boldyreff, Kevin Crowston, Björn Lundell et Anthony I. Wasserman, éditeurs : *OSS*, volume 299 de *IFIP*, pages 170–184. Springer-Verlag, 2009.
9. Juan Fernández-Ramil, Daniel Izquierdo-Cortazar et Tom Mens : Relationship between size, effort, duration and number of contributors in large FLOSS projects. Rapport technique CS-Report 08-30, Technische Universiteit Eindhoven, décembre 2008.
10. Mathieu Goeminne et Tom Mens : Towards a framework for analysing and visualizing open source software ecosystems. *In Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*, September 2010.
11. Mathieu Goeminne et Tom Mens : Evidence for the pareto principle in open source software activity. *In* Magiel Bruntink et Kostas Kontogiannis, éditeurs : *CSMR 2011 Workshop on Software Quality and Maintainability (SQM)*, volume 701, pages 74–82. CEUR-WS.org, 2011.

# Sessions des groupes de travail

# Session de l'action AFSEC

Approches Formelles des Systèmes Embarqués Communicants

# Divide and Recycle: Types and Compilation for a Hybrid Synchronous Language*†

Albert Benveniste[1], Timothy Bourke[1 2], Benoît Caillaud[1], and Marc Pouzet[2]

[1] INRIA, Campus de Beaulieu, 263 avenue du Général Leclerc, 35 042 Rennes cedex.
[2] École normale supérieure, 45 rue d'Ulm, 75230 Paris cedex 05.

## 1 Introduction

Models of complex embedded systems involve more than just discrete control and software components: they usually also include physical devices and elements of operating environments. And thus, Hybrid system modelers, which allow discrete-time reactive or dynamical systems to be mixed with continuous-time ones, have become increasingly important in their development. The best known examples of such modelers are Simulink[1], most notably for explicit (or causal) models composed of Ordinary Differential Equations (ODEs), and Modelica[2], which addresses more general implicit (or acausal) models defined by Differential Algebraic Equations (DAEs). In our work to date, we have focused on modelers for explicit hybrid systems. We refer the reader to [4] for an overview of tools related to hybrid systems modeling and analysis.

We do not address the verification of hybrid systems—an area that has already received much attention—but rather treat hybrid system modelers from a programming language perspective. That is, we bring existing and novel typing, semantic, and compilation techniques to bear on certain aspects of these tools, including simulation reproducibility, interactions between discrete and continuous components, causally-related cascades of zero-crossings, inefficiencies in generated code, and the fidelity of simulations and generated code with respect to source models. While some imperfections during simulation are inevitable due to the approximations made by numerical solvers, or simply because signals are not mathematically integrable, others arise from the lack of a strong typing discipline to separate continuous parts, which are exercised by a numerical solver, from discrete parts, which must be guaranteed not to change during computations within a solver.

Our approach focuses on developing a hybrid extension to a synchronous language. There are two reasons for adapting a synchronous language: 1. it can be used directly to specify discrete components; 2. it can serve as target languages for compilation. This approach is novel, with respect to existing solutions (like Simulink), because it shows that hybrid modelers need not be monolithic and highly-specialized tools, but that, in fact, they can be built from existing synchronous languages and their compilers.

We start with a minimal, yet full featured, Lustre-like synchronous language, and extend it conservatively so that, in a single program, data-flow equations can be mixed with ODEs. Then, we propose a novel type system to *divide* such hybrid programs into their continuous and discrete parts, and we *recycle* existing tools, viz. compilers and numerical solvers, to execute them. The recycling of existing tools involves two interrelated developments. The first is a source-to-source transformation that translates the full language into its synchronous subset While most approaches to integrating variable-step numerical solvers involve specific conventions on blocks within a model (to separate the output computations from modifications of internal states), we are able, by exploiting a standard synchronous language compiler like the one of Lustre [5] or of Signal [2], to generate a single, more efficient, step function that modifies discrete states in place [3]. The second development is an interface between a compiled routine and a single off-the-shelf numerical solver. Programs can then be simulated by cycling between continuous phases

* This extended abstract summarises work already presented in [1].
† This work was supported by the SYNCHRONICS large scale initiative of INRIA.
[1] http://www.mathworks.com/products/simulink/
[2] http://www.modelica.org/

where a numerical solver approximates the state, and discrete phases where the consequences of zero-crossing events are computed. We have implemented our approach in a prototype system based that exploits the SUNDIALS CVODE library [6].

*Dividing (Typing):* The type system distinguishes, at compile time, *discrete* computations from *continuous* ones and ensures that all signals are used in their proper domain. The principle is to give a kind $k \in \{\mathtt{A}, \mathtt{D}, \mathtt{C}\}$ to every expression. An expression has kind `D` when it must be activated on a *discrete* clock, `C` when it must be activated on a *continuous* clock (i.e. when it must be approximated by a numerical solver), and `A` when it can be activated on any clock. A clock is termed *discrete* either if it has been so declared or if it is the result of a zero-crossing or a sub-sampling of a discrete clock. Otherwise, it is termed *continuous*. The type system is conservative with respect to that of the basic synchronous language, that is, synchronous programs are typed *discrete*; and they are compiled in the usual way.

The language includes traditional synchronous data-flow expressions like $v$ `fby` $e$ and `pre`$(e)$ which are of kind `D`, as is, more generally, any non-combinatorial LUSTRE function. And also ODEs `der`$(x)$ $=$ $e$ `init` $e_0$[3] which are assigned kind `C`. Discrete expressions can only be embedded into a continuous context when they are activated on a zero-crossing condition, `up`$(e)$.[4] On the other hand, a continuous expression must never be placed under a zero-crossing, that is, on a discrete clock, since it must be activated continuously.

As an example, consider the following synchronous function, that counts the number of `tick`s between two `top`s, is well typed.

```
let node counter(top, tick) = o where
      o = if top then i else 0 fby o + 1
  and i = if tick then 1 else 0
```

Its type signature is `bool` $\times$ `bool` $\xrightarrow{\mathtt{D}}$ `int` because the equations defining `o` and `i` are both of kind `D`. Imagine now, that we wanted to embed the counter in continuous time, by, for instance, executing it every ten seconds. First we define a continuous signal `time` of slope 1/10. Then we look for zero-crossings on the expressions `time -. 1.0`, and use them both to reset the continuous signal and to activate the discrete counter.

```
let hybrid counter_ten(top, tick) = o where
      der(time) = 1.0 /. 10.0 init 0.0 reset 0.0 every zero
  and zero = up(time -. 1.0)
  and o = counter(top, tick) every zero init 0
```

The type signature of this function is `bool` $\times$ `bool` $\xrightarrow{\mathtt{C}}$ `int`. The construction `counter(top, tick) every zero init 0` is an *activation condition*:[5] the function `counter(top, tick)` is called when `zero` is true; otherwise, `o` keeps its previous value. The initial value is 0. As each of the three equations is of kind `C` then so is their composition.

A variation of the standard bouncing ball example can be programmed in a similar way. In this case the ball has initial position $(x_0, y_0)$ and initial velocity $(x'_0, y'_0)$:

```
let hybrid bouncing(x0,y0,x'0,y'0) = (x,y) where
     der(x)  = x' init x0
 and der(x') = 0.0 init x'0
 and der(y)  = y' init y0
 and der(y') = -. g init y'0 reset -. 0.9 *. last y' every up(-. y)
```

Every time the ball hits the ground (when $-y$ passes upward through zero) its velocity is reset to `-. 0.9 *. last y'`, that is, the direction of the left limit (`last y'`) is reversed and its magnitude is reduced by 10%. The function has type: `float` $\times$ `float` $\times$ `float` $\times$ `float` $\xrightarrow{\mathtt{C}}$ `float` $\times$ `float`.

[3] Which denotes $\dot{x} = e \wedge x(0) = e_0$

[4] Zero-crossings are the mechanism usually used by numerical solvers to identify the occurrence of discrete events. In our context an event occurs when the value of an expression passes 'upward' through zero.

[5] Like, for instance, those of SCADE, or the triggered blocks of SIMULINK.

*Recycling (compilation and execution):* Hybrid programs can be compiled though a source-to-source transformation from the extended language into its discrete subset. This transformation removes all continuous computations, that is, all ODE and zero-crossing expressions, yielding a standard synchronous language program that can be compiled using existing compilers. As an example, consider the transformation of `counter_ten`:

```
let node counter_ten([z], [ltime], (top, tick)) = (o, [upz], [time], [dtime])
  where
      time = 0.0 every z default ltime init 0.0
  and dtime = 1.0 /. 10.0
  and o = counter(top, tick) every z init 0
  and upz = time -. 1.0
```

Additional inputs and outputs have been added so that (a compiled version of) this function can be passed directly to a numerical solver. Two new inputs have been added: for signaling zero-crossings, there is a vector of boolean conditions (`[z]`); and for passing approximated values of continuous states, there is a vector of floats (`[ltime]`). Three new outputs have also been added: the values of zero-crossing expressions are returned in a vector of floats (`[upz]`); new continuous state values are returned in a second vector of floats (`[time]`); and the instantaneous derivatives of continuous states are returned in a third vector of floats (`[dtime]`). The discrete state of the function, that is, the registers implied by the delay operators, only change when the zero-crossing variable is true. The function is otherwise completely combinatorial. The `counter` function is not modified by the translation since it does not contain any continuous states or zero-crossings. This new, discrete version of `counter_ten` can, modulo syntactic details, be processed by any synchronous compiler, and the generated transition function will satisfy the invariant:

> The discrete state never changes when all of the zero-crossing conditions are false.

The execution of compiled programs involves alternating between two phases:

1. a continuous phase, where continuous states (`[ltime]`, in the example) are approximated by a numerical solver, which also monitors the zero-crossing expressions (`[upz]`) for events, and,
2. a discrete phase, where the effect of zero-crossing events are computed.

The discrete phase is triggered by the detection of events during numerical approximation, the ensuing changes may result in further zero-crossings which are detected and processed immediately without reentering the continuous phase. The precise treatment of these so called cascades is a distinguishing characteristic of our approach.

## References


1. A. Benveniste, T. Bourke, B. Caillaud, and M. Pouzet. Divide and recycle: types and compilation for a hybrid synchronous language. In *ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES'11)*, Chicago, USA, April 2011.
2. A. Benveniste, P. LeGuernic, and Ch. Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16:103–149, 1991.
3. D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation of synchronous data-flow languages. In *ACM Int. Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, Tucson, Arizona, June 2008.
4. L.P. Carloni, R. Passerone, A. Pinto, and A.L. Sangiovanni-Vincentelli. Languages and tools for hybrid systems design. *Foundations and Trends in Electronic Design Automation*, 1(1/2), 2006.
5. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.
6. A.C. Hindmarsh, P.N. Brown, K.E. Grant, S.L. Lee, R. Serban, D.E. Shumaker, and C.S. Woodward. SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Trans. on Mathematical Software*, 31(3):363–396, September 2005.

# Building Tight Occurrence Nets from Reveals Relations

Sandie Balaguer, Thomas Chatain, and Stefan Haar
{balaguer, chatain, haar}@lsv.ens-cachan.fr

INRIA & LSV (CNRS & ENS Cachan)

Occurrence nets are a well known partial order model for the concurrent behavior of Petri nets. The causality and conflict relations between events, which are explicitly represented in occurrence nets, induce logical dependencies between event occurrences: if an event $e$ occurs in a run, then all its causal predecessors also occur, and no event in conflict with $e$ occurs. But these structural relations do not express all the logical dependencies between event occurrences in *maximal* runs. Indeed, in this context, concurrency does not necessarily mean logical independency: it is possible that the occurrence of an event implies the occurrence of another one, which is structurally concurrent. This happens with events $a$ and $c$ in Fig. 2: since $a$ is in conflict with $b$ and any maximal run contains either $b$ or $c$, if $a$ occurs in a maximal run, then $b$ does not occur and eventually $c$ necessarily occurs. Yet $c$ and $a$ are not causally related. The *reveals* relation has been introduced in [4] to express this dependency. Here we generalize the reveals relation to express more general dependencies, involving more than two events, and we introduce a logic to express them as boolean formulas. Finally we answer the synthesis problem that arises: given a logical formula $\varphi$, is there an occurrence net $\mathcal{N}$ such that $\varphi$ describes exactly the dependencies between the events of $\mathcal{N}$?

*Occurrence nets and maximal runs:* A *net* is a triple $(P, T, F)$ where $P$ and $T$ are disjoint sets of *places* and *transitions*, respectively, and $F \subseteq (P \times T) \cup (T \times P)$ is the *flow relation*. A *Petri net* (PN) is a tuple $(P, T, F, M_0)$, where $(P, T, F)$ is a finite net and $M_0 \subseteq P$ is the *initial marking*. A Petri net can be unfolded into an occurrence net, ON $= (B, E, F)$, an acyclic net without self-conflict and with an initial event $\bot$, which represents its executions. Fig. 1 gives an example of ON. The structure of an ON induces three relations over its events [5].

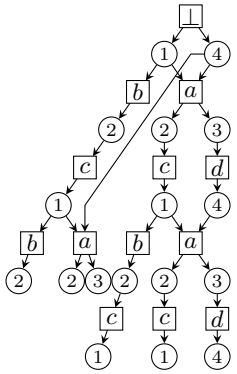



**Fig. 1.** Occurrence net

**causality,** $\leq$**:** $s \leq t \overset{def}{\Leftrightarrow} s\ F^*\ t$, *causal past* of $t$: $\lceil t \rceil \overset{def}{=} \{s \mid s \leq t\}$
**conflict,** $\#$**:** $s \# t \overset{def}{\Leftrightarrow} \exists s' \in \lceil s \rceil, t' \in \lceil t \rceil : s' \#_d t'$, where $s \#_d t \overset{def}{\Leftrightarrow} {}^\bullet s \cap {}^\bullet t \neq \emptyset$
is the *direct conflict*, *conflict set* of $t$: $\#[t] \overset{def}{=} \{s \mid s \# t\}$
**concurrency,** $co$**:** $s\ co\ t \overset{def}{\Leftrightarrow} \neg(s \# t) \wedge \neg(s \leq t) \wedge \neg(t \leq s)$

A *run* of an ON is a conflict-free and causally closed set of events, i.e. $\omega \subseteq E$ is a run iff $\forall e \in \omega, (\#[e] \cap \omega = \emptyset) \wedge (\lceil e \rceil \subseteq \omega)$. A run is *maximal* iff it is maximal w.r.t. $\subseteq$. The set of maximal runs is denoted by $\Omega$.

*Reveals relation and facets abstraction:* Event $e$ *reveals* event $f$, written $e \triangleright f$, iff $\forall \omega \in \Omega, (e \in \omega \Rightarrow f \in \omega)$. Moreover, when we consider maximal runs, $e \triangleright f \Leftrightarrow \#[f] \subseteq \#[e]$.

A *facet* [4] is a set of events that reveal each others, see Fig. 2. For any facet and for any run, either all events in the facet are in the run or no event in the facet is in the run. Therefore, facets can be seen as events. In the sequel, we consider *reduced ONs* [4], i.e. ONs reduced by contracting facets into events, see Fig. 3. The relations $\leq$, $\#$, $co$, and $\triangleright$ naturally extend to the set of facets.

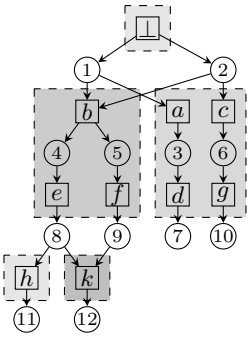



**Fig. 2.** Squared events are in the same facet.

*A logic for occurrence nets:* We introduce a logic, called *ERL* for *Event Reveal Logic*, that describes the properties of the runs of an ON by giving relations between event occurrences. Facets are used as boolean variables: $a$ stands for the presence of facet $a$ in a run. Our logic is designed so that it allows us to express complex dependencies between event occurrences, which can be given in

particular by constraints using an appropriate *extended reveals relation*, $\rightarrowtriangle$. Two kinds of binary minimal constraints are particularly useful: the *immediate conflicts* $a \#_i b \stackrel{def}{\Leftrightarrow} a \# b \wedge \nexists c : (c \neq a \wedge a \triangleright c \wedge c \# b) \vee (c \neq b \wedge b \triangleright c \wedge c \# a)$, and the *immediate reveals*, $\triangleright_i$, transitive reduction of the reveals relation. They are special cases of minimal constraints with $\rightarrowtriangle$.

*From ONs to ERL Formulas:* For a given reduced ON $\mathcal{N}$, the formula which describes the set of maximal runs of $\mathcal{N}$, $\Phi_{\mathcal{N}}$, can be built as follows:

$$\Phi_{\mathcal{N}} = \underbrace{\bigwedge_{a,b\in\Psi, a\lessdot b} (b \rightarrow a)}_{\text{causal closure}} \wedge \underbrace{\bigwedge_{a,b\in\Psi, a\#_d b} (\neg a \vee \neg b)}_{\text{conflict-freeness}} \wedge \underbrace{\bigwedge_{a\in\Psi} \Big( \overbrace{\big( \bigwedge_{b\in\Psi, b\lessdot a} b \big)}^{a \text{ enabled}} \rightarrow \big( a \vee \bigvee_{c\in\Psi, c\#_d a} c \big) \Big)}_{\text{progress assumption}}$$

where $\lessdot$ is the transitive reduction of $\leq$ and the progress assumption (maximality) means "for any facet $a$, if $a$ is enabled, then $a$ or a direct conflict with $a$ has to fire", see Fig. 4.

*From ERL formulas to ONs:* We give a procedure to build a net, $\mathrm{CN}(\varphi)$, from an ERL formula $\varphi$. First, a set of binary minimal constraints is extracted from $\varphi$ (this can be solved quite efficiently in practice by SAT-solvers), then, $\mathrm{CN}(\varphi)$, is built from these constraints: each binary minimal constraint is represented by a condition connected to the facets that appear in the constraint. If $\mathrm{CN}(\varphi)$ is a reduced ON, then $\Phi_{\mathrm{CN}(\varphi)}$ is computed and compared with $\varphi$. We prove that there exists a finite reduced ON $\mathcal{N}$ such that $\Phi_{\mathcal{N}} \equiv \varphi$ iff $\mathrm{CN}(\varphi)$ is a reduced ON and $\Phi_{\mathrm{CN}(\varphi)} \equiv \varphi$.

We call *tight net* a reduced ON in which all binary logical dependencies among facets (reveals relations) are represented as causalities. Since we represent the immediate reveals as causalities, $\mathrm{CN}(\varphi)$ is a tight net. This synthesis lets us tackle two problems: given a formula $\varphi$, does there exist a reduced ON $\mathcal{N}$ such that $\Phi_{\mathcal{N}} \equiv \varphi$? And given a reduced ON $\mathcal{N}$, build the associated canonical tight net. For example, the canonical tight net associated with the reduced ON of Fig. 4(a) is shown in Fig. 4(b).

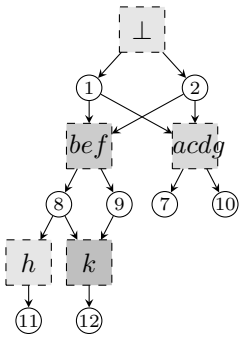
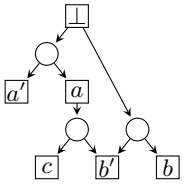



**Fig. 3.** The corresponding reduced ON.

## References


1. Robin Bergenthum, Jörg Desel, Robert Lorenz, and Sebastian Mauser. Synthesis of Petri nets from finite partial languages. *Fundam. Inform.*, 88(4):437–468, 2008.
2. Joost Engelfriet. Branching processes of Petri nets. *Acta Inf.*, 28(6):575–591, 1991.
3. J. Esparza, S. Römer, and W. Vogler. An improvement of McMillan's unfolding algorithm. *Formal Methods in System Design*, 20(3):285–310, 2002.
4. Stefan Haar. Types of asynchronous diagnosability and the *reveals*-relation in occurrence nets. *IEEE Transactions on Automatic Control*, 55(10):2310–2320, 2010.
5. Mogens Nielsen, Gordon D. Plotkin, and Glynn Winskel. Petri nets, event structures and domains, part I. *Theor. Comput. Sci.*, 13:85–108, 1981.

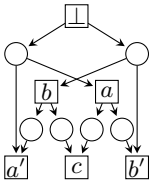




(a) Reduced ON $\mathcal{N}$

$$\begin{aligned}\Phi_{\mathcal{N}} \equiv\ & (c \rightarrow a) \wedge (b' \rightarrow a) \\ & \wedge (\bar{a}' \vee \bar{a}) \wedge (\bar{b}' \vee \bar{b}) \wedge (\bar{c} \vee \bar{b}') \\ & \wedge \bot \wedge (\bot \rightarrow (b' \vee b)) \\ & \wedge (\bot \rightarrow (a' \vee a)) \\ & \wedge (a \rightarrow (c \vee b')) \\ & \wedge ((a \wedge \bot) \rightarrow (c \vee b' \vee b))\end{aligned}$$

minimal constraints:

$$\begin{aligned}(a \triangleright_i \bot, b \triangleright_i \bot)\\ c \triangleright_i a\\ c \triangleright_i b\\ a' \triangleright_i b\\ b' \triangleright_i a\\ a \#_i a'\\ b \#_i b'\end{aligned}$$




(b) Synthesized net, $\mathrm{CN}(\Phi_{\mathcal{N}})$

**Fig. 4.** Example

# Flow Control with (Min,+) Algebra

Euriell Le Corronc, Bertrand Cottenceau, and Laurent Hardouin

Laboratoire d'Ingénierie des Systèmes Automatisés, Université d'Angers,
62, Avenue Notre Dame du Lac, 49000 Angers, France,
{euriell.lecorronc,bertrand.cottenceau,laurent.hardouin}@univ-angers.fr,
WWW home page: http://www.istia.univ-angers.fr/LISA/

## 1 Introduction

According to the theory of Network Calculus based on the (min,+) algebra (see [3] and [4]), analysis and measure of worst-case performance in communication networks can be made easily. In this context, this paper deals with traffic regulation and performance guarantee of a network *i.e.* with flow control. More precisely, the optimal window size of a window flow controller is given by considering the following configuration: The data stream (from the source to the destination) and the acknowledgments stream (from the destination to the source) are assumed to be different and the service provided by the network is assumed to be known in an uncertain way, more precisely it is assumed to be in an interval.

## 2 Network Calculus

In the theory of Network Calculus, a communication network is seen as a black-box denoted $S$, with an input flow $u$ constrained by an arrival curve $\alpha$, and an output flow $y$. Moreover, the service provided by $S$ is constrained by a lower curve $\underline{\beta}$ and an upper curve $\overline{\beta}$. These constraints combined with the following operations of the (min,+) algebra (see [2]) provide bounds on worst-case performance measures. Let $f$ and $g$ be two non-decreasing functions from $\mathbb{R}$ to the dioid $\overline{\mathbb{R}}_{min} = (\mathbb{R} \cup \{-\infty, +\infty\})$, such that $f(t) = 0$ and $g(t) = 0$ for $t \leq 0$, these operations are:

- pointwise minimum : $(f \oplus g)(t) = \min[f(t), g(t)]$,
- pointwise maximum : $(f \wedge g)(t) = \max[f(t), g(t)]$,
- inf-convolution : $(f * g)(t) = \min_{\tau \geq 0} \{f(\tau) + g(t - \tau)\}$,
- deconvolution : $(f \oslash g)(t) = \max_{\tau \geq 0} \{f(\tau) - g(\tau - t)\}$,
- subadditive closure : $f^{\star}(t) = \min_{\tau \geq 0} \ f^{\tau}(t)$ with $f^{0}(t) = e$.

## 3 Window flow control

First, a difference is made between the data stream represented by network $S_1$, and the acknowledgments stream represented by network $S_2$. Indeed, the

acknowledgments stream requires considerably less bandwidth than the data itself (see [1]), so the computation of the window size will have benefit of this profit of bandwidth.

Second, the service provided by the network is assumed to be included in interval, *i.e.* into $[\ \underline{\beta}_1\ ,\ \overline{\beta}_1\ ]$ for $S_1$ and $[\ \underline{\beta}_2\ ,\ \overline{\beta}_2\ ]$ for $S_2$. In that way, the size of the window can be computed as well as for the worst case than for the best case of traffic without damaging the service provided.

Finally, let $\gamma_w$ be the representative function of the window size $w$ ($\gamma_w(t) = w$ for $t < 0$ and $+\infty$ for $t \geq 0$).

The service curve of the whole system is included in the interval:

$$[\ \underline{\beta}_1(\gamma_w\underline{\beta}_2\underline{\beta}_1)^\star\ ,\ \overline{\beta}_1(\gamma_w\overline{\beta}_2\overline{\beta}_1)^\star\ ].$$

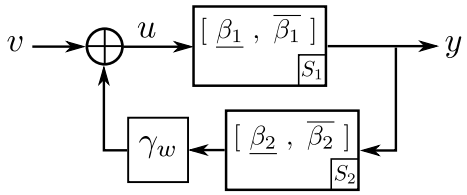



Fig. 1: Configuration of the window flow control system.

The chosen point of view is to compute a minimal window size such that the global network behavior, *i.e.* the controlled one, is the same as the open-loop network behavior, *i.e.* the one of $S_1$ only. This objective can be stated as follows:

$$\hat{\gamma_w} \ = \ \bigoplus\{\gamma_w \mid \underline{\beta}_1(\gamma_w\underline{\beta}_2\underline{\beta}_1)^\star = \underline{\beta}_1 \quad \text{and} \quad \overline{\beta}_1(\gamma_w\overline{\beta}_2\overline{\beta}_1)^\star = \overline{\beta}_1\}. \qquad (1)$$

**Proposition 1.** *In order to obtain a behavior of the closed-loop system unchanged in comparison to the one of the open-loop (see equation (1)), the optimal window size $\hat{w}$ represented by function $\hat{\gamma_w}$ is given below:*

$$\hat{\gamma_w} = (\underline{\beta}_1 \oslash \underline{\beta}_1 \not\oslash (\underline{\beta}_2\underline{\beta}_1)) \wedge (\overline{\beta}_1 \oslash \overline{\beta}_1 \not\oslash (\overline{\beta}_2\overline{\beta}_1)).$$

## References

1. Agrawal R, Cruz RL, Okino C, Rajan R (1999) Performance bounds for flow control protocols. IEEE/ACM Transactions on Networking (TON), IEEE Press 7(3):310–323.
2. Bouillard A, Thierry E (2008) An algorithmic toolbox for network calculus. Journal of Discrete Event Dynamic Systems, Springer 18(1):3–49.
3. Chang CS (2000) Performance guarantees in communication networks. Springer Verlag.
4. Le Boudec J-Y, Thiran P (2001) Network calculus: a theory of deterministic queuing systems for the internet. Springer.

# Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures et Langages

# Faire levier sur les architectures logicielles pour guider et vérifier le développement d'applications SCC ⋆

Damien Cassou[1], Emilie Balland[2], Charles Consel[2], and Julia Lawall[3]

[1] Software Architecture Group, Hasso-Plattner-Institut, Potsdam, Germany,
[2] INRIA/University of Bordeaux, France,
[3] DIKU, University of Copenhagen, Denmark

**Résumé** Une architecture logicielle décrit la structure d'un système informatique en spécifiant ses composants et leurs interactions. Projeter une architecture logicielle sur une implémentation est une tâche reconnue difficile. Un élément crucial de cette projection est la description architecturale des interactions entre les composants. La caractérisation de ces interactions peut être plutôt abstraite ou très concrète, fournissant plus ou moins de support de programmation et de possibilités de vérifications statiques.

Nous explorons un point dans l'espace de conception entre les spécifications abstraites et concrètes des interactions de composants. Nous introduisons la notion de *contrat d'interactions* qui exprime les interactions autorisées. Cette déclaration architecturale permet la génération de support de programmation qui assure la conformité entre l'architecture et l'implémentation, et favorise diverses vérifications. Nous instancions notre approche sur un langage de description d'architectures pour les applications *Sense/Compute/Control* et décrivons les stratégies de compilation et de vérification associées.

## 1 Introduction

Une application *Sense/Compute/Control* (SCC) est une application qui interagit avec un environnement extérieur. Ces applications se retrouvent dans des domaines comme la domotique, la robotique et l'informatique autonome. Développer une application SCC est complexe car l'implémentation doit prendre en compte l'interaction avec l'environnement. De plus, la correction est essentielle puisque un changement dans l'environnement peut avoir des conséquences irréversibles.

Une application SCC peut être définie suivant un style architectural comprenant quatre types d'éléments organisés en couches : (1) les *sources*, en bas, obtiennent les informations de l'environnement ; (2) les *opérateurs de contexte* traitent ces informations ; (3) les *opérateurs de contrôle* utilisent ces informations raffinées pour contrôler (4) les *actions*, en haut, qui impactent finalement l'environnement. Projeter une architecture logicielle ayant un tel niveau d'abstraction vers une implémentation et maintenir cette projection sont des tâches reconnues difficiles.

Dans cet exposé nous proposons une approche pour lier architecture et implémentation qui vise les applications SCC. Cette approche introduit la notion de *contrat d'interactions* permettant à un architecte de déclarer quelles sont les interactions qu'un élément de l'architecture a le droit de réaliser (Section 2). Cette notion de contrat d'interactions est dédiée au style architectural SCC dans le sens où un contrat d'interactions ne peut, syntaxiquement, décrire que les interactions autorisées par le style. Les contrats d'interactions sont utilisés pour générer un support de programmation qui va guider le travail d'implémentation par les développeurs tout en maintenant la conformité avec l'architecture (Section 3). L'architecte peut aussi utiliser les contraintes exprimées par les contrats d'interactions pour vérifier un ensemble de propriétés allant au delà de la conformité (Section 4).

## 2 Contrats d'interactions

Le but d'un contrat d'interactions est de décrire les interactions autorisées d'un opérateur au sein d'une application SCC. Ce contrat d'interactions est un triplet constitué des informations

⋆. Ces deux pages résument l'article *"Leveraging Software Architectures to Guide and Verify the Development of Sense/Compute/Control Applications"* publié à ICSE '11

suivantes : *la condition d'activation* permet d'indiquer quelles sont les interactions capables d'activer l'opérateur ; *les données requises* permettent d'indiquer les interactions supplémentaires autorisées pour chaque condition d'activation ; *les actions à entreprendre* permettent d'indiquer la réponse appropriée à chaque activation (émission d'une information pour un opérateur de contexte ou commande d'une action pour un opérateur de contrôle). En résumé, les contrats d'interactions guident le travail de l'architecte en lui proposant un cadre de spécification dédié au style SCC.

## 3 Support de programmation

Nous avons intégré les contrats d'interactions dans DiaSpec, un langage de description d'architectures dédié aux applications SCC. À partir d'une architecture en DiaSpec, un générateur de code produit un *framework* de programmation Java dédié. Ce *framework* de programmation généré contient une *classe abstraite* pour chaque élément de l'architecture. Cette classe abstraite générée contient des méthodes pour faciliter l'implémentation des éléments ainsi que des déclarations de méthodes abstraites permettant d'implémenter la logique applicative. Implémenter un élément DiaSpec nécessite donc de créer une sous-classe de la classe abstraite générée correspondante. En conséquence, dans cette approche, un architecte peut changer l'architecture et générer un nouveau *framework* de programmation sans écraser le code des développeurs. Les changements dans l'architecture qui ont un effet sur le code déjà implémenté sont révélés par le compilateur Java assurant par là la conformité de l'implémentation avec l'architecture.

Chaque contrat d'interactions d'un opérateur se projette vers la déclaration d'une méthode abstraite dans la classe abstraite générée correspondante à l'opérateur. En particulier, (1) la condition d'activation influence le nom de la méthode abstraite ainsi que son premier paramètre ; (2) les données requises se projettent vers autant de paramètres représentant des fonctions permettant d'exécuter l'interaction supplémentaire ; (3) les actions à entreprendre se projettent vers un ou plusieurs paramètres supplémentaires ainsi que vers le type de retour de la méthode.

Le *framework* de programmation est généré de façon à guider les développeurs dans l'implémentation de l'application ainsi qu'à les limiter à ce que l'architecture autorise. En particulier, un contrat d'interactions se projette vers une méthode abstraite qui fournit en paramètre tout ce qui est nécessaire à l'implémentation de la logique applicative de l'opérateur. De plus, la déclaration de cette méthode impose au développeur de respecter les contraintes de l'architecture, assurant ainsi la conformité entre l'architecture et l'implémentation.

## 4 Support de vérification

Les contrats d'interactions rendent explicites des informations sur le flot de données et permettent des vérifications statiques. Par exemple, avec les contrats d'interactions, il est possible de savoir au moment de la conception tous les opérateurs qui seront éventuellement activés par la publication d'une information par une source. De plus, notre stratégie de génération assure que ces propriétés seront préservées au niveau de l'implémentation.

Les contrats d'interactions permettent aussi de vérifier des *invariants d'interactions* qui sont des propriétés vérifiées à tout moment de l'exécution. Nous caractérisons la progression d'une application SCC par son flot de données et utilisons la logique temporelle linéaire (LTL) pour définir ces invariants. Pour vérifier ces invariants, une architecture DiaSpec est automatiquement traduite en un modèle pour le *model checker* SPIN. Si un invariant n'est pas satisfait, SPIN donne un contre exemple sous la forme d'une trace d'exécution qui guide l'architecte dans la correction de son architecture. Cet exemple montre que les contrats d'interactions rendent explicites les concepts clés du style architectural SCC et donc facilitent les analyses sur le flot de données.

## 5 Conclusion

Nous avons introduit la notion de contrat d'interactions exprimant les interactions autorisées au sein d'une architecture SCC. Les contraintes exprimées par ces contrats d'interactions permettent des vérifications et guident l'implémentation de l'architecture, tout en assurant la conformité.

# Modèles réactifs pour le contrôle de reconfiguration dans le modèle à composants Fractal

Gwenaël Delaval[1] and Éric Rutten[2]

[1] LIG/Université de Grenoble, France, gwenael.delaval@inria.fr
[2] LIG/INRIA Grenoble, France, eric.rutten@inria.fr

**Résumé** Nous présentons une méthode pour la conception de contrôleurs de reconfiguration dans le modèle à composants Fractal. Cette méthode permet d'obtenir une boucle de contrôle assurant automatiquement des propriétés de sûreté concernant les interactions entre composants, telles que l'exclusion mutuelle, ou le fait d'interdire ou d'imposer certaines séquences. Nous utilisons un langage de programmation pour les systèmes réactifs, muni d'un mécanisme de contrats permettant d'exprimer les propriétés à assurer. La compilation de ce langage inclut une phase de synthèse de contrôleurs discrets, qui permet de générer automatiquement le contrôleur de reconfiguration assurant ces propriétés. Nous montrons la pertinence de cette approche en l'illustrant sur un problème de gestion de ressources pour un serveur HTTP.

## 1 Contexte et motivations

Les systèmes d'informations doivent de nos jours être de plus en plus *adaptatifs* : ils doivent accomplir des reconfigurations, autant que possible dynamiques, en réaction à des changements de leur environnement. Ces changements peuvent concerner, typiquement, l'énergie ou l'alimentation disponible, la bande passante, la qualité de service demandée, ou encore la fiabilité et la tolérance aux fautes du système. Une autre motivation des systèmes ≪adaptatifs≫ ou ≪autonomiques≫ est la complexité de l'administration, induisant le besoin de techniques automatiques pouvant remplacer ou suppléer l'administration manuelle [6].

Les méthodes de conception et d'implémentation de stratégies d'adaptations sont actuellement l'objet de recherches. Les systèmes autonomiques [5] en sont une approche, dans laquelles des fonctionnalités sont définies au niveau du système d'exploitation ou du middleware, permettant de tester l'état du système, et de décider à partir de cet état d'actions de reconfiguration à réaliser.

## 2 Contrôle de reconfiguration en boucle fermée

Dans ce contexte, la gestion de l'adaptativité dynamique peut être vue comme une boucle de contrôle fermée, continue ou discrète. Cette boucle consiste en

un composant de contrôle qui, sur la base d'informations sur l'état courant du système ou de son environnement, et à partir d'un modèle de ce système, va imposer une politique d'adaptation définie, en décidant des actions de reconfigurations à réaliser [6].

La conception correcte de boucles de contrôle fermées, et l'étude de leurs propriétés, est l'objet de l'automatique. Les applications de méthodes issues de l'automatique continue aux systèmes d'information ont été largement explorés [4]. En revanche, les aspects logiques, tels que traités par l'automatique discrète, ou par les systèmes hybrides combinant aspects dynamiques discrets et continus, n'ont été considérés que très récemment pour les systèmes adaptatifs ; et ce uniquement concernant des propriétés logiques spécifiques (absence d'interblocage à l'exécution [7]).

Nous proposons d'utiliser le langage de programmation BZR [2], afin de concevoir des boucles de contrôle discrètes de systèmes adaptatifs.

Nous nous intéressons au modèle à composants Fractal [1], où sont définis des notions de cycle de vie et de contrôleur locaux à chaque composant, et qui servent à gérer leurs interactions et leur coordination. La classe de changements dynamiques considérée ici est la commutation à l'exécution entre des configurations caractérisées par des états stables, dans lesquels une activité de calcul donnée est exécutée. Dans le cadre de Fractal, nous nous intéressons donc à faire des transitions d'un assemblage de composants à un autre [3]. Les actions de base sont l'ajout ou le retrait de composant, et l'établissement ou la déconnexion de liens. Nous appliquons cette approche sur un problème de gestion de ressources pour un serveur HTTP en Cecilia, implémentation sur C du modèle Fractal.

## Références


1. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The fractal component model and its support in java. *Software – Practice and Experience (SP&E)*, 36(11-12), sep 2006.
2. G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. of the ACM Conf. on Languages, Compilers and Tools for Embedded Systems, LCTES*, 2010. `http://hal.inria.fr/inria-00436560`.
3. G. Delaval and E. Rutten. Reactive model-based control of reconfiguration in the fractal component-based model. In *Proc. of the 13th Int. Symp. on Component Based Software Engineering (CBSE),* Prague, 23-25 June, 2010.
4. J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback Control of Computing Systems*. Wiley-IEEE, 2004.
5. J. O. Kephart and D. M. Chess. The vision of autonomic computing. *IEEE Computer*, 36(1), 2003.
6. S. Krakowiak. *Middleware Architecture with Patterns and Frameworks*. electronic book, 2009. Chap. 10, `http://sardes.inrialpes.fr/~krakowia/MW-Book`.
7. Y. Wang, S. Lafortune, T. Kelly, M. Kudlur, and S. Mahlke. The theory of deadlock avoidance via discrete control. In *ACM Symp. on Principles of Programming Languages (POPL '09)*, Jan. 2009.

# Une approche orienté aspect allant du modèle d'exigences au modèle de conception

## Définition d'un processus itératif d'ingénierie logicielle

Sébastien Mosser[1], Gunter Mussbacher[2], Mireille Blay–Fornarino[3], and Daniel Amyot[4]

[1] INRIA Lille–Nord Europe, LIFL (UMR CNRS 8022), Univ. Lille 1, France
[2] SCE, Carleton University, Canada
[3] I3S (UMR CNRS 6070), Université Nice–Sophia Antipolis, France
[4] SITE, University of Ottawa, Canada

**Résumé** Des approches orientées aspects sont aujourd'hui disponibles à chaque phase du développement d'un logiciel : analyse des exigences, conception, ou encore implémentation. Passer d'une phase à l'autre en conservant les aspects identifiés au préalable reste un défi majeur, pourtant peu étudié. Nous proposons dans [2] une approche itérative et dirigée par les préoccupations, permettant de transformer un modèle d'exigences orienté aspect en un modèle de conception lui aussi orienté aspect, et ceci de manière automatique. Cette approche est mise en œuvre en utilisant AoURN (*"use case maps"*) pour le modèle d'exigence et Adore pour le modèle de conception (orchestrations SOA). Elle permet l'encapsulation continue des préoccupations identifiées lors des exigences, transformées en artefact de conception. Nous proposons de plus une boucle de rétro–action permettant de remonter dans les modèles d'exigences des défauts constatés dans le modèle de conception, supportant ainsi un processus de développement itératif.

*Introduction.* De nombreuses méthodologies "aspects" (AO⋆) ont été proposées dans la littérature pour supporter la séparation des préoccupations au cours du cycle de vie du logiciel. Cependant, peu de travaux ont lié ces approches entre elles, alors que par essence l'AO⋆ vise a conserver une telle séparation tout au long du processus de développement. Nous proposons d'étudier ce problème dans le contexte des Architectures Orientées Services (SOA), en prenant pour cible deux phases du processus : *(i)* analyse des exigences (au travers de modèles d'exigence AoURN [3]) et *(ii)* conception (au travers de modèles d'orchestration de services Adore [1]). Les objectifs de ce travail sont illustrés en figure 1 : *(1)* fournir une approche automatisée (*i.e.*, implémentée par un algorithme) permettant de transformer les aspects identifiés au niveau des exigences en aspects utilisables lors de la phase de conception, et *(2)* remonter dans le modèle d'exigences les modifications apportées (*e.g.*, pour corriger des défauts) lors de la conception.

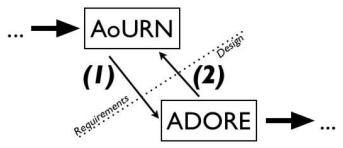
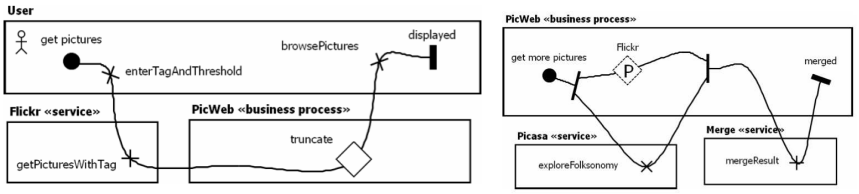


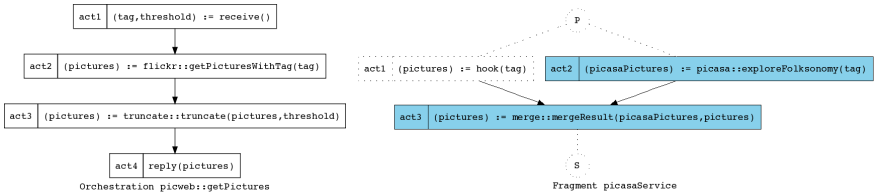

**Figure 1.** AoURN ⇌ Adore

*Étude de cas.* Nous illustrons cette approche à l'aide de PicWeb, une application SOA de référence utilisée par plusieurs universités (Nice, Lille 1, Univ. Du Texas à Austin) comme support d'enseignement. L'objectif de cette application est représenté en terme de modèle d'exigence en figure 2(a) : lors de la réception d'un `tag` et d'un `seuil`, PicWeb va interroger le service Flickr à la recherche d'images associées à ce `tag`, puis restreindre le jeu d'images récupérées au `seuil` donné avant de le renvoyer. Une nouvelle "préoccupation" est l'utilisation du service de stockage Picasa en addition du service rendu par Flickr. Une telle préoccupation est représentée en figure 2(b) : en parallèle de l'invocation de Flickr, nous ajoutons une invocation à Picasa, suivi de la concaténation des résultats obtenus avant de poursuivre le comportement pré-préexistant. En terme SOA, les modèles d'exigences définis précédemment peuvent être mis en œuvre sous la forme d'orchestrations de services. Nous utilisons la plate–forme Adore pour modéliser l'orchestration PicWeb (figure 3(a)) ainsi que l'aspect d'ajout du service Picasa (figure 3(b)).




(a) `getPictures` (b) `picasa`

**Figure 2.** Modèle d'exigence orienté–aspect ("Use Case Maps" AoURN)




(a) `getPictures` (b) `picasa`

**Figure 3.** Modèle de conception orienté–aspect (Orchestration Adore)

*Mise en œuvre et Bénéfices.* Nous proposons dans [2] un algorithme supportant la transformation automatique de modèles AoURN en modèles Adore. Grâce à cet algorithme, nous bénéficions des avantages suivants dans le cadre du développement des applications :

Exigences → Conception : Les modèles de conception sont générés automatiquement à partir des modèles d'exigences. De fait, il n'y a pas de divergences entres les différents artefacts, ce qui augmente la cohérence entre les différentes phases de développement. De plus, les analyses effectuées sur les modèles d'exigences permettent d'assurer des propriétés sur les modèles de conception (*e.g.*, respect de règles métiers validées en termes d'exigences). Pour finir, les règles d'ordonnancement d'aspects identifiées à un très haut niveau d'abstraction sont automatiquement adaptées pour être prise en compte par les aspects de composition.

Conception → Exigences : Le grain plus fin des modèles de conception permet d'identifier des interactions invisibles lors de la phase précédente. Ces informations peuvent être remontées dans les modèles d'exigences afin d'en renforcer la sémantique. Dans le même esprit, des flots de données inconsistants peuvent être identifiés, illustrant des oublis dans les modèles d'exigences. Pour finir, des choix de conception (*e.g.*, enrichissement d'interface de services) peuvent être réinjectés au niveau précédent pour être discutés en terme d'exigences avec le client.

*Références.*

[1] Mosser, S. : Behavioral Compositions in Service-Oriented Architecture. Ph.D. thesis, Université Nice - Sophia Antipolis, ED STIC, Nice, France (Oct 2010), http ://nyx.unice.fr/publis/mosser :2010.pdf

[2] Mosser, S., Mussbacher, G., Blay-Fornarino, M., Amyot, D. : From Aspect-oriented Requirements Models to Aspect-oriented Business Process Design Models. In : 10th international conference on Aspect Oriented Software Development (AOSD'11) AR=20% , long paper : 1st round. , ACM, Porto de Galinhas (Mar 2011)

[3] Mussbacher, G., Amyot, D., Araújo, J., Moreira, A. : Requirements Modeling with the Aspect-oriented User Requirements Notation (AoURN) : A Case Study. In : Katz, S., Mezini, M., Kienzle, J. (eds.) Transactions on Aspect-Oriented Software Development VII. Lect. Notes Comp. Sci., vol. 6210, pp. 23–68. Springer (2010)

# Session du groupe de travail Compilation

# Compiling for a Heterogeneous Vector Image Processor*

Fabien Coelho and François Irigoin

CRI, Maths & Systems, MINES ParisTech, France
*firstname.lastname*@mines-paristech.fr

**Abstract.** We present a new compilation strategy, implemented at a small cost, to optimize image applications developed on top of a high level image processing library for an heterogeneous processor with a vector image processing accelerator. The library provides the semantics of the image computations. The pipelined structure of the accelerator allows to compute whole expressions with dozens of elementary image instructions, but is constrained as intermediate image values cannot be extracted. We adapted standard compilation techniques to perform this task automatically. Our strategy is implemented in PIPS, a source-to-source compiler which greatly reduces the development cost as standard phases are reused and parameterized for the target. Experiments were run on the hardware functional simulator. We compile 1217 cases, from elementary tests to full applications. All are optimal but a few which are mostly within a mere accelerator call of optimality. Our contributions include: 1) a general low cost compilation strategy for image processing applications, based on the semantics provided by library calls, which improves locality by an order of magnitude; 2) a specific heuristic to minimize execution time on the target vector accelerator; 3) numerous experiments that show the effectiveness of our strategy.

## 1 Introduction

Heterogeneous hardware accelerators, based on GPU, FPGA or ASIC, are used to reduce the execution time, the energy used and/or the cost of a small set of application specific computations, or even the cost of a whole embedded system. They can also be used to embed the intellectual property of manufacturers or to ensure product perennity. Thanks to Moore's law, their potential advantage increases with respect to standard general-purpose processors which do not gain anymore from the increase in area and transistor number. But all these gains are often undermined by large software development cost increases, as programmers knowledgeable in the target hardware must be employed, and as this investment is lost when the next hardware generation appears.

We present a compilation strategy to map image processing applications developed on top of a high-level image library onto a heterogeneous processor with

* This work is funded by the French ANR through the FREIA project [2].

a vector image processing accelerator. This approach is relatively inexpensive as mostly-standard and reusable compilation techniques are involved: only the last code generation phase is fine-tuned and target-specific.

Our hardware target, the SPoC vector image processing accelerator [9], currently runs on a FPGA chip as part of a SoC. The hardware accelerator implements directly some basic image operators, possibly part of the developer visible API: this hardware-level API characterizes the accelerator instruction set. Dozens of elementary image operations such as dilatations, erosions, ALUs, thresholds and measures, can be combined to compute whole image expressions per accelerator call. However these capabilities come with constraints: only two images can be fed into the accelerator internal pipeline structure, and two images can be extracted after various image operations performed on the fly. The accelerator is a set of chained vector units. It does not hold a single image but only a few lines (2 lines per unit) which are streamed in and out of the main memory. There is no way to extract intermediate image values from the pipeline.

The application development relies on the FREIA image processing library API [2]. A software implementation on top of Fulguro [8], a portable open-source image processing library, is used for functional tests. The developer has no knowledge of the target accelerator hardware. Operators of the FREIA image library must be programmed specifically for the chosen target accelerator, either by simply calling basic hardware accelerated operators (basic hardware operator library implementation), or, better, with a specialized implementation (hardware optimized library implementation) that takes advantage of the hardware by composing basic operations. Although the library layer provides functional application portability over accelerators, it does not provide all the time, energy and cost performance expected from these pieces of hardware.

In order to reach better performance, library developers may be tempted to increase the sizes of API's to provide more opportunities for optimized code to be used, but this is an endless process leading to over-bloated libraries and possibly non-portable code: up to thousands of entries are defined in VSIPL [1], the Vector Signal Image Processing Library. In contrast to this library-restricted approach, we use the basic hardware operator library implementation, but the composition of operations needed to derive an efficient version is performed by the compiler for the whole application. We see the image API as a domain specific programming language, and we compile this language for the low-level target architecture.

The keys to performance improvement are to lower the control overhead and to increase data locality at the accelerator level, so that larger numbers of operations are performed for each memory load. This is achieved by merging successive calls to the accelerator, with no or few memory transfers for the intermediate values. To detect which calls to merge, techniques have been developed such as loop fusion or complex polyhedral transformations. Such techniques cannot be applied usefully on a well-designed, highly modular software library such as Fulguro: loops and memory accesses are placed in different modules and loop

nests are not adjacent: size checks, type dispatch and dynamic allocations of intermediate values are performed between image processing steps.

Instead of studying the low-level source code and trying to guess its semantics with respect to the available hardware operators, we remain at the higher image operation level. We inline high-level API function calls not directly implemented in the accelerator, unroll loops, flatten the code, so as to increase the size of basic blocks. These basic blocs are then analyzed to build expression DAGs using the instruction set of the accelerator. They are optimized by removing common subexpressions and propagating copies. Up to here, the hardware accelerator is only known by the operations it implements. We then consider hardware constraints, such as the number of vector units, data paths, code size or local memory available, and split these expression DAGs into parts as large as possible, but meeting these constraints. Finally, using the expression DAGs as input, we generate the configuration code ands calls to a runtime library activating the accelerator, and replace the expressions by these calls.

The whole optimization strategy is automated and implemented in PIPS [17,4], a source-to-source compiler, which let the user see the C source code that is generated. This greatly helps compiler debugging. We compile 1217 test cases, from elementary tests to full applications, all of which are optimal but a few. Experiments were run with the SPoC functional simulator. The results on the running example included in this paper show a speed-up of 16.5 over the most naïve use of the accelerator, and a speed-up of 3 over the use of the optimized library.

In the remainder of this paper, we first introduce our running example which is a short representative of the application domain (Section 2) and present the target architecture (Section 3). Then we show how the user source code is preprocessed to obtain basic blocks with optimization opportunities (Section 4). Next, compiler middle-end optimizations for locality are described (Section 5), and the back-end SPoC specific hardware configuration generation is detailed (Section 6). We finally present our implementation and experimental results obtained with a SPoC simulator (Section 7), and discuss the related work (Section 8).

## 2 Applications and Running Example

The FREIA project aims at mapping efficiently an image processing applications developed on top of a high-level API onto different hardware accelerators. The image applications use all kind of image processing operations, such as: `AND`-ing an image with a mask to select a subregion; `MAXLOC`-cating where is the hottest point; `THR`-esholding an image with values to select regions of interest; mathematical morphology (MM) [20] operators. The MM framework created in the 1960's provides a well-founded theory to image analysis, with algorithms described on top of basic image operators. The project targets high performance, possibly hardware accelerated, very often embedded, high-throughput image processing. For this purpose, the software developer is ready to make some efforts in order reach the expected high performances for critical applications on selected hardware. Current development costs are high, as application must be optimized


**Fig. 1.** License plate *(LP)*: character extraction


**Fig. 2.** Out of position *(OOP)*: airbag ok or not

**Fig. 3.** Video survey *(VS)*: motion detection

from the high-level algorithmic choices down to the low-level assembler code and memory transfers for every hardware targets. The project aims at reducing these costs through optimizing compilation and careful runtime designs. Typical applications extract informations from one image or from a stream of images, such as a license plate in a picture (LP, Figure 1), whether a car passenger is out of position and could be harmed if the airbag is triggered (OOP, Figure 2), or whether there is some motion under a surveyance camera (VS, Figure 3).

The high-level FREIA image API has several implementations. The first one is pure C, based on the Fulguro [8] open-source image processing library, and is used for the functional validation of the applications. There are two implementations for the SPoC vector hardware accelerator (Section 3), which can run over a functional simulator or on top of the actual FPGA-based hardware: One uses SPoC for elementary functions, which are directly supported by the SPoC instruction set, one elementary operator at a time. The other is hand-optimized at the library call level by taking full advantage of the SPoC vector hardware capability to combine operations. Other on going versions of the library are optimized for the Terapix [5] SIMD accelerator, and for OpenCL targeting graphics hardware (GPGPU).

The code in Figure 4 was defined as part of the FREIA project to provide a short test case significant both for the difficulties involved and for the optimization potential, with the two hardware accelerators in mind. The test case contains all the steps of a typical image processing code: an image is read, intermediate images are allocated and processed, and results are displayed. As it is short enough to fit in a paper, we use it as running example, together with extracts from larger applications. Optimization opportunities at the `main` level of our test case are very limited. The `min` and `vol` function calls correspond to two SPoC instructions. Since they are next to each other and use the same input argument, they can be merged into a unique call to SPoC. The `dilate` and `gradient` functions are not part of the SPoC instruction set. They are implemented in the non-optimized SPoC version of the FREIA library, using calls to elementary functions. Since these calls are not visible in the main function, no optimization is possible in this case. With the naïve elementary function based implementation, 33 calls to the accelerator are used per frame, hidden in the callees. A hand-optimized SPoC implementation of the FREIA image library results in 6 accelerator calls only, because calls to elementary functions can be merged within the implementation of the FREIA functions.

## 3 SPoC Architecture

Figure 5 outlines the structure of the SPoC processor. It can be seen as a simplified version of the 30 year old CDC Cyber 205 [16], specialized for image processing instead of floating point computation. A MicroBlaze provides a general purpose scalar host processor and a streaming unit, the SPoC pipeline, made of several image processing vector units, constitutes the image processing accelerator. It also contains a DDR3 memory controller, DMA engines, FIFOs to

```c
#include <stdio.h>
#include <freia.h>

int main(void) {
  freia_dataio fin, fout;
  freia_data2d *in, *og, *od;
  int32_t min, vol;
  // initializations
  freia_common_open_input(&fin, 0);
  freia_common_open_output(&fout, 0, ...)
  in = freia_common_create_data(fin.bpp, ...);
  od = freia_common_create_data(fin.bpp, ...);
  og = freia_common_create_data(fin.bpp, ...);
  // get input image
  freia_common_rx_image(in, &fin);
  // perform some computations
  freia_global_min(in, &min);
  freia_global_vol(in, &vol);
  freia_dilate(od, in, 8, 10);
  freia_gradient(og, in, 8, 10);
  // output results
  printf("input global min = %d\n", min);
  printf("input global volume = %d\n", vol);
  freia_common_tx_image(od, &fout);
  freia_common_tx_image(og, &fout);
  // cleanup
  freia_common_destruct_data(in);
  freia_common_destruct_data(od);
  freia_common_destruct_data(og);
  freia_common_close_input(&fin);
  freia_common_close_output(&fout);
  return 0;
}
```

**Fig. 4.** FREIA API running example

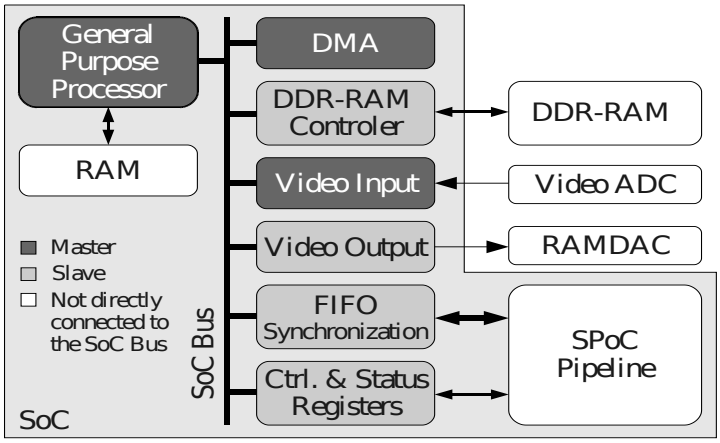



**Fig. 5.** SPoC architecture

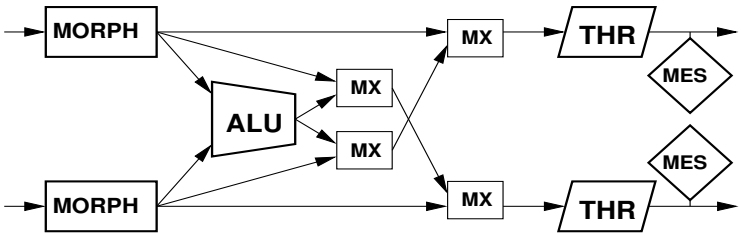



**Fig. 6.** One SPoC vector unit, to be chained

synchronize memory transfers and vector computations and the host, a gigabit Ethernet interface and video converters for I/Os.

Figure 6 shows one vector unit of the SPoC pipeline, with two inputs and two outputs of 16 bit-per-pixel images. The units are chained linearly, directly one to the next, using their outputs and inputs: there is no apparent vector image registers. The first inputs and last outputs are connected to the external memory by DMA engines. A vector unit is made of several operators, but the interconnection is not free: the data paths are quite rigid, with some control by multiplexers *MX*. One morphological operator *MORPH* can be applied to each input. Their results can be combined by an arithmetic and logic unit, *ALU*. Two outputs are selected among those three results by the four multiplexers which control the stream of images. Then a threshold operator, *THR*, can be applied to each selected output and the reduction engine *MES* compute reductions such as maximal or sum of the passing pixels, the result of which can be extracted if needed after the accelerator call. To sum up, each micro-instruction can perform concurrently up to 5 full image operations and a number of reductions, equivalent to 29 pixel operations per tick. A `NOP` micro-instruction is available to copy the two inputs on the two outputs. It is useful when some vector units of the SPoC pipeline are unused.

The host processor controls the vector units by sending them one micro-instruction each and by configuring the four DMA engines for loading and storing pixels. The host processor can also retrieve the reduction results from the vector units. The control overhead remains small because images are always large enough to generate very long pixel vectors. A low resolution image, for instance $320 \times 240$, is equivalent to a 76800 element vector.

When considering FPGA implementations, the number of vector micro-instructions that can be executed concurrently, i.e. the number of vector units, ranges from 4 to 32. The limiting factor is the internal RAM available. Our reference target hardware includes 8 vector processing units, but the solution we suggest below is parametric with respect to this number. In practice, this vector depth provides a reasonable cost-performance trade-off as it fits patterns of iterated erosions and dilatations on few images that are often found in typical applications, but is yet not too expensive when these patterns are not found. With a specific set of application in mind, several vector depth can be tested to choose the best setting. The total number of image operations that can be executed at a given time is 5 times the number of units, not counting the reductions. So the compiler must chain 40 image operations of the proper kind and order to obtain the peak performance. Unlike the Cray vector register architecture, only two inputs are available. Unlike the CDC 205, no general interconnection is present between elementary functional unit. Chaining and register allocation are very much constrained as each vector processing unit is pipelined: delay lines help compute $3 \times 3$ morphological convolutions, including a transparent and accurate management of image boundaries which are out of the stencil. Thus the size of the output image is equal to the input image size, contrary to repeated stencil computations [11] which usually reduce the image size. This is

another reason why low-level loop transformation-based approaches are likely to fail. Micro-instruction scheduling and compaction is easy once the order of operations is determined.

To sum up, the useful hardware constraints are 1) the structure of the micro-instruction set and the structure of the vector unit data paths, 2) the maximal number of chained microinstructions, i.e. the number of vector units, and 3) the number of image paths, two. Furthermore, the operations must be as packed as possible to reduce the number of micro-instructions. With 8 vector units, up to 40 full image operations can be performed for two loads and two stores, which leads to 10 SPoC operations per memory access, including high-level morphological convolutions which require more than 20 elementary operations each, and not counting the many reductions. So between 50 and 100 elementary operations can be executed per memory access.

## 4 Phase 1 – Application Preprocessing

The FREIA API [2] and its Fulguro [8] implementation are designed to be general with respect to the connectivity, the image sizes and the pixel representation. Standard or advanced loop transformations cannot take advantage of such source code because the loops are distributed into different functions and because elementary array accesses are hidden into function calls to preserve the abstraction over the pixel structure.

To build large basic blocks of elementary image operations, control flow breaks such as procedure call sites, local declarations, branches and loops must be removed by using key parameters such as connectivity and image size set up by the `main` and propagated to callees such as the image dilatation. Several source-to-source transformations help achieve this goal: 1) inlining to suppress functional boundaries, 2) partial evaluation to reduce the control complexity and 3) constant propagation to allow full loop unrolling, 4) dead code elimination to remove useless control, 5) declaration flattening to suppress basic block breaks. Safety tests are automatically eliminated as the application is assumed correct before its optimization is started. The order of application of these five transformations is chosen to maximize the available information so as to simplify the code and obtain larger basic blocks. Figure 7 shows the resulting code after automatic application of these transformations on the `main` function in Figure 4. It contains a sequence of elementary image operators mixed with scalar operations and temporary image allocations and deallocations.

## 5 Phase 2 – DAG Optimization

The basic blocks of the image application are analyzed to build an expression DAG as the one in Figure 8 (on next page), which is then optimized for locality. The vertices are the operations to perform, which may be image operations (*MORPH* as rectangles, *ALU* as trapezium, *THR* as parallelogram, *MES* as

```
// perform some computations
freia_aipo_global_min(in, &min);
freia_aipo_global_vol(in, &vol);
freia_aipo_dilate_8c(od, in, k8c);
freia_aipo_dilate_8c(od, od, k8c);
// previous line repeated 10 times...
I_0 = 0;
tmp = freia_common_create_data(...);
freia_aipo_dilate_8c(tmp, in, k8c);
freia_aipo_dilate_8c(tmp, tmp, k8c);
// previous line repeated 10 times...
freia_aipo_erode_8c(og, in, k8c);
freia_aipo_erode_8c(og, og, k8c);
// previous line repeated 10 times...
freia_aipo_sub(og, tmp, og);
freia_common_destruct_data(tmp);
```

**Fig. 7.** Excerpt of the `main` of Figure 4 after preprocessing

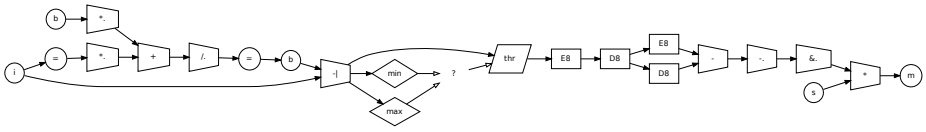



**Fig. 8.** DAG example from *Video Survey* – the left-end updates background $b$, the remainder detects movements in hot regions

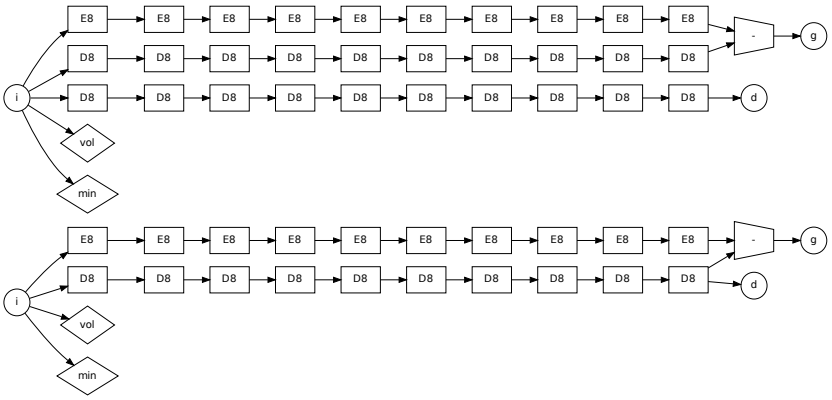



**Fig. 9.** Initial and optimized expression DAG for Figure 4

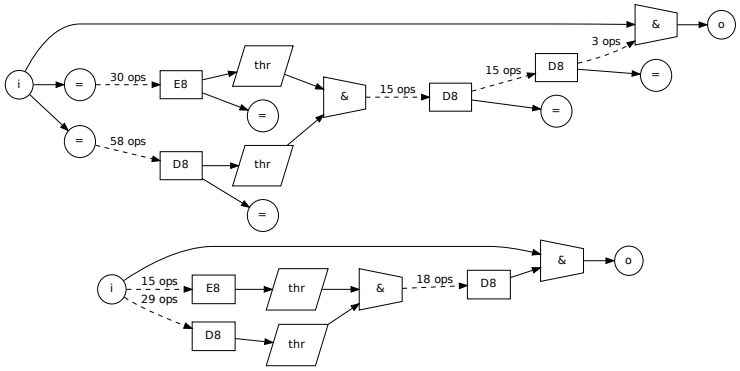



**Fig. 10.** Extract of initial and optimized DAG for *License Plate*

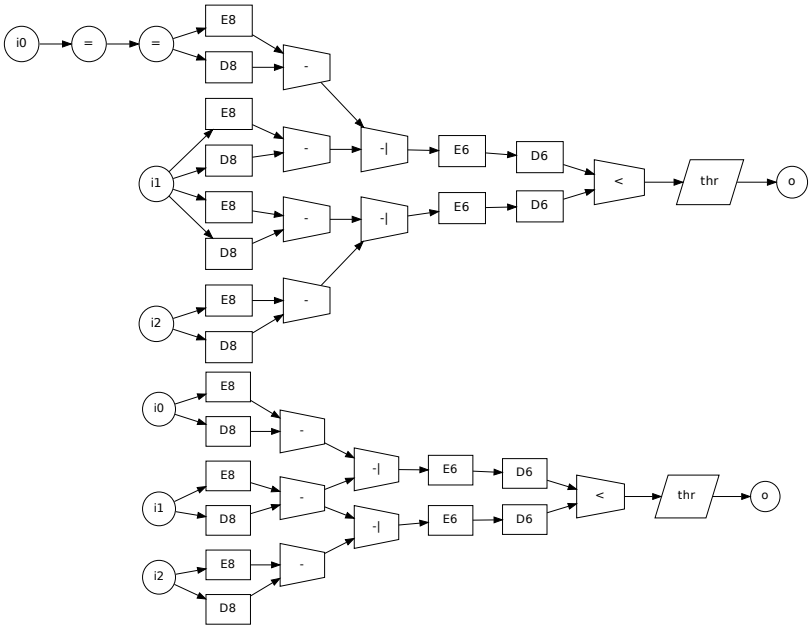



**Fig. 11.** Initial and optimized DAG from application *OOP*

diamond, copy and input/output images as circles) or intermediate scalar operations depicted as question marks. The arcs represent the dependencies between operations, when a piece of data defined at the source node end is used at the sink node. Arcs shown as black arrows embed image dependencies, and white arrows represent scalar dependencies. For instance the result of reductions on an image is used after some computation for thresholding it.

The DAG derived from our running example is shown in the upper part of Figure 9. The third row of dilatations is the call to the `freia_dilate` function with connectivity 8 and size 10. This DAG is then optimized in a target independent manner, with standard compilation techniques: common sub-expressions elimination and copy propagation are applied at the image level. Operator commutativity is taken into account to perform CSE, thanks to the image operator semantics which is available by recognizing the calls. Simple information about parameters, image or scalar, input and/or output, are derived automatically from C source stubs. Image copies are propagated forward toward their uses, with the exception of copies on output images which are propagated backward to their producer so that they are directly generated instead of using a temporary image. Remaining input and output image copies are extracted from the DAG to be performed outside of the accelerator. In our running example, the optimization detects that all operations of the *dilatation* are also performed within the *gradient*, so they are removed in the lower part of Figure 9, and the intermediate result is simply extracted.

Other challenges are found in the DAG for the *license plate* application in Figure 10, where repeated operations are denoted as dashed arrows. In this debug version of the code, every two operations are image copies either inserted within the computations or diverted to extract intermediate images. All these useless copies are removed from the optimized DAG. Eliminating such diversions is important for our target vector accelerator because extractions break the computation pipeline, thus inducing more accelerator calls. The DAG in Figure 11 is extracted from the *OOP* application. The optimized version removed both copies and a common subexpression involving 3 operations. These redundancies are not obvious to spot in the source code. The optimization of the DAG in Figure 8 removes both copy operations on input and within the graph.

The result of this phase is an optimized image expression DAG ready to be mapped onto the available hardware.

## 6 Phase 3 – SPoC Hardware Configuration

Finally, an accelerator specific compilation phase generates the hardware configuration, i.e. the micro-instructions for evaluating the optimized DAG resulting from the previous phase. We have two code generators; one for SPoC described hereafter, and one under development for the Terapix SIMD accelerator.

### Problem Description

The SPoC hardware accelerator [9] constraints discussed in Section 3 must be met: The computations in the hardware accelerator must only involve two live images at any single point because only two data paths are available (Figure 6). Actual computations must be scheduled on components so that live images can still reach their use or the end of the path. If all available vector units in the SPoC pipeline are used for a computation and more operations remain, the pipeline spilling must be managed. The optimality criterion is to minimize the number of calls to the accelerator, taking into account its actual number of vector units, as one call lasts about the same time whatever the operations performed in the pipeline.

The problem of mapping an image expression DAG onto the SPoC accelerator is very close to the pebble game problems used in register allocation, with in our case only two registers. However, unlike register allocation problems, our spill code is to interrupt our computation pipeline, resulting in both registers to be spilled at the same cost as one of they occur simultaneously. So although mapping scalar expression DAG onto a register machine [6,3] is NP-complete, these results do not apply directly to our case.

We conjecture nevertheless that our problem is NP-complete, because of the close similarity with the code generation problem for register-machines. First, the setting is highly combinatorial if one enumerates all possible evaluation orders compatible with the dependencies when there is a high degree of parallelism available in the DAG. Second, evaluating the cost of a proposed solution is reasonably easy: given an order of operations, one can detect in one pass over the vertices when an infinite pipeline should be cut because an operation would create more than two live images; if the finite number of vector units is considered, instruction compaction can tell when the pipeline is full.

### Code Generation

Given the combinatorial nature of the problem, our heuristic consists in breaking down the problem into three successive stages. Each stage satisfies one of the constraints independently, and there is no guarantee of global optimality. First, we meet the two live image constraint with a decomposition of the expression DAG into sub-DAGs, where each resulting sub-DAG operations are ordered by the decomposition process so that their evaluation in *that* order only requires two live images. Then, instructions are compacted in a conceptually infinite pipeline, which is finally cut according to the number of available vector units. We chose to avoid a global combinatorial optimization because this simple heuristic, which satisfies each constraint one after the other, leads to excellent experimental results (Section 7).

The optimized expression DAG is first split into sub-DAGS with no more than two live images and no internal scalar-carried dependencies. As noted above, this is very similar to evaluating an expression with only two registers. We use the simple *list scheduling of basic blocks* technique described in the *Dragon book*, with

a prioritized topological order which focuses on the critical resource, namely the small number of data paths. Scalar dependencies, cannot be handled within one hardware accelerator call as images are processed concurrently, so the needed result would not be available at the start of the dependent computation: they must be split across distinct sub-DAGs. The greedy list scheduling heuristic expands a subgraph as much as possible, and never backtracks. The priority choices favor the immediate use of computed images in the pipeline: reductions that do not update their source are performed first, then operations that use up an image and define another one, ordered by the number of uses, then other operations. The result of the first pass is a list of DAGs, each with an *ordered* list of operations that require no more than two live images if processed in that particular order along the pipe.

Each sub-DAG is then mapped onto a pipeline with a conceptually infinite number of vector units by compacting operations into microinstructions. We do not allow much freedom at this stage because the order of operations cannot be modified without putting at risk the two live image constraint. It is kept unchanged. Microinstruction compaction is performed at the same time because the packing constraints are very easy to meet: structural, control and data pipeline hazards are avoided by the hardware, hence sophisticated microinstruction scheduling and compaction are not required. The compaction is achieved by scheduling operations in the first available slot. When only one image is needed by the pipeline, it is sent on both input paths so as to help the compaction at the beginning of the pipe. Path selection implies the multiplexer configuration. It must ensure that computed images reach the operators that process them, which may shift a computation further down in the pipeline in some cases. Under these assumptions, this compaction stage could be proven optimal, that is the number of units used is minimal, by induction on the structure of the pipeline, as we choose the first available operator at each iteration. However this optimality is weak because it requires that there is no reordering of the operations, which could improve the result if allowed. Moreover this optimality is local, and taking this constraint in the previous stage could help improve the overall solution.

The third stage of the code generation process is to map the open-ended pipeline onto the available vector units. This is simply achieved by cutting the micro-instructions sequence at the number of available vector units, and to perform another activation of the SPoC pipeline for the remainder, until all sub-DAG operations are performed. This stage of the process is trivially optimal if the compaction is optimal.

This heuristic phase for the SPoC accelerator reuses standard compilation techniques to generate most of the time optimal results. It is followed by a quick cleanup of intermediate images which are not used anymore by the function. The techniques are applied on very long vector flows of pixels from images, whereas they were originally designed for scalars in registers. This works well because the SPoC architecture takes care of pipeline hazards and performs stencil computations without reducing the image size: images are equivalent to scalar variables.

## 7 Discussion, implementation and experiments

There is a cost performance tradeoff in choosing the number of vector units, as longer pipeline are less efficiently used when no operations can be scheduled and add to the overall latency of accelerator calls. The solution to this tradeoff depends on the actual applications and on the user ability to select optimal hardware. It is not taken into account here as we assume that the number of vector units is a given, with 8 a typical figure.

**Phase 1** application preprocessing – enlarge basic blocs
1. inlining of FREIA library functions
2. partial evaluation
3. constant propagation and loop full unrolling
4. dead code elimination
5. block flattening

**Phase 2** DAG optimization
1. DAG construction per sequence
2. common sub-expression elimination, with commutativity
3. forward or backward copy propagation
4. extraction of remaining copies
5. dead image operation removal

**Phase 3** SPoC configuration: map DAG onto hardware
1. DAG splitting and scheduling of sub-DAGs
2. instruction compaction and path selection
3. pipeline overflow management
4. unused image cleanup

**Fig. 12.** Outline of our compilation strategy: phases and stages

Our optimization strategy is implemented in PIPS [10], for a small development cost measured hereafter with the KLOCs *(line of codes)* involved. Figure 12 summarizes the different phases presented in detail in the previous sections. Transformations of Phase 1 are standard in an advanced optimizing compiler. Phase 2 operations are also standard, but are used here for full image processing calls although the usual scope is on elementary scalar processor operations. Its implementation uses about 2 KLOCs for representing the FREIA elementary operator semantics, plus building and optimizing the DAG representation. Phase 3 is the back-end specific code generation. It uses about 1.6 KLOCs including DAG splitting, scheduling, wiring, and SPoC configuration. It produces acceleration functions to be called from the initial application. Each generated pipeline configuration function (see excerpt in Figure 13) is called from the `main` with the appropriate arguments (in Figure 14). Other applications may require more preprocessing phases, such as while loop unrolling or code hoisting, to obtain longer basic blocks.

```
void helper_0(f_data2d *o0, f_data2d *o1,
  f_data2d *i0, int32_t *red0, int32_t *red1,
  int32_t * kern2 /* ... up to kern16 */)
{
  // SKIPPED: declarations & initializations
  // - si & op: micro instructions
  // - sp & par: operation parameters
  // - redres & reduc: reduction results

  // set state of MUX stage 0 number 0
  si.mux[0][0].op = SPOC_MUX_IN0;
  // set state of POC stage 1 side 0
  si.poc[1][0].op = SPOC_POC_DILATE;
  si.poc[1][0].grid = SPOC_POC_8_CONNEX;
  // and its kernel
  for(i=0 ; i<9 ; i++)
    sp.poc[1][0].kernel[i] = kern2[i];
  // SKIPPED: more configurations...

  // actual call to the hardware accelerator
  // instructions, params, 2 images out, 2 images in
  f_cg_process_2i_2o(op, par, o0, o1, i0, i0);
  // extract reductions results
  f_cg_read_reduction_results(&redres);
  *red0 = (int32_t)reduc.measure[0][0].minimum;
  *red1 = (int32_t)reduc.measure[0][0].volume;
}
```

**Fig. 13.** One stub source code (excerpt) for Figure 4

```
// perform some computations
helper_0(od, og, in, &min, &vol, k8c, /* 18 more k8c args */);
helper_1(od, og, od, og, k8c, k8c, k8c, k8c, k8c);
```

**Fig. 14.** Code in Figure 4 is reduced to two stub calls

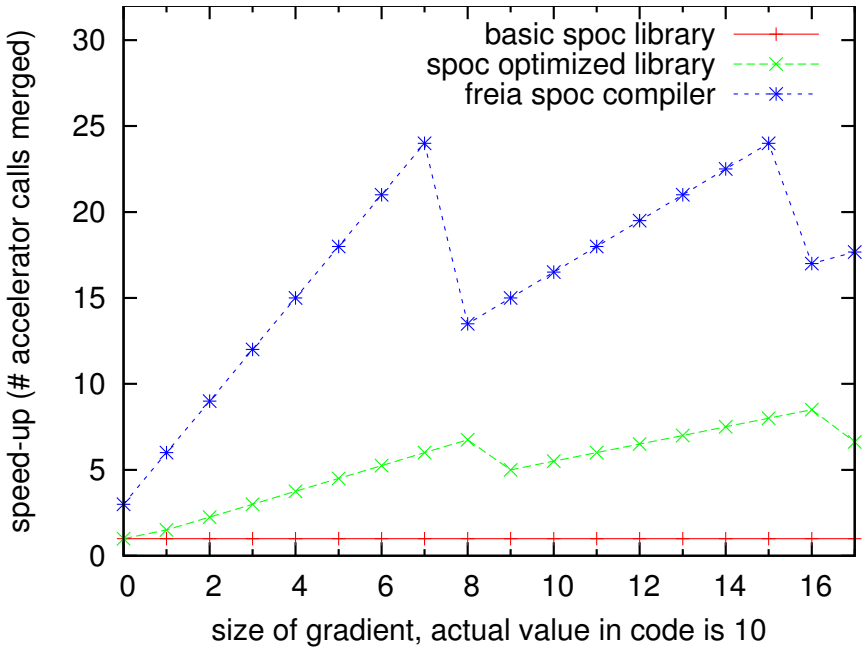



**Fig. 15.** Speed-ups on SPoC with 8 vector units for Figure 4

Figure 15 compares speed-ups obtained on versions the running example Figure 2 with differing numbers of iterations (the source code selected executes 10 iterations). In the baseline version, each call to the accelerator performs only one operation. In the optimized library version, each call to a FREIA operator is optimized independently. Finally the PIPS version is generated with the techniques described in this paper to optimized the whole application. It fares better than any other versions, thanks to the extracted common sub-expression and the optimal (in this case) hardware mapping which combines elementary operations whenever possible. Note that the optimized version runs out of vector units for a size 8 gradient operation, whereas the version using the SPoC optimized library goes down for 9: the first vector unit is used up by the optimized version for the volume and minimum measurements in the input image, hence the shift of the discontinuity between the two versions. Our compiler was tested on 1217 cases, comprising 1005 combinatorial tests (3 to 6 ops), 105 elementary tests (1 to 13 ops), 40 atomic tests (1 op for which we generate the hardware accelerated version) and finally 57 significant applications or functional blocks (5 to 135 ops) tested with various parameters. Only 15 results are not optimal for the target SPoC accelerator: 14 are one call from optimality, and one is non optimal by 3 calls. Most of these non optimality cases are linked to the greedy nature of the heuristic coupled with pipeline spilling effects.

## 8 Related Work

The related work is rather limited because people developing hardware accelerators in an academic environment usually do not have the resources required to develop a full programming tool chain. They either design a specialized language, or use pragmas to guide the optimizer, or build an optimized library, which may grow with each new application to include the application-specific API that leads to good performance, or they develop applications with target-knowledgeable people and do not advertise it. We break the related work in two parts, software development for accelerators and optimization of expression evaluation.

**Software development for accelerators**

Specialized languages have been designed to address various needs of application domains and target architectures which are not well served by general purpose languages. The OpenCL recent standard aims at providing portability across accelerators, especially GPGPU targets, but it is quite large and pretty low-level: application developers should be able to ignore it. It remains an interesting output language for a source-to-source tool like PIPS, or for implementing an efficient runtime to be called by the generated code. Array-OL [13] is an example of a domain-specific language designed for signal processing and for accelerator programming. On the one hand, Array-OL is not general enough to write a whole application, and on the other hand it is still hard to compile efficiently for a given target: parts of the application must be isolated and coded in Array-OL, and the Array-OL optimization process be performed under human supervision using a graphical tool.

Pragma annotations on top of a standard language are used to preserve the portability of applications and allow their functional validation in a standard environment. OpenMP allows the developer to hint about the program semantics, say loop parallelism or critical sections, but does not yet address all the requirements of hardware accelerators, especially when the hardware accelerator must be programmed. HMPP [7] is another pragma set designed by CAPS Enterprise to provide higher level pragmas. It can be used to program an accelerator such as Nvidia Tesla or AMD FireStream, including the use of several accelerators linked to a unique host, issue which is not addressed by our technique. However the set of directives is very specific and requires deep architectural insight from the developer to be exploited fully.

Another way of achieving high performance on specialized hardware and still retain portability is to use domain-specific libraries which can be implemented for various targets. VSIPL [1] in the signal processing field was developed as an open standard by an industry, government and research consortium. It contains thousands of functions, and various level of partial implementations are defined in the standard, starting from the 127 functions core lite profile, followed by the 513 functions core profile, but implementations do not necessarily implement these profiles in full. As the functions are not independent and orthogonal, the

developer must choose an implementation strategy which may result in different performances with differing library implementations, and may impair the portability when all functions are not available. Moreover, we observed in the Ter@ops project that an API has a direct impact on the application structure, which may not lead to good performance on a new piece of hardware. A library has been designed for vector-based instruction set additions such as Altivec or Intel SSE extension family. To optimize its functions, application-level loops had to be moved down into the library to improve data re-use. When the application was ported on a new MPSoC, without any vector operation support but with multiple processors, loops were moved back up across functional boundaries [15] to re-optimize the application differently. When performance is a concern, a fixed API cannot really remain target independent. Although our approach relies on an API, it is used to provide the underlying application semantics, and the generated code does not have to respect the API; the compiler restructures the computations to fit the target hardware.

Application-specific instructions can be added to an existing general-purpose instruction set. For instance, the Video Specific Instruction Set Processor [18] has special instructions for computational intensive parts such as inter-block prediction but also uses co-processors for specific tasks such as entropy encoding. This is close to our case, although these instructions are very algorithm-specific, while we have generic elementary operators.

**Optimization of expression evaluation**

We use commutativity to detect more common subexpressions, but we do not currently attempt to use advanced algebraic properties [21], mainly because none of our test cases would benefit from these complex combinatorial optimizations. However we would consider using them if we had a motivating example that would be really improved by such optimizations. Basic block enlargement is useful for trace scheduling [12] and obtained by different code transformations, including code hoisting and code sinking [14]. For image processing applications, code hoisting and sinking do not seem useful. Our technique is close to the optimization of expression evaluation and vector instruction chaining [19], although in our case we must preliminary meet the pipeline constraints of our target hardware.

## 9 Conclusion and Future Work

We have shown how standard compilation techniques can be efficiently reused and adapted to optimize applications based on an image processing library for a domain-specific hardware accelerator composed of multiple chained vector units. Applications can be developed in C by any programmer competent in the image processing field, but without knowledge of the hardware accelerator, and are automatically optimized for the specific target system without any of the traditional hurdles such as the procedure calls imposed by the different APIs

used. The source code transformations and the high-level optimization strategy is simple, it properly combines and adapts existing techniques to perform a wide range of loop fusions based on semantical information. This simplicity is an asset, as it greatly reduces the development costs of the compiler and bring large speedups.

Some experimental results are even better than expected. The PIPS automatically optimized version of the running example beats the hardware expert first-cut hand-optimized version, because common sub-expression elimination opportunities were not considered. It is up to three times faster than the version based on the hand optimized FREIA library implementation, and it is optimal, as most of our 1217 test cases. The compiler also generates, as a side effect, the basic hardware accelerated library version by considering elementary operations as a whole application. The full library hardware accelerated version is more subtle, as it dynamically adapts the generated configuration to the parametric number of iterated operations (see the dilatation) and the available hardware pipeline depth. These are known to the compiler when considering a full applications in context, but not when simply looking at a function library implementation.

What are the underlying reason of our success? Firstly, the application domain uses one large type of data, images, and a limited set of operators executed on whole images, with a lot of implicit locality and parallelism. Secondly, the architectural choices of the hardware with the high level instruction set provided by SPoC [9] takes advantage of these opportunities to provide a potentially high performance pipeline, which, although not as convenient as a crossbar which would enable any operator chaining, fits the kind of DAG found in applications, and is accessible through runtime calls which handle low-level details but enable all necessary configurations. Thirdly, the library API is reasonably small (about 40 basic operations and about 20 higher level combined operations), and is both relevant to the application developers who can find high level operations and develop functional blocks, and still easily mapped onto the hardware which implements directly most of the elementary operations. Thus the gap is small enough to be compatible with a simple compilation strategy, allowing a low cost fast development and integration in an existing source-to-source compiler.

This does not preclude the implementation of the same approach on more traditional SIMD hardware accelerators or GPGPU targets, because the high level API provides all the semantical information needed to generate code and perform many classical compiler optimizations. However it may need to be combined with more traditional loop transformation techniques to produce optimized combined operation microcode for these targets, or to develop a specific runtime which takes advantage of the available hardware once high-level optimizations and choices are performed. Such work is already underway. A second direction is to test our approach on more real-life applications. We also have to look at the impact on our strategy on domains with multiple data types, such as signal processing applications. A third direction is to reuse the semantical loop fusion and emulate its the schedule of our target to benefit from the locality increase

for general-purpose processors. The technique used by our accelerator to handle image boundaries by maintaining a constant image size could also be useful.

**Acknowledment**

We are indebted to Christophe Clienti who designed the SPoC accelerator, its simulator, the Fulguro library and the FREIA interface, and who ported existing applications on the FREIA interface. We also thank Serge Guelton, designer, implementer and maintainer of the source-to-source inlining phase in PIPS, Laurent Daverio who contributed the code flattening transformation, Michel Bilodeau who provided application codes. Finally, we want to thank Alain Darte for a long discussion about NPC problems.

## References

1. VSIPL Std v1.3, January 2008. Vector Signal Image Processing Library.
2. FREIA: FRamework for Embedded Image Applications, 2008–2011. French ANR-funded project.
3. Alfred V. Aho, Stephen C. Johnson, and Jeffrey D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.
4. Mehdi Amini, Corinne Ancourt, Fabien Coelho, François Irigoin, Pierre Jouvelot, Ronan Keryell, Pierre Villalon, Béatrice Creusillet, and Serge Guelton. PIPS Is not (just) Polyhedral Software. In *International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Chamonix, France, April 2011.
5. Philippe Bonnot, Fabrice Lemonnier, Gilbert Edelin, Gérard Gaillat, Olivier Ruch, and Pascal Gauget. Definition and SIMD Implementation of a Multi-Processing Architecture Approach on FPGA. In *Design Automation and Test in Europe*, pages 610–615. IEEE, December 2008.
6. John L. Bruno and Ravi Sethi. Code generation for a one-register machine. *J. ACM*, 23(3):502–510, 1976.
7. CAPS enterprise. HMPP, Manycore Portable Programming, 2008.
8. Christophe Clienti. Fulguro image processing library. Source Forge, 2008.
9. Christophe Clienti, Serge Beucher, and Michel Bilodeau. A system on chip dedicated to pipeline neighborhood processing for mathematical morphology. In *EUSIPCO: European Signal Processing Conference*, August 2008.
10. CRI, MINES ParisTech. PIPS, 1989–2011. Open source, under GPLv3.
11. Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *SC'08: Conference on Supercomputing*, pages 1–12. IEEE Press, 2008.
12. J.A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, 1981.
13. Calin Glitia, Philippe Dumont, and Pierre Boulet. Array-ol with delays, a domain specific specification language for multidimensional intensive signal processing. *Multidimensional Systems and Signal Processing*, 23(3), 2009.
14. Rajiv Gupta. A code motion framework for global instruction scheduling. In *International Conference on Compiler Construction, LNCS 1383*, pages 219–233. Springer Verlag, 1998.

15. Mary H. Hall, , Saman P. Amarasinghe, Brian R. Murphy, Shih-Wei Liao, and Monica S. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Conference on High Performance Networking and Computing, Proceedings of the 1995 Conference on Supercomputing*. ACM, December 1995.
16. Roger W. Hockney and Chris R. Jesshope. *Parallel Computers 2: architecture, programming, and algorithms*. Adam Hilger, IOP Publishing Ltd, 1988.
17. François Irigoin, Pierre Jouvelot, and Rémi Triolet. Semantical interprocedural parallelization: An overview of the pips project. In *1991 International Conference on Supercomputing, Cologne, June 1991*, 1991.
18. S.D. Kim, C.J. Hyun, and M.H. Sunwoo. VSIP: Implementation of Video Specific Instruction-set Processor. In *APCCS: Asia Pacific Conference on Circuits and Systems*, pages 1075–1078, Singapore, December 2006. IEEE.
19. T. Rauber. Optimal evaluation of vector expression trees. In *JCIT: Proceedings of the fifth Jerusalem conference on Information technology*, pages 467–473, 1990.
20. Pierre Soile. *Morphological Image Analysis: Principles and Applications*. Springer-Verlag, 2003.
21. Julien Zory and Fabien Coelho. Using algebraic transformations to optimize expression evaluation in scientific codes. In *PACT: Parallel Architectures and Compilation Techniques*, pages 376–384, Paris, December 1998. IEEE.

# Analyse statique par découverte de chemin

Julien Henry

CNRS - Verimag
Julien.Henry@imag.fr

## 1 Introduction

L'analyse statique consiste à trouver à la compilation des propriétés sur un programme, comme l'ensemble des valeurs possibles pour chacunes des variables durant l'exécution. Ceci permet par exemple de prouver que le programme ne va pas avoir de débordements arithmétiques, à calculer des invariants de boucle, à faire des optimisations de code lors de la compilation, etc.

L'Interprétation Abstraite est une méthode classique d'analyse statique, qui consiste à calculer une surapproximation de l'ensemble des états possibles d'un programme numérique. L'état d'un programme est caractérisé par la valeur de chacunes de ses variables numériques, et par la position à laquelle on se trouve dans le code. Un cas particulier d'Interprétation Abstraite est l'analyse par relations linéaires : on surapproxime l'ensemble des états possibles du programme à un certain point par un polyhèdre convexe, pour découvrir des propriétés linéaires entre les différentes variables numériques du programme. Cette méthode implique des surapproximations, nécessaires pour rendre l'analyse moins coûteuse : il faut alors essayer de trouver le bon compromis entre la précision et le coût. De nombreux travaux consistent à trouver de nouvelles techniques pour améliorer cette précision à moindre coût. La méthode proposée ici consiste à s'aider des progrès dans la recherche sur le SMT-solving pour raffiner la méthode d'Interprétation Abstraite.

## 2 Interprétation abstraite

L'application de l'Interprétation Abstraite à l'analyse de relations linéaires consiste à attacher à chaque point de contrôle du programme un polyhèdre convexe dont les dimensions sont les variables numériques. On travaille sur le graphe de flot de contrôle du programme (Fig. 1) : le programme peut être vu comme un graphe où chaque transition est étiquetée par une ou plusieurs instructions, et peut être soumise à condition(s). On calcule itérativement les différents polyhèdres $A_i$ à chaque noeud $s_i$ du graphe de la façon suivante :

Au départ, le polyhèdre de l'état initial est l'espace entier (aucune contrainte sur les variables), et celui des autres états est le polyhèdre vide (l'état est considéré comme étant inaccessible).

Puis, on itère ce calcul pour tous les états du graphe, tant que la suite des polyhèdres $A_i$ est croissante pour la relation d'inclusion :

- On part d'un état $i$, et on met à jour chaque état $j$ où $j$ est un successeur de $i$ dans le graphe, en calculant l'image $Y$ du polyhèdre $A_i$ par la transition $i \to j$.
- Le nouveau polyhèdre $A_j$ est l'enveloppe convexe de $Y$ avec l'ancienne valeur de $A_j$, notée $Y \sqcup A_j$, qui est le plus petit polyhèdre convexe incluant $A_j$ et $Y$. Les valeurs successives de $A_j$ au cours de l'analyse définissent donc bien une suite croissante.

Après un certain nombre d'itérations, chaque polyhèdre a atteint une limite, et l'analyse s'arrête [1]. Les polyhèdres résultats permettent de déduire des invariants sur les variables du programme.

Dans l'exemple de la figure 1, en supposant que $n > 0$, l'analyse donne :

- $A_1$ est le polyhèdre $\{n > 0\}$ : au point $s_1$, on sait seulement que $n > 0$.
- $A_2$ est le polyhèdre $\{0 \leq x \leq 2n,\ 0 \leq i \leq n\}$.
- $A_3$ est le polyhèdre $\{0 \leq x < 2n,\ 0 \leq i < n\}$.
- $A_4$ est le polyhèdre $\{i = n,\ x = 2n\}$. On connaît donc directement la valeur des variables à la fin du programme, ce qui permet des optimisations de compilation.

[1] En fait, pour garantir la terminaison, on a recours à un opérateur dit d'élargissement, qui fait une extrapolation lorsqu'un polyhèdre risque de croître indéfiniment.

```
// position s1
x = 0;
i = 0;
//position s2
while (i < n) {
        // position s3
        i++;
        x = x+2;
}
// position s4
```

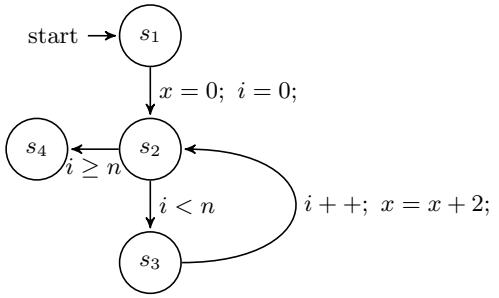



**Fig. 1.** Exemple de programme, et son graphe de flot de contrôle associé

## 3 Découverte de chemin

Nous proposons un raffinement de la méthode d'analyse par Interprétation Abstraite, en guidant plus astucieusement les itérations de point fixe. En effet, la vitesse et la précision de l'analyse dépendent de l'ordre dans lequel on traite les noeuds du graphe.

Au lieu de travailler sur le graphe de flot de contrôle et de calculer un polyhèdre à chacun des noeuds, on choisit un sous-ensemble $P$ des noeuds et on fait le calcul de point fixe sur le graphe ne contenant que les noeuds de $P$, et où une transition correspond à un chemin possible dans le graphe de départ.


**Fig. 2.** En noir, les noeuds de $P$. À gauche, un graphe de flot de contrôle, et à droite son équivalent avec uniquement les noeuds de $P$. L'un des noeuds contient deux boucles, chacune correspondant à un chemin possible dans le graphe de gauche.

Ce nouveau graphe contient moins d'états, mais un nombre de transitions potentiellement exponentiellement plus grand que dans le graphe d'origine. C'est pourquoi la technique consiste à ne pas construire explicitement ce graphe, et à procéder comme suit :

Lors de l'analyse d'un noeud $i \in P$ du graphe, on encode la question suivante sous forme de formule logique : "Existe-t-il un chemin entre $i$ et $j \in P$, tel que les variables numériques sont dans $A_i$ au noeud $i$, et tel que les variables après le passage par ce chemin ne sont pas dans $A_j$ ?", et on la résout avec un SMT-solver. Si la réponse est "oui", celà veut dire qu'on a trouvé un chemin qui fait avancer l'analyse, et on peut alors calculer la transformation associée à ce chemin, et mettre à jour $A_j$. L'analyse d'un noeud $i$ s'arrête lorsque plus aucun chemin n'est trouvé.

## 4 Conclusion

Nous proposons une méthode pour améliorer la précision de l'analyse par Interprétation Abstraite, en guidant l'analyse vers les chemins du graphe à explorer. Le fait de considérer des chemins complets et non pas de simples transitions permet de gagner en précision, et l'utilisation du SMT-solving permet de limiter le coût des calculs et la consommation mémoire.

# Session du groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

# Vues de sécurité pour documents XML

Iovka Boneva[12], Benoît Groz[123], Sławomir Staworko[14],
Anne-Cecile Caron[12], Yves Roos[12], and Sophie Tison[12]

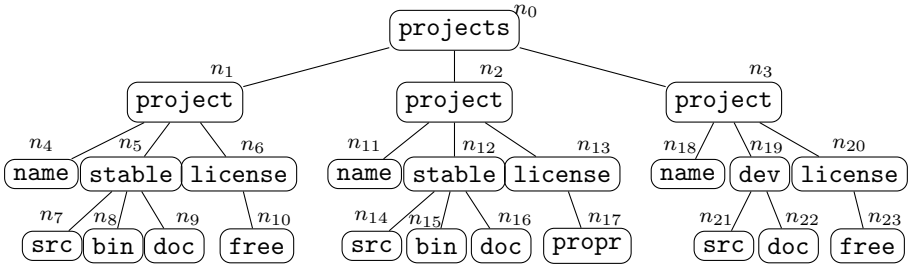
[1] Projet Mostrare, INRIA Lille Nord-Europe & LIFL (CNRS UMR8022)
[2] Université Lille 1
[3] ENS Cachan
[4] Université Lille 3

La spécification de politiques de sécurité ainsi que leur application sont bien connues dans les bases de données relationnelles. Les vues de sécurité définies à l'aide d'une requête SQL sont une manière de préserver la confidentialité des données. Lorsqu'on définit une vue de sécurité, on doit traduire les requêtes de l'utilisateur (sélections ou mises à jour) sur cette vue non matérialisée en requêtes sur la donnée source. Notons que le problème est plus complexe quand il s'agit de mises à jour, car la traduction peut entraîner des modifications des données cachées, ou rendre visible/invisibles des données qui ne l'étaient pas. Le fait d'utiliser des vues dans un objectif de sécurité pour des documents XML a été étudié à la fois d'un point de vue pratique et théorique par de nombreux auteurs [9,1,6,2,10,3,8]. Comme dans le modèle relationnel, une vue sur un document XML peut-être définie à l'aide d'une requête. Une vue définie par une requête $Q$ sur un arbre XML $t$ est un arbre XML $V(Q,t)$ contenant tous les noeuds de $t$ sélectionnés par la requête $Q$. Dans [5], nous utilisons le langage Regular XPath [7] pour définir une vue de sécurité. Fan et al. [1,2,3] définissent une vue grâce à une DTD annotée, comme dans l'exemple ci-dessous, mais nous montrons que si $A$ est une DTD annotée par des requêtes Regular XPath, $A$ peut toujours être traduite en une seule requête Regular XPath $Q_A$ qui sélectionne les noeuds que l'on souhaite visibles.

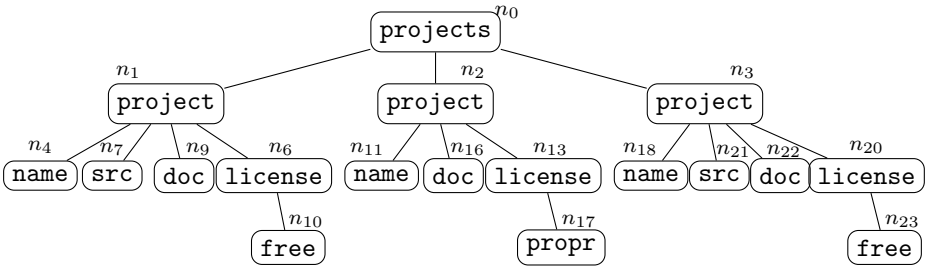
*Example 1.* Voici une DTD $D_0$, qui modélise des projets de développement, ainsi que leurs fichiers. Ces projets peuvent être libres ou propriétaires, stables ou en cours de développement ; les fichiers peuvent être de la documentation, des sources et des binaires. Les éléments qui n'apparaissent pas en partie gauche des règles de la DTD sont soit des éléments vides (`free` et `propr`) soit des chaînes (`PCDATA`) décrivant des url d'accès aux fichiers (`doc`, `src`, `bin`). Cette DTD est annotée par une fonction $X_0$ qui à chaque couple d'éléments $(e_1, e_2)$ indique la visibilité de l'élément $e_2$ lorsqu'il est fils de $e_1$. La visibilité peut être `true`, `false`, ou conditionnée par un prédicat `[condition]` qui signifie que le noeud est visible ssi la condition est satisfaite. Lorsque $X_0(e_1, e_2)$ n'est pas spécifié, alors $e_2$ hérite la visibilité de $e_1$. Cette DTD annotée sera désignée par la suite par $A_0 = (D_0, X_0)$.

`projects` → `project`$^*$
`project` → `name`, (`stable` | `dev`), `license`
  $X_0$(`project`, `stable`)=false
  $X_0$(`project`, `dev`)=false
`license` → `free` | `propr`
`stable` → `src`, `bin`, `doc`
  $X_0$(`stable`, `src`)=[⇑$^*$::`project`/⇓$^*$::`free`]
  $X_0$(`stable`, `doc`)=true
`dev` → `src`, `doc`
  $X_0$(`dev`, `src`)=[⇑$^*$::`project`/⇓$^*$::`free`]
  $X_0$(`dev`, `doc`)=true

La fonction $X_0$ donne accès à tous les projets, en cachant l'information `stable` ou `dev`, les binaires des projets, et les sources des projets propriétaires. Voici un arbre $t_0$ conforme à la DTD $D_0$ :



et voici la vue non matérialisée $V(Q_{A_0}, t_0)$ :




Nous montrons qu'une requête Regular XPath sur une vue peut toujours être traduite en une requête Regular XPath sur l'arbre XML source. Dans ce travail, la politique de sécurité peut cacher certains noeuds sans cacher leurs fils ; dans ce cas, la vue est construite en rattachant chaque noeud visible à son premier ancêtre visible, en sachant que la racine du document est toujours visible (ainsi $V(Q, t)$ est un arbre). Remarquons que, dans ce contexte, le langage de vues $V(Q, L)$ d'un langage d'arbres $L$ validé par une DTD peut ne pas être reconnaissable.

Dans [4], nous étudions la traduction des mises à jour faites sur une vue, en mises à jour sur le document XML source. Cette fois, nous considérons une classe de vues qui permet d'enlever des sous-arbres d'un document ou de renommer des noeuds. Un programme de mise à jour consiste en une collection de mises à jour atomiques qui sont appliquées simultanément, conformément à la sémantique du langage XQuery Update Facility (XQUF)[11]. A la fois les vues et les programmes de mises à jour que nous manipulons sont des langages reconnaissables d'arbres, où les arbres considérés sont des scripts d'édition. Nous montrons qu'on peut résoudre le problème de la traduction de mises à jour sur une vue lorsque l'on ne donne aucune contrainte sur le document XML source. Par contre, ce problème ne peut en général pas être résolu si l'on ajoute des contraintes sur la source.


1. W. Fan, C.-Y. Chan, and M. N. Garofalakis. Secure XML querying with security views. pages 587–598, 2004.
2. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. SMOQE : A system for providing secure access to XML. pages 1227–1230. ACM, 2006.
3. W. Fan, F. Geerts, X. Jia, and A. Kementsietsidis. Rewriting regular XPath queries on XML views. pages 666–675, 2007.
4. Benoit Groz, Iovka Boneva, Yves Roos, Sophie Tison, Anne-Cécile Caron, and Slawomir Staworko. View update translation for XML. In *14th International Conference on Database Theory (ICDT)*, Uppsala, Sweden, March 2011.
5. Benoit Groz, Slawomir Staworko, Anne-Cécile Caron, Yves Roos, and Sophie Tison. XML Security Views Revisited. In Philippa Gardner and Floris Geerts, editors, *International Symposium on Database Programming Languages*, volume LNCS, pages pp 52–67, LYON, France, August 2009. Springer.
6. G. Kuper, F. Massacci, and N. Rassadko. Generalized XML security views. In *SACMAT '05 : Proceedings of the tenth ACM Symposium on Access Control Models and Technologies*, pages 77–84. ACM, 2005.
7. M. Marx. XPath with conditional axis relations. pages 477–494, 2004.
8. N. Rassadko. Query rewriting algorithm evaluation for XML security views. In *Secure Data Management (VLDB Workshop)*, volume 4721 of *Lecture Notes in Computer Science*, pages 64–80. Springer, 2007.
9. A. Stoica and C. Farkas. Secure XML views. In *IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security*, volume 256 of *Research Directions in Data and Applications Security*, pages 133–146. Kluwer, 2002.
10. R. Vercammen, J. Hidders, and J. Paredaens. Query translation for XPath-based security views. In *EDBT Workshops*, volume 4254 of *Lecture Notes in Computer Science*, pages 250–263. Springer, 2006.
11. W3C. XQuery update facility 1.0, 2011. `http://www.w3.org/TR/xqupdate/`.

# Systèmes de Réécriture Fonctionnels pour le *Model-Checking* Symbolique

Y. Boichut, J.-M. Couvreur, and D. Nguyen

LIFO, Université d'Orléans, France

## 1 Introduction

Dans les années 90, les industries des composants électroniques, dans leur recherche d'outils pour améliorer le niveau de confiance de leurs produits, ont adopté les diagrammes de décisions binaires (BDD) [2] pour traiter des composants de plus en plus complexes. Les BDD sont des structures codant des fonctions booléennes. Ils peuvent être vus comme des arbres où les états représentent des choix de valeurs de variables booléennes ; un ordre total sur les variables garantit l'unicité du codage d'une fonction. Les techniques de partage de structures, combinées à des méthodes de réductions, conduisent à des implémentations extrêmement efficaces en pratique [12,10]. Ainsi, des vérifications exhaustives ont pu être réalisées sur des systèmes comprenant des billions d'états [12,10]. Le pouvoir d'expression des BDD est suffisant pour manipuler une grande classe de systèmes finis [3]. De plus, certains systèmes dynamiques peuvent être pris en compte avec ce type de techniques [11]. Comme le nombre de variables des systèmes étudiés est un facteur critique, de nombreuses structures "à la BDD" ont été proposées [13,4].

Dans le cadre de projets industriels, nous avons conçu une nouvelle structure à la BDD, les *Diagrammes de Décisions de Données* (DDD) [6]. Notre objectif était de fournir un outil flexible qui peut être autant que possible adapté pour la vérification de tout type de modèles et qui offre des capacités de traitement similaires aux BDD. A la différence des BDD, les opérations sur nos structures ne sont pas prédéfinies, mais une classe d'opérateurs, appelée *homomorphismes*, est introduite pour permettre à un utilisateur de concevoir ses propres opérations. Ces travaux ont été depuis généralisés dans [7,14] par la définition d'un opérateur de point de fixe qui permet d'accéler significativement les calculs mais aussi par une version hiérarchique des DDD, les *Set Decision Diagrams* (SDD) et les *Hierarchical Set Decision Diagrams* (HSDD) qui permettent de coder façon plus concise des ensembles des états de systèmes.

Dans [1], nous avons étendu une contribution récente [8] sur les *Tree Data Decision Diagrams* (TDDDs). En résumé, TDDDs sont des termes (représentant des structures de données) sur lesquels la mise à jour est faite à l'aide de techniques de réécriture. En réalité, nous utilisons un type de système de réécriture particulier que nous avons nommé *Systèmes de réécriture fonctionnels* (SRF). Bien que ces systèmes de réécriture semblent restrictifs en apparence (voir Définition 1), nous avons démontré dans [1] qu'ils sont aussi expressifs que les systèmes de réécriture de manière générale. De plus, ces systèmes de réécriture sont bien adaptés pour le contrôle du mécanisme de réécriture (par exemple demande de calcul de points-fixes locaux). L'originalité de notre approche

réside dans la façon de guider les calculs de points-fixes. Cette stratégie est aussi définie sous forme d'un SRF. Nous avons de plus implémenté un prototype dont les résultats obtenus sur des protocoles comme LEP (*Leader Election Protocol*) ou TAP (*Tree Arbiter Protocol*) s'avèrent prometteurs.

## 2 SRF et Model-Checking Symbolique

Nous distinguons tout d'abord deux catégories de symboles fonctionnels : $\mathcal{F}_{NT}$ sont appelés les symboles non-terminaux (en lettres capitales) et $\mathcal{F}_{bin}$ l'ensemble des symboles terminaux. Sémantiquement, les symboles de $\mathcal{F}_{NT}$ sont considérés comme des symboles de fonctions dont l'implémentation seront décrites par des systèmes de réécriture fonctionnels. De plus, ces symboles ont comme arité 1. Les symboles de $\mathcal{F}_{bin}$ sont les constructeurs de la structure de données que nous mettrons à jour grâce à des systèmes de réécriture fonctionnels. Plus précisément, ces symboles sont soit d'arité deux, soit la constante $\perp$. La définition suivante décrit ce qu'est un système de réécriture fonctionnel.

**Definition 1.** *Un système de réécriture fonctionnel (SRF) $\mathcal{R}_\lambda$ est un ensemble de règles $(l, r) \in \mathcal{T}(\mathcal{F}_{bin} \cup \mathcal{F}_{NT}, X) \times \mathcal{T}(\mathcal{F}_{bin} \cup \mathcal{F}_{NT}, X)$ (notées également $l \to r$) tel que chaque règle soit de l'une des formes décrites ci-dessous :*

1. $F(a(x, y)) \to \alpha$ *avec* $a \in \mathcal{F}_{bin}$, $F \in \mathcal{F}_{NT}$, $x, y \in \mathcal{X}$, $x \neq y$, $\alpha \in \mathcal{T}(\mathcal{F}_{bin} \cup \mathcal{F}_{NT}, X)$ *et* $\mathcal{V}ar(\alpha) = \{x, y\}$
2. $F(\perp) \to \alpha$ *et* $\alpha \in \mathcal{T}(\mathcal{F}_{bin} \cup \mathcal{F}_{NT})$
3. $F(a(x, \perp)) \to x$ *avec* $x \in \mathcal{X}$, $F \in \mathcal{F}_{NT}$ *et* $a \in \mathcal{F}_{bin}$.

L'application d'un SRF est particulière. En effet, dans le mécanisme de réécriture considéré, nous ne pouvons substituer une variable que par un terme ne contenant que des terminaux. Par exemple, prenons le SRF $\mathcal{R}_\lambda$ tel que $\mathcal{R}_\lambda = \{F(a(x, y)) \to b(y, x)\}$. Soient $t$ et $t'$ deux termes de $\mathcal{T}(\mathcal{F}_{bin} \cup \mathcal{F}_{NT})$ tels que $t = F(a(c(\perp, \perp), \perp))$ et $t' = F(a(G(c(\perp, \perp)), \perp))$. Clairement, $F(a(x, y)) \to b(y, x)$ pourrait être appliquée dans le cadre classique de la réécriture à la racine du terme $t'$ en substituant respectivement les variables $x$ et $y$ par les termes $G(c(\perp, \perp))$ et $\perp$. Ce n'est donc pas le cas pour les SRF. Par contre, cette même règle peut tout à fait s'appliquer sur le terme $t$ et ainsi obtenir le terme $b(\perp, c(\perp, \perp))$.

Dans [1], nous avons de plus défini une classe particulière de symboles non-terminaux : $\mathcal{F}^\star_{NT}$. Soient $F^\star \in \mathcal{F}^\star_{NT}$, $F \in \mathcal{F}_{NT}$ et $\mathcal{R}_\lambda$ un SRF implémentant la "*fonction*" $F$. Le terme $F^\star(t)$ avec $t \in \mathcal{T}(\mathcal{F}_{bin})$ modélise le fait que l'on applique la fonction $F$ un nombre de fois non borné. L'application ou l'exécution de la fonction $F$ est régie par le SRF $\mathcal{R}_\lambda$. Ainsi, comme un symbole de $\mathcal{F}^\star_{NT}$ est considéré comme un non-terminal avant tout, des stratégies de réécriture (ou de calcul) peuvent être définies à l'aide de SRF. Par exemple une règle de la forme $G(a(x, y)) \to a(F^\star(x), y)$ appliquée sur le terme $G(a(t_1, t_2))$ avec $t_1, t_2 \in \mathcal{T}(\mathcal{F}_{bin})$ engendrera un ensemble de termes correspondant à un calcul de point-fixe sur le sous-terme $t_1$. Ces stratégies permettent de réaliser des accélérations du calcul des termes accessibles.

Notons aussi que pour le terme $t'$ précédemment défini, l'ensemble des termes de $\mathcal{T}(\mathcal{F}_{bin})$ accessibles est vide. Par conséquent, des propriétés d'invariant peuvent être

exprimées avec notre formalisme. En effet, prenons la négation de l'invariant et considérons la comme un prédicat à vérifier sur chaque terme accessible. L'implémentation de ce prédicat est faite par un SRF $\mathcal{R}_{\neg\phi}$ et un non-terminal est associé à ce prédicat (par exemple, $F_{\neg\phi}$). Considérons d'autre part un SRF $\mathcal{R}_\lambda$ représentant la relation de transition du système étudié, un non-terminal $Succ$ dont la sémantique est de calculer le successeur d'un état du système et $Succ^\star$ un non-terminal représentant le calcul de tous les successeurs. A partir d'un état initial $t_0 \in \mathcal{T}(\mathcal{F}_{bin})$, nous pouvons déduire que l'invariant est satisfait sur la globalité du système si et seulement si l'ensemble des termes dans $\mathcal{T}(\mathcal{F}_{bin})$ accessibles à partir du terme $F_{\neg\phi}(Succ^\star(t_0)))$ en appliquant le SRF $\mathcal{R}_\lambda \cup \mathcal{R}_{\neg\phi}$ est bien vide.

## 3 Principaux Résultats & Conclusions

Nous avons implémenté un prototype (FTRSMC pour *Functional Term Rewriting System Model-Checker*). Clairement, lorsque les stratégies de points-fixes locaux (LFP) sont bien conçues pour le problème à traiter, nous sommes capables de traiter des systèmes dont l'espace des états peut être très grand(colonne $\#Confs$). Le tableau ci-dessous présente une comparaison avec d'autres outils de réécriture : Maude [5] et Timbuk [9].

| $\Sigma$ | N | $\#Confs$ | TRS | | TDDD | FTRS model-checker | |
|---|---|---|---|---|---|---|---|
| | | | Timbuk | Maude | | NoLFP | LFP |
| TAP | $2^5$ | $8.539 * 10^8$ | $> 3h$ | $> 3h$ | $> 3h$ | $> 3h$ | 0.109 |
| | $2^{10}$ | $1.989 * 10^{273}$ | - | - | - | - | 0.196 |
| | $2^{20}$ | $5.350 * 10^{278807}$ | - | - | - | - | 0.256 |
| | $2^{400}$ | ? | - | - | - | - | 2.866 |
| PP | $2^5$ | 11047 | 0.013 | 1.479 | 0.302 | 0.128 | 0.018 |
| | $2^{10}$ | $8.445 * 10^{135}$ | 0.146 | $> 3h$ | $> 3h$ | $> 3h$ | 0.053 |
| | $2^{20}$ | $4.508 * 10^{139402}$ | 0.268 | - | - | - | 0.102 |
| | $2^{500}$ | ? | 216.184 | - | - | - | 0.396 |
| LEP | $2^5$ | 16 | 0.112 | 0.457 | 0.212 | 0.108 | 0.093 |
| | $2^{10}$ | 512 | 0.162 | $> 3h$ | 0.310 | 0.195 | 0.159 |
| | $2^{20}$ | $5.243 * 10^5$ | 0.304 | - | 1.038 | 0.347 | 0.256 |
| | $2^{400}$ | $1.291 * 10^{120}$ | 333.813 | - | 573.265 | 393.969 | 138.699 |
| PHILOS | $2^2$ | 81 | $> 1G$ | 2.013 | 1.283 | 0.081 | 0.003 |
| | $2^3$ | 6561 | - | $> 3h$ | 3.361 | 0.181 | 0.007 |
| | $2^5$ | $1.853 * 10^{14}$ | - | - | 311.648 | 25.871 | 0.029 |
| | $2^6$ | $3.434 * 10^{30}$ | - | - | 3505.274 | 574.508 | 0.869 |
| | $2^7$ | $1.179 * 10^{61}$ | - | - | $> 3h$ | $> 3h$ | 1.199 |
| | $2^{10}$ | $3.734 * 10^{488}$ | - | - | - | - | 2.278 |
| | $2^{30}$ | ? | - | - | - | - | 25.284 |
| | $2^{40}$ | ? | - | - | - | - | 499.282 |
| | $2^{50}$ | ? | - | - | - | - | 10697.804 |

Au delà des performances, même si la forme des règles d'un SRF semble contraignante, nous avons démontré dans [1] que l'expressivité des SRF est la même que celle des systèmes de réécriture.

## Références


1. Y. Boichut, J-M Couvreur, and D-T Nguyen. Functional term rewriting systems Towards Symbolic Model-Checking. International Journal of Critical Computer-Based Systems, 2011.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on computers*, C-35(8) :677–691, August 1986.
3. J.R. Burch, E.M. Clarke, and K.L. McMillan. Symbolic model checking : $10^{20}$ states and beyond. *Information and Computation (Special issue for best papers from LICS90)*, 98(2) :153–181, 1992.
4. G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state-space construction for asynchronous systems. In *ICATPN'2000*, volume 1825 of *LNCS*, pages 103–122. Springer Verlag, 2000.
5. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. F. Quesada. Maude : Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

   `http ://maude.cs.uiuc.edu`.
6. Couvreur, J-M. and Encrenaz,E. and PaviotAdet,E. and Poitrenaud, D. and Wacrenier, P. (2002) *Data Decision Diagram for Petri Net Analysis.* ICATPN, volume 2360, pages 101-120, Springer Verlag.
7. J-M. Couvreur and Y. Thierry-Mieg. Hierarchical decision diagrams to exploit model structure. In *FORTE*, pages 443–457, 2005.
8. J-M. Couvreur and D-T. Nguyen. Tree Data Decision Diagrams. In *VeCOS*, 2008.
9. T. Genet and V. Viet Triem Tong. Reachability Analysis of Term Rewriting Systems with *timbuk*. In *Proc. 8th LPAR Conf., Havana (Cuba)*, volume 2250 of *LNAI*, pages 691–702. Springer-Verlag, 2001.

   `http ://www.irisa.fr/celtique/genet/timbuk/`.
10. H. Hulgaard, P. F. Williams, and H. R. Andersen. Equivalence checking of combinational circuits using boolean expression diagrams. *IEEE Transactions of Computer-Aided Design*, 18(7), 1999.
11. T. Kolks, B. Lin, and H. De Man. Sizing and verification of communication buffers for communicating processes. In *ICCAD'93*, volume 1825, pages 660–664, 1993.
12. S. Minato, N. Ishiura, and S. Yajima. Shared binary decision diagrams with attributed edges for efficient boolean function manipulation. In *DAC'90*, pages 52–57. ACM/IEEE, IEEE Computer Society Press, 1990.
13. A.S. Miner and G. Ciardo. Efficient reachability set generation and storage using decision diagrams. In *ICATPN'99*, volume 1639 of *LNCS*, pages 6–25. Springer Verlag, 1999.
14. Y. Thierry-Mieg, A. Hamez, and F. Kordon. Building efficient model checkers using hierarchical set decidion diagrams and automatic saturation. *Fundamenta Inforaticae Petri Nets*, 1-25, 2008.

# Session de l'action IDM

Ingénierie dirigée par les modèles

# L'ingénierie Dirigée par les Modèles : Où en est-on ?

Thierry Millan[1], Agusti Canals[2]

[1] IRIT - Université Paul Sabatier
118, Route de Narbonne - 31062 TOULOUSE Cedex 9
thierry.millan@irit.fr
[2] CS Communication & Systèmes
Parc de la Grande Plaine - 5 Rue Brindejonc des Moulinais
BP 15872 - 31506 Toulouse Cedex 5
agusti.canals@c-s.fr

L'ingénierie dirigée par les modèles et, en particulier, les processus de développement logiciel à base de modèles ont toujours été au centre des préoccupations des Journées Neptune. Ces procédures correspondent à un paradigme dans lequel le code source n'est plus considéré comme l'élément central d'un logiciel, mais comme un élément dérivé d'éléments de modélisation.

Cette approche prend toute son importance dans le cadre des architectures logicielles et matérielles dirigées par les modèles utilisant des standards tels que les spécifications MDA (Model-Driven Architecture) proposées par l'OMG. De telles architectures s'intègrent tout naturellement dans un processus de développement à base de modèles s'assurant, à chaque niveau de modélisation, que les modèles obtenus et réutilisés ont les qualités requises. Cette démarche met le modèle au centre des préoccupations des analystes/concepteurs. Leur élaboration devient donc centrale et le choix du formalisme revêt une importance capitale.

Les travaux concernant l'IDM menés jusqu'à ce jour dans le cadre de collaborations académiques/industrielles montrent que ces technologies intéressent de plus en plus d'industriels. Cet intérêt grandissant a donné lieu à la réalisation de nombreux projets aussi bien nationaux qu'internationaux et à la réalisation de nombreux outils industriels et open-source. Dans ce contexte, il nous a semblé pertinent, pour l'édition 2011 des journées NEPTUNE, de revenir sur des problématiques que nous avons déjà abordées par le passé afin d'une part de présenter les avancées en matière de vérification et de transformation de modèles et, d'autre part, de présenter les nouveaux projets phares dans le domaine des modèles, ainsi que des retours d'expérience d'entreprises ayant utilisé les technologies et les outils IDM dans le cadre de leurs projets. En effet, un nombre important de travaux et de projets ont vu le jour au cours des cinq dernières années et beaucoup avaient, entre autre, comme objectif le transfert technologique des connaissances issues du monde académique vers le monde industriel et le passage à l'échelle.

Le mode de sélection des présentations basé sur la cooptation nous permet chaque année d'analyser les différents projets issus de l'ANR, de l'IST, des Pôles de Compétitivités et des grands laboratoires Français pour réaliser notre programme. Cette stratégie offre l'avantage d'avoir un programme fortement ciblé sur des problématiques qui intéressent notre communauté, présente un atout important qui est de nous permettre de dresser chaque année un instantané des problématiques de l'IDM qui font l'objet d'une forte activité aussi bien de la part des académiques que

des industriels, et nous permet de proposer une première base pour les débats qui ont lieu à l'issue de chaque journée.

La présentation qui sera faite lors de ce séminaire du GDR est notre vision et celle des participants aux journées NEPTUNE. Elle peut être vue comme un élément de débat.

# Modeling Wizards : École d'automne dédiée à la modélisation

Ileana Ober[1] and Sébastien Gérard[2]

[1] IRIT, Université de Toulouse, 118 Route de Narbonne
31062 Toulouse
Ileana.Ober@irit.fr
[2] CEA LIST
91191 Gif-sur-Yvette
sebastien.gerard@cea.fr

**Résumé** Lors des rencontres avec les partenaires industriels, ainsi que à l'occasions des symposium des enseignants dans le cadre de la conférence MODELS, on évoque le manque de formation des équipes en termes de notions de modélisation comme l'une des raisons principales de la non-généralisation de l'utilisation de la modélisation. L'organisation d'écoles pour les doctorants est un moyens pour palier à ce problème. Nous présentons ici des retours sur l'organisation de la première édition d'une école d'automne dédiée à la modélisation. Cette école, qui est en train d'être intégrée à la prochaine édition européenne de la conférence MODELS, se propose de devenir un forum d'échange pour les jeunes chercheurs.

Lors des rencontres avec les partenaires industriels, ainsi que à l'occasions des symposium des enseignants dans le cadre de la conférence MODELS, on évoque le manque de formation des équipes en termes de notions de modélisation comme l'une des raisons principales de la non-généralisation de l'utilisation de la modélisation.

Pour répondre aux besoins de formation aux techniques de modélisation, du 30 Septembre au 2 Octobre 2010, à Oslo, a eu lieu la première édition de l'école d'automne *Modeling Wizards : 1st International Master Class on Model-Driven Engineering*. L'objectif de cette école internationale est de réunir des personnes intéresses par la thématique de la modélisation, typiquement des thésards mais non seulement et de les mettre dans un contexte qui facilite les échanges.

L'organisation de cette école a été possible grâce au travail de son Conseil Scientifique, qui regroupe des personnalités connues du domaine de la modélisation : *Robert France*, de Colorado State University, États-Unis, *Oystein Haugen*, SINTEF, Norvège, *Alexander Pretschner*, Université Technique de Kaiserslautern, Allemagne, *Andy Schuerr*, Université Technique de Darmstadt, Allemagne, *Bran Selic*, Canada, *Naoyasu Ubayashi*, Kyushu Institute of Technology, Japon, *Jon Whittle*, Lancaster University, Royaume Uni. Ce Conseil Scientifique a été présidé pour cette première édition par *Ileana Ober* (IRIT) et *Sébastien Gérard* (CEA).

Les conférenciers choisis par le Conseil Scientifique pour cette première édition, sont des personnalités connues dans notre domaine, qui ont réussi à réaliser

de présentations qui étaient à la fois accessibles par le public non initié dans la thématique précise de ces présentations, tout en introduisant des éléments qui tiennent du travail prospectif. Les présentations, qui ont eu lieu dans le cadre de cette école, sont les suivantes :

- *Issues in Domain Specific Languages (DSLs) and Model Driven Interoperability (MDI)* effectué par Jean Bézivin de l'Université de Nantes ;
- *DSML use in industrial projects* effectué par Steven Kelly, de la Société Metacase (Finlande) ;
- *DSLs : techniques and tools* effectué par Frédéric Fondement de l'Université de haute Alsace ;
- *DSL and UML profiles* effectué par Sébastien Gérard de CEA ;
- *Integrating Ontologies and UML/SysML models* effectué par Nicolas Rouquette de NASA (États-Unis) ;
- *Model Mappings and Transformations : from Theory to Practice* effactué par - Krzysztof Czarnecki de l'Université de Waterloo (Canada) ;
- *DSL formalisation* effectué par Janos Sztipanovits de Vanderbilt University (États-Unis) ;
- *A model-driven software factory to support enterprise application product-lines* effectué par Vinay Kulkarni, de Tata Consultancy Services (Inde) ;
- *Standardising variability* effectué par Øystein Haugen de l'Université d'Oslo (Norvège).

La plupart de ces présentations sont disponibles sur le site web de l'école [1].

En plus de ces présentations, nous avons organisé plusieurs sessions de mini-projet. le but de ces sessions était de permettre aux participants à l'école de travailler sur un problème concret et d'avoir la possibilité d'appliquer les connaissances acquises pensant les sessions de cours, dans le cadre d'un projet pratique. Les étudiants ont eu accès a la spécification d'un problème à résoudre, et on leur a donné la possibilité de le faire en choisissant parmi trois techniques. Ainsi, le travail en mini-projet a eu lieu en trois sous-groupes, qui ce sont réunit à la fin pour échanger leurs expériences. Cette organisation, qui s'est avéré très intéressante pour les étudiants, a été possible uniquement grâce à l'investissement effectué par ceux qui ont eu en charge l'organisation du module pratique : Frédérique Fondement, Steven Kelly et Florian Noyrit. La spécification qui a fait l'objet de ces travaux pratiques, ainsi que quelques éléments de solution et des renseignements relatifs aux outils employés sont disponibles sur le site de l'école.

Toujours dans le but d'intensifier les échanges entre les participants à l'école, nous avons organisé une session de posters. Le recueil de ces posters est disponible en tant que rapport technique [3].

L'organisation de cette école a été facilitée par des aides reçues de la part d'ARTIST (Network of Excellence in Embedded System Design), du laboratoire IRIT (Institut de Recherche en Informatique de Toulouse), de l'École doctorale MITT de Toulouse, et du CEA. En plus, l'action IDM a offert deux bourses de 500 Euros chacune à des étudiants, Faten ATIGUI et Hassan Ait LAHCEN, pour leur faciliter la participation à cette école. A leur retour de l'école, ces étudiants ont préparé un résumé, qui est disponible sur le site de l'action IDM [2].

Les retours que nous avons pu avoir de la part des participants et conférenciers ont été particulièrement positives et encourageantes, nous incitant à renouveler cette expérience et d'aller vers une meilleure intégration avec la conférence ACM/IEEE MODELS. Suite à ces retours, nous avons contacté de steering committee de la conférence MODELS pour leur proposer une meilleure intégration des deux événements. Leur accueil enthousiaste, nous laissent espérer que la prochaine édition européenne de la conférence ACM/IEEE Models, qui aura lieu en automne 2012 à Innsbruck en Autriche, comptera parmi ses événements satellites la deuxième édition de l'école Modeling Wizards. La nouvelle formule a été soumise à l'approbation des organismes ACM et IEEE, qui patronnent la conférence MODELS et nous sommes optimistes quant à leur retour. Indépendamment de ça, on peut d'ores et déjà annoncer la deuxième édition de l'école Modeling Wizards, qui aura lieu fin septembre 2012 au cœur des Alpes Autrichiens, à presque 2000 m d'altitude, dans les magnifiques locaux de l'Université d'Innsbruck de Obergurgl.

En 1996 à la Conférence ICSE 1996 Tony Hoare, remarquait, par rapport au décalage entre la théorie et la pratique, que "teaching reduces the gap and research increases it again". Il est bien trop tôt pour évaluer l'effet de Modeling Wizards dans ce sens. Cependant, nous sommes persuadés que cette école offre à ses participants une opportunité intéressante et enrichissante pour des échanges différentes que celles qui ont lieu traditionnellement lors des conférences.

## Références

1. http ://modelingwizards.isti.cnr.it, 2010.
2. Faten ATIGUI and Hassan Ait LAHCEN. Compte-rendu de l'École d'automne modeling wizards 2010 (oslo). http ://www.actionidm.org, 2010.
3. Ileana Ober. Modeling Wizards 1st International Master Class on Model-Driven Engineering Poster Session Companion. Rapport de recherche IRIT/RR-2010-20-FR, IRIT, Université Paul Sabatier, Toulouse, septembre 2010.

# Overview of an Approach for Deviation Detection in Modeling Activities of MDE Processes

Reda Bendraou[1], Marcos Aurélio Almeida da Silva[1], Xavier Blanc[2] and Marie-Pierre Gervais[1,3]

1 LIP6, UPMC Paris Universitas, France {firstname.LastName@Lip6.fr}

2 Labri, Université de Bordeaux 1{xavier.blanc@labri.fr}

3 UPO, Université Paris Ouest, France {marie-pierre.gervais@lip6.fr}

## 1. Problem Statement

In the context of large-scale industrial projects, it has been recognized that adopting software process models represents a good means to foster the respect of project's deadlines and reliability criteria [4]. For a long while, such models have been used only for training and communication purposes. However, recently, they are more and more used as inputs to PSEEs (Process-centered Software Engineering Environments) to be enacted. The enactment of a process model consists in improving the coordination between the project's developers by automatically affecting the activities to the appropriate roles, routing the artifacts from one activity into another and ensuring the monitoring of activity's deadlines. The role of the PSEE is also to enforce the fact that the developers respect the execution order of the process's activities and to make sure that they deliver the right outcomes. In case of any violation of the execution order or in the absence of an activity's outcome, a warning is raised so that the project manager can act accordingly and manage to handle this process deviation.

However, what happens during the execution of some creative tasks such as modeling is left to the developer and remains outside the PSEE's control. The developer usually indicates to the PSEE that he started the modeling activity; performs some modeling actions and then submits his outcome to the PSEE at the end of the activity. Nevertheless, if the developer is performing inappropriate modeling actions in his activity, this will never be reported by the process execution engine. If these modeling actions are in contradiction with the process guidelines or company's best practices, they will straightforwardly impact the other activity's outcomes and may induce delays in the project schedule.

Unluckily, in current PSEEs, the effects of such deviations are discovered very late in the process execution, usually, at the end of long design activities when the delivered model is submitted to be structurally checked either manually or with the help of model checkers. At this time, if the model is inconsistent, the developer will be asked to rework the modeling activity which can be costly in terms of time and may disturb the project's organization.

Lanubile and Vissagio [5] conducted an empirical study that showed that most agents do not follow an assigned process 100% of time. Main reasons for that are because the developers may be confronted to unanticipated situations which are not represented in the process model or their willing to accomplish the activity according to their experience and intuitions. Visser et al in [6] also concluded that experts usually deviate and act opportunistically, using the process as a guide without necessarily reducing the quality of their work.

Therefore, whatever the process model, it yet would be too restrictive to assume that the process is fixed and that it anticipates all possible situations. Instead of forcing developers to follow a potentially sound process model, a better solution would be to provide PSEE with the necessary flexibility for dealing with developer's deviations. Our intuition is that, if the developer is warned instantly as soon as he is deviating from the process, he can anticipate many errors and manage to come back to the nominal process. This can also avoid reworking the modeling activity that causes the deviation and thus disturbing the project organization.

Unfortunately, none of the current approaches for process modeling and execution provides such flexibility. Present solutions succeed only in detecting if the developers are violating the execution order of the process activities. They do not provide a support for detecting developer's deviations inside modeling activities. Detecting such deviations as soon as they occur can improve the quality of the delivered software and prevent from many risks of project failure and unnecessary delays. This is what we tried to prove in the context of the works presented in [1] and in [2]. While the first paper focused on presenting our approach for detecting developer deviations during process execution, the second paper described an empirical study we conducted in order to assess the effect of providing developers with a PSEE that supports process deviation detections.

## 2. An Approach for Detecting Process Deviations

In our approach, we propose to deal with process deviations by: (i) providing developers the ability to perform their activities even though that may cause deviations but inside the control of the PSEE; and (ii) tracking developer's deviations instantly and warning him with possible ways to repair them.

To achieve these goals, in [1] we introduced the requirements and architecture of a PSEE ensuring such functionalities. The architecture is generic and is composed of three main components: 1) the Process Execution Engine (PEE) for process model execution; 2) The Modeling Action Listener (MAL) for tracing the actions performed by the developer during modeling activities. At this aim, the MAL is integrated to the modeling tool used by the developer; and to the PEE to capture in which process's activity these actions where performed. The traces of modeling actions are represented using our language Praxis, which represents the construction of a model as a sequence of elementary editing actions (e.g. create a model element; add a reference to a class, etc.) [3].

3) Finally, the Deviation Detection Engine (DDE) which is responsible of detecting developer's deviations. The DDE uses the traces of developer's modeling actions captured by the MAL (Praxis editing actions), the information given by the PEE about the activity being executed and the most important, a description of allowed actions within the activity, provided with the process model. While in current approaches, activity's allowed actions are given in natural language (i.e. in form of guidelines), we promote the use of process activity rules. These rules are expressed using a language we defined called Praxis Rules and acts as constraint language on top of modeling actions represented using Praxis. Praxis and Praxis rules are briefly presented in the next subsection.

***a. Praxis and Praxis Rules***

As stated earlier, Praxis is used to represent each of the elementary modeling actions performed by the developer for building a model whatever the modeling language used at this aim. Thus, a model construction is represented as a sequence of Praxis editing actions. In order to check if the developer is performing the appropriate actions for building a model or if the model he delivered at the end of the activity is free of inconsistencies, we apply Praxis Rules, a rule-based logical language, on top of this sequence.

For preventing from developer deviations during the execution of an activity, we attach to the process activity's description a Praxis Rule *Invariant*. This invariant is used to indicate the set of allowed actions in this activity and optionally the order of their occurrence. An example would be for instance to say that *"during the application of the MVC design pattern in activity X, the developer is not allowed to create other packages then the Model, View and Control packages"*. If the developer creates a package called "Foo" a deviation is instantly detected and the developer is warned with its cause and a repair option. The process activity invariant rule can be more precise by stating the order of package creations (e.g., first "Model", then "Control" and finally "View"). If the order is not respected, a deviation is also raised. Of course, the developers can decide to ignore these warnings and come back later to repair his deviations.

With Praxis Rules it is also possible to express an activity *Post-Condition* to prevent from the fact that a developer may deliver an inconsistent model at the end of an activity. An example of post-condition would be for instance *"at the end of activity Y, the model should contain the Notify() operation otherwise, the model is inconsistent or is in conflict with another model"*. If the post-condition is violated, this is considered as deviation and a warning is displayed to the developer.

For process modelers, the originality of our approach comes from the fact that it is process modeling language, process execution engine and modeling tool independent. In our experiment we used UML activity diagrams for process modeling, the Eclipse cheat sheets as process execution engine and Papyrus as modeling language. The cost of adopting our approach is related to the efforts of learning Praxis Rules, defining process activity rules to be attached with the

process model and to the customization of the MAL for a given modeling tool. More details on adoption costs can be found in the paper [1]

***b. Validation, results and Perspectives***

To validate our approach, we first built a PSEE on top of Eclipse Cheat sheets and Praxis Rules and we used UML activity diagrams as a process modeling language. Secondly, we realized a case study and an empirical evaluation to prove the effectiveness of our approach. This was done by submitting our tool and a process example to our master students in order to evaluate the impact of using our PSEE. The study demonstrated that the students that used our tool get their delivered model better and faster than those that did not used our tool. More details on the empirical study we conducted can be found in [1][2].

Therefore, we can consider using our tool and approach in the two following cases. The first one is to use our tool as a possible means to continuously improve the process as described by the CMMI maturity levels. Indeed, detecting recurrent developer deviations may be a way to deduce a better way of achieving the process, to understand why developers deviate and how we can improve the process model in the future. The second one is to provide organizations having the CMMi's Defined Level or more the means to preserve their maturity level by preventing the developers for possible deviations during process enactment. As a perspective of this work we are currently studying the resolution of the optimal path to reconcile the developer with the process description in case late deviation detections i.e. the early deviation detection is turned off by the developer. We also plan to generalize the approach for more generic process activities and not to be only restricted to modeling activities.

## 3. References

[1] Almeida da Silva M.A., Bendraou R., Blanc X. and Gervais M-P., "Early Deviation Detection in Modeling activities of MDE Processes", in proceedings of MoDELS 2010, LNCS, Springer Verlag, October 3-8, 2010, Oslo, Norway

[2] Almeida da Silva M.A., Mougenot A., Bendraou R., Robin J. and Blanc X. "Artifact or Process Guidance, an Empirical Study", in proceedings of MoDELS 2010, LNCS, Springer Verlag, October 3-8, 2010, Oslo, Norway

[3] Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Detecting model inconsistency through operation-based model construction. In Robby, ed.: Proc. Int'l Conf. Software engineering (ICSE'08). Volume 1., ACM (2008) 511-520.

[4] Chrissis, M.B., Konrad, M., Shrum, S., 2003. CMMI: Guidelines for Process Integration and Product Improvement. Addison-Wesley, Boston, MA

[5] Lanubile F. and Visaggio G. Evaluating defect detection techniques for software requirements inspections, 2000.

[6] Visser W. More or less following a plan during design: Opportunistic deviations in speci_cation. International Journal of Man-Machine Studies, 33(3):247-278, 1990.

# Session du groupe de travail LaMHA

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

# BSP-WHY: an intermediate language for deductive verification of BSP programs

Jean Fortin and Frédéric Gava

Laboratory of Algorithms, Complexity and Logic, University of Paris-East
61 avenue du Général de Gaulle, 94010 Créteil cedex – France
fortin@u-pec.fr and frederic.gava@univ-paris-est.fr

## 1 Introduction

The correctness of parallel programs is of paramount importance, especially considering the growing number of parallel architecture (GPUs, multi-cores, *etc.*) and the cost of conducting large-scale simulations — the losses due to fault program, unreliable results or crashing simulations. Formal verification tools that display parallel concepts are thus useful for program understanding and incisive debugging and such tools are long overdue. Given the strong heterogeneity of these massively parallel architectures and their complexity, a frontal attack of the problem of verification of parallel programs is a daunting task that is unlikely to materialize. An approach would be to consider well-defined subsets that include interesting structural properties. A BSP program is executed as a sequence of *super-steps* [1]. In a super-step, each processor performs a pure sequential computation followed by a global communication phase. It looks like a good candidate for formal verification: its model of execution is simpler to understand than any concurrent model.

Also, avoiding deadlocks is not sufficient to ensure that the parallel programs will not crash. Mainly, we need to check buffer and integer overflows (safety) and liveness and thus a better trust in the code: are results as intended? Verification Condition Generator (VCG) tools, as Why[2] is one of the solutions. They take an annotated program (Hoare logic spirit) as input and produce verification conditions (proof obligations to provers) as output to ensure correctness of the properties given in the annotation. An advantage of this approach is to allow the mixing of the manual proof of properties using proof assistants and automatised checks of simple properties using automatic decision procedures.

The goal of BSP-Whyis to be a tool for the verification of properties of a special class of parallel programs by providing annotations and generation of proof obligations using a VCG.

## 2 How BSP-Why works ?

We used the Why language as a back-end of our own BSP-Whylanguage (The prototype BSP-Why tool is available at http://lacl.fr/fortin/BSP-why/ ). BSP-Why extends the syntax of Whywith BSP primitives (message passing and synchronisation). A special syntax for BSP annotations is also provided which is simple to use and seems sufficient to express conditions in most practical programs.

The transformation of BSP-Why programs into Why ones is based on the fact (proved using an operationel semantics) that, for each super-step, if we execute sequentially each computation of each processor and them perform the simulation of the communications by copy data, we have the same results that really doing the execution in parallel.

The first step of the transformation is a decomposition of the program into blocks of sequential instructions. In order to obtain the best efficiency, we are trying to isolate the largest blocks of code that remain sequential. Once that we have regrouped the sequential parts of the program into blocks, we create a "for" loop, to execute the code block sequentially on each processor. Special care must be taken to generate correct loop invariants and variant in the "for" loop executing the sequential code. If the invariant is not strong enough, it will not be possible to prove the resulting program using Why. The invariant also keeps track of which variable are modified, and which are not. Since we are using arrays to represent the variables on each processors, it is necessary to say that we only modify a variable on the current processor, and that the remaining of the array stays unchanged after the iteration of the loop.

Aso, when transforming a `if` or `while` structure, there is a risk that a global synchronous instruction might be executed on a processor and not on the other. We generate an assertion to forbid this case, ensuring that the condition associated with the instruction will always be true on every processor at the same time (and thus fordid deadlocks). The parallel instructions are not directly translated in an equivalent sequential code. They are replaced by calls to the parameters axiomatized in a prelude file. The details and examples are available in [3].

## 3 Future Work

First, we intend to add a companion tool for C and Java programs as in [2]. Second, BSP features a cost model for an estimation of the execution time: formally giving these costs by extended pre-, post-condition and invariants is an interesting challenge. Third, we are currently working on adapt BSP-Why for sub-set synchronisation: having super-steps for only subpart of the processors. Last, there are many more MPI programs than BSP ones: our tool is intended to manage MPI programs that are BSP-like (*e.g.* MPI's collective routines).

## References

1. R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
2. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer-Verlag, 2007.
3. J. Fortin and F. Gava. BSP-Why: an intermediate language for deductive verification of BSP programs. In *4th International Workshop on High-level Parallel Programming and Applications (HLPP 2010, affiliated to conference ICFP 2010).* ACM Press, 2010.

# Principes et Pratiques de la Programmation Parallèle en Pi-calcul

Frédéric Peschanski - UPMC - LIP6 - APR

## 1 Introduction

*« On ne rase plus gratis ! »* [12]. L'avénement des architectures multi-coeurs grand-public à remis au goût du jour la concurrence et le parallélisme [1]. Il y a quelques nouveautés : (1) le faible coût du parallélisme matériel, (2) de nouvelles techniques bas-niveau (algorithmes *lock-free*, *wait-free*, mémoires transactionnelles, etc.) et (3) le développement d'une théorie de la concurrence (algèbres de processus, réseaux de Petri, etc.). Notre travail met l'emphase sur ce troisième point : nous nous basons sur la théorie du pi-calcul [2], un langage minimaliste, naturellement concurrent, et accompagné d'un vaste *corpus* théorique. Ce dernier aspect n'est pas toujours un avantage car cela conduit - comme pour le lambda-calcul - à un nombre assez vertigineux de variantes syntaxiques et/ou sémantiques, un labyrinthe dans lequel il est parfois peu aisé de se repérer.

Notre réflexion imite le cheminement qui conduit du lambda-calcul théorique aux machines abstraites puis aux compilateurs efficaces pour les langages fonctionnels, mais transposé au pi-calcul pour les *langages à processus*. Nous proposons les *pi-threads*, une variante appliquée du pi-calcul comme langage intermédiaire dans une chaîne de compilation. Ce langage « noyau » est présenté en Section 2. Dans les langages à processus, la problématique de l'ordonnancement efficace est centrale. L'enjeu ici est de ne pas limiter le nombre de processus tout en garantissant un taux « d'occupation par cœur » le plus élevé possible. Nous discutons en Section 3 de l'ordonnanceur parallèle des pi-threads qui se démarque par son caractère fortement décentralisé. Autre aspect fondamental : la récupération automatique de la mémoire. L'enjeu à ce niveau est de « paralléliser » au maximum sans pour autant tomber dans une complexité technique rédhibitoire. Le ramasse-miettes des pi-threads est basé sur un principe de détection de terminaison partielle de processus, qui le rend à la fois simple conceptuellement et naturellement parallèle. Cet aspect est discuté en Section 4.

**Remarque** : Cet article est un résumé étendu basé sur deux articles précédemment publiés, l'un aux JFLA en 2010 [4] et l'autre à DAMP 2011 (Declarative Aspects of Multicore Programming) [5].

## 2 Le langage intermédiaire

Le langage intermédiaire sur lequel nous nous basons suit la syntaxe suivante :

| | | | |
|---|---|---|---|
| **Définition** | `def` $D(x_1, \ldots, x_n)$=$P$ | | |
| **Processus** | $P$`::=` | `end` | terminaison |
| | | $\mid \sum_i [g_i]\alpha_i, P_i$ | choix |
| | | $\mid D(v_1, \ldots, v_n)$ | appel (terminal) |
| **Action** | $\alpha$`::=` | `tau` | silence |
| | | $\mid c$`!`$v$ | émission |
| | | $\mid c$`?(`$x$`)` | réception |
| | | `\| new(`$c$`)` | création de canal |
| | | `\| spawn{`$P$`}` | création de processus |

Un programme est formé d'un ensemble de définition suivi d'une expression de processus. Le langage est clairement spartiate avec peu de constructeurs : la terminaison, le choix non-déterministe (un lointain héritier du célèbre *select*) et les appels terminaux. Chaque branche d'un choix non-déterministe est préfixé par un garde (expression booléenne), une action (atomique) et une continuation. La sémantique informelle est que la première branche (dans

1. L'exercice dialectique qui conciste à séparer les notions de concurrence et de parallélisme tomble souvent dans la rhétorique - simplifions en disant que la première découle naturellement de la seconde.

l'ordre spécifié) avec un garde valide et une action exécutable est choisie. Si aucune branche ne peut-être choisie alors le choix est bloquant. Le dernier constructeur concerne les appels (éventuellement récursifs) dont la principale qualité est d'être systématiquement en position terminale. C'est une différence fondamentale avec les langages fonctionnels usuels qui ne contraignent pas les appels en position terminale (même s'ils sont vivement encouragés). Les actions atomiques les plus importantes concernent la communication synchrone. L'action $c!v$ qui tente d'émettre sur un canal $c$ une « valeur » $v$ (en fait une expression) n'est exécutable que si au moins un autre processus effectue une action complémentaire $c?(x)$ de réception sur le même canal $c$. Dans ce cas une synchronisation se produit entre les deux processus, et la valeur $v$ est communiquée au récepteur puis liée à la variable $x$. Ces actions sont complétées par une action silence `tau` (plus intéressante qu'il n'y paraît puisqu'elle permet de « forcer » une branche de choix), une action de création de canal `new(`$c$`)` et une action de création de processus fils `spawn{`$P$`}` exécutant le processus $P$. On ajoute à ce langage toutes les « décorations » d'un calcul appliqué : types et valeurs de base saupoudrés de sucres syntaxiques qui ne seront pas discutées en détails ici.

Ce langage intrinsèquement parallèle est très expressif, notamment par le biais du passage de canal, principe consistant à considérer les canaux de communications comme des valeurs de première classe. Voici un exemple de programme [2] :

```
def TaskPool(nb:int, enter:chan<chan<>>, leave:chan<>) =
  [true] leave?, TaskPool(nb+1,enter,leave)
  + [nb>0] enter?(release), spawn{Permit(release,leave)}, TaskPool(nb-1,enter,leave)

def Permit(release:chan<>, leave:chan<>) = release?,leave!,end

def Task(enter:chan<>) =
  new(rel:chan<>), enter!(rel), /* tâche à exécuter */, rel!,end

new(e:chan<chan<>>),spawn{Task(e)},spawn{Task(e)},spawn{Task(e)},spawn{Task(e)},
new(l:chan<>), TaskPool(2,e,l)
```

Il s'agit de l'implémentation d'une version robuste des célèbres sémaphores. Dans notre exemple, on lance 4 tâches en parallèle mais en s'assurant qu'au plus 2 d'entre elles peuvent réellement s'exécuter « simultanément ». Une tâche peut demander un permis au `TaskPool` en envoyant sur le canal `enter` un canal privé `release` qui sera utilisé en fin d'exécution pour relâcher le permis. Le type `chan<chan<>>` du canal `enter` est symptomatique du pi-calcul (`chan<>` représente la synchronisation pure à la CCS, sans passage de valeur). Si un permis est disponible (compteur `nb` positif dans le `TaskPool`) alors un processus `Permit` est lancé pour contrôler la tâche. Lorsqu'aucun permis n'est disponible les tâches sont implicitement mises en attente et la seule action autorisée est le relâchement d'un permis déjà attribué par une tâche en fin d'exécution.

Cet exemple à l'avantage d'être non-trivial, relativement concis et utile en pratique (c'est un *pattern* de base) ; surtout, il exploite l'ensemble des constructeurs du langage. De nombreux autres exemples peuvent être trouvés dans la littérature, par exemple [2, 11, 7, 4, 5].

## 3 Les principes d'ordonnancement

Les trois mots d'ordre concernant l'ordonnaceur des pi-threads sont : (1) la fidélité, (2) l'efficacité et (3) le parallélisme. Le caractère fidèle - à la théorie du pi-calcul - n'est pas uniquement une contrainte artificielle, il s'agit de relier la problématique de modélisation et de vérification (de propriétés) à la problématique de programmation. Les pi-threads sont en effet à rapprocher de nos travaux dans le domaine de la modélisation et la vérification à base de Pi-calcul, cf. [6, 8, 9]. Concernant l'efficacité il s'agit d'efficacité algorithmique, que l'on doit croiser avec la problématique de parallélisation de ces algorithmes. Pour l'implémentation de l'ordonnanceur, on recherchera avant tout une décentralisation maximale de la logique de contrôle. On peut schématiser la structure d'un pi-thread par un triplet noté $[\Gamma;\delta] : P$ où $\Gamma$ est un ensemble d'engagements (*commitments*), $\delta$ est un environnement local (reliant comme de coutûme variables et valeurs) et $P$ est une expression de processus. Une caractéristique fondamentale observable à ce niveau est que l'environnement $\delta$ est « plat » c'est-à-dire que contrairement au lambda-calcul, le pi-calcul ne permet pas aux environnements d'être emboîtés. Si l'on ajoute ce fait au caractère terminal des appels récursifs, alors on obtient une particularité du modèle de calcul proposé : on peut se passer de structure de pile pour

2. Rappelons que le langage présenté n'est pas un langage de programmation mais un langage noyau, cible pour le *frontend* du compilateur, et source du *middleend*.

l'évaluer (ou le compiler). Cette caractéristique *stackless* est anecdotique en séquentiel mais prend tout son intérêt si l'on pense au parallélisme. Si la pile de chaque thread est séparée on obtient des threads « lourds » (*heavyweight*) et si on la partage (comme dans le langage Go [1]) on se retrouve avec un point de centralisation (vraiment) complexe à exploiter.

Les engagements sont similaires à des captures de continuation, et indiquent comment « débloquer » un processus en attente. Si un pi-thread est en attente avec un engagement d'émission $c \Leftarrow e : Q$ alors on peut le débloquer par une réception sur le canal $c$. Dans ce cas, l'expression $e$ sera évaluée pour le receveur et l'émetteur « réveillé » continuera avec l'expression $Q$. Avec un engagement de réception $c \Rightarrow x$ (canal $c$, variable $x$) le principe est symétrique. Contrairement à l'implémentation Pict du pi-calcul [10] les pi-threads permettent d'enregistrer plusieurs engagements simultanément, ce qui augment considérablement l'expressivité du langage (tout en complexifiant la problématique d'ordonnancement). Ces engagements explicites permettent un ordonnancement en temps « théoriquement » constant. Mais la problématique de parallélisation se heurte au caractère centralisé d'une opération fondamentale de l'ordonnanceur : lorsqu'un pi-thread est réveillé, il doit annuler tous ses engagenements et cela peut concerner de multiples canaux. Pour paralléliser cet algorithme, l'idée est de mettre en place une invalidation implicite des engagements. Pour cela, chaque engagement enregistré sur un canal encapsule une horloge logique correspondant au « moment » ou le pi-thread concerné s'est engagé. Lorsque ce dernier est réveillé, l'horloge logique est incrémentée, ce qui a pour effet immédiat d'invalider tous les engagements enregistrés, sans aucune destruction explicite. En pratique, il faut implémenter quelques « trucs » comme le recyclage des engagements invalides et l'implémentation efficace (i.e. *lock-free*) de l'horloge logique. Au final, on obtient un système d'ordonnancement fortement décentralisé dans lequel les pi-threads ne se synchronisent jamais directement

# 4 Le ramasse-miettes

Le ramasse-miettes est sans doute la contribution la plus « évidente » de notre travail, de pas son originalité et surtout sa simplicité. La sémantique formelle du ramasse-miettes des pi-threads (cf. [4, 5]) exploite un minimum d'information. Chaque canal est associé à un compteur qui enregistre le nombre de threads le « connaissant ». On remarquera qu'il ne s'agit pas de comptage de référence classique puisque seuls les threads sont comptés (et non les références proprement dites). La première règle sémantique indique que tout canal possédant un compteur à zéro (et donc connu d'aucun thread) peut être ramassé, en toute logique. La seconde règle de sémantique, légèrement plus complexe, explique que toute clique de pi-threads bloqués sur des canaux uniquement connu des threads de la clique peut être récupérée intégralement par le ramasse-mietttes. Un cas dégénéré simple est lorsqu'un thread tente d'émettre ou recevoir sur un canal dont le compteur est à 1. Cela signifie qu'aucun autre thread ne connait le canal concerné, et donc aucune synchronisation ne pourra se faire le pi-thread est bloqué. En toute logique il faut le « ramasser ». Un autre cas dégénéré concerne les structures cycliques, le contre-exemple classique des ramasse-miettes à comptage de références. Dans la sémantique proposée, aucune structure particulière n'est identifiée, à part la séparation de la clique à ramasser du monde des threads actifs. Les cycles sont donc ramassés « gratuitement ». On peut montrer assez simplement que ces deux règles de sémantiques sont à la fois nécessaires (on ne peut pas les inter-définir) et suffisantes pour résoudre le problème du ramasse-miettes dans le langage proposé. Algorithmiquement parlant, la problématique de ramassage des miettes est donc réinterprétée en une détection de terminaison partielle. Ce problème peut être résolu de façon efficace avec un algorithme parallèle relativement simple.

# 5 Conclusion

Programmer avec du pi-calcul peut sembler à première vue une idée assez saugrenue. Le langage « théorique » (par exemple celui de [2]) semble en effet bien trop abstrait du point de vue de la programmation. Mais comme pour l'illustre lambda-calcul, un minimum de « décoration » du langage permet d'obtenir un langage intermédiaire - ou langage noyau - à la fois minimaliste, très expressif et surtout implémentable de façon efficace. Nos travaux montrent ainsi que d'un point de vue algorithmique l'efficacité est au rendez-vous. En pratique, nous disposons de plusieurs implémentations des pi-threads. La bibliothèque LuaPi [3] propose une implémentation centralisée à base de coroutines. Outre son intérêt pédagogique (les principes de programmation y sont largement documentés), cette implémentation nous permet d'expérimenter l'algorithme d'ordonnancement discuté en Section 3. La bibliothèque JavaPi [3] exploite la couche multi-thread de l'environnement Java et permet donc de tirer parti des architectures

3. cf. `http://code.google.com/p/javapi/`

multi-coeurs. En revanche, chaque pi-thread étant représenté par un thread Java, l'approche est *heavyweight*. Une troisième implémentation - en cours de développement - propose une chaîne de compilation complète permettant d'exploiter le multi-coeur avec des pi-threads légers.

## Références


[1] The Go authors. *The Go Programming Language Specification*, 2009. `http://golang.org`.

[2] Robin Milner. *Communicating and Mobile Systems : The π-Calculus*. Cambridge University Press, 1999.

[3] Frédéric Peschanski. (lua)pi-threads tutorial. Technical report, UPMC Paris Universitas – LIP6, `http://luaforge.net/docman/view.php/505/5768/LuaPiTut.pdf`, 2008.

[4] Frédéric Peschanski. Principes et pratiques de la programmation en pi-calcul. In *JFLA 2010*, Studia Informatica Universalis, pages 245–273. Hermann éditeurs, 2010.

[5] Frédéric Peschanski. Parallel computing with the pi-calculus. In Manuel Carro and John H. Reppy, editors, *DAMP*, pages 45–54. ACM, 2011.

[6] Frédéric Peschanski and Joël-Alexis Bialkiewicz. Modelling and verifying mobile systems using pi-graphs. In *Sofsem'09*, volume LNCS 5404, pages 437–442. Springer, 2009.

[7] Frédéric Peschanski and Samuel Hym. A stackless runtime environment for a pi-calculus. In *VEE '06*, pages 57–67. ACM Press, 2006.

[8] Frederic Peschanski, Hanna Klaudel, and Raymond Devillers. A decidable characterization of a graphical pi-calculus with iterators. In *Infinity*, volume 39 of *EPTCS*, pages 47–61, 2010.

[9] Frederic Peschanski, Hanna Klaudel, and Raymond Devillers. A Petri net interpretation of open reconfigurable systems. In *Petri Nets*, volume to appear of *LNCS*, 2011.

[10] Benjamin C. Pierce and David N. Turner. Pict : a programming language based on the pi-calculus. In *Proof, Language, and Interaction*, pages 455–494. The MIT Press, 2000.

[11] Davide Sangiori and David Walker. *The pi-calculus : a Theory of Mobile Processes*. Cambridge University Press, 2001.

[12] Herb Sutter. The free lunch is over : a fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, March 2005.

# SGL - Programmation parallèle hétérogène et hiérarchique

Chong Li[1,2] et Gaétan Hains[1,2]

[1] LACL, Université Paris-Est, 94000 Créteil, France
[2] EXQIM SAS, 24 rue de Caumartin, 75009 Paris, France
chong.li@exqim.com gaetan.hains@u-pec.fr

**Résumé.** Cet article présente un modèle pour la programmation parallèle base de diffusion-contraction et réalisé sour la forme d'un langage impératif simple appelé *scatter-gather language* ou (SGL). Sa conception est basée sur l'expérience avec la programmation BSP. Les nouvelles caractéristiques de SGL sont motivées par l'évolution de la dernière décennie vers les architectures parallèles multi-niveaux et hétérogènes contenant des processeurs multi-cœurs, des accélérateurs graphiques et les réseaux de routage hiérarchiques. La conception de SGL est conforme au modèle Multi-BSP de Valiant, tout en offrant une interface de programmation encore plus simple que les primitives de BSML (bulk-synchronous parallel ML). SGL semble couvrir un grand sous-ensemble des algorithmes BSP tout en centralisant la sémantique des communications. Comme tous les systèmes inspirés de BSP, il permet l'équilibrage de charge systématique et des performances prévisibles, portables et évolutives.

**Mots-clé:** BSP, langage de programmation parallèle, *bridging parallel model*, sémantique opérationelle, modèle de performance

## 1 Introduction

La programmation parallèle et les algorithmes parallèles ont été les principales techniques au cœur du calcul haute-performance depuis plusieurs décennies. Des chercheurs comme M. Cole[2] et H. Kuchen ont développé le paradigme des squelettes algorithmiques pour la programmation parallèle déterministe et sans interblocage. Vers 1990, L. Valiant a présenté son modèle Bulk-Synchronous Parallel (BSP)[8] qui est un *bridging model* reliant les algorithmes parallèles aux architectures matérielles. Ce modèle est à la fois réaliste ce qui a permis à McColl et al.[6] de définir des versions BSP de tous les algorithmes parallèles importants, de les mettre en œvre et de vérifier la portabilité et de l'extensibilité de leurs performances telles que prévues par le modèle. Loulergue, Hains et Foisy ont conçu BS-lambda[5] comme un modèle de calcul minimal avec les opérations BSP. Ensuite une bibliothéque pour le langage OCaml, appelé BSML (Bulk Synchronous Parallel ML), a été développée par Loulergue et al.[4].

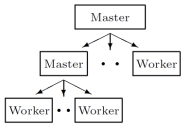
Bien que BSML soit en évolution et que beaucoup d'expérience pratique avec des algorithmes BSP ait été accumulée, l'une de ses hypothèse de base sur les

matériels parallèles a été changée. La vue " à plat " d'une machine parallèle comme un ensemble des machines séquentielles communicantes reste utile, mais est de plus en plus incomplète. Nous allons introduire ici *Scatter-Gather Parallel Language* (SGL)[3], qui est à la fois un *bridging model* hiérarchique et un langage de programmation complet pour les algorithmes parallèlles. SGL généralise Multi-BSP[9] pour des systèmes hétérogènes, réalise un modèle de programmation parallèlle et réduit les primitives de BSML à trois: *scatter*, *gather* et *pardo*.

## 2 Le modèle SGL

La machine abstraite de SGL commence par un ensemble de processeurs composés d'un élément de calcul ("cœur") et d'un élément de stockage ("mémoire"). Les processeurs sont disposés dans une structure arborescente avec la racine appelée "maître" et de ses fils qui sont soit maîtres eux-mêmes (nœuds) soit des "travailleurs" (feuilles). Le nombre de travailleurs-fils n'est pas limité de sorte que le concept BSP d'un $p$-vecteur plat des processeurs est facilement récupérable dans SGL. Différentes formes sont possibles:

1. un travailleur, sans maître → une machine séquentielle
2. un maître + des travailleurs → e.g. un ordinateur BSP
3. une machine hiérarchique → c.f. figure à droite




La structure logique d'un ordinateur SGL satisfait les contraints suivantes. Un système doit avoir un et un seul racine-maître. Un maître coordonne ses enfants. Un travailleur est contrôlé par un et un seul maître. Toutes les communications sont entre un maître et ses enfants.

Un programme de SGL est composé d'une séquence de *super-étapes*. Chaque *super-étape* est composée de 4 phases: 1. une phase de communication *scatter* initialisée par le maître; 2. une phase de calcul *pardo* asynchrone effectuée par les enfants; 3. une phase de communication *gather* centralisée au maître; 4. une phase de calcul local sur le maître. Chacune des quatre phases peut être nulle. La phase *pardo* peut elle-même être un programme SGL.

L'objectif principal de SGL est d'exprimer des algorithmes parallèles ce qui nécessite une notion précise du temps d'exécution de chaque super-étape: la fonction $Cost$ définie récursivement sur la structure de la machine SGL. $Cost_{Master} = \max_{i=1}^{p}(Cost_{child_i}) + w_0 * c_0 + k_\downarrow * g_\downarrow + k_\uparrow * g_\uparrow + 2l$ et $Cost_{Worker} = w_i * c_i$. La définition des paramètres peut être trouvée dans [3]. Notre modèle couvre la possibilité d'une architecture hétérogène.

## 3 La sémantique du langage

SGL est un langage impératif pour lequel nous avons défini une sémantique opérationnelle de "big-step"[10]. La définition complète peut être trouvée dans [3]. Nous résumons ici seulement les spécificités de SGL.

Les valeurs sont des nombres entiers, les booléens et des tableaux construits à partir de ces valeurs. Les expressions élémentaires définissent des opérations

vectorielles et des conversions scalaire-vecteur. Les commandes du langage comprennent les constructions séquentielles classiques avec les 3 primitives de SGL:
**Com** ::= **skip** | $X := a$ | $\overrightarrow{V} := v$ | $\widetilde{W} := w$ | $c$ ; $c$ | **if** $b$ **then** $c$ **else** $c$ |
**for** $X$ **from** $a$ **to** $a$ **do** $c$ | **scatter** $w$ **to** $\overrightarrow{V}$ | **scatter** $v$ **to** $X$ | **gather** $\overrightarrow{V}$ **to** $\widetilde{W}$ |
**gather** $X$ **to** $\overrightarrow{V}$ | **pardo** $c$ | **if master** $c$ **else** $c$.

Les environnements associent aux variables impératives des valeurs de la sorte correspondante. La valeur $Pos \in Nat$ est appelée emplacement (relatif) dans la machine SGL: $Pos = 0$ désigne la position du maître, et $Pos = i \in \{1..p\}$ représente la position du i-ème fils. Elle est l'analogue récursif du `pid` de BSP ou de MPI. $\sigma : NatLoc \to Pos \to Nat$, donc $\sigma(X_{pos}) = \sigma_{pos}(X) \in Nat$; $\sigma : VecLoc \to Pos \to Vec$, donc $\sigma(\overrightarrow{V}_{pos}) = \sigma_{pos}(\overrightarrow{V}) \in Vec$.

La sémantique des expressions classiques sur les vecteur ne mérite pas d'explications particulières. Les primitives vectorielles propres à SGL sont:

$$\frac{\langle w,\ \sigma\rangle \to <v_1,v_2,\ldots v_\ell> \quad \forall_{i=1}^{numChd} \langle \overrightarrow{V}_i := v_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter } w \textbf{ to } \overrightarrow{V},\ \sigma\rangle \to \sigma'} \qquad \frac{\langle \widetilde{W} := <\overrightarrow{V}_1, \overrightarrow{V}_2, \ldots \overrightarrow{V}_{numChd}>,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather } \overrightarrow{V} \textbf{ to } \widetilde{W},\ \sigma\rangle \to \sigma'}$$

$$\frac{\langle v,\ \sigma\rangle \to <n_1,n_2,\ldots n_\ell> \quad \forall_{i=1}^{numChd} \langle X_i := n_i,\ \sigma\rangle \to \sigma'_i}{\langle \textbf{scatter } v \textbf{ to } X,\ \sigma\rangle \to \sigma'} \qquad \frac{\langle \overrightarrow{V} := <X_1,X_2,\ldots X_{numChd}>,\ \sigma\rangle \to \sigma'}{\langle \textbf{gather } X \textbf{ to } \overrightarrow{V},\ \sigma\rangle \to \sigma'}$$

$$\frac{\forall_{i=1}^{numChd} \langle c,\ \sigma_i\rangle \to \sigma'_i}{\langle \textbf{pardo } c,\ \sigma\rangle \to \sigma'}$$

Nous avons modélisé une grappe SGI Altix ICE 8200EX de 128 cœurs en une machine SGL deux niveaux [3]: niveau nœud en InfiniBand avec 16 nœuds, et niveau cœur en Front-Side Bus (FSB) avec 8 cœurs par nœud. Les mesures montrent que si la machine est modélisée en machine BSP plate, on perd à peu près 0,5 nanosecondes par 32 bits pour la communication soit environ 15% de la vitesse transfert de données.

Nous avons testé des variantes de SGL de certains algorithmes parallèles de base les plus importants: réduction parallèle, préfixe parallèle et tri parallèle [3]. Les mesures montrent une correspondance quasi parfaite (erreur moyenne relative $\approx 1\%$) des performances mesurées avec les performances prévues. Les pseudo-codes et les mesures soutiennent fortement nos revendications que SGL est simple à utiliser et que son modèle de performance est fiable.

## 4 Conclusion

Les opérations parallèles de diffusion et contraction (scatter-gather) sont des concepts anciens et bien-compris. SGL les revisite en considérant qu'elles sont les primitives idéales pour correspondre aux architectures haute-performance et multi-niveaux. Les avantages de BSP pour les logiciels parallèles peuvent être renforcés par la structure hiérarchique et hétérogène de la machine SGL, tout en simplifiant la programmation par le remplacement des messages point à point par des communications logiquement centralisées.

Un problème restant ouvert est le traitement implicite de la communication horizontale: il reste déterminer comment programmer et exécuter des algorithmes comme le tri par échantillonage [7]. Nous pensons que la structure simplifiée des communcations en SGL n'est pas seulement nécessaire pour la fiabilité

des logiciels, mais aussi cohérente avec l'un des concepts les plus fondamentaux en informatique soit la récursion.

Les travaux en cours pour SGL visent à développer un compilateur du langage SGL avec LLVM, et optimiser SGL en se basant sur le modèle de prévision de performance et par l'équation fondamentale de modélisation: $T_{total} = T_{comp} + T_{comm} - T_{overlap}$ [1].

## Remerciements

Le premier auteur remercie EXQIM SAS pour une bourse industrielle de doctorat CIFRE. Le deuxième auteur remercie Bill McColl de lui avoir présenté le modèle BSP au début des années 1990. SGL est inspiré par le travail de collègues qui sont trop nombreux à citer ici, mais Frédéric Loulergue et al. mérient une mention pariculière pour leurs travaux sur BSML. Les deux auteurs tiennent à remercier Frédéric Gava pour ses commentaires utiles et EXQIM pour l'accès au matériel SGI sur lequel nous avons mené nos expériences.

## References

1. Kevin J. Barker, Kei Davis, Adolfy Hoisie, Darren J. Kerbyson, Michael Lang, Scott Pakin, and Jose Carlos Sancho. Using performance modeling to design large-scale systems. *Computer*, 42:42–49, 2009.
2. M. Cole. *Algorithmic Skeletons, Structural Management of Parallel Computation*. MIT Press, 1989.
3. C. Li and G. Hains. A simple bridging model for high-performance computing. Rapport interne TR-LACL-2010-12, LACL, Université Paris-Est Créteil (UPEC), December 2010. `http://lacl.u-pec.fr/Rapports/TR /TR-LACL-2010-12.pdf`.
4. F. Loulergue. *Conception de langages fonctionnels pour la programmation massivement parallèle*. doctorat, Université d'Orléans, LIFO, 2000.
5. F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.
6. William F. McColl and David Walker. Theory and algorithms for parallel computation. In David J. Pritchard and Jeff Reeve, editors, *Euro-Par*, volume 1470 of *Lecture Notes in Computer Science*, pages 863–864. Springer, 1998.
7. H. Shi and J. Schaeffer. Parallel sorting by regular sampling. *Journal of Parallel and Distributed Computing*, 14:361–372, 1992.
8. L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, 1990.
9. Leslie G. Valiant. A bridging model for multi-core computing. In *Proceedings of the 16th annual European symposium on Algorithms*, ESA '08, pages 13–28, Berlin, Heidelberg, 2008. Springer-Verlag.
10. Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT Press, Cambridge, MA, USA, 1993.

# Session du groupe de travail LTP

Langages, Types et Preuves

# Secure the Clones * †

## Static Enforcement of Policies for Secure Object Copying

Thomas Jensen, Florent Kirchner, and David Pichardie

INRIA Rennes – Bretagne Atlantique, France
firstname.lastname@inria.fr

**Abstract.** Exchanging mutable data objects with untrusted code is a delicate matter because of the risk of creating a data space that is accessible by an attacker. Consequently, secure programming guidelines for Java stress the importance of using defensive copying before accepting or handing out references to an internal mutable object. However, the implementation of a copy method (like clone()) is entirely left to the programmer. It may not provide a sufficiently deep copy of an object and is subject to overriding by a malicious sub-class. Currently no language-based mechanism supports secure object cloning. This paper proposes a type-based annotation system for defining modular copy policies for class-based object-oriented programs. A copy policy specifies the maximally allowed sharing between an object and its clone. We present a static enforcement mechanism that will guarantee that all classes fulfill their copy policy, even in the presence of overriding of copy methods, and establish the semantic correctness of the overall approach in Coq. The mechanism has been implemented and experimentally evaluated on clone methods from several Java libraries.

Exchanging data objects with untrusted code is a delicate matter because of the risk of creating a data space that is accessible by an attacker. Consequently, secure programming guidelines for Java such as those proposed by Sun and CERT stress the importance of using defensive *copying* or *cloning* before accepting or handing out references to an internal mutable object. There are two aspects of the problem:

1. If the result of a method is a reference to an internal mutable object, then the receiving code may modify the internal state. Therefore, it is recommended to make copies of mutable objects that are returned as results, unless the intention is to share state.
2. If an argument to a method is a reference to an object coming from hostile code, a local copy of the object should be created. Otherwise, the hostile code may be able to modify the internal state of the object.

A common way for a class to provide facilities for copying objects is to implement a `clone()` method that overrides the cloning method provided by `java.lang.Object`. However, relying on calling a polymorphic `clone` method to ensure secure copying of objects may prove insufficient, for two reasons. First, the implementation of the `clone()` method is entirely left to the programmer and there is no way to enforce that an untrusted implementation provides a sufficiently *deep* copy of the object. It is free to leave references to parts of the original object being copied in the new object. Second, even if the current `clone()` method works properly, sub-classes may override the `clone()` method and replace it with a method that does not create a sufficiently deep clone. To quote from the CERT guidelines for secure Java programming: *"Do not carry out defensive copying using the clone() method in constructors, when the (non-system) class can be subclassed by untrusted code. This will limit the malicious code from returning a crafted object when the object's clone() method is invoked."* Clearly, we are faced with a situation where basic object-oriented software engineering principles (sub-classing and overriding) are at odds with security concerns.

To reconcile these two aspects in a manner that provides semantically well-founded guarantees of the resulting code, this paper proposes a formalism for defining *cloning policies* by annotating classes and specific copy methods, and a static enforcement mechanism that will guarantee that all classes of an application adhere to the copy policy. We do not enforce that a copy method will always return a

* This work was supported by the ANSSI, the ANR, and the *Région Bretagne*, respectively under the Javasec, Parsec, and Certlogs projects.

† This paper is a short version of the eponymous article presented to the 20th European Symposium on Programming, held as part of the Joint European Conference on Theory and Practice of Software.

target object that is functionally equivalent to its source. Rather, we ensure non-sharing constraints between source and targets, expressed through a copy policy, as this is the security-critical part of a copy method in a defensive copying scenario.

Our first contribution is a proposal for a set of semantically well-defined program annotations, whose purpose is to enable the expression of policies for secure copying of objects. Introducing a copy policy language enables class developers to state explicitly the intended behavior of copy methods. In the basic form of the copy policy formalism, fields of classes are annotated with `@Shallow` and `@Deep`. Intuitively, the annotation `@Shallow` indicates that the field is referencing an object, parts of which may be referenced from elsewhere. The annotation `@Deep(X)` on a field `f` means that *a*) the object referenced by this field `f` cannot itself be referenced from elsewhere, and *b*) the field `f` is copied according to the copy policy identified by `X`. Here, `X` is either the name of a specific policy or if omitted, it designates the default policy of the class of the field. For example, the following annotations:

```
class List  { @Shallow V value;  @Deep List next;  ...}
```

specifies a default policy for the class `List` where the `next` field points to a list object that also respects the default copy policy for lists. Any method in the `List` class, labelled with the `@Copy` annotation, is meant to respect this default policy.

The second major contribution of our work is to make the developer's intent, expressed by copy policies, statically enforceable using a type system. We formalize this enforcement mechanism by giving an interpretation of the policy language in which annotations are translated into graph-shaped type structures. For example, the annotations of the `List` class defined above will be translated into the graph that is depicted to the right in Fig. 1 (`res` is the name given to the result of the copy method). The left part shows the concrete heap structure, and the finely dotted lines show the correspondences that will be checked by the enforcement mechanism.

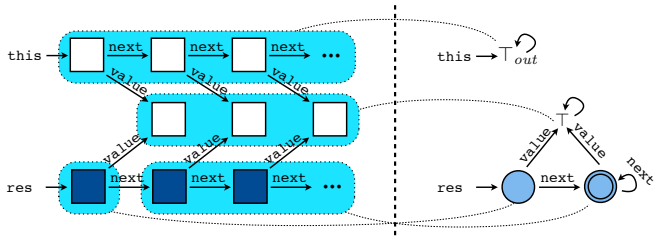



Fig. 1: A linked structure (left part) and its abstraction (right part).

By specializing the shape analysis process, we take into account the programming methodologies and practice for copy methods, and design a type system specifically tailored to the scalable enforcement of copy policies. This means that the underlying analysis must be able to track precisely all modifications to objects that the copy method allocates itself (directly or indirectly) in a flow-sensitive manner. Conversely, as copy methods should not modify non-local objects, the analysis will be designed to be more approximate when tracking objects external to the method under analysis, and the type system will accordingly refuse methods that attempt such non-local modifications. As a further design choice, the annotations are required to be verifiable modularly on a class-by-class basis without having to perform an analysis of the entire code base, and at a reasonable cost.

To conclude, this work constitutes the formal foundations for a secure cloning framework. A large part of the proofs have been mechanized in the Coq proof assistant; both the policy extraction and enforcement components have been implemented and shown to scale up, making use of the Javalib/Sawja static analysis libraries[1] to derive annotations and intermediate code representations. Several issues merit further investigations in order to develop a full-fledged software security verification tool, including the extension of copy annotations to virtual method calls, and of the soundness property to interleaving multi-threading semantics.

[1] `http://sawja.inria.fr`

# A Certified Weakest Precondition Calculus

Paolo Herms[1,2,3], Claude Marché[2,3], and Benjamin Monate[1]

[1] CEA, LIST, Software Safety Laboratory, Gif-sur-Yvette F-91191
[2] INRIA Saclay - Île-de-France, 4 rue Jacques Monod, Orsay, F-91893
[3] Lab. de Recherche en Informatique, Univ Paris-Sud, CNRS, Orsay, F-91405

**Abstract.** Deduction-based software verification tools have reached a maturity level allowing them to be used in industrial context where a very high level of assurance is required. This raises the question of the level of confidence we can grant to the tools themselves.
In this paper we focus on Dijkstra's weakest precondition calculus which is a central component in several verification tools. We develop, in the Coq proof assistant, a version of this calculus, prove it correct and extract from it an executable version.

## 1 Introduction

Formal specification languages allow to express complex properties of the expected behavior of programs. When these are given using an expressive logic, say at least first-order logic, then it is required to apply proof techniques for verifying that a given program meets its specification.

In the case of purely applicative programs, the proof techniques are well established because programs are written in a language close to the logic. Highly expressive logics such as the calculus of inductive constructions, implemented in the Coq proof assistant [5], allow to express programs and proofs in the same language.

For programs involving side-effects, well-established techniques originate from Floyd-Hoare logic [18, 20] and Dijkstra's weakest precondition (abbreviated as WP below) calculus [15]. Nevertheless, these approaches make an implicit assumption, often not well understood: the references (or mutables variables) must not be *aliased*, in other words sharing of references is forbidden. This is illustrated by the following toy example (in the Caml syntax) equipped with a post-condition (in curly brackets):

```
let f (x:int ref) (y:int ref) =
  x := !x + 1; y := !y + 1
  { !x = old(!x) + 1 and !y = old(!y) + 1 }
```

The validity of the post-condition can be established under the assumption that $x$ and $y$ are distinct: a call to `f` of the form

```
let z = ref 0 in f z z
```

must be forbidden, otherwise the post-condition of `f` would say that `z` is incremented by 1 instead of 2. The Why verification condition (abbreviated as VC) generator [16, 17],

computes WPs and handles such a case using a type system which forbids the above call to `f`.

The Why platform incorporates a set of tools to handle source codes written in mainstream programming languages: Krakatoa [17] for Java code and the Jessie plugin [26] of Frama-C [13] for C code. Specifications of such source programs are given as special kind of comments, using existing specification languages such as ACSL [3] or JML [8]. C or Java sources are *compiled* into the Why intermediate language on which VC generation by WP calculus is performed. Transforming input programs, where pointer aliasing is allowed, into the alias-free intermediate language of Why, is performed using a so-called memory model, which typically encodes the operations on the memory heap by functional updates [24]. A very similar approach is available in the Boogie tool [1] which also has several front-ends such as VCC [14] for C code or Spec# [2] for C#. Esc/Java2 [12] follows also such an approach, and the KeY system [4] follows a somewhat similar technique but where updates are represented directly in logic formulas, using the so-called dynamic logic.

Deductive verification systems like those mentioned above gained a lot of efficiency, and thus a lot of interest, in the past decade, partly because of the progress made in efficiency and automation of back-end provers such as SMT solvers. In industry dealing with critical systems, such verification tools can now partly replace testing at lower cost. However, the high complexity of the verification processes involved raises the question of the trust we can have in their implementations. This paper addresses the issue of formally certifying a tool chain of this form, in the Coq proof assistant. The specific goal in this paper is the certification of a weakest precondition calculus, which plays a central role. We want to follow a similar approach as CompCert [6, 7, 21] for building a certified compiler, by extracting trustable executable from a Coq proof.

In Section 2, we give an informal description of the considered input language. Section 3 formalizes this language in Coq, and defines its operational semantics. Section 4 defines the WP computation and proves its soundness: the main result is Theorem 2 stating that if for each function of a program, its pre-condition implies the WP of its post-condition, then all their annotations are satisfied. Section 5 considers the problem of extracting an executable code. We conclude in Section 6. The sources of the underlying Coq development are available as `http://www.lri.fr/~herms/Whycert.tar.gz`.

## 2 Informal Presentation of a Why-style Language

We describe informally the core language on which we want to compute weakest preconditions. We follow the design choices of the input language of Why, that we reduce to an even more basic set of constructs, nevertheless remaining enough expressive to encode any higher-level sequential algorithm. We follow an ML-style syntax, in particular there is no distinction between expressions and instructions: the latter are simply expressions returning the type `unit`. A program in this language is defined by a finite set of global mutable variables called *references*, denoted $r$ below ; a finite set of exceptions names, denoted $ex$, holding a value of a given declared type ; and a finite set of function definitions, denoted $f$ below, which can be mutually recursive.

We assume given a background logic which contains at least multi-sorted first-order logic with equality, and may contain any additional theories such as built-in integer or real arithmetic, or theories axiomatized by declaring sorts, functions and predicates symbols (denoted F below) and axioms. The grammars for terms $t$ and predicates $p$ is

| | |
|---|---|
| $t ::= s$ | logic constant |
| $\mid (F\ t\ \cdots\ t)$ | symbol application |
| $\mid \texttt{let}\ v = t\ \texttt{in}\ t$ | local binding |
| $\mid v$ | local name |
| $\mid\ !\ r$ | dereferencing |
| $\mid r@l$ | dereferencing at label |
| $p ::= t$ | atomic proposition |
| $\mid \texttt{let}\ v = t\ \texttt{in}\ p$ | local binding |
| $\mid \neg p \mid p \wedge p \mid p \vee p \mid p \rightarrow p$ | connectives |
| $\mid \forall v : \tau,\ p \mid \exists v : \tau,\ p$ | quantifications |

The concrete grammar of expression is

| | |
|---|---|
| $e ::= t$ | term |
| $\mid \texttt{let}\ v = e\ \texttt{in}\ e$ | local binding |
| $\mid f(t, ..., t)$ | function call |
| $\mid \texttt{if}\ t\ \texttt{then}\ e\ \texttt{else}\ e$ | conditional branching |
| $\mid \texttt{r:=t}$ | assignment of a reference |
| $\mid \texttt{label}\ l : e$ | labeled expression |
| $\mid \texttt{assert}\ \{p\}$ | local assertion |
| $\mid \texttt{raise}\ ex(t)$ | exception throwing |
| $\mid \texttt{try}\ e\ \texttt{catch}\ ex(v)\ e$ | exception catching |
| $\mid \texttt{loop}\ \{\texttt{invariant}\ p\}\ e$ | infinite loop |

where $v$ and $l$ denote identifiers for local names and local labels respectively. Following again the Why design, our core language contains an exception mechanism, providing powerful control flow structures. However exceptions are not first class values. As we will see these can be handled by weakest pre-condition calculus without major difficulty. Loops are infinite ones, with a given invariant. The only way to exit them is by using exceptions.

A definition of a function follows the structure

$$\texttt{fun}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau = \{\ p\ \}\ e\ \{\ p\ \}$$

where the predicates $p$ are the pre- and the post-condition. The types are those declared in the background logic. In the post-condition, the special reserved name $result$ is locally bounded, and denotes the result of the function, of type $\tau$ ; and the label $old$ is also bounded to the pre-state. Note that exceptions are not supposed to escape function bodies. We could easily support such a feature by adding a family of post-conditions indexed by exceptions as in Why [16].

We use $e; e$ as a shortcut for $\texttt{let}\ v = e\ \texttt{in}\ e$ when the variable $v$ is unused.

*Example 1.* The classical Hoare-logic example for computing the square root of non-negative integers (rounded down) is written as follows in our core language.

```
ref count, sum : int
exception Break(unit)
fun isqrt (x : int) : int =
  { x ≥ 0 }
  count := 0; sum := 1;
  try loop { invariant !count ≥ 0 ∧ x ≥ (sqr !count)∧
                  !sum = (sqr (!count + 1)) }
       if !sum > x then raise Break () else
         (count := !count + 1; sum := !sum + 2 * !count + 1)
  catch Break(v) !count
  { result ≥ 0 ∧ (sqr result) ≤ x ∧ x < (sqr (result + 1)) }
ref tmp : int
fun test () =
  { !tmp ≥ 0 }
  let a = isqrt(42) in assert { a = 6 } ;
  let b = isqrt(!tmp) in tmp := b
  { !tmp ≤ tmp@old }
```

Here the logic is assumed to contain integer arithmetic (with symbols applied in infix syntax for readability) and the $sqr$ function denoting $x \mapsto x^2$.

Note that since we accept only global references and not references local to functions, we forbid reference aliasing, thus avoiding the problem mentioned in introduction. The goal is now to formalize this language in Coq, define its operational semantics, define a WP calculus and show its soundness, so that we could prove that an example like above is correct with respect to its annotations.

## 3 Formalization

A design choice in our formalization is to define terms and expressions such that they are well typed by construction. This simplifies the definition of the semantics and the weakest precondition calculus on such expressions, as we don't need to handle malformed constructions at those points.

As we want terms to be always well-typed, we must begin to define the atoms of the language as well-typed, too.

### 3.1 Dependently Typed de Bruijn Indexes

Since the language involves binders and bound variables, we use the so-called dependently typed de-Bruijn indexes following the preliminary approach of Herms[19] and documented in [10].

Dependent indexes are intended to access elements in heterogeneous list. In such a heterogeneous list each element may have a different type but the type of the heterogeneous list depends on the list of types of its elements. Then we can define dependent indexes such that their type depends on the type of the element they point to and to the list of the types of all elements.

In Coq we can express these constraints with the following inductive definition. The definition constrains the type of the first index to match the first element in the type list and recursively for the other elements.

```
Variable S : Type.
Variable T : S → Type.
Inductive idx A : list S → Type :=
| I0 E : idx A (A :: E)
| IS B E : idx A E → idx A (B :: E).
Inductive hlist : list S → Type :=
| Hnil : hlist []
| Hcons A E : T A → hlist E → hlist (A :: E).
```

Then we can write the function accsidx which given an index and an hlist returns the element in the list pointed by the index. Its type is

```
∀S:Type, ∀T:S → Type, ∀A:S, ∀E:list S,
  lidx A E → hlist T E → T A
```

*Example 2.* We can declare a list of typed symbols as

```
Let l : hlist [nat; bool; nat → nat] :=
   [ 5; true; pred ]%hlist.
```

In environment $l$ we can check the type of de Bruijn indexes:

```
Check I0 : idx nat [nat;bool;nat→nat].
Check IS I0 : idx bool [nat;bool;nat→nat].
Check IS (IS I0) : idx (nat → nat) [nat;bool;nat→nat].
```

and access their values as follows

```
Compute accsidx I0 l = 5 : nat.
Compute accsidx (IS I0) l = true : bool.
Compute accsidx (IS (IS I0)) l = pred : nat → nat.
```

In the formalization of our language we use these indexes and heterogeneous lists to represent local variables and evaluation environments.

### 3.2 Syntax of background logic

The language is generic and depends on certain parameters that are introduced below as needed. The first parameter of the language is $type$. It represents all the possible types of values in our language. We assume it contains at least $tbool$ for booleans, $tunit$ for the singleton type of expressions returning no values, $tprop$ for propositional symbols and $tarrow : type \to type \to type$ for functions. In the following we will use the infix $\rightarrowtail$ for $tarrow$. The second parameter is $sym_A$ representing the symbols of the language. Each symbol $s$ has a given type. For instance if we had a symbol $+$ it would have the type $sym_{int \rightarrowtail int \rightarrowtail int}$. The parameter $ref_A$ represents the global references,

$t_{L,E,A}$ ::= Tconst $sym_A$
Tvar $var_{A,E}$
Tderef $ref_A$
Tapp $t_{E,B\rightarrowtail A}\, t_{E,B}$
Tlet $t_{E,B}\, t_{B::E,A}$
Tat $label_L\, ref_A$

$p_{L,E}$ ::= Peq $t_{A,E}\, t_{A,E}$
Pand $p_{L,E}\, p_{L,E}$
Pimply $p_{L,E}\, p_{L,E}$
Pforall $p_{L,B::E}$
Plet $t_{L,E,B}\, p_{L,B::E}$
Pfalse
Pterm $t_{L,E,tprop}$

**Fig. 1.** Inductive definitions of terms and propositions

where $A$ is the type of the value stored in the state for such a reference. The parameter $exn_A$ represents exceptions that, when raised, carry a value of type $A$.

The syntax of terms and propositions which are used in expressions is defined in Figure 1. Terms $t_{L,E,A}$ and propositions $p_{L,E}$ depend on the parameters $E$ and $L$, denoting respectively the typing environment and the highest index of a valid label. Terms additionally depend on the parameter $A$, the type of the value they denote. Variables are represented by our dependent indexes $var_{A,E} := idx_{A,E}$. The constructor `Tlet` can be used to express let-blocks at the term level. As usual with de Bruijn indexes, no variable name is given and the body of the block is typed in a typing environment that is enriched by the type of the term to be remembered. The symbol application is formalized in a curryfied style.

The constructor `Tat` dereferenciates a reference at a former state pointed by the given label. Labels will be introduced at the expression level.

For the propositions we define only the ones needed within the WP calculus. The constructor `Pterm` allows to define propositions in terms of user defined predicates, i.e $<$,$\vee$. The `Pforall` and the `Plet` bind a new de Bruijn variable.

*Example 3.* To formalize Example 1 we pose the type $integer : type$ and the symbols $ge : sym_{integer\rightarrowtail integer\rightarrowtail tprop}$, $zero : sym_{integer}$ and $sqr : sym_{integer\rightarrowtail integer}$. The representation of the formula $x \geq (sqr\,!count)$, assuming $x$ has index 0, is `Pterm (Tapp (Tapp (Tconst ge) (Tvar v0)) (Tapp (Tconst sqr) (Tderef count)))`

### 3.3 Semantics of background logic

The semantics of our generic language depends on the interpretation given to types and symbols. This is formalized by requiring the parameters $dentype : type \rightarrow Type$ which provides an interpretation for each $type$ in terms of a Coq $Type$ ; $densym : sym_A \rightarrow dentype\ A$ which provides an interpretation for each symbol ; $denarrow : dentype\ (A \rightarrowtail B) \rightarrow dentype\, A \rightarrow dentype\, B$ which transforms an application of symbols to a Coq application ; and $denprop : dentype\, tprop \rightarrow Prop$ which gives the interpretation of $tprop$ a propositional contents.

The semantics of terms and propositions are given under an evaluation environment $\Gamma$ and a state $S$. An evaluation environment $\Gamma$ of type $env_E$ is a heterogeneous lists as defined above where the parameter $S$ is $type$ and $T$ is $dentype$. A memory state $S$ of type $state_L$ is a vector (of size $L$) of mappings from references $ref_A$ to values

$$
\begin{aligned}
[\![Tconst\, s]\!]_{\Gamma,S} &::= densym\, s \\
[\![Tvar\, v]\!]_{\Gamma,S} &::= accsidx\, \Gamma\, v \\
[\![Tderef\, r]\!]_{\Gamma,S} &::= Here\, S\, r \\
[\![Tapp\, t_1\, t_2]\!]_{\Gamma,S} &::= denarrow\, [\![t_1]\!]_{\Gamma,S}\, [\![t_2]\!]_{\Gamma,S} \\
[\![Tlet\, t_1\, t_2]\!]_{\Gamma,S} &::= [\![t_2]\!]_{[\![t_1]\!]_{\Gamma,S}::\Gamma,S} \\
[\![Tat\, l\, r]\!]_{\Gamma,S} &::= At\, S\, l\, r
\end{aligned}
\qquad
\begin{aligned}
[\![Peq\, t_1\, t_2]\!]_{\Gamma,S} &::= [\![t_1]\!]_{\Gamma,S} = [\![t_2]\!]_{\Gamma,S} \\
[\![Pand\, p_1\, p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \wedge [\![p_2]\!]_{\Gamma,S} \\
[\![Pimply\, p_1\, p_2]\!]_{\Gamma,S} &::= [\![p_1]\!]_{\Gamma,S} \to [\![p_2]\!]_{\Gamma,S} \\
[\![Pforall\, p]\!]_{\Gamma,S} &::= \forall b : B,\, [\![p]\!]_{b::\Gamma,S} \\
[\![Plet\, t\, p]\!]_{\Gamma,S} &::= [\![p]\!]_{[\![t]\!]_{\Gamma,S}::\Gamma,S} \\
[\![Pfalse]\!]_{\Gamma,S} &::= \bot \\
[\![Pterm\, t]\!]_{\Gamma,S} &::= denprop\, [\![t]\!]_{\Gamma,S}
\end{aligned}
$$

**Fig. 2.** Denotational semantics of terms and propositions

$$
\begin{aligned}
e_{L,E,\_} ::= {} & \text{Eterm } t_{L,E,A} : e_{L,E,A} \\
\mid {} & \text{Elet } e_{L,E,B}\ e_{L,B::E,A} : e_{L,E,A} \\
\mid {} & \text{Eassign } ref_A\ t_{L,E,A} : e_{L,E,tunit} \\
\mid {} & \text{Eassert } p_{L,E} : e_{L,E,tunit} \\
\mid {} & \text{Eraise } ex_{Aex}\ t_{L,E,Aex} : e_{L,E,tunit} \\
\mid {} & \text{Eif } t_{L,E,A}\ e_{L,E,A}\ e_{L,E,A} : e_{L,E,A} \\
\mid {} & \text{Eloop } p_{L,E,A}\ e_{L,E,B} : e_{L,E,tunit} \\
\mid {} & \text{Etry } e_{L,E,A}\ ex_{Aex}\ e_{L,Aex::E,A} : e_{L,E,A} \\
\mid {} & \text{Elab } e_{L+1,E,A} : e_{L,E,A} \\
\mid {} & \text{Ecall } f_{A,P}\ (t_{L,E,P_1}, ..., t_{L,E,P_n}) : e_{L,E,A}
\end{aligned}
$$

**Fig. 3.** Inductive definition of expressions

of type $dentype\, A$. The first element denotes the current state whereas the $(l+1)$-nth element denotes state labeled by $l$. We define the shortcuts $Here\ S = S[0]$ and $At\ S\ l = S[l+1]$. Then the update operation $S[r/a]$ simply replaces the topmost mapping for $r$.

The semantics of terms is defined by structural recursion (Figure 2). As correct typing is ensured by construction its definition is straightforward.

### 3.4 Syntax of expressions

Expressions are defined in Figure 3. Like terms, expressions $e_{L,E,A}$ depend on the parameters $A$, $E$ and $L$, denoting respectively the evaluation type, the typing environment and the highest index of a valid label. Unlike terms, the parameter $A$ changes between constructors – it is for Coq not a parameter but an annotation of the inductive type. We give assignments, assertions, exception raisings and loops the evaluation type $tunit$. The types of the other constructions depend on their contents.

Additionally expressions depend on the parameter $F$ meaning the list of signatures of the functions in the program the expression can appear in. We will usually omit this parameter as clearly it doesn't change within a program. A signature is a pair of the return type of the function and the list of the function's parameters. $F$ appears within expressions in function calls where we use dependent indexes to refer to functions, $f_{A,P} := idx_{\langle A,P\rangle,F}$. A function identifier is therefore an index pointing to an element with the signature $\langle A, P\rangle$ within a heterogeneous list of types $F$. This heterogeneous

list is precisely the representation of a program $prog_F := hlist_{func\,F,F}$, where each element is a function $func_{F,\langle A,P\rangle}$.

A function $func_{F,\langle A,P\rangle}$ consists of a body $e_{F,1,E,A}$, a pre-condition $p_{0,P}$ and a post-condition $p_{1,A::P}$. In the latter, the label index is 1 to represent $old$ and the $A$ in environment is to denote the type of the implicitly bound $result$. Note that in the definition of programs the parameter $F$ appears twice: once as parameter of $hlist$, to define the signatures of the functions in the program, and once as parameter of $func$ to constrain expressions in function bodies to refer only to functions with a signature appearing in $F$. This way we ensure the well-formedness of the graph structure of programs.

### 3.5 Operational Semantics

The operational semantics is defined in big-step style following the approach of Leroy and Grall [22]. A first set of inference rules inductively defines the semantics of termi-

$$\frac{}{\Gamma, S, t \Rightarrow S, [\![t]\!]_{\Gamma,S}}$$

$$\frac{\Gamma, S, e_1 \Rightarrow S', v \quad v :: \Gamma, S', e_2 \Rightarrow S'', o}{\Gamma, S, \texttt{let}\, e_1\, \texttt{in}\, e_2 \Rightarrow S'', o}$$

$$\frac{\Gamma, S, e_1 \Rightarrow S', ex\,(v)}{\Gamma, S, \texttt{let}\, e_1\, \texttt{in}\, e_2 \Rightarrow S', ex\,(v)}$$

$$\frac{}{\Gamma, S, r := t \Rightarrow S\,[r/[\![t]\!]_{\Gamma,S}]\,, [\![t]\!]_{\Gamma,S}}$$

$$\frac{[\![p]\!]_{\Gamma,S}\ \mathsf{valid}}{\Gamma, S, assert\{p\} \Rightarrow S, ()}$$

$$\frac{}{\Gamma, S, \texttt{raise}\, ex\, t \Rightarrow S, ex\,([\![t]\!]_{\Gamma,S})}$$

$$\frac{\Gamma, S, \mathsf{if}\ [\![t]\!]_{\Gamma,S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Rightarrow S', o}{\Gamma, S, \texttt{if}\, t\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Rightarrow S', o}$$

$$\frac{[\![p]\!]_{\Gamma,S}\ \mathsf{valid} \quad S, e \Rightarrow S', v \quad S', \texttt{loop}\,\{\texttt{invariant} : p\}\, e \Rightarrow S'', o}{S, \texttt{loop}\,\{\texttt{invariant} : p\}\, e \Rightarrow S'', o}$$

$$\frac{[\![p]\!]_{\Gamma,S}\ \mathsf{valid} \quad S, e \Rightarrow S', ex\,(v)}{S, \texttt{loop}\,\{\texttt{invariant} : p\}\, e \Rightarrow S', ex\,(v)}$$

$$\frac{S, e_1 \Rightarrow S', o \quad o \neq ex}{S, \texttt{try}\, e_1\, \texttt{catch}\, ex\, \texttt{in}\, e_2 \Rightarrow S', o}$$

$$\frac{S, e_1 \Rightarrow S', ex\,(v) \quad v :: \Gamma, S', e_2 \Rightarrow S'', o}{S, \texttt{try}\, e_1\, \texttt{catch}\, ex\, \texttt{in}\, e_2 \Rightarrow S'', o}$$

$$\frac{\Gamma,\ S\!\uparrow, e \Rightarrow S', o}{\Gamma, S, \texttt{label}\, e \Rightarrow\ S'\!\downarrow, o}$$

$$\frac{[\![\mathsf{pre}_f]\!]_{\Gamma_f,S}\ \mathsf{valid} \quad \Gamma_f, S, \mathsf{body}_f \Rightarrow S', v \quad [\![\mathsf{post}_f]\!]_{v::\Gamma_f,S'}\ \mathsf{valid}}{\Gamma, S, f\ (t_1, ..., t_n) \Rightarrow S', v}\ \Gamma_f := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}]$$

**Fig. 4.** Operational semantics of terminating expressions

$$\frac{\Gamma, S, e_1 \Rightarrow \infty}{\Gamma, S, \texttt{let}\, e_1\, \texttt{in}\, e_2 \Rightarrow \infty}$$

$$\frac{\Gamma, S, e_1 \Rightarrow S', v \quad v :: \Gamma, S', e_2 \Rightarrow \infty}{\Gamma, S, \texttt{let}\, e_1\, \texttt{in}\, e_2 \Rightarrow \infty}$$

$$\frac{\Gamma, S, \mathsf{if}\, [\![t]\!]_{\Gamma,S}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Rightarrow \infty}{\Gamma, S, \texttt{if}\, t\, \texttt{then}\, e_1\, \texttt{else}\, e_2 \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S}\, \mathsf{valid} \quad S, e \Rightarrow \infty}{S, \texttt{loop}\, \{\texttt{invariant} : p\}\, e \Rightarrow \infty}$$

$$\frac{[\![p]\!]_{\Gamma,S}\, \mathsf{valid} \quad S, e \Rightarrow S', v \quad S', \texttt{loop}\, \{\texttt{invariant} : p\}\, e \Rightarrow \infty}{S, \texttt{loop}\, \{\texttt{invariant} : p\}\, e \Rightarrow \infty}$$

$$\frac{S, e_1 \Rightarrow \infty}{S, \texttt{try}\, e_1\, \texttt{catch}\, ex\, \texttt{in}\, e_2 \Rightarrow \infty}$$

$$\frac{S, e_1 \Rightarrow S', ex\,(v) \quad v :: \Gamma, S', e_2 \Rightarrow \infty}{S, \texttt{try}\, e_1\, \texttt{catch}\, ex\, \texttt{in}\, e_2 \Rightarrow \infty}$$

$$\frac{\Gamma,\, S{\uparrow}, e \Rightarrow \infty}{\Gamma, S, \texttt{label}\, e \Rightarrow \infty}$$

$$\frac{[\![\mathsf{pre}_f]\!]_{\Gamma_f,S}\, \mathsf{valid} \quad \Gamma_f, S, \mathsf{body}_f \Rightarrow \infty}{\Gamma, S, f\ (t_1, ..., t_n) \Rightarrow \infty}\ \Gamma_f := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}]$$

**Fig. 5.** Operational semantics of non-terminating expressions

nating expressions (Figure 4) and a second set defines the semantics of non-terminating expressions, co-inductively (Figure 5). Judgement $\Gamma, S, e \Rightarrow S', o$ expresses that in environment $\Gamma$ and state $S$, the execution of expression $e$ terminates, in a state $S'$ with outcome $o$. The outcome is either a normal value $v$ or an exception $ex(v)$ where $v$ is the value hold by it. There are two rules for $let\, v = e_1 in\, e_2$ depending on the outcome of $e_1$. The rule for assignment use the update function on state defined in Section 3.3. A labeled expression is evaluated in an enriched state $S{\uparrow}$ where the current state is copied on top of the vector. The resulting state $S'{\downarrow}$ is obtained by deleting second position of the vector what corresponds to "forget" the previously copied current state. The rule for function call requires the pre-condition to be valid in the starting state and, if the function terminates normally, the validity of the post-condition in the returning state to be valid too. The execution will block if the body exits in exception (because no rule is given).

The main feature to notice is that execution blocks whenever an invalid assertion is met: the rules for assertions, loops and function calls checks the annotations. In other words, a program respects its annotations if it has a semantics by the given rules: we take this as the definition of "A program respects its annotation".

# 4 Weakest precondition calculus

## 4.1 Effect inference

To carry out the weakest precondition calculus we need to know for each expression its *effect*, i.e. the references it may modify. The only expression to modify a reference is the assignment $r := t$, we just need to collect all the assignments in the sub-expressions and this could be done by structural recursion if wasn't for the function calls. An expression that calls a function propagates the modifications on all the references that may be modified by the function body, which itself may call other functions.

Since the functions can be mutually recursive, we compute their effects by iterating a function `iter` collecting additional effects, until it reaches a fix-point. First we define the types

```
Definition rset := (set (sigT ref)).
Definition effects := list rset.
```

denoting respectively a set of references (as the type of a reference $ref_A$ is indexed by its type $A$, in Coq we use an existential type inside the set: $set\ \exists A, ref_A$) and an environment of effects for each function, indexed by their de Bruijn indexes.

Then, the function `writes` computes the effect of a given expression e assuming given as argument the environment $\epsilon$ of function effects. It is defined by structural recursion on e.

```
Fixpoint writes (ϵ:effects) L E A (e:expr L E A) : rset :=
match e with
| Eterm t | Eassert p | Eraise ex t => empty
| Eassign r t => singleton r
| Eloop inv e | Elab l e =>  writes e
| Elet e1 e2 | Eif t e1 e2 | Etry e1 ex e2 =>
    union (writes e1) (writes e2)
| Ecall i => accslist i ϵ
end.
```

The iteration for computing the correct effects of function is then done as follows:

```
Definition iter (ϵ:effects) : effects :=
  map (fun f, writes ϵ f.(body)) p.
Program Fixpoint infer_n e { measure ... } :=
  let e' := iter e in
  if e == e' then e else infer_n e'.
Definition infer := infer_n empty_effects.
```

where p denotes the list of functions of our program. Function `infer_n` is defined by recursion with respect to a measure which allows to prove the termination: the number of references appearing in the program is finite and at each iteration we add some of them to some set in the list.

$$
\begin{aligned}
WP\,(t, Q; R)\ \Gamma &= \lambda S,\ Q\,S\,[\![t]\!]_{\Gamma,S} \\
WP\,(\texttt{let}\,e_1\,\texttt{in}\,e_2, Q; R)\ \Gamma &= WP\,e_1\ (\lambda S\,a,\ WP\,e_2\,Q\,R\ (a :: \Gamma)\ S)\ R\,\Gamma \\
WP\,(r := t, Q; R)\ \Gamma &= \lambda S,\ Q\ (S\,[r/[\![t]\!]_{\Gamma,S}])\ () \\
WP\,(\texttt{assert}\,p, Q; R)\ \Gamma &= \lambda S,\ [\![p]\!]_{\Gamma,S} \wedge Q\,S\,() \\
WP\,(\texttt{raise}\,ex\,(t), Q; R)\ \Gamma &= \lambda S,\ R\,S\,ex\,[\![t]\!]_{\Gamma,S} \\
WP\,(\texttt{if}\,t\,\texttt{then}\,e_1\,\texttt{else}\,e_2, Q; R)\ \Gamma &= WP\ (\mathsf{if}\ [\![t]\!]_{\Gamma,S}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2)\ Q\,R\,\Gamma \\
WP\,(\texttt{loop}\,\{\texttt{invariant}:p\}\,e, Q; R)\ \Gamma &= \lambda S,\ [\![p]\!]_{\Gamma,S} \wedge \forall S', S \overset{e}{\rightsquigarrow} S' \implies [\![p]\!]_{\Gamma,S'} \\
&\quad \implies WP\,e\ (\lambda S''\,(),\ [\![p]\!]_{\Gamma,S''})\ R\,\Gamma\,S' \\
WP\,(\texttt{try}\,e_1\,\texttt{catch}\,ex\,\texttt{in}\,e_2, Q; R)\ \Gamma &= WP\,e_1\,Q\ (\lambda S'\,ex'\,a,\ \mathsf{if}\ ex = ex' \\
&\quad \mathsf{then}\ WP\,e_2\,Q\,R\ (a :: \Gamma)\ S'\ \mathsf{else}\ R\,ex'\,a)\ \Gamma \\
WP\,(\texttt{label}\,e, Q; R)\ \Gamma &= \lambda S,\ WP\,e\,Q\,R\,\Gamma\ \ S\uparrow \\
WP\,(f\ (t_1, ..., t_n), Q; R)\ \Gamma &= \lambda S,\ [\![pre_f]\!]_{\Gamma_{args},S} \wedge \forall S'\,a, S \overset{body_f}{\rightsquigarrow} S' \\
&\quad \implies [\![post_f]\!]_{(a::\Gamma_{args}),(S',S)} \implies Q\,S'\,a \\
&\quad\quad , \Gamma_{args} := [[\![t_1]\!]_{\Gamma,S}, ..., [\![t_n]\!]_{\Gamma,S}]
\end{aligned}
$$

**Fig. 6.** Definition of WP-calculus

**Lemma 1 (soundness of effects inference).**

```
forall P A (i:lidx (signature A P) F),
  accslist i (infer p) === writes (accsfunc i).(body).
```

Finally, we can now define the predicate $assigns$ that relates two states that are different only in the references appearing in a given set and prove that it is correct with respect to the semantics of expressions.

```
Definition assigns L (S:state L) (ϵ: rset) (S':state L):=
  forall A (r:ref A),
    (¬ In (A&r) ϵ → Here S' r = Here S r)
    ∧ ∀l, At l S' r = At l S r.
```

**Lemma 2 (assigns correct).** *If* $\Gamma, S, e \Rightarrow S', o$ *then* $\rightarrow S \overset{e}{\rightsquigarrow} S'$ *where* $S \overset{e}{\rightsquigarrow} S'$ *means* $assigns\,S\,(writes\,e)\,S'$.

### 4.2 Definition of WP-calculus

We can calculate the WP of an expression given a post-condition by structural recursion over expressions (Figure 6). We admit several post-conditions, $NOP_{L,A} : state_L \rightarrow dentype_A \rightarrow Prop$ for regular execution and $EOP_L : state_L \rightarrow \forall Aex, exn_{Aex} \rightarrow dentype_{Aex} \rightarrow Prop$ for exceptional behavior. So our calculus has the type $WP : e_{L,E,A} \rightarrow NOP_{L,A} \rightarrow EOP_L \rightarrow state_L \rightarrow Prop$.

In the case of a loop, the precondition is calculated using the loop invariant and in the case of a function call we use the pre- and post-condition of that function. In both cases we quantify over all states that may be reached by normal execution starting from the given state. Here we take profit of our assigns predicate $S \overset{e}{\rightsquigarrow} S'$.

**Definition 1.** *The set of verification conditions of a program is*

$$\begin{aligned} \mathit{VCGEN} := \forall f : idx_{\langle A,P\rangle,F}\, \Gamma\, S, \\ [\![pre_f]\!]_{\Gamma,S} \implies \mathit{WP}\, body_f\ (\lambda S'\, v, [\![post_f]\!]_{v::\Gamma,S'})\ False\, \Gamma\, S \end{aligned}$$

*The False as exceptional post-conditions imposes to show that no function body exits in exception.*

### 4.3 Soundness Results

A first property is that after a terminating execution, the post-conditions are respected if the WP is valid.

**Lemma 3.** *for all environment $\Gamma$, expression $e_{L,E,A}$, $R_L$ initial and final states $S$, $S'$ outcome $o$ and post-conditions $Q_{L.A}$, $R_L$, if $(WP\, e\, Q\, R\, S)$ and $\Gamma, S, e \Rightarrow S', o$ then (1) if $o$ is a normal outcome $v$ then $(Q\, S'\, v)$, and (2) if $o$ is an exceptional outcome $ex(v)$ then $(R\, S'\, ex\, v)$*

*Proof.* Induction over the derivation of $\Gamma, S, e \Rightarrow S', o$.

The main theorem states that if the VCs hold for all functions then any expression having a valid WP safely execute, that is either it terminates or it loops forever.

**Theorem 1 (soundness).** *If VCGEN is valid then for any $\Gamma, S$, if $(WP\, e\, Q\, R\, \Gamma\, S)$ then either $\Gamma, S, e \Rightarrow \infty$ or $\exists S', o,\ \Gamma, S, e \Rightarrow S', o$*

*Proof.* By co-induction, using the axiom of excluded middle to distinguish whether the execution of an expression does or does not terminate, following the guidelines of [22].

Notice that this requires to proof the verification conditions for each function separately, even if all functions can be mutually recursive – no circular reasoning is required.

The important corollary below states that if the VCs hold for all functions then any their bodies safely execute, that is either they terminate or loop forever. By definition of the semantics, this implies that all assertions, invariants and pre- and post-conditions in a given program are verified if the verification conditions are valid.

**Theorem 2.** *If VCGEN is valid then for any $\Gamma, S$, if $[\![pre_f]\!]_{\Gamma,S}$ then either $\Gamma, S, body_f \Rightarrow \infty$ or exist $S', v$ such that $\Gamma, S, e \Rightarrow S', v$ and $[\![post_f]\!]_{v::\Gamma,S'}$*

*Proof.* Corollary of the lemma and the theorem above.

## 5 Extraction of a certified verification tool

The obtained generator of verification conditions is not extractable. Given a program $prog_F$ we obtain a Coq term *VCGEN* of type $Prop$ which must be proven valid to show the correctness of the program. The process thus remains based on Coq for making the proofs. In this section we show how to extract the calculus to into a separate tool so that proofs could be performed with other provers, e.g. SMT solvers.

### 5.1 Concrete WP computation

To achieve this we need the WP calculus to produce a formula in abstract syntax instead of a Coq $Prop$. We define another function

$$wp : e_{L,E,A} \to p_{L,A::E} \to (exn_{Aex} \to p_{L,Aex::E}) \to prop_{L,E}$$

which, given an expression $e$, a proposition $Q$ about the variables in $e$ and the result and a family of propositions $R$, one for each exception about the variables in $e$ and the value carried by the exception, returns a proposition about the variables in $e$ with the following property:

**Theorem 3.** *if* $[\![wp\, e\, Q\, R]\!]_{\Gamma,S}$ *then* $WP\ e\ (\lambda S\, v, [\![Q]\!]_{v::\Gamma,S})\ (\lambda S\, ex\, v, [\![R\, ex]\!]_{v::\Gamma,S})\ \Gamma\ S$

*Proof.* By induction over the expression, where in each case the required commutation lemmas are applied.

With the help of this function we can define a concrete verification-condition generator.

**Definition 2.** $vcgen\ f := abstrv\ (Pimply\, pre_f\ (wp\, body_f\, post_f\ Pfalse))$
*where* $abstrv$ *is a function that generalizes all the references in a given proposition*

**Theorem 4.** *If* $(\forall f, [\![vcgen\ f]\!]_{[],[]})$ *then VCGEN*

*Proof.* Corollary of the theorem above and by commutation of $abstrv$ with quantification over states.

That is, we are in the hypothesis of the soundness theorem, if we can prove valid the formulas generated by $vcgen$ for all the functions in the program.

### 5.2 Producing concrete syntax with explicit binders

Still, these formulas are represented by a de Bruijn-style abstract syntax. To print out such formulas we need to transform them into a concrete syntax with identifiers for variables by generating new names for all the binders. This can be done on the fly in an unproven pretty-printer but as a non trivial transformation it is better to do it in a certified way directly after the generation.

We therefore formalized a front-end syntax, similar to the one informally presented in section 2, along with its semantics for well-typed terms and propositions. We then defined a compilation from de Bruijn-style terms and propositions to the front-end syntax and proved preservation of semantics.

### 5.3 Extraction and experimentation

For experimentation purposes we also defined a compilation in the opposite direction, i.e. from programs in front-end syntax to the corresponding program in de Bruijn syntax, provided that the former is well typed.

We use the extraction mechanism of Coq to extract an Ocaml function that, given an AST of our front-end syntax representing a program, produces a list of ASTs representing the VCs for the program.

Since we do not yet have a front-end with concrete syntax for programs, we need to manually enter the AST as Ocaml code. As an example we coded the Ocaml representation of the isqrt function of Example 1. We finally combine the VC generator with a hand-written pretty printer that produces Why3 syntax (the new release of Why). This allows us to call automated provers on it, and on the isqrt example the VCs are proved automatically (See the Coq development at the URL given in introduction).

## 6 Conclusions and Future Work

We formalized a core language for deductive verification of imperative programs. Its operational semantics is defined co-inductively so that we can accept possibly non-terminating functions. The annotations are taken into account in the semantics so that validity of a program with respect to its annotations is by definition the progress of its execution. We used an original formalization of binders so that only well-typed programs can be considered, allowing to simplify to rest of the formalization.

The weakest precondition calculus is defined by a structurally recursive function, assuming the given function contracts. Even if there is an apparent cyclic reasoning, this approach is shown sound by a co-inductive proof.

By additionally formalizing abstract syntax for terms and formulas, and relating their semantics with respect to the Coq proposition, we defined a concrete variant of the WP calculus which can be extracted to OCaml code, thus obtaining a trustable and executable VC generator close to Why or Boogie.

Our work provides a way of proving imperative programs in a certified way. Other approaches for dealing with imperative programs in Coq include the Ynot approach [27, 11] and the CFML approach [9]. Other approaches exist in other systems like in Isabelle/HOL [25]. They can accept more expressive programs as input, e.g. including higher-order functions, but on the other hand the proofs must be made inside the underlying proof assistant, whereas our approach provides a certified tool independent of Coq, and proofs can be made by automatic provers. Of course, in the latter case another part of the certification of the tool chain is the certification of automatic provers, for which good progress was obtain in the recent years, see e.g. [23].

Future work is to certify the remaining part of a complete chain from C programs to proof obligations. A first step is the formalization of a front-end like Frama-C/Jessie which compiles annotated C to intermediate Why code. We plan to reuse the C semantics defined in CompCert [21] and incorporating ACSL [3] annotations into it. The main issue in this compilation process is the representation of the C memory heap by Why global references using a memory heap modeling.

## References


1. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
3. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.
4. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.
5. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
6. S. Blazy, Z. Dargaye, and X. Leroy. Formal verification of a C compiler front-end. In *Proceedings of Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, 2006.
7. S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.
8. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer*, 2004.
9. A. Charguéraud. *Characteristic Formulae for Mechanized Program Verification*. PhD thesis, Université Paris 7, 2010. `http://www.charglueraud.org/arthur/research/2010/thesis/`.
10. A. Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.
11. A. Chlipala, G. Malecha, G. Morrisett, A. Shinnar, and R. Wisnesky. Effective interactive proofs for higher-order imperative programs. In *Proceedings of ICFP'09*, 2009.
12. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
13. P. Cuoq, J. Signoles, P. Baudin, R. Bonichon, G. Canet, L. Correnson, B. Monate, V. Prevosto, and A. Puccetti. Experience report: Ocaml for an industrial-strength static analysis framework. In *Proceedings of the 14th ACM SIGPLAN international conference on Functional programming (2009)*, pages 281–286, 2009.
14. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.
15. E. W. Dijkstra. *A discipline of programming*. Series in Automatic Computation. Prentice Hall Int., 1976.
16. J.-C. Filliâtre. Verification of non-functional programs using interpretations in type theory. *Journal of Functional Programming*, 13(4):709–745, July 2003.
17. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.

18. R. W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, *Mathematical Aspects of Computer Science*, volume 19 of *Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.
19. P. Herms. Certification of a chain for deductive program verification. In Y. Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, 2010.
20. C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580 and 583, Oct. 1969.
21. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
22. X. Leroy and H. Grall. Coinductive big-step operational semantics. *Inf. Comput.*, 207:284–304, February 2009.
23. S. Lescuyer. *Formalisation et développement d'une tactique réflexive pour la démonstration automatique en Coq*. Thèse de doctorat, Université Paris-Sud, 2011.
24. C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics*, volume 3603 of *Lecture Notes in Computer Science*, pages 179–194. Springer, Aug. 2005.
25. F. Mehta and T. Nipkow. Proving pointer programs in higher-order logic. In F. Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer, 2003.
26. Y. Moy and C. Marché. *Jessie Plugin Tutorial,* Beryllium *version*. INRIA, 2009. `http://www.frama-c.cea.fr/jessie.html`.
27. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.

# Preuves Formelles de Systèmes de Calculs Locaux

Pierre Castéran, Vincent Filou, and Allyx Fontaine

Université Bordeaux 1, LaBRI, UMR 5800
Bât A30, 351 Cours de la Libération, 33405 Talence cedex
pierre.casteran@labri.fr, vincent.filou@labri.fr, allyx.fontaine@labri.fr

Les algorithmes distribués sont ardus à concevoir et encore plus à vérifier, notamment quand des processus qui n'ont qu'une connaissance *locale* de la situation (l'état de leurs voisins par exemple) doivent tendre vers une propriété *globale* (c'est-à-dire portant sur tout le réseau) comme l'élection d'un nœud, une tolérance aux pannes, la terminaison, etc. Il est par exemple très difficile de se convaincre de la correction d'algorithmes comme celui de Mazurkiewicz construisant une énumération des sommets d'un graphe.

La complexité des algorithmes distribués rend nécessaire la définition et la mise en œuvre d'outils qui assistent le concepteur dans le développement et la vérification de la correction de ses programmes, avec le plus d'automatisation possible. Un moyen d'atteindre un niveau de confiance extrêmement élevé est de considérer l'adaptation au cadre distribué de techniques de preuve formelle de programmes séquentiels.

Parmi les modèles de calculs distribués, les *systèmes de calculs locaux* offrent un cadre général permettant d'exprimer de nombreux problèmes de l'algorithmique distribuée : choix de primitives de communication, détection de la terminaison des calculs par les processeurs, influence de l'état initial des processeurs, etc.

Dans ce modèle, un réseau est représenté par un graphe fini, ses processeurs par les sommets de ce graphe, et les liens de communication par ses arêtes. L'état local d'un processeur (resp. lien) est représenté par une étiquette attachée au sommet (resp. à l'arête) correspondant. L'exécution d'un algorithme se présente sous la forme d'une suite d'interactions locales : chaque processeur exécute de façon asynchrone le même code et ne peut interagir qu'avec ses voisins immédiats. Nous considérons dans cette étude des réseaux statiques, où les interactions s'expriment sous forme de changement d'état des processeurs et des liens, et non d'une modification de la structure du réseau.

Nous décrivons dans cet article une formalisation de la théorie des calculs locaux, sous forme d'une bibliothèque pour l'assistant à la preuve *Coq*. Une des premières utilisations de cette bibliothèque est l'obtention de preuves formelles d'algorithmes distribués. L'étude de la littérature sur les algorithmes distribués, et plus particulièrement les systèmes de calculs locaux, suggère de s'intéresser à d'autres propriétés que la simple correction d'algorithmes, par exemple l'impossibilité de réaliser certaines spécifications avec telle ou telle sous-classe d'algorithmes, ou la validation de transformations d'algorithmes.

**Le Modèle des Calculs Locaux** Nous présentons les notions principales du modèle des calculs locaux adaptées de Métivier *et al.*

Nous considérons des réalisations de tâches par des suites de *réétiquetages locaux*, c'est-à-dire de modifications de l'étiquetage du graphe limitées au voisinage immédiat d'un sommet. Nous supposons de plus que l'applicabilité d'un tel pas de réétiquetage ne dépend que de l'état local de ce voisinage et non de l'état du reste du graphe. Afin de rendre les réétiquetages indépendants de la représentation concrète des graphes, nous imposons que chaque réétiquetage commute avec tout isomorphisme de graphes étiquetés. Les systèmes de réétiquetage satisfaisant ces conditions sont appelés *systèmes localement engendrés*. Ce formalisme est bien adapté à la spécification de problèmes classiques en algorithmique distribuée, en décrivant les tâches associées à l'élection d'un sommet dans un graphe quelconque, à l'élection dans un arbre avec ou sans connaissance initiale du degré des sommets, au calcul d'arbre recouvrant, etc.

Les systèmes considérés sont définis à l'aide de *règles de réétiquetage* décrivant l'interaction d'un sommet du graphe avec ses voisins immédiats. Suivant Bauderon *et al.*, nous classifions les

règles de réétiquetage selon l'ensemble des étiquettes pouvant être prises en compte et/ou modifiés lors d'une interaction locale.

- Une règle **LC0** opère sur deux sommets adjacents du graphe considéré. Cela correspond à un *rendez-vous* entre ces deux sommets. Ce type de calcul correspond aussi au modèle client-serveur de nombreuses applications réseau.
- Dans une règle **LC1**, toutes les étiquettes d'un sommet $c$, de ses voisins et des arêtes reliant $c$ à ses voisins sont prises en compte pour décider si la règle est applicable. En revanche, seules les étiquettes du centre $c$ et des arêtes reliant $c$ à ses voisins peuvent être modifiées.

**Modes de Détection de la Terminaison** La spécification d'un calcul distribué par une tâche ne concerne que la relation entre données et résultats d'une exécution. Or certaines spécifications d'algorithmes distribués précisent des propriétés d'états intermédiaires, comme par exemple la connaissance que chaque sommet-processeur peut avoir de l'état de terminaison d'une exécution. On peut ainsi spécifier que dans un processus d'élection tout processeur puisse savoir s'il est définitivement élu ou battu (*détection locale de la terminaison*). De même on peut spécifier que tout processeur doit finir par savoir si un autre sommet est déjà élu (*détection observée de la terminaison*). Des *modes de détection de la terminaison* peuvent ainsi être définis de façon générique et s'appliquent à n'importe quelle spécification exprimée sous forme de tâche globale. Nous pouvons considérer ces modes comme des modificateurs de spécifications.

**Formalisation en *Coq*** La formalisation en *Coq* de la théorie des calculs locaux[1] s'appuie sur une description des systèmes de réétiquetage de graphes comportant les traits suivants :

- Bibliothèque sur les graphes, et les graphes étiquetés.
- Formalisation de la théorie des calculs locaux ; classification des systèmes de calculs locaux selon les types de règles de réétiquetage et les modes de détection de la terminaison.

Nous avons ainsi déjà pu obtenir deux types de résultats :

- Des transformations certifiées de systèmes localement engendrés servant de conversions entre diverses classes de règles et de modes de détection de la terminaison,
- Des invariants portant sur toute une classe de systèmes ou de règles, et utilisés dans des preuves de l'impossibilité de réaliser certaines tâches.

Nous avons en outre formellement prouvé la correction de plusieurs algorithmes distribués : élection, calcul d'arbre recouvrant, calcul du degré des sommets d'un graphe, avec des variantes portant sur le type de règle utilisée et sur le mode de détection de la terminaison choisi. La correction se prouve par l'intermédiaire d'invariants, et la terminaison par des variants définis sur un ordre bien fondé.

Par ailleurs, nous avons prouvé l'impossibilité de réaliser certaines spécifications, comme par exemple l'élection d'un sommet dans un arbre non étiqueté avec détection locale de la terminaison, par des règles de type **LC0**. Cette preuve est un exemple d'application directe d'invariants de la classe de règles **LC0**.

**Conclusion** Le modèle des calculs locaux permet de traiter dans un formalisme simple plusieurs notions importantes en algorithmique distribuée : répartition des données initiales, types de synchronisation, détection de la terminaison (locale ou globale) des calculs. Loin d'être indépendantes, ces notions interagissent fortement et conditionnent la réalisabilité de tâches. L'environnement que nous construisons permet d'écrire dans un cadre unique des spécifications, d'en certifier formellement des réalisations, mais aussi de prouver et d'appliquer des résultats abstraits, pour déterminer par quels types d'algorithmes elles sont ou non réalisables.

[1] `www.labri.fr/~casteran/Loco/`

# Session du groupe de travail MFDL

Méthodes Formelles dans le Développement Logiciel

# Assistance à la conception de modèles à l'aide de contraintes

Marie de Roquemaurel[1], Thomas Polacsek[2], Jean-François Rolland[1], Jean-Paul Bodeveix[1], and Mamoun Filali[1]

[1] IRIT CNRS, Université de Toulouse,
118 Route de Narbonne, F-31062 Toulouse Cedex 9
firstname[DOT]lastname[AT]irit[DOT]fr,
[2] ONERA, Toulouse,
BP 4025 - 31055 Toulouse Cedex 4
Firstname[DOT]Lastname[AT]onera[DOT]fr

**Résumé** La conception système est une des premières étapes du processus de conception dite amont. Lors de cette étape, il s'agit de raisonner à un haut niveau de granularité (en faisant abstaction du code) en terme des fonctionnalités à implanter et de l'architecture cible. Pour assister cette phase de conception, nous proposons une approche basée sur les techniques de l'IDM. Le passage à l'échelle de cette approche est envisagé à l'aide d'outils issus de la programmation par contraintes. Nous avons adopté le méta modèle Ecore pour la description des modèles et le langage OCL pour l'expression des contraintes. Nous présentons dans cet article des approches différentes pour exprimer le problème des contraintes sur un modèle, puis nous exposons les apports d'une approche dans la lignée de la programmation par contraintes.

## 1 Introduction

Dans la modélisation orientée objet, les modèles graphiques, comme le diagramme de classe, ne sont pas suffisamment expressifs pour pouvoir exprimer l'intégralité d'une spécification précise. Si de plus en plus d'outils intègrent le langage OCL [23], pour Object Constraint Language, nous nous sommes rendus compte que face à des modèles issus du monde aéronautique et du monde spatial un véritable problème de performances se pose. Nous allons ici voir des approches différentes pour exprimer le problème des contraintes sur un modèle, puis, présenter les apports d'une approche dans la lignée de la programmation par contraintes.

Dans la seconde partie de cet article, nous dressons une liste des principaux outils qui intègrent le langage OCL. La troisième partie présente un exemple de problème de sûreté de fonctionnement. La quatrième partie s'intéresse au problème de vérification d'un modèle. Nous traduisons l'exemple en un problème d'Ingénierie dirigée par les Modèles puis nous présentons le problème du passage à l'échelle où comment, sur de grosses instances de modèle, l'interprétation de requêtes OCL s'avère peu performante ainsi que les possibilités d'une réécriture vers Java. La cinquième partie introduit notre approche qui cherche à exprimer le problème de validation de modèle suivant des contraintes OCL en un problème de satisfaction de contraintes. Nous décrivons notre

méthode et nous montrons comment elle permet de résoudre l'exemple. Puis nous présentons une fonctionnalité qui découle directement de notre approche, la possibilité de compléter un modèle pour qu'il valide des contraintes données et de prendre en compte un critère à minimiser. Enfin, la dernière partie montre comment on peut automatiser le processus de traduction d'un modèle et d'une contrainte et prendre en compte les résultats du solveur au niveau modèle.

## 2 Exprimer des contraintes

Les modèles graphiques malgré leur expressivité ne suffisent généralement pas pour exprimer toute la complexité de ce qu'ils représentent, il est souvent nécessaire de décrire des contraintes supplémentaires sur les objets des modèles. Dans ce cas, ces contraintes sont souvent décrites en langage naturel. Cependant, la spécification en langage naturel soulève bien des problèmes d'ambiguïté. Pour faire face à cela, des langages formels de spécification de contraintes ont été développés. Généralement ces langages utilisent des notations complexes qui nécessitent une connaissance mathématique de la part de ceux qui les utilisent, ce qui peut présenter un problème majeur si le but de la modélisation est d'être compris et utilisé par des acteurs différents. Le langage OCL a été développé pour combler cette lacune. C'est un langage formel de modélisation qui a été conçu pour être facile à lire et à écrire. Dans notre approche, nous avons adopté le métamodèle Ecore [3] pour la description des modèles et le langage OCL pour l'expression des contraintes.

OCL est un langage purement descriptif, par conséquent, une expression OCL est sans effet de bord : elle n'affecte pas le modèle. Il faut bien comprendre que s'il est possible d'exprimer des contraintes à respecter, par contre, le langage ne permet pas d'exprimer la façon de programmer une méthode. Dans le cadre d'une modélisation objet, le langage OCL sert principalement à :

- définir des invariants sur les classes ;
- décrire les pré et post conditions sur les opérations et les méthodes ;
- décrire les gardes sur les transitions dans les diagrammes d'états ou sur les messages dans les diagrammes de séquences et de collaborations ;
- définir des stéréotypes.

Millan et al. [17] proposent aussi d'utiliser OCL pour contrôler la cohérence entre plusieurs modèles.

Même si OCL fait totalement partie de l'UML, l'intégration de celui-ci dans les outils de modélisation ne s'est pas encore généralisée. De plus, la sémantique associée au langage OCL n'est pas exactement la même, en d'autres termes l'implémentation diffère, d'un outil à l'autre. Gogolla et al. [10] étudient différentes implémentations et les différences de résultats d'une implémentation à une autre. Leur étude est mise à disposition par Buttelmann et al. [4]. Nous allons ici présenter certains d'entre eux issus, pour la plupart, du monde académique.

3. Un métamodèle peut être défini comme un modèle d'un langage de modélisation qui permet d'exprimer des concepts communs à l'ensemble des modèles d'un même domaine. Ecore est un langage d'écriture de métamodèles.

Le Model Development Tools [15] est un plugin développé dans le cadre du projet Eclipse. Il vise a fournir une implémentation générique paramétrable par un métamodèle dans le cadre industriel. Il fournit un ensemble d'outils pour la modélisation et la manipulation de métamodèle. Plus particulièrement, le plugin MDT propose une implémentation d'un évaluateur OCL sur des modèles exprimés dans différents langages comme EMF [4] ou UML. Cette implémentation fournit des APIs en Java qui permettent notamment de construire, de valider et d'interpréter une requête OCL sur une instance d'un métamodèle exprimé en EMF.

Un autre plugin Eclipse pour la création et l'évaluation de contraintes OCL est RoclET présenté par Jeanneret et al. [12]. Il comprend un éditeur graphique pour les diagrammes de classe UML et pour les diagrammes d'objets. Il dispose d'un éditeur pour les fichiers OCL qui réalise la vérification de type et d'un interpréteur OCL. RoclET est implémenté à partir de transformations QVT [5]. Jeanneret et al. [13] proposent des opérations de *refactoring* du modèle mettant automatiquement à jour les contraintes OCL dépendant des éléments de modélisation modifiés.

USE (UML-based Specification Environment), développé par Gogolla et al. [9], est une application à part entière qui permet de manipuler et d'interpréter des contraintes OCL sur des métamodèles définis directement dans le langage USE ou importés depuis d'autres outils.

Les langages de transformations de modèles comme ATL [2], ou de méta modélisations comme Kermeta [20] utilisent OCL ou un langage très proche pour décrire des règles de transformations. Il est possible de détourner leur usage et de les utiliser comme interprètes OCL.

## 3 Un exemple tiré des architectures de systèmes embarqués

Cette section propose un problème issu de vérification de propriétés d'architectures dans le cadre de la sûreté de fonctionnement que nous utiliserons dans les sections suivantes. Nous avons un système décomposé en deux parties, une partie logique et une partie physique, et nous devons vérifier certaines propriétés de sûreté de fonctionnement. La partie logique de l'architecture correspond à des instances de processus et leurs liens logiques et la partie physique aux processeurs et les bus les reliant. A cela, nous rajoutons une relation binaire qui apparie ensemble deux à deux les éléments de la couche logique à ceux de la couche physique.

Nous cherchons à vérifier que pour tous les liens logiques entre deux processus, il existe un chemin physique entre les deux composants physiques liés aux composants logiques. Nous focalisons notre exemple sur cette unique contrainte mais notons que, sans rentrer plus en détail, d'autres contraintes sont nécessaires, comme par exemple le

4. Eclipse Modeling Framework est un plugin pour l'IDE Eclipse qui permet, à partir de la définition d'un modèle objet, de faciliter sa manipulation et de générer automatiquement l'éditeur de modèles utilisateur.

5. Queries View Transformation est un standard, défini par l'OMG, pour la transformation de modèle. En ingénierie des modèles une transformation de modèle consiste à passer d'un modèle $A$ conforme à un métamodèle $MA$ en un modèle $B$ conforme à un métamodèle $MB$.

nombre maximal de processus appariés sur un processeur, pour éliminer les solutions triviales où tous les processus sont liés à un unique processeur.

Pour résoudre ce problème, nous considérons une architecture où le chemin physique n'est pas toujours unique et où il ne correspond pas toujours à un lien logique et nous cherchons à vérifier qu'il existe pour chaque lien logique un chemin physique.

Nous représentons ce problème d'architecture comme un problème de mapping entre deux graphes tel que le système est décomposé en un graphe d'architecture logique $G_l = \langle N_l, A_l \rangle$ et un graphe d'architecture physique $G_p = \langle N_p, A_p \rangle$ où les nœuds sont respectivement les processus et les processeurs et où les arcs sont respectivement les liens logiques et les bus. Nous allons définir le mapping entre ces deux graphes de manière à vérifier facilement que pour tout arc logique $a$, il existe un arc ou une succession d'arcs physiques qui permettent d'aller de l'image de la source de $a$ à l'image de la destination de $a$. Ainsi, le mapping sera un morphisme du graphe d'architecture logique dans la fermeture réflexive transitive du graphe d'architecture physique. Avant de définir formellement le mapping, nous introduisons une relation qui nous permet de construire un ensemble de nœuds successeurs.

**Définition 1 (Relation)** *Soit un graphe $G = \langle N, A \rangle$ et un ensemble de nœuds $N' \subseteq N$. La relation $R_G(N')$ est l'ensemble des nœuds destinations des arcs ayant leurs nœuds sources dans $N'$ : $R_G(N') = \{d \mid \exists s \in\ N' \wedge (s,d) \in\ A\}$.*

*La relation $R^*_G(N')$ est l'ensemble des nœuds accessibles par un ou une succession d'arcs à partir des nœuds de $N'$, i.e. l'ensemble des nœuds destinations des arcs de la fermeture réflexive transitive de $G$ et ayant leurs nœuds sources dans $N'$ : $R^*_G(N') = R_{G^*}(N') = \{d \mid \exists s \in\ N' \wedge (s,d) \in\ A^*\}$ avec $G^* = \langle N, A^* \rangle$.*

**Définition 2 (Mapping simple)** *Le mapping simple est défini sur les nœuds d'un graphe logique $G_l = \langle N_l, A_l \rangle$ dans un graphe physique $G_p = \langle N_p, A_p \rangle$ par la fonction mapping : $N_l \rightarrow N_p$ telle que*[6] *:*

*$\forall\ a \in A_l$, mapping(dst(a)) $\in R_{G_p}$(mapping(src(a))). Cette définition correspond au morphisme du graphe $G_l$ dans $G_p$.*

**Définition 3 (Mapping réflexif transitif)** *Le mapping réflexif transitif est défini sur les nœuds d'un graphe logique $G_l = \langle N_l, A_l \rangle$ dans la fermeture réflexive transitive $G^*_p = \langle N_p, A^*_p \rangle$ du graphe physique $G_p = \langle N_p, A_p \rangle$ par la fonction mapping : $N_l \rightarrow N_p$ telle que $\forall\ a \in A_l$, mapping(dst(a)) $\in R_{G^*_p}$(mapping(src(a))). Cette définition correspond au morphisme du graphe $G_l$ dans $G^*_p$.*

## 4 Vérifier un modèle

La phase de conception système nécessite de raisonner à un haut niveau de granularité (en faisant abstraction du code) en terme des fonctionnalités à implanter et de l'architecture cible. Il s'agit d'exprimer et de prendre en compte des propriétés globales

6. Nous définissons $src$, respectivement $dst$, comme la source et la destination d'un arc. Soit un graphe $G$, pour tout arc $a$ de $G$, $src(a)$ désigne le nœud source de l'arc $a$ et $dst(a)$ le nœud destination de l'arc $a$.

liées au placement des fonctions sur l'architecture. Pour assister cette phase de conception, nous proposons de traiter l'exemple par une approche basée sur les techniques de l'ingénierie dirigée par les modèles, de mettre en évidence le problème de passage à l'échelle sur cet exemple puis d'aborder une piste d'optimisation.

### 4.1 Notre exemple par l'ingénierie dirigée par les modèles

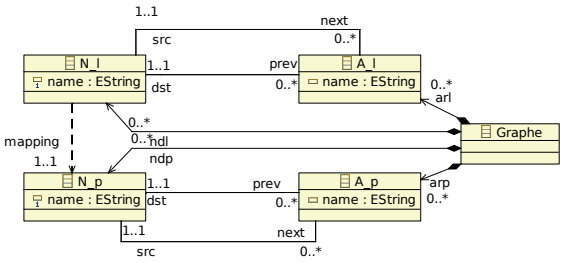



**FIGURE 1.** Méta-modèle du problème de mapping entre deux graphes $G_l = \langle N_l, A_l \rangle$ et $G_p = \langle N_p, A_p \rangle$

La figure 1 donne le méta-modèle correspondant à l'exemple décrit dans la section précédente, c'est-à-dire un problème de mapping entre deux graphes $G_l = \langle N_l, A_l \rangle$ et $G_p = \langle N_p, A_p \rangle$. Nous cherchons à vérifier que pour tout arc $a$ de $A_l$, il existe un arc ou une succession d'arcs de $A_p^*$ qui permettent d'aller de l'image de la source de $a$ à l'image de la destination de $a$. Cette contrainte correspond exactement à la définition du mapping donnée précédemment. Cette contrainte peut s'écrire sous la forme d'une formule mathématique : $\forall x, y \in N_l : (x, y) \in A_l \Rightarrow$ mapping(y) $\in \{d \mid$ (mapping(x),d) $\in A_p^*\}$ et qui correspond à un morphisme du graphe logique dans la fermeture réflexive transitive du graphe physique. Nous pouvons la traduire en OCL avec la notion de fermeture (bornée dans notre cas par le nombre de nœuds physiques) :

**Listing 1.1.** Contrainte exprimée en OCL

```
-- Pour chaque arc logique
  context A_l inv:
-- à partir du mapping de la source de l'arc
    self.src.mapping
-- la fermeture des noeuds accessibles
    -> closure(x : N_p | x.next.dst)
-- doit contenir le mapping de la destination de l'arc
    -> includes(self.dst.mapping)
```

### 4.2 Le problème du passage à l'échelle

Comme nous l'avons vu, le langage OCL permet d'exprimer une contrainte et de tester si un modèle la satisfait. Si le temps d'évaluation par une machine ne pose pas réellement de problème sur des exemples simples et, plus précisément, sur des modèles peu volumineux, nous avons été confrontés à des problèmes de performance avec des modèles issus du monde aéronautique et du monde spatial. Face a des modèles de plusieurs milliers d'objets et des contraintes imbriquant plusieurs boucles, il n'est pas toujours possible avec les évaluateurs actuels de disposer d'une réponse dans un temps acceptable.

Aujourd'hui, plusieurs pistes d'optimisations sont envisagées avec d'un côté le travail de Sánchez Cuadrado et al. [24] sur la formule OCL et de l'autre celui de Mezei et al. [16] sur les algorithmes d'évaluation. Les travaux de Sánchez Cuadrado et al. [25] ont mis en évidence les différences de performances pour l'évaluation d'une formule OCL suivant la façon dont elle est formée. Ainsi, ils préconisent différentes techniques ad'hoc comme l'usage du court-circuit, qui consiste à ne pas évaluer une expression qui ne peut pas modifier le résultat de l'opération logique, pour les formules utilisant le "and" et le "or".

Nous allons illustrer ici le problème du passage à l'échelle à l'aide de notre exemple et nous utilisons le métamodèle présenté dans la figure 1. Nous définissons un $N$-modèle comme un modèle de notre exemple avec $N$ instances de nœuds logiques et de nœuds physiques et $N^2$ arcs logiques et arcs physiques. Pour tester les performances d'un interpréteur OCL, nous utilisons Model Development Tools (MDT) avec la contrainte OCL portant sur les mappings de graphes logiques sur les graphes physiques. Pour éviter toute optimisation due au fait que le mécanisme de court circuit n'est pas implémenté dans les évaluateurs nous nous intéressons seulement à des exemples où la contrainte est satisfaite.

Plus $N$ grandit, plus l'interpréteur a d'objets à manipuler. Comme nous ne voulons pas seulement avoir des performances brutes, mais pouvoir les comparer à ce que l'on pourrait qualifier d'optimum, nous avons écrit un programme Java qui est la traduction exacte de notre contrainte. Ce programme ne comporte pas d'optimisation et n'est pas généré automatiquement. Nous pouvons considérer qu'il est un étalon et qu'une solution idéale pour la validation d'un modèle suivant des contraintes devrait nous donner les mêmes temps de réponses.

La figure 2 présente l'évolution du temps de résolution mis par l'interpréteur OCL de MDT et par le programme Java qui traduit la contrainte testée pour différents $N$-modèles. Ce graphique montre que l'interpréteur OCL est loin des performances idéales et il n'est plus capable de donner de réponses dès que le modèle comporte une grande quantité d'objets. La différence de performance entre le code Java et l'interprétation d'une formule OCL équivalente n'est pas inhérente à MDT. En effet, les outils que nous avons pu tester qui réalisent une interprétation, et non une transformation vers un langage exécutable, restent dans la même échelle de grandeur en terme de temps pour la validation d'une contrainte.

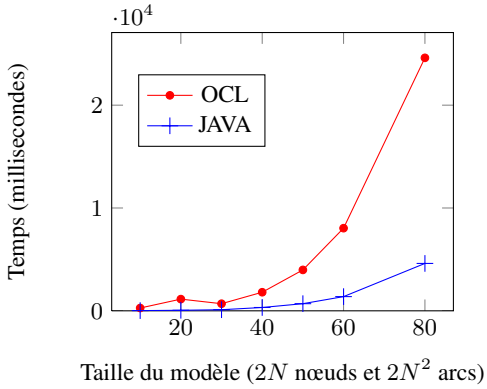



**FIGURE 2.** Test de performance sur le problème de graphe

### 4.3 Java, le problème de l'expressivité

Comme nous venons de le voir si le langage OCL est très expressif, les différentes implémentations d'interpréteurs qui existent présentent un problème de performance face à de gros modèles avec des contraintes récursives. Une solution possible serait d'écrire directement des contraintes dans un langage compilé. Si cette solution présente l'avantage d'ajouter un gain en temps d'exécution, elle entraîne une perte d'expressivité. Une contrainte écrite dans un langage comme par exemple Java est beaucoup moins compréhensible qu'en langage OCL. Une piste possible, que nous avons évoquée, est d'écrire des contraintes en OCL et de les traduire automatiquement en Java.

Notons que traduire automatiquement le problème de validation d'un modèle en fonction de contraintes OCL en un programme exécutable n'est pas nouveau. Si la plupart des approches relèvent de l'interprétation, notons que dès 1999 Demuth et Hussmann [6] cherchaient à transformer des expressions OCL en SQL et Finger [8] donnait les bases d'une traduction vers Java. Depuis le langage OCL a évolué et d'autres travaux (Octopus [21]) ont cherché à formaliser une transformation de OCL vers du code exécutable. En pratique, le plupart de ces approches vont vers Java. Ainsi, le projet Key de Ahrendt et al. [1] fournit des méthodes de spécification et vérification de programmes développés en utilisant UML. La spécification formelle utilise OCL et Key réalise la traduction de cette spécification vers un code exécutable en Java Card. Citons aussi les travaux de Dzidek et al. [7] qui étudie les différents problèmes soulevés par la génération de code à partir d'une formule OCL, mais leur outil, "OCL2J approach", n'est pas disponible. Notons que toutes ces approches restent pour le moment non exhaustives et aucun outil ne permet le passage de OCL à Java sans aucune restriction. Parmi les projets existants, Dresden OCL2 semble nettement s'imposer, cette remarque s'appuyant sur deux constats. Premièrement la vitalité du projet qui contrairement à Octopus fournit régulièrement des mises à jour. Deuxièmement, Dresden OCL2 sert de base à de nombreux projets importants comme : Borland Together, Poseidon ou Rational Rose.

Dresden OCL2 Toolkit est un outil développé par l'université de Dresden qui permet de transformer un modèle UML2, métamodèle de Eclipse Model Development Tools

Project, ou un métamodèle Ecore avec ses contraintes OCL en Aspect Java. Cette transformation s'effectue en essayant de traduire, autant que faire ce peu, les types OCL en types Java. Par exemple, le type Integer devient int, Real devient float et Boolean devient boolean. Cependant, OCL dispose aussi de différentes collections Bag, OrderedSet, Sequence, et Set qui ne se traduisent pas de façon adéquat dans des types Java existant. C'est pourquoi ils introduisent un plugin Eclipse, tudresden.ocl20.pivot.ocl2-java.types qui contient une implémentation en Java des collections de OCL.

Si la piste d'écrire des contraintes en OCL et de les traduire automatiquement en Java est prometteuse, nous aimerions dans la suite de cet article proposer une autre traduction : valider des contraintes sur un modèle en un problème de satisfaction de contraintes.

## 5 Vérifier un modèle à l'aide de contraintes

Cette section montre comment nous pouvons utiliser le formalisme des problèmes de satisfaction de contraintes pour vérifier des contraintes exprimées en OCL sur des modèles. Si, dans un premiers temps, cette approche est motivée par un gain possible de performance, nous verrons dans la section suivante qu'elle ouvre surtout de nouvelles perspectives dans le domaine de l'aide au design.

### 5.1 Le problème de satisfaction de contraintes

Le formalisme des problèmes de satisfaction de contraintes (CSP) présenté par Montanari [18] offre un cadre puissant pour la représentation et la résolution efficace de nombreux problèmes. Ces problèmes sont représentés par un ensemble de variables qui doivent, chacune, être affectées dans leur domaine respectif, tout en satisfaisant un ensemble de contraintes qui expriment des restrictions sur les différentes affectations possibles.

Aujourd'hui, de nombreux solveurs CSP existent et ne cessent d'être perfectionnés. Des améliorations aussi bien au niveau algorithmique qu'au niveau des structures de données sont sans cesse proposées et une compétition internationale [7] mettant en jeu les performances de chaque prouveur contribuent à l'évolution de ceux-ci. La librairie CHOCO développée par Jussien et al. [14] est une librairie Java qui permet de modéliser et de résoudre efficacement des CSP. Régulièrement amélioré et complété, CHOCO est devenu un outil de plus en plus utilisé dans le cadre d'autres domaines de recherche.

C'est dans la droite ligne de cette approche que nous cherchons à exprimer le problème de la validation de modèle suivant des contraintes OCL en un problème CSP et nous ferons appel à CHOCO pour le modéliser sous forme de contraintes puis pour le résoudre. Notons que notre approche n'est pas unique puisque l'outil UMLtoCSP de Cabot et al. [5] permet, à partir d'un diagramme UML et de contraintes OCL associées, de vérifier des invariants OCL ainsi que des propriétés UML, en transformant tout le diagramme de classes UML et les invariants OCL associés en un problème de satisfaction de contraintes. Par contre, leur approche nécessite de fournir en paramètre l'espace

7. Pour plus d'information voir le site : `http://www.cril.univ-artois.fr/CPAI09/`

de recherche pour obtenir une recherche complète. De plus, alors que cette approche se limite à l'aspect vérification d'une contrainte, nous cherchons en plus à faire de la synthèse et prendre en compte un critère à minimiser.

### 5.2 Méthode

Pour valider un modèle et faire une synthèse partielle de solution, nous transformons la contrainte écrite en OCL et le modèle associé en un problème de satisfaction de contraintes, puis résolvons le problème avec le solveur CHOCO.

**Sous-ensemble d'OCL considéré** Dans cette étude préliminaire, nous considérons seulement un extrait d'OCL mais nous travaillons actuellement à augmenter ce sous-ensemble. Nous définissons comme langage source le sous-ensemble du langage OCL que nous pouvons transformer. Les contraintes OCL sont définies comme des invariants. Les expressions OCL sont basées sur les types entier, booléen et les collections.

Les opérateurs sur les collections pris en compte sont forAll, exists, select, reject, collect, includesAll, excludesAll, union, intersection, notEmpty, isEmpty, includes, excludes, sum, size et count.De plus, nous considérons les itérations sous la forme normalisée suivante :

```
iterate(x, acc : T = a_0 |
  let z_1 = f_1(x) in
  ...
  let z_n = f_n(x) in
  if cond(x) then f'(x, i, acc) else acc
  endif)
```

où $f'$ est une fonction associative et cumulative et $i$ le nombre d'occurences de l'élément $x$.

**Traduction des données en CSP** Pour vérifier des contraintes sur un modèle, nous avons besoin d'extraire une partie des données contenues dans le modèle. Afin de limiter la taille du CSP, nous traduisons en CSP seulement les données qui sont partiellement connues, les éléments totalement connus sont stockés dans des structures de données comme des constantes. Pour manipuler des entiers, nous associons un identifiant à chaque objet et à chaque attribut utilisé.

*Exemple* Prenons comme exemple le problème de graphe défini dans la sous-section 4. Nous noterons $n_l$ le nombre de nœuds logiques et $n_p$ le nombre de nœuds physiques. Dans ce problème, nous avons besoin des arcs logiques et physiques qui sont fixés dans le modèle. Ces deux données sont stockées dans deux collections de couples de nœuds : `al` pour les arcs logiques et `ap` pour les arcs physiques. Nous considérons également comme fixé le calcul de la fermeture transitive réflexive du graphe physique car elle est

basée sur les arcs physiques qui sont connus. Le calcul est effectué avec l'algorithme de Warshall et son résultat est stocké dans un tableau de constantes booléennes `closure` :

$$\begin{aligned}\texttt{closure}[i,j] = (i,j) \in al \vee (i = j) \\ \vee (\exists k \in \{0,..,n_p - 1\} : \texttt{closure}[i,k] \wedge \texttt{closure}[k,j])\end{aligned} \quad (1)$$

Nous modélisons par des variables du CSP le mapping du graphe logique dans le graphe physique qui est partiellement connu. A chaque nœud logique $x \in N_l$, le mapping associe un et un seul nœud physique $y \in N_p$. Nous créons un tableau de variables entières `mapping` :$\{0,..,n_l - 1\} \rightarrow \{0,..,n_p - 1\}$ où l'indice représente l'identifiant du nœud logique et la case contient l'identifiant de l'image de ce nœud. Le domaine de chacune de ces variables est l'ensemble des identifiants des nœuds physiques. Lorsque le mapping est défini, les variables sont affectées, sinon l'affectation d'une valeur du domaine à chaque variable est confiée au solveur. Le listing 1.2 donne la traduction du mapping en CHOCO et l'affectation au nœud d'identifiant 0 l'image d'identifiant 1 :

**Listing 1.2.** Mapping exprimé en CHOCO

```
--definition du mapping
IntegerVariable[] mapping = new IntegerVariable[nl] ;
for(int i = 0 ; i < nl ; i++){
  mapping[i] = Choco.makeIntVar("M"+i, 0, np-1) ;
  csp.addVariables(mapping[i]) ;
}
--affectation: mapping[0]=1
csp.addConstraint(Choco.eq(mapping[0],1)) ;
```

Cette traduction des données introduit $n_l$ variables, dont la taille du domaine est $n_p$, et autant de contraintes unaires que de mappings connus.

**Traduction des contraintes en CSP** Une fois définies les constantes et les variables nécessaires au problème, nous pouvons traduire les contraintes OCL sous forme de contraintes CSP.

*Exemple* La contrainte qui cherche à vérifier que, pour tout arc logique $(i,j)$, il existe un chemin entre l'image de $i$ et l'image de $j$ peut se traduire avec notre représentation par :

$$\forall (i,j) \in A_l, \texttt{closure}[\texttt{mapping}[i]][\texttt{mapping}[j]] \quad (2)$$

Nous obtenons la traduction de l'équation 2 en CHOCO par l'introduction de Card$(al)$ contraintes binaires comme le montre le listing 1.3.

**Listing 1.3.** Contrainte exprimée en CHOCO

```
--pour tous les arcs logiques
for(Couple c:al.getCouples()) {
  int i=c.getFirst();
  int j=c.getSecond();
```

```
  -- ajout contrainte: closure[mapping[i]][mapping[j]]=true
  csp.addConstraint(
    Choco.feasPairAC(mapping[i], mapping[j], closure));
};
```

Cette modélisation en un problème de satisfaction de contraintes produit un CSP qui contient $n_l$ variables entières dont le domaine a une taille $n_p$, autant de contraintes unaires que de mappings connus et Card$(al)$ contraintes binaires.

### 5.3 Résultats

Pour comparer notre approche avec un interpréteur OCL et une solution idéale codée en Java, nous reprenons le test de performance effectué dans la sous-section 4.2. La figure 3 montre que notre méthode permet un gain en temps d'exécution par rapport aux OCL checker et que nous sommes capable de traiter un grand nombre d'objets dans des temps raisonnables par rapport à la solution idéale représentée en Java.

Lorsque le modèle comporte $2 * 80^2$ arcs, notre méthode est même plus rapide que Java, ce qui peut s'expliquer par un traitement différent pour le calcul de la fermeture. En Java les fermetures sont calculées pour chaque nœud d'un arc logique lors de l'évaluation de la contrainte OCL, augmenter le nombre d'arcs augmente les possibilités de recalculer plusieurs fois la même fermeture. Dans notre méthode, nous calculons préalablement les fermetures de chaque nœud avant d'évaluer la contrainte, augmenter le nombre d'arcs diminue les possibilités de calculer des fermetures inutilisées. Ainsi, plus le modèle contient d'objets et notamment d'arcs, plus notre méthode est avantageuse en terme de temps d'éxécution.

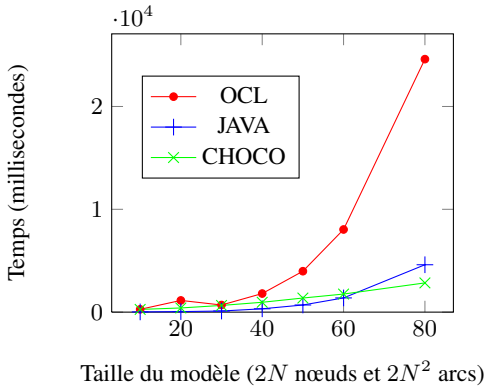



**FIGURE 3.** Test de performance sur le problème de graphe

Sur cet exemple, la traduction du problème de validation de modèles en un problème de satisfaction de contraintes permet d'éviter le problème du passage à l'échelle. Il semble alors pertinent d'étudier une traduction automatique du problème de validation

d'une contrainte OCL sur un modèle en CSP pour tester notre méthode sur d'autres problèmes. La section suivante montre que notre approche ne se limite pas à valider des modèles mais permet d'aider l'utilisateur lors de la conception de modèles.

## 6 Compléter un modèle partiel et minimiser un critère

Nous avons montré comment utiliser un solveur CSP pour valider un modèle. CHOCO ne se contente pas de savoir si une contrainte est satisfaite ou non, mais peut nous donner un modèle pour cette contrainte s'il existe. L'idée ici est d'utiliser le solveur CHOCO pour proposer une modification du modèle à l'utilisateur. Les valeurs inconnues du modèle sont décrites à l'aide de variables afin de réaliser une synthèse partielle de solution pour une contrainte OCL. Dès lors, il est possible d'exprimer le problème sous forme de contraintes et que CHOCO nous renvoie un modèle solution qui valide ces contraintes. A l'aide de transformations inverses, la solution nous revient dans un format compréhensible. Nous avons donc un mécanisme qui nous permet de synthétiser une solution.

Dans l'exemple précédent, cette méthode nous permet de compléter un mapping partiel tout en respectant les contraintes données.

D'autre part, CHOCO peut prendre en compte un critère à minimiser décrit par une variable objective comme nous l'illustrons dans l'exemple suivant.

*Exemple* Nous cherchons à minimiser, dans le problème précédent, le cardinal de l'image de l'ensemble des nœuds logiques par la fonction mapping : Card$(\{y \mid \exists x \in N_l, y = \text{mapping}(x)\})$. Ce critère consiste à compter le nombre de nœuds physiques liés par un mapping. Le listing 1.4 montre la traduction de ce critère en OCL.

**Listing 1.4.** Critère de minimisation exprimé en OCL

```
N_l.allInstances().mapping->asSet()->count()
```

Pour prendre en compte ce critère dans le CSP, comme le montre le listing 1.5, nous créons la variable images dont la valeur représente l'ensemble des nœuds physiques liés par un mapping, nous ajoutons $n_p$ contraintes spécifiant que toutes les images du mapping appartiennent à l'ensemble images. Nous déclarons comme variable à minimiser, une variable entière cardImages qui représente la cardinalité de la variable images.

**Listing 1.5.** Critère de minimisation exprimé en CHOCO

```
-- Definition de l'ensemble des images
SetVariable images = Choco.makeSetVar("s", 0, np);
for (int i=0; i<np; i++) {
  csp.addConstraint(Choco.member(images, mapping[i]));
};
-- Definition du critere
IntegerVariable cardImages =
  Choco.makeIntVar("CardImages", 0, np, "cp:objective");
csp.addConstraint(Choco.eqCard(images, cardImages));
```

Lorsqu'il existe plusieurs solutions au problème, CHOCO retourne une solution qui minimise notre critère. La traduction de cette solution nous permet de compléter le modèle initial en satisfaisant les contraintes et le critère.

A travers cet exemple, nous montrons que notre méthode permet de faire de la vérification mais surtout d'aider à la conception de modèles en complétant des modèles partiels ou en minimisant un critère.

## 7 Vers une automatisation du processus de traduction

Dans la partie précédente, on a montré que l'on pouvait exprimer une contrainte OCL et le modèle qui lui est associé en Choco. On travaille actuellement à l'automatisation de cette traduction grâce à différentes techniques. Les contraintes n'étant pas intégrées au modèle, nous avons pris le parti de gérer leur traduction indépendamment. La contrainte est traduite en utilisant des méthodes de réécriture supportées par le langage Tom présenté par Moreau et al. [19]. On génère la partie données à partir du modèle source grâce à des outils de transformations de modèles.

Tom est une extension de langages hôtes comme Java, c, ou Python, permettant de manipuler des structures arborescentes par réécriture. Tom apporte un mécanisme de reconnaissance de motifs avancé qui facilite l'écriture des règles de transformations permettant de passer des expression OCL aux expressions Choco. De plus, il définit des stratégies de parcours génériques afin d'appliquer des règles de transformations simples à toutes les sous-expressions d'une formule OCL.

Le MDT OCL nous permet d'analyser le texte d'une contrainte OCL, et de la représenter sous la forme d'un arbre abstrait d'objets Java. On sait interpréter les objets Java comme des termes Tom. La description de cette correspondance est en cours, elle nous permettra d'implanter en Tom les règles de transformation des expressions OCL.

La traduction des données en Choco est effectuée par une transformation de modèles. Son but est double : générer une partie du code Choco, et modifier le modèle source en fonction des résultats synthétisés par le solveur de contraintes. En effet, dans le cas ou on souhaite compléter un modèle suivant un critère de minimisation, on veut pouvoir intégrer la solution proposée au niveau du modèle. On doit donc définir une transformation bidirectionnelle permettant dans un premier temps de générer le code Choco, puis de compléter le modèle source. Le langage QVT définit par l'OMG [22] permet de définir des transformations de modèles relationnelles ou opérationnelles. Les implantations de ce standard ne couvrent pas tout le langage, souvent seule la partie opérationnelle est utilisée. Une implantation de ce langage, MediniQvt développé par ikv++ developers [11], prend en compte la partie relationnelle de QVT et nous autorise donc à décrire des transformations bidirectionnelles.

La première étape, avant de décrire la transformation, est de définir un métamodèle pour les données Choco. Les types de données manipulés peuvent être de trois sortes, entier, réel, ou ensemble. On peut ensuite définir des tableaux, et enfin on doit préciser la nature de notre donnée, variable ou constante. La transformation est ensuite implantée suivant les règles définies dans la partie 5.

Une fois que le problème est exprimé sous la forme de contraintes Choco, la résolution peut être lancée. On doit encore inclure le résultat dans notre modèle choco, puis le remonter au niveau du modèle de graphe.

La figure 4 représente le processus complet proposé dans cette partie.

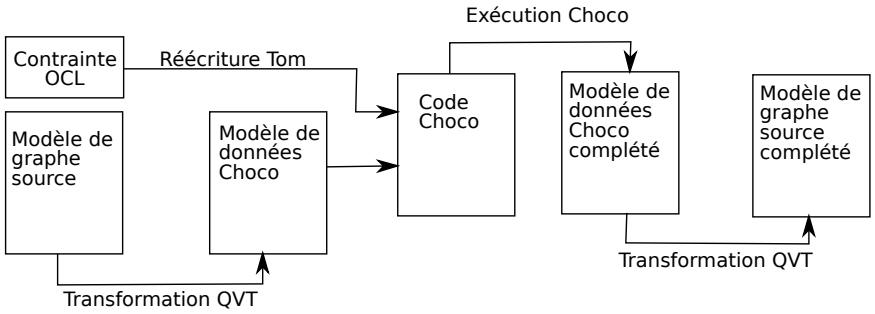



**FIGURE 4.** Processus de complétion d'un modèle basé sur Choco.

## 8 Conclusion

Dans cet article, nous avons étudié un cadre de travail basé sur les formalismes et outils de l'ingénierie dirigée par les modèles pour valider et synthétiser des modèles. Pour supporter le passage à l'échelle de ces derniers, nous avons étudié l'adaptation de techniques issues de la programmation par contraintes. Cette approche nous semble intéressante car elle offre une continuité avec les chaînes de conception dites aval (plus concernée par les aspects génération de code et déploiement) qui sont aussi de plus en plus basées sur les outils de l'ingénierie des modèles.

Il s'agit là d'une étude préliminaire. Dans le futur, nous envisageons d'offrir un meilleur support pour l'analyse de propriétés du domaine de l'embarqué, e.g., celles définies pour le placement par AADL et par la suite des plateformes avioniques présenté par Bieber et al. [3]. Une voie qui nous semble prometteuse est l'amélioration du codage des structures de données du méta modèle et du modèle en prenant en compte des contraintes structurelles métier.

## Références

1. Ahrendt, W., Baar, T., Beckert, B., Bubel, R., Giese, M., Hähnle, R., Menzel, W., Mostowski, W., Roth, A., Schlager, S., Schmitt, P.H. : The key tool. Technical report in computing science no. 2003-5, Department of Computing Science, Chalmers University and Göteborg University (2003)
2. Atlas transformation language, http ://www.eclipse.org/m2m/atl/

3. Bieber, P., Bodeveix, J., Castel, C., Doose, D., Filali, M., Minot, F., Pralet, C. : Constraint-based design of avionics platform - preliminary design exploration. In : ERTS, Toulouse. SEE (January 2008)
4. Buttelmann, B., Hamann, L., Jolk, F., Sun, B., Wang, H., Xia, L. : Evaluating a benchmark for ocl engine accuracy, determinateness, and efficiency (2008), http ://db.informatik.uni-bremen.de/publications/Buettelmann_2008_BMEVAL.pdf
5. Cabot, J., Clarisó, R., Riera, D. : Verification of uml/ocl class diagrams using constraint programming. In : Proceedings of ICST Workshop on Model Driven Engineering, Verification and Validation : Integrating Verification and Validation in MDE (MoDeVVa'2008) (2008)
6. Demuth, B., Hussmann, H. : Using UML/OCL constraints for relational database design. In : France, R., Rumpe, B. (eds.) UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings. LNCS, vol. 1723, pp. 598–613. Springer (1999)
7. Dzidek, W., Briand, L., Labiche, Y. : Lessons learned from developing a dynamic ocl constraint enforcement tool for java. In : Bruel, J.M. (ed.) MoDELS Satellite Events. Lecture Notes in Computer Science, vol. 3844, pp. 10–19. Springer (2005)
8. Finger, F. : Design and Implementation of a Modular OCL Compiler. Ph.D. thesis, Technische Universität Dresden (March 2000)
9. Gogolla, M., Büttner, F., Richters, M. : Use : A uml-based specification environment for validating uml and ocl. Sci. Comput. Program. 69(1-3), 27–34 (2007)
10. Gogolla, M., Kuhlmann, M., Büttner, F. : A benchmark for ocl engine accuracy, determinateness, and efficiency. In : MoDELS'08 : Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems. pp. 446–459. Springer-Verlag, Berlin, Heidelberg (2008)
11. ikv++ developers : Medini QVT. ikv++ technologies ag, http ://projects.ikv.de/qvt (2008)
12. Jeanneret, C., Eyer, L., Markovié, S., Baar, T. : RoclET-A Tool for Wrestling with OCL Specifications. In : Model Driven Engineering Languages and Systems, 9th International Conference, MoDELS 2006 (2006)
13. Jeanneret, C., Eyer, L., Markovié, S., Baar, T. : RoclET : Refactoring OCL Expressions by Transformations. In : Software & Systems Engineering and their Applications,19th International Conference, ICSSEA 2006 (2006)
14. Jussien, N., Rochart, G., Lorca, X. : The choco constraint programming solver. In : CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08). Paris, France (Jun 2008)
15. Model development tools, http ://www.eclipse.org/modeling/mdt/
16. Mezei, G., Lengyel, L., Levendovszky, T., Charaf, H. : Optimization algorithms for ocl compilers. In : Proceedings of the 5th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems. pp. 55–60. WSEAS, Stevens Point, Wisconsin, USA (2006)
17. Millan, T., Sabatier, L., Le Thi, T.T., Bazex, P., Percebois, C. : An OCL extension for checking and transforming UML Models. In : WSEAS - International Conference on Software Engineering, Parallel and Distributed Systems (SEPADS'09), Cambridge, United Kingdom, 21/02/09-23/02/09. pp. 144–150. WSEAS Press, http ://www.wseas.org/ (2009), (Conférencier invité)
18. Montanari, U. : Networks of constraints : Fundamental properties and applications to picture processing. Information Sciences 7, 95–132 (1974)

19. Moreau, P.E., Ringeissen, C., Vittek, M. : A Pattern Matching Compiler for Multiple Target Languages. Tech. rep., INRIA (2002)
20. Mottu, J., Barais, O., Skipper, M., Vojtisek, D., Jézéquel, J. : Intégration du support ocl dans kermeta. spécifiez la sémantique statique de vos méta-modèles. 10ième Anniversaire de la Conférence Francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'07) (Jun 2007)
21. Octopus : Http ://sourceforge.net/projects/octopus/
22. OMG : MOF QVT Final Adopted Specification. Object Modeling Group (June 2005), http://fparreiras/papers/mof_qvt_final.pdf
23. OMG : Object Constraint Language, OMG Available Specification, Version 2.0 (2006)
24. Sánchez Cuadrado, J., Jouault, F., García Molina, J., Bézivin, J. : Deriving ocl optimization patterns from benchmarks. In : Proceedings of the 8th International Workshop on OCL Concepts and Tools (OCL 2008) at MoDELS 2008. Electronic Communications of the EASST (2008), http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/178/175
25. Sánchez Cuadrado, J., Jouault, F., García Molina, J., Bézivin, J. : Optimization patterns for ocl-based model transformations. In : Models in Software Engineering, Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008. Reports and Revised Selected Papers. pp. 273–284. Springer-Verlag (2008)

# Formalisation de contextes et d'exigences pour la validation formelle de logiciels embarqués

**Philippe Dhaussy[*] — Frédéric Boniol[+] — Pierre-Yves Pillain[*] — Amine Raji[*] — Yves Le Traon[**] — Benoit Baudry[***]**

[*] *UEB, Laboratoire LISyC, ENSIETA, BREST, F-29806 cedex 9,*
*{dhaussy, pillaipi, rajiam }@ensieta.fr,*

[+] *Onera/Cert, Toulouse,*
*boniol@onera.fr,*

[**] *Université du Luxembourg, Campus Kirchberg,*
*Yves.letraon@Uni.lu,*

[**] *Equipe Triskell, l'IRISA, RENNES, F-35042,*
*bbaudry@IRISA.fr*

RÉSUMÉ. *Un défi bien connu dans le domaine des méthodes formelles est d'améliorer leur intégration dans les processus de développement industriel. Dans le contexte des systèmes embarqués, l'utilisation des techniques de vérification formelle par model-checking nécessitent tout d'abord de modéliser le système à valider, puis de formaliser les propriétés devant être satisfaites sur le modèle et enfin de décrire le comportement de l'environnement du modèle. Ce dernier point que nous nommons « contexte de vérification » est souvent négligé. Il peut être, cependant, d'une grande importance afin de réduire la complexité de la vérification. Dans notre contribution, nous cherchons à proposer à l'utilisateur une aide pour la formalisation de ce contexte en lien avec la formalisation des propriétés. Dans ce but, nous proposons et expérimentons un langage, nommée CDL (Context Description Language), pour la description des acteurs (ou composants) de l'environnement, basée sur des diagrammes d'activités et de séquence et des patrons de définition des propriétés à vérifier. Les propriétés sont modélisées et reliées à des régions d'exécution spécifiques du contexte. Nous illustrons notre contribution sur un exemple et décrivons des résultats sur plusieurs applications industrielles embarquées.*

ABSTRACT. *A well known challenge in the formal methods domain is to improve their integration with practical engineering methods. In the context of embedded systems, model checking re-*

*quires first to model the system to be validated, then to formalize the properties to be satisfied, and finally to describe the behavior of the environment. This last point which we name as the proof context is often neglected. It could, however, be of great importance in order to reduce the complexity of the proof. The question is then how to formalize such a proof context. We experiment a language, named CDL (Context Description Language), for describing a system environment using actors and sequence diagrams, together with the properties to be checked. The properties are specified with textual patterns and attached to specific regions in the context. Our contribution is a report on several industrial embedded system applications.*

MOTS-CLÉS : *Vérification, contexte, patrons de propriétés.*

KEYWORDS: *Verification, context, property pattern.*

## 1. Introduction

Dans le domaine des systèmes embarqués, les architectures logicielles doivent être conçues pour assurer des fonctions critiques soumises à des contraintes très fortes en termes de fiabilité et de performances temps réel. En raison de ces contraintes, les architectures logicielles embarquées sont soumises à un processus de certification qui nécessite un développement très rigoureux. Toutefois, en raison de la complexité croissante des systèmes, leur conception reste une tâche difficile.

Dans le but d'accroître la fiabilité des logiciels, les méthodes formelles contribuent à l'apport de solutions rigoureuses et puissantes pour aider les concepteurs à produire des systèmes non défaillants. À cet effet, les méthodes de vérification formelle de comportement de modèles ont été explorées depuis plusieurs années par de nombreuses équipes de recherche et expérimentées par des industriels. Mais aujourd'hui, les logiciels embarqués intègrent des fonctionnalités de plus en plus complexes ce qui rend difficile la mise en œuvre de ces méthodes. Malgré la performance croissante des outils de model-checking, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel.

Les techniques de vérification formelle par model-checking souffrent aussi du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifié. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre de composants logiciels. Un moyen pour restreindre les comportements du modèle est de spécifier le comportement de son environnement dans lequel il est plongé. Le système est ensuite synchronisé fortement avec son environnement lors des phases de vérification des exigences. Ce contexte correspond à des phases opérationnelles bien identifiées, comme, par exemple, l'initialisation, les reconfigurations, les modes dégradés, etc. Une autre difficulté est liée à l'expression formelle des exigences sous forme de propriétés nécessaire à leur vérification. Cette expression peut s'effectuer à l'aide de formalismes de type logique temporelle comme LTL (Pnueli, 1977) ou CTL (Clarke *et al.*, 1986). Bien que ces langages aient une grande expressivité, ils ne sont pas faciles à utiliser par des ingénieurs lors de leurs projets. Pour surmonter ce problème, certaines approches (Dwyer *et al.*, 1999), (Smith *et al.*, 2002), (Konrad *et al.*, 2005) ont été proposées pour formaliser les propriétés temporelles à l'aide de patrons d'expression plus proche des types de langages que les ingénieurs ont l'habitude de manipuler.

Face à ce constat, nous avons proposé (Dhaussy *et al.*, 2008), (Dhaussy *et al.*, 2009) de définir un cadre structurant permettant à l'utilisateur de décrire formellement des contextes de vérification grâce à l'utilisation d'un langage de description

de contextes, CDL (Context Description Language). Ce DSL[1] permet de spécifier des contextes sous la forme de scénarios (diagrammes d'activités et de séquences) et de propriétés temporelles spécifiées à l'aide de patrons de définition de propriété. De plus, CDL offre la possibilité à l'utilisateur d'associer chaque propriété à une phase du comportement de l'environnement du système, c'est-à-dire un contexte spécifique. Les contextes de vérification sont ensuite exploités pour générer automatiquement des programmes formels assimilables par des outils de vérification.

Dans cet article, nous rendons compte d'un retour d'expérience sur l'application de notre approche et de ce langage sur plusieurs cas d'étude du domaine aéronautique. Nous présentons notre proposition et décrivons les résultats sur des expérimentations de mise en œuvre de preuves formelles par des ingénieurs. Nous montrons, tout d'abord, l'intérêt de spécifier le contexte dans lequel le système sera utilisé lors de la simulation en vue de réduire l'explosion combinatoire. Ensuite, nous montrons comment formaliser, avec CDL, les contextes et les propriétés et ensuite comment relier ces propriétés à des régions spécifiques du contexte.

Pour faciliter la compréhension de notre approche, nous l'illustrons par un exemple simple déduit d'un cas industriel réel (le logiciel du système $S\_CP$ montré en figure 1). $S\_CP$ contrôle les modes internes d'un système relié à des périphériques (radars, capteurs, effecteurs, ...) et répond à des signaux provenant de composants (que nous nommons ici "acteurs") de l'environnement ($HMI, Device$).

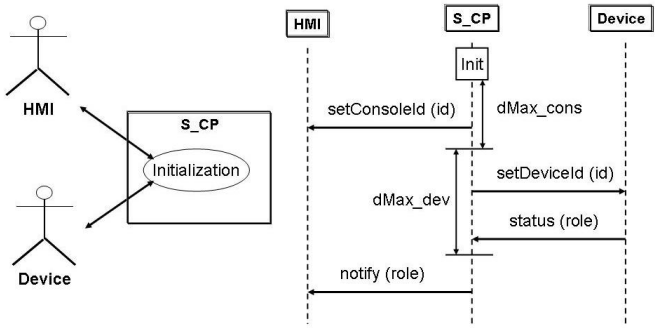



**Figure 1.** *Le système $S\_CP$ : un diagramme de séquences et un cas utilisation (partiel) décrivant le comportement du système et son environnement lors de la phase d'initialisation.*

L'article est organisé comme suit : la section 2 décrit le périmètre de nos travaux dans le domaine de la pratique actuelle des techniques de vérification formelle et présente des travaux connexes. La section 3 décrit le langage CDL avec sa syntaxe et sa sémantique formelle pour la spécification des contextes et des propriétés. La méthodologie proposée utilisée pour les expérimentations est présentée en section 4, ainsi que l'outillage mis en œuvre. En section 5, nous donnons quelques résultats sur plusieurs

1. Domain Specific Language

études de cas industriels. Enfin, nous concluons, en section 6, par une discussion sur notre approche et des travaux futurs.

## 2. Objectifs et travaux connexes

Une difficulté de mise en œuvre d'une technique par model-checking est de pouvoir exprimer les propriétés à vérifier de façon aisée. Les langages à base de logique temporelle permettent en théorie une grande expressivité des propriétés. Mais en pratique dans un contexte industriel et au regard de la grande majorité des documents d'exigences à manipuler, ces langages sont souvent difficiles, voire impossible à utiliser tels quels. En effet, une exigence peut référencer de nombreux événements, liés à l'exécution du modèle ou de l'environnement, et est dépendante d'un historique d'exécution à prendre en compte au moment de sa vérification. Ce constat est illustré avec un exemple d'exigence R (cf. Listing 1) extrait du cahier de charge du système $S\_CP$. On peut constater que la formalisation de R évoque beaucoup d'événements et d'actions opérées par le système et l'environnement et exprime un historique d'exécution. Due à sa formalisation ambiguë, son interprétation peut très vite poser problème. Si le recours à des formules d'une logique temporelle peut permettre une expression formelle, leur utilisation n'est pas aisée car les expressions peuvent être d'une grande complexité. Elles demandent beaucoup d'expertise, ce qui les rend difficilement lisibles ou manipulables par des ingénieurs.

**Exigence R :** *« Au cours de la procédure d'initialisation, $S\_CP$ doit associer un identificateur à chaque périphérique (Device) du système, avant une unité de temps $dMax\_dev$. Il doit également associer un identificateur pour chaque console (IHM), avant $dMax\_cons$ unités de temps. $S\_CP$ transmet un message $notifyRole$ à chaque périphérique connecté et chaque console connectée. L'initialisation s'achève avec succès, lorsque $S\_CP$ a affecté tous les identificateurs de périphérique et de console et lorsque tous les messages $notifyRole$ ont été envoyés. »*

**Listing 1.** Une exigence d'initialisation du système $S\_CP$.

Il est donc nécessaire de faciliter l'expression des exigences avec des langages adéquats permettant d'encadrer l'expression des propriétés et d'abstraire certains détails, au prix de réduire l'expressivité. De nombreux auteurs ont fait ce constat depuis longtemps et certains (Dwyer *et al.*, 1999), (Smith *et al.*, 2002), (Konrad *et al.*, 2005) ont proposé de formuler les propriétés à l'aide de patrons de définition. Le but est de proposer un mode d'expression plus proche des langages que les ingénieurs ont l'habitude de manipuler. Nous reprenons cette approche en réutilisant les patrons de Dwyer et Smith. Nous les étendons et nous les implantons dans le langage de description de contextes CDL.

Une autre difficulté à contourner est de gérer la complexité des vérifications due à l'explosion du nombre de comportements du modèle à traiter lors de la vérification. Nous faisons alors un premier constat. Nous avons vu précédemment que dans les documents d'exigences d'un système, celles-ci sont souvent exprimées, informellement,

dans un contexte donné de l'exécution du système. Les exigences sont associées à des phases d'utilisation spécifique du système (initialisation, reconfiguration, modes dégradés, changement d'état, etc.). Il n'est donc pas nécessaire de les vérifier sur tous les scénarios de l'environnement. Dans les travaux décrits dans (Dwyer *et al.*, 1999), (Konrad *et al.*, 2005), les auteurs ont proposé d'identifier la portée ($scope$) d'une propriété en permettant à l'utilisateur de préciser le contexte temporel d'exécution de la propriété à l'aide d'opérateurs ($Global, Before, After, Between, After - Until$). Ceux-ci permettent de localiser les exigences à vérifier dans un contexte temporel particulier d'exécution du modèle à valider. Le scope indique si la propriété doit être prise en compte, par exemple, durant toute l'exécution du modèle, avant, après ou entre des occurrences d'évènements.

Mais en pratique, dans des contextes d'exécution complexes lorsque, par exemple, l'environnement d'un système est composé de plusieurs acteurs s'exécutant en parallèle, ces opérateurs ne sont pas aisés à utiliser pour spécifier cette localisation. Le déroulement des phases d'exécution de l'environnement peut être alors difficile à décrire avec les opérateurs proposés à cause de l'enchaînement des interactions entre l'environnement et le modèle. Nous proposons donc de mettre en œuvre un lien entre chaque exigence ou propriété à vérifier et une phase d'exécution du modèle à valider. Les propriétés sont localisées explicitement dans un contexte temporel déterminé par le cahier des charges. Dans le cas de documents d'exigences industriels que nous avons eu à traiter, ce lien exigence-contexte est rarement explicité formellement. Parfois il transparait dans des descriptions disséminées dans plusieurs documents. Nous proposons donc de lier explicitement et formellement les propriétés formalisées à leur contexte d'exécution spécifique pour limiter la portée de la propriété. L'avantage est de spécifier explicitement les conditions, c'est-à-dire, le contexte d'exécution, pour lesquelles une propriété donnée doit être vérifiée. Ceci a pour conséquence de simplifier l'expression de la propriété et d'être facilement plus compréhensible.

Ce qui nous amène à un deuxième constat. Pour diminuer l'explosion combinatoire, un moyen peut être de partitionner l'activité de vérification en un ensemble d'actions de vérification. Chaque vérification d'un ensemble de propriétés est exécutée en restreignant le nombre de comportement du modèle en le plongeant explicitement dans un contexte spécifique qui se synchronise avec le modèle. Lors d'une vérification, le nombre d'états du système pour lesquels les propriétés sont à vérifier est alors considérablement réduit. Pour que cette approche soit fondée, le processus de développement du système doit inclure une étape de spécification de l'environnement permettant d'identifier explicitement des ensembles de comportements finis mais de manière complète. L'hypothèse forte que nous faisons pour mettre en œuvre ce processus méthodologique est que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. Nous justifions cette hypothèse, en particulier dans le domaine de l'embarqué, par le fait que le concepteur d'un système doit connaître précisément et complètement le périmètre (contraintes, conditions) de son utilisation pour pouvoir le développer correctement.

Nous mettons en œuvre cette approche en proposant le formalisme CDL comme un langage prototype qui permet à l'ingénieur de décrire les contextes et les propriétés qu'il souhaite vérifier sur son modèle. Nous présentons maintenant, dans les grandes lignes, ce langage.

## 3. Le langage CDL

Un modèle CDL permet à l'utilisateur de décrire le comportement de l'environnement du modèle à valider et les propriétés devant être vérifiées. Le comportement est considéré comme des enchainements de scénarios qui décrivent les interactions entre le modèle soumis à validation et des entités composant l'environnement de ce modèle. Ce DSL, basé sur UML2, permet, d'une part, de décrire plusieurs entités (nommés $acteurs$) contribuant à l'environnement et pouvant s'exécuter en parallèle, comme les acteurs $Device$ et $HMI$ de la figure 1. D'autre part, il intègre un langage de description de propriétés reposant sur la notion de patron. Un méta modèle de CDL a été défini et une sémantique décrite (Dhaussy *et al.*, 2010) en terme de traces s'inspirant des travaux de (Haugen *et al.*, 2005) et (Whittle, 2002).

### 3.1. *Description hiérarchique du contexte*

Un modèle CDL[2] permet de décrire formellement un environnement d'un système. D'un point de vue syntaxique, un modèle CDL est structuré, de manière graphique et hiérarchique, en 3 niveaux. Au premier niveau, des diagrammes de cas d'utilisation décrivent, par des diagrammes d'activités et de manière hiérarchique, des enchaînements d'activités des entités s'exécutant en parallèle et constituant l'environnement. Les diagrammes de ce niveau font référence à des diagrammes du même niveau ou des scénarios décrits au niveau 2 également sous la forme de diagrammes d'activités. Les diagrammes de niveau 2 décrivent des enchainements de scénarios, ceux-ci étant décrits au niveau 3 par des diagrammes de séquence UML2.0 simplifiés (ITU, 1996). Il est à noter qu'une version textuelle du langage est aujourd'hui en cours de spécification.

Lors de la compilation d'un modèle CDL, celui-ci sera déplié (mis à plat). Nous pouvons donc considérer que le modèle est structuré par un ensemble de diagrammes de séquences (MSCs) reliés entre eux à l'aide de trois opérateurs : celui de séquencement ($seq$), l'opérateur parallèle ($par$) et l'alternative ($alt$). Une fois le contexte décrit par un ensemble de MSCs et déplié, il est partitionné de telle manière à générer un ensemble de scénarios à composer avec le système et les propriétés à vérifier : les observateurs. Ces observateurs sont décrits dans une syntaxe particulière et dérivent des travaux de (Dwyer *et al.*, 1998).

2. Pour la syntaxe détaillée, voir (Dhaussy *et al.*, 2010) et http ://www.obpcdl.org.

Ce langage a des similitudes avec les $HMSC$ (ITU, 1996) et les étend dans plusieurs directions : D'abord il permet l'ajout de compteurs sur le MSC permettant ainsi un partitionnement de scénarios finis. L'objectif des compteurs est de permettre de limiter les boucles d'exécution des entités de l'environnement. Chacun d'entre eux est associé à un nœud d'un diagramme. La gestion des compteurs permet d'assurer, comme décrit dans (Roger, 2006), un dépliage fini lors de la construction des automates du contexte dans le langage d'implantation de l'outil de vérification choisi. Des variables permettent, quant à elles, de mémoriser des états de l'environnement. Elles peuvent être référencées dans les gardes conditionnant le séquencement dans les diagrammes d'activités.

CDL permet aussi de rattacher des propriétés sur des MSCs spécifiques de manière à simplifier la formalisation des propriétés. Dans cette article, le contexte est modélisé à l'aide d'une sous-partie du langage CDL suffisante ici pour la description de l'environnement du système présenté. En particulier, ni les compteurs, ni les gardes CDL ne sont utilisés. La figure 2 illustre graphiquement un modèle partiel du contexte du système $S\_CP$. Le cas d'utilisation et le diagramme de séquence de la figure 1 sont modélisés et complétés pour créer le modèle de contexte.

Dans les cas d'applications industrielles, compte tenu de la complexité potentielle des interactions entre le modèle et son environnement, la construction d'un seul modèle CDL peut être difficile. Il est donc souhaitable que l'utilisateur spécifie un ensemble de modèles CDL, chacun correspondant à des cas d'utilisation du modèle à valider.

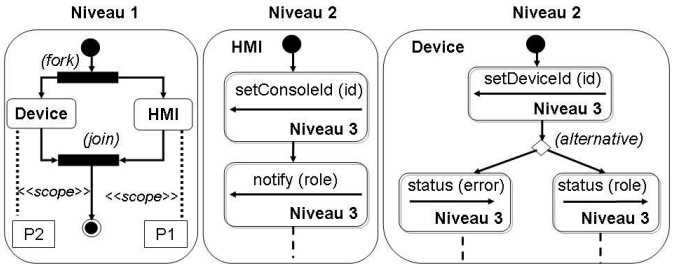



**Figure 2.** *Illustration d'un modèle CDL partiel du contexte du système $S\_CP$*

### 3.2. *Formalisation de CDL.*

Un contexte est décrit sous la forme d'une composition séquentielle, parallèle ou alternative, de diagrammes de séquences. D'un point de vue syntaxique, on peut résumer les opérateurs du langage CDL aux trois types d'opérateurs précités : $seq$, $par$ et $alt$. Ceux-ci permettent de structurer l'expression du comprtement d'un environnement par combinaison entre eux.

Formellement, un contexte est un processus fini produisant et recevant des événements décrits par la grammaire suivante :

$$\begin{array}{lll} C & ::= & M \mid C_1.C_2 \mid C_1 + C_2 \mid C_1 \| C_2 \\ M & ::= & \mathbf{0} \mid a!; M \mid a?; M \text{ with } a \neq \textit{null}_\sigma \end{array}$$

Un contexte est soit un simple MSC $M$ composé d'une séquence d'évènements d'émission $a!$ et de réception $a?$ terminée par un MSC terminal qui ne fait plus rien ($\mathbf{0}$), soit une composition séquentielle de deux contextes ($C_1.C_2$), soit une alternative entre deux contextes ($C_1 + C_2$), soit enfin une composition parallèle de deux contextes ($C_1 \| C_2$).

Par exemple, considérons l'exemple de la figure 2. Nous considérons ici que l'environnement est composé de 3 acteurs : un acteur $Device$ et deux acteurs $HMI_1$ et $HMI_2$. Le modèle peut être formalisé, avec la grammaire précédente, de la façon suivante [3] :

$$\begin{array}{lll} C & = & Device \parallel HMI_1 \parallel HMI_2 \\ Device & = & setDevice\ (id)\ ?\ ; \\ & & \quad ((status\ (error)\ !\ ;\ \ldots\ ;\ \mathbf{0}) + (status\ (role)\ !\ ;\ \ldots\ ;\ \mathbf{0})) \\ HMI_1 & = & setConsoleId\ (1)\ ?\ ;\ notify\ (role)\ ?\ ;\ \ldots\ ;\ \mathbf{0} \\ HMI_2 & = & setConsoleId\ (2)\ ?\ ;\ notify\ (role)\ ?\ ;\ \ldots\ ;\ \mathbf{0} \end{array}$$

### 3.3. *Sémantique de CDL*

Pour décrire la sémantique de CDL, considérons la fonction *wait*$(C)$ associant un contexte $C$ avec l'ensemble des évènements attendus dans son état initial.

$$\begin{array}{lll} \textit{Wait}\ (\mathbf{0}) \stackrel{\text{def}}{=} \emptyset & \textit{Wait}\ (a!; M) \stackrel{\text{def}}{=} \emptyset & \textit{Wait}\ (a?; M) \stackrel{\text{def}}{=} \{a\} \\ \textit{Wait}\ (C_1 + C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1) \cup \textit{Wait}\ (C_2) & \textit{Wait}\ (C_1.C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1) & \\ \textit{Wait}\ (C_1 \| C_2) \stackrel{\text{def}}{=} \textit{Wait}\ (C_1) \cup \textit{Wait}\ (C_2) & & \end{array}$$

Un contexte peut être assimilé à un processus communiquant de façon asynchrone avec le système mémorisant ses événements d'entrée (reçus du système) dans un *buffer*. La sémantique du langage CDL est définie par une relation $(C, B) \xrightarrow{a} (C', B')$ pour exprimer que le contexte $C$, associé à un *buffer* $B$ (une file d'attente d'événements émis par le système à destination de son environnement), "produit l'action" $a$ (qui peut être une émission ou une réception, voire l'événement $\textit{null}_\sigma$ si $C$ n'évolue pas), avant de devenir le nouveau contexte $C'$ soumis à un nouveau *buffer* $B'$. Cette relation est définie par les règles suivantes (cf figure 3) (dans l'ensemble de ces règles, $a$ représente un événement différent de $\textit{null}_\sigma$) :

3. Pour l'illustration dans ce papier, nous considérons ici que les comportements des acteurs se prolongent, ce qui est noté par les pointillés « ... ».

$$\frac{}{(a!; M, B) \xrightarrow{a!} (M, B)}\ [\text{pref1}] \qquad \frac{}{(a?; M, a.B) \xrightarrow{a?} (M, B)}\ [\text{pref2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1, B) \xrightarrow{a} (C_1', B')}{(C_1.C_2, B) \xrightarrow{a} (C_1'.C_2, B')}\ [\text{seq1}] \qquad \frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{(C_1.C_2, B) \xrightarrow{a} (C_2, B')}\ [\text{seq2}]$$

$$\frac{C_1' \neq \mathbf{0} \quad (C_1, B) \xrightarrow{a} (C_1', B')}{\begin{array}{l}(C_1 \| C_2, B) \xrightarrow{a} (C_1' \| C_2, B') \\ (C_2 \| C_1, B) \xrightarrow{a} (C_2 \| C_1', B')\end{array}}\ [\text{par1}] \qquad \frac{(C_1, B) \xrightarrow{a} (\mathbf{0}, B')}{\begin{array}{l}(C_1 \| C_2, B) \xrightarrow{a} (C_2, B') \\ (C_2 \| C_1, B) \xrightarrow{a} (C_2, B')\end{array}}\ [\text{par2}]$$

$$\frac{(C_1, B) \xrightarrow{a} (C_1', B')}{\begin{array}{l}(C_1 + C_2, B) \xrightarrow{a} (C_1', B') \\ (C_2 + C_1, B) \xrightarrow{a} (C_1', B')\end{array}}\ [\text{alt}] \qquad \frac{a \notin \mathit{wait}(C)}{(C, a.B) \xrightarrow{\mathit{null}_\sigma} (C, B)}\ [\text{discard}_C]$$

**Figure 3.** *Sémantique d'un contexte CDL*

– La règle *pref1* (sans pré-condition) spécifie qu'un MSC commence par une émission $a!$ puis poursuit le reste du MSC.

– La règle *pref2* (sans pré-condition) spécifie qu'un MSC commence par une réception $a?$ et, confronté à un *buffer* d'entrée contenant en tête cet événement $a$, il consomme cet événement puis poursuit le reste du MSC.

– La règle *seq1* spécifie qu'une séquence $C_1.C_2$ se comporte comme $C_1$ tant que celui-ci n'est pas terminé. La règle *seq2* spécifie en revanche que si $C_1$ s'achève (devient $\mathbf{0}$), la séquence devient $C_2$.

– La règle *alt* spécifie le choix entre deux contextes : $C_1 + C_2$ se comporte soit comme $C_1$, soit comme $C_2$. Mais une fois le choix fait, celui-ci est définitif (notons que cette règle a deux conclusions).

– Les règles *par1* et *par2* spécifie que la sémantique de l'opérateur parallèle est basée sur une sémantique d'entrelacement asynchrone des exécutions.

– Enfin, la règle *discard*$_C$ spécifie que, si un événement $a$ en tête du *buffer* d'entrée du contexte n'est pas attendu par celui-ci, alors il est perdu (supprimé de la tête du *buffer*).

$$\frac{(s,\mathcal{S},B_2) \xrightarrow[\sigma]{} (s',\mathcal{S},B_2')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[\sigma]{null_e} < (C,B_1.\sigma)|(s',\mathcal{S},B_2') >} \text{[cp1]}$$

$$\frac{(C,B_1) \xrightarrow{a!} (C',B_1')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[null_\sigma]{a} < (C',B_1')|(s,\mathcal{S},B_2.a) >} \text{[cp2]}$$

$$\frac{(C,B_1) \xrightarrow{a?} (C',B_1')}{< (C,B_1)|(s,\mathcal{S},B_2) > \xrightarrow[null_\sigma]{null_e} < (C',B_1')|(s,\mathcal{S},B_2) >} \text{[cp3]}$$

**Figure 4.** *Règle sémantique de la composition d'un contexte CDL et d'un système*

Notons que le caractère asynchrone de la composition parallèle (qui correspond à l'asynchronisme de l'environnement réel entourant le système) induit en pratique une explosion du nombre des traces possibles d'un contexte (ses scénarios). Par exemple, le contexte $C$ suivant décrit 5040 scénarios différents :

$$C \quad = \quad (r1?;(s1! + s2!)) \parallel (s3!;r3?) \parallel (s4!;r4?) \parallel (s5!;r5?)$$

### 3.4. *Composition du contexte avec le système*

Nous pouvons maintenant définir, comme une composition, la « fermeture » par un contexte $< (C,B_1) \mid (s,\mathcal{S},B_2) >$ d'un système $\mathcal{S}$ dans un état $s$ avec un buffer d'entrée $B_2$, avec son contexte $C$ avec un buffer d'entrée $B_1$ (notons que chaque composant, système et contexte, a son propre buffer).

L'évolution de $\mathcal{S}$ fermé par $C$ est donnée par la relation :

$$< (C,B_1)|(s,B_2) > \xrightarrow[\sigma]{a} < (C',B_1')|(s',B_2') >$$

pour exprimer que $\mathcal{S}$ dans l'état $s$ évolue vers l'état $s'$ en recevant l'événement $a$ (produit par le contexte) et en produisant la séquence d'événements $\sigma$ (vers le contexte). La fermeture du système par son contexte est ainsi définie par les trois règles suivantes : (cf figure 4) :

– Règle *cp1* : si $\mathcal{S}$ peut produire une séquence d'évènement $\sigma$, alors $\mathcal{S}$ évolue et $\sigma$ est placé en fin du *buffer* de $C$.

– Règle *cp2* : si $C$ peut émettre $a$, $C$ évolue et $a$ est placé en fin du *buffer* de $\mathcal{S}$.

– Règle *cp3* : si $C$ peut consommer $a$, alors il évolue tandis que $\mathcal{S}$ reste inchangé.

Notons que la composition de fermeture entre un système et son contexte est assimilable à une composition parallèle asynchrone : les comportements de $C$ et de $\mathcal{S}$ sont entrelacés et la communication est réalisée par échanges asynchrones via des *buffers*.

On note $< (C, B)|(s, B') > \not\rightarrow$ pour exprimer que le système et son contexte, associés à leurs *buffers* respectifs, ne peuvent plus évoluer selon les règles ci-dessus (le système fermé est bloqué ou le contexte est terminé). Nous définissons alors l'ensemble des traces (appelées des *runs*) du système fermé par son contexte et à partir d'un état $s$, par la fonction suivante :

$$\begin{aligned}[\![C \mid (s, \mathcal{S})]\!] \stackrel{\text{def}}{=} \{ & a_1 \cdot \sigma_1 \cdot \ \ldots \ a_n \cdot \sigma_n \cdot \mathit{end}_C \mid \\ < (C, \mathit{null}_\sigma) \mid (s, \mathit{null}_\sigma) > & \xrightarrow[\sigma_1]{a_1} < (C_1, B_1) \mid (s_1, B'_1) > \\ & \xrightarrow[\sigma_2]{a_2} \ \ldots \ \xrightarrow[\sigma_n]{a_n} < (C_n, B_n) \mid (s_n, B'_n) > \not\rightarrow \ \}\end{aligned}$$

$[\![C|(s, \mathcal{S})]\!]$ est l'ensemble des exécutions du système $\mathcal{S}$ fermé par le contexte $C$ et en considérant $s$ comme l'état de départ de $\mathcal{S}$.

Notons que, les contextes étant formés de compositions séquentielles ou parallèles de MSC finis, les traces des contextes sont donc nécessairement finies. En conséquence les *runs* du système fermé par son contexte sont nécessairement finis. Nous étendons chaque *run* de $C|(s, \mathcal{S})$ par un événement terminal spécifique $\mathit{end}_C$ permettant à l'observateur d'observer la fin d'un scénario et de "tester" d'éventuelles propriétés de vivacité (une propriété du type "un jour $a$" se traduira sous cette restriction de finitude des contextes par "un jour $a$ avant $\mathit{end}_C$").

### 3.5. *Formalisation des observateurs*

La formalisation de CDL concerne également l'expression des propriétés intégrée au langage. Nous considérons, dans cet article, que les propriétés sont modélisées comme des observateurs. Un observateur est un automate qui observe l'ensemble des évènements échangés entre le système $\mathcal{S}$ et son contexte $C$ (et les évènements survenant dans une trace (*run*) de $[\![C|(\mathit{init}, \mathcal{S})]\!]$) et qui produit un évènement *reject* quand la propriété devient fausse.

Nous montrons, dans l'exemple illustré de la propriété $P1$ au paragraphe 3.6, que l'automate l'observateur de la figure 5 contient un nœud reject. Ce nœud est atteint après la détection de l'événement $S\_CP\_hasReachState\_Init$ si la séquence $sendSetConsoleIdToHMI1$ et $sendSetConsoleIdToHMI2$ n'est pas produite dans l'ordre et avant $dMax\_cons$ unités de temps.

Nous considérons, dans la suite, qu'un observateur est un automate émettant un seul évènement de sortie (*reject*), pouvant capter des évènements, produits et reçus

par le système et son contexte, et telles que toutes les transitions étiquetées *reject* arrivent dans un état spécifique d'erreur (état de rejet).

**Semantique.** Soit un système $\mathcal{S}$ et un contexte $C$. Soit un observateur $\mathcal{O}$. $\mathcal{S}$ dans l'état $s \in \Sigma$ fermé par $C$ satisfait $\mathcal{O}$, noté $C|(s,\mathcal{S}) \models \mathcal{O}$, si et seulement si aucune exécution de $\mathcal{O}$ confrontée aux runs $r$ de $[\![C|(s,\mathcal{S})]\!]$ ne produit l'événement *reject*. C'est-à-dire :

$$C \mid (s,\mathcal{S}) \models \mathcal{O} \iff \forall r \in [\![C \mid (s,\mathcal{S})]\!],$$
$$(\mathit{init}_o, \mathcal{O}, r) \xrightarrow[\mathit{null}_\sigma]{} (s_1, \mathcal{O}, r_1) \xrightarrow[\mathit{null}_\sigma]{} \dots \xrightarrow[\mathit{null}_\sigma]{} (s_n, \mathcal{O}, r_n) \not\to$$

**Remarque.** Exécuter $\mathcal{O}$ sur un run $r$ de $C|(s,\mathcal{S})$ revient simplement à remplir le *buffer* de $\mathcal{O}$ par $r$, et dérouler son exécution à partir de son état initial. Si la propriété est satisfaite, alors cette exécution n'émettra que des événements vides ($\mathit{null}_\sigma$) (et donc jamais *reject*).

### 3.6. *Spécification des propriétés*

Des patrons de définition capturent, sous forme textuelle, des types de propriétés usuellement rencontrées dans les documents d'exigences. Dans (Dwyer *et al.*, 1999) et (Konrad *et al.*, 2005), les patrons sont classés en familles de base et prennent en compte les aspects temporisés des propriétés à spécifier. Les patrons identifiés dans une première approche permettent d'exprimer des propriétés de réponse (Response), de pré-requis (Precedence), d'absence (Absence), d'existence (Existence). Les propriétés font référence à des évènements détectables (Dhaussy *et al.*, 2009) comme des envois ou des réceptions de signaux, des actions, des changements d'état. Les formes de base peuvent être enrichies par des options ($Pre-arity, Post-arity, Immediacy, Precedency, Nullity, Repeatability$) à l'aide d'annotations (Konrad *et al.*, 2005). Dans CDL, nous enrichissons les patrons avec la possibilité d'exprimer des gardes sur les occurrences d'évènements exprimées dans les propriétés. En effet, il est souvent utile de pouvoir permettre ou non la prise en compte de la détection d'un évènement en fonction de l'état de l'environnement. Une occurrence d'évènements exprimée dans une propriété peut donc être associée à une garde référençant des variables déclarées dans le modèle CDL. Une autre extension apportée aux patrons est la possibilité de manipuler des ensembles d'évènements, ordonnés ou non ordonnés comme dans la proposition de (Janssen *et al.*, 1999). Les opérateurs $AN$ et $ALL$ précisent respectivement si un évènement ou tous les événements, ordonnés ($Ordered$) ou non ($Combined$), d'un ensemble d'évènements sont concernés par la propriété. Nous illustrons dans le listing 2, pour notre cas d'étude $S\_CP$, une propriété $P1$ qui correspond à une exigence $R1$ provenant de la décomposition d'exigence du listing 1 comme expliqué au paragraphe 4 :

**Exigence** $R1$ **:** *« Au cours de la procédure d'initialisation, $S\_CP$ doit associer un identificateur aux $NC$ consoles ($HMI$), avant un délai $dMax\_cons$. »*

R1 référence la communication entre $S\_CP$ et les consoles ($HMI$). Dans le diagramme de séquence de la figure 1, l'association à d'autres périphériques n'a aucun effet sur R1. Pour l'étude de cas, le nombre de consoles ($HMI$) considéré ici est deux ($NC$ = 2). $R1$ décrit une observation des occurrences d'événements. $S\_CP\_hasReachState\_Init$ fait référence à un changement d'état d'un processus du modèle à valider. $sendSetConsoleIdToHMI1$ et $sendSetConsoleIdToHMI2$ fait référence à des envois de signaux décrits dans le modèle CDL (figure 2).

```
Property P1;
     exactly one occurence of S_CP_hasReachState_Init
     eventually  leads-to  [0..dMax_cons]
     ALL Ordered
            one or more occurence of sendSetConsoleIdToHMI1
            exactly one occurence of sendSetConsoleIdToHMI2
     end
     S_CP_hasReachState_Init may never occurs
     one of sendSetConsoleIdToHMI1
                cannot occur before S_CP_hasReachState_Init
     one of sendSetConsoleIdToHMI2
                cannot occur before S_CP_hasReachState_Init
     repeatibility : true
```

**Listing 2.** Le système $S\_CP$ : la propriété de type de réponse correspondant à l'exigence $R1$.

Tel que mentionné dans la section 4, l'outil OBP (Dhaussy *et al.*, 2008) transforme chaque propriété en un automate observateur (Halbwachs *et al.*, 1993), incluant un nœud de rejet $reject$. La figure 5 illuste l'automate observateur généré pour la propriété P1.

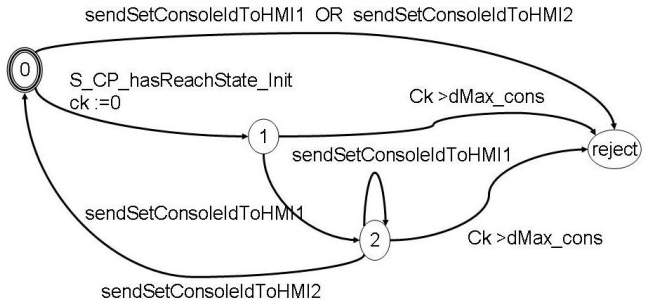



**Figure 5.** *Automate observateur généré pour la propriété P1.*

Avec les observateurs, nous pouvons traiter des propriétés de type sûreté et vivacité bornée. Une analyse d'accessibilité consiste en la recherche d'états reject qui sont

atteint par un observateur. Dans notre exemple, le nœud reject de l'observateur est atteint après la détection de l'événement $S\_CP\_hasReachState\_Init$ si la séquence $sendSetConsoleIdToHMI1$ et $sendSetConsoleIdToHMI2$ n'est pas produite dans l'ordre et avant $dMax\_cons$ unités de temps. A l'inverse, le nœud de reject n'est pas atteint si l'événement $S\_CP\_hasReachState\_Init$ n'est jamais reçu, ou si la séquence des deux événements ci-dessus est produite correctement (dans le bon ordre et avec le délai spécifié). Par conséquent, une telle propriété peut être vérifiée à l'aide une analyse de l'accessibilité mis en œuvre dans un vérificateur de modèle.

## 4. Méthodologie et outillage OBP

### 4.1. *Le processus de vérification*

Le processus méthodologique de vérification de modèles que nous proposons est basé sur une activité de vérification d'exigences sur un modèle de conception. Les vérifications ont pour but de contrôler que le modèle conçu est conforme à ses spécifications, c'est à dire à un ensemble d'exigences. Pour établir la vérification de l'ensemble des exigences, nous supposons que celles-ci puissent être formalisées sous la forme de propriétés logiques (comportementales de nature fonctionnelle ou non fonctionnelle). Nous supposons également que l'environnement du modèle (le contexte) à valider ainsi que les interactions entre l'environnement et le modèle soient modélisées formellement. Enfin, en vue d'utiliser un vérificateur formel, le modèle de conception doit pouvoir être simulable. Les propriétés, l'environnement et le modèle simulable constituent les données pertinentes et suffisantes pour conduire les vérifications d'exigences sur le modèle. La méthode que nous identifions inclut donc les phases suivantes (figure 6) :

– A partir d'un modèle de conception fourni par l'industriel, un modèle de conception formel est généré (manuellement ou semi-automatiquement) par une extraction des données utiles du modèle de conception. Ces données permettent d'obtenir un modèle comportemental formel simulable.

– A partir d'exigences fournies par le document d'exigences, des propriétés sont formalisées et un ou des modèles du comportement de l'environnement sont construits. Cette formalisation est réalisée à partir d'une interprétation d'exigences textuelles et d'une description de l'environnement du système modélisé. La rédaction d'exigences formelles et la modélisation du contexte sont d'autant plus aisées que, d'une part, la description du comportement de l'environnement est déjà formalisée (cas d'utilisation, scénarios, diagramme de séquence, etc.) et que, d'autre part, les exigences sont exprimées sans ambigüité. Dans notre approche, l'utilisateur rédige ses exigences avec les patrons de propriétés proposés dans CDL. Dans l'état actuel du langage CDL, certaines exigences du document d'exigences peuvent ne pas correspondre aux patrons proposés.

A ce niveau, il nous semble important que les exigences et le contexte soient bien dissociés des éléments du modèle de conception et formalisés de manière complète

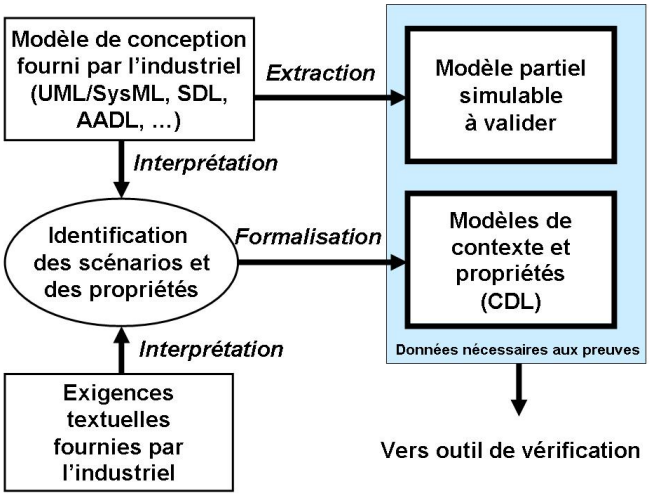



**Figure 6.** *Processus de prise en compte des données et des modèles du concepteur.*

et non ambigüe. Ils doivent également porter, de manière non ambiguë, sur des éléments bien identifiés du modèle de conception et à un même niveau d'abstraction. Le modèle quant à lui doit pouvoir être interprété de manière unique et être exécutable par un calculateur, moyennant une transformation (sémantique) de modèle adéquate. Dans la définition du processus de vérification, nous avons identifié le concept d'*Unité de Vérification* (UdV) (Dhaussy *et al.*, 2007) comme une structure de données regroupant toutes les données nécessaires et suffisantes pour conduire la vérification d'un ensemble de propriétés sur un modèle dans un contexte donné. Une UdV assure le lien entre le modèle CDL et le modèle à valider (figure 7). A partir des données contenues dans une UdV, les codes formels assimilables par les outils de vérification sont générés automatiquement.

Une condition préalable pour la mise en œuvre de ce processus est de disposer, de la part de l'industriel, de spécifications qui permettent d'identifier les contextes et les exigences qui peuvent être soumises à la vérification et qui peuvent être reliées à un contexte. Dans beaucoup de documents industriels que nous avons traités, les informations portant sur les contextes étaient très souvent implicites ou réparties dans plusieurs documents. Des discussions ont donc été nécessaires avec les ingénieurs pour comprendre les différents contextes du système et les capturer dans les modèles CDL. Le processus de développement doit inclure une étape de spécification de l'environnement permettant d'identifier un ensemble complet de toutes les interactions entre l'environnement et le modèle, ce qui doit assurer un taux de couverture de 100%. L'atteinte de cette couverture correspond à l'hypothèse formulée précédemment, qui postule que le concepteur est capable d'identifier toutes les interactions possibles entre le système et son environnement. L'ensemble des interactions doit lui être fourni formellement comme un résultat du processus d'analyse de l'architecture logicielle conçue,

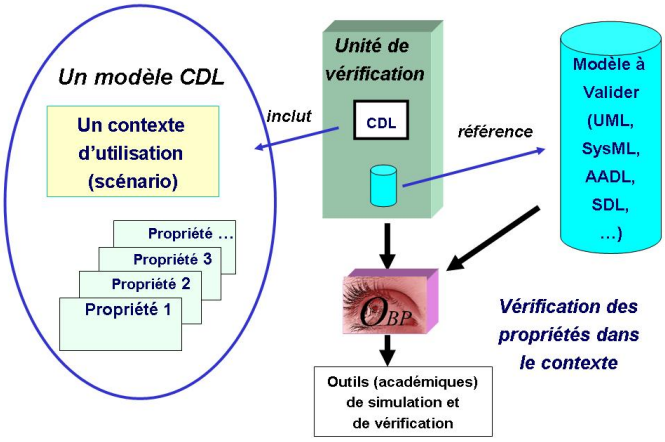



**Figure 7.** *Modèle CDL et Unité de Vérification.*

et ceci dans un processus de développement encadré et outillé. Compte tenu de la complexité de l'ensemble des interactions, il est préférable que l'utilisateur construise un ensemble structuré de modèles CDL spécifiques, chacun correspondant à des cas d'utilisation spécifiques, ce qui implique l'exploitation d'un ensemble d'unités de vérification.

En ce qui concerne la formalisation des propriétés, de nombreuses exigences comportaient beaucoup d'informations et ont dû être décomposées en exigences élémentaires pour simplifier leur formalisation avec les patrons. Par exemple, dans l'étude de cas décrit précédemment, l'exigence $R$ (listing 1) peut être décomposée en quatre sous-exigences comme suit :

**R1 :** *« Au cours de la procédure d'initialisation, $S\_CP$ doit associer un identificateur aux $NC$ consoles (IHM), avant un délai $dMax\_cons$. »*

**R2 :** *« Ensuite, $S\_CP$ doit associer un identificateur aux NE périphérique (Device), avant un délai $dMax\_dev$. »*

**R3 :** *« Chaque périphérique renvoie un message statusRole à $S\_CP$ avant un délai $dMax\_ack$.»*

**R4 :** *« $S\_CP$ transmet un message notifyRole à chaque périphérique et chaque console connectés. L'initialisation s'achève avec succès, lorsque $S\_CP$ a affecté tous les identificateurs de périphérique et de console et lorsque tous les messages notifyRole ont été envoyés. »*

Une fois ce travail de décomposition des exigences réalisé, la formalisation avec les patrons CDL est beaucoup plus aisée.

### 4.2. *Le prototype OBP (Observer-Based Prover)*

Les modèles CDL sont traduits automatiquement, dans l'outil OBP (figure 8), en codes assimilables par un vérificateur. Dans nos expérimentations, OBP génère actuellement deux types de programmes : Des programmes basés sur des automates temporisés (Alur *et al.*, 1994) au format IF2 (Bozga *et al.*, 2002) en vue d'alimenter le simulateur IFx (Bozga *et al.*, 2002) et des programmes au format Fiacre (Farail *et al.*, 2008) permettant de connecter les outils TINA (Berthomieu *et al.*, 2004) ou CADP (Fernandez *et al.*, 1996). L'utilisation d'IFx implique de transformer les exigences spécifiées en automates observateurs. IFx génère, lors de l'exploration d'un modèle IF2, un système de transitions qui est analysé par OBP. Celui-ci délivre à l'utilisateur un résultat de vérification compréhensible (état de véracité des exigences et contre exemples filtrés) suite à une analyse d'accessibilité. Pour Tina, les propriétés doivent être exprimées sous la forme de formules logiques SELT (Berthomieu *et al.*, 2004). SELT vérifie ces formules suite à l'exploration.

L'utilisation des outils de vérification connectés à l'outil OBP implique des transformations des modèles d'entrée de l'utilisateur et des modèles de contexte CDL vers les programmes formels exploités par les outils de vérification, IF2 ou Fiacre. Des chaînes de transformation sont implantées dans OBP. Actuellement, les modèles de conception à valider sont importés dans OBP (figure 8) au format IF2 ou Fiacre. Pour importer des modèles dans des formats UML 2, AADL (Feiler *et al.*, 2006) ou SDL (ITU, 1992), il est nécessaire de mettre en œuvre des traducteurs adéquats comme ceux étudiés dans les projets TopCased[4], Oméga[5] ou VerifMe[6]. Ils permettront de générer automatiquement, à partir des modèles de conception de l'utilisateur, des programmes formels IF2 ou FIACRE.

OBP importe les modèles CDL, les déplie les partitionne pour produire des graphes acycliques de contexte au format IF2 ou Fiacre. Ceux-ci représentent des ensembles des chemins d'exécution de l'environnement. Ce sont ces graphes qui sont ensuite composés avec le modèle à valider au sein des explorateurs IFx ou TINA. C'est cette partition du contexte en un ensemble de graphes d'exécution qui permet d'aboutir, lors de la composition, à des graphes d'états de taille limitée rendant possible l'analyse d'accessibilité (dans le cas de IFx) ou le model-cheking (dans le cas de TINA). L'outil OBP génère, par une technique de transformation de modèles, les automates observateurs à partir des propriétés qui sont décrites dans le modèle CDL.

La génération des graphes de contexte temporisés implique plusieurs phases automatisées et implantées par des programmes de transformation de modèles écrits en java :

– Le dépliage des contextes en des automates nommés contextes concrets. Le dépliage prend en compte l'ensemble des valeurs des compteurs, ce qui assure la termi-

4. http ://www.topcased.org
5. http ://www-Omega.imag.fr
6. http ://www.verifme.net

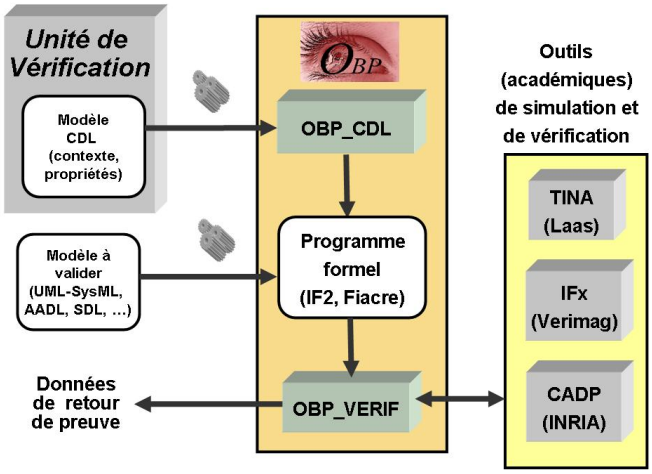



**Figure 8.** *L'outil OBP (Observer-Based Prover).*

naison de la génération du contexte concret.

– La génération d'un graphe résultant de l'entrelacement des contextes concrets avec la prise en compte de leurs exécutions parallèles.

– Le partitionnement du graphe en un ensemble de sous-graphes : cette phase consiste à produire des graphes acycliques permettant de découper le contexte en des graphes d'exécution finis et traduisibles en langage IF2 ou Fiacre.

– Enfin, chaque graphe généré est transformé en un automate IF2 ou Fiacre et composé avec le système et les automates observateurs (dans le cas de IFx).

Dans la configuration actuelle, OBP délivre à l'utilisateur un retour de vérification lui indiquant si chaque propriété à vérifier est détectée comme vraie ou fausse. Pour cela, dans le cas d'IFx, une analyse d'accessibilité est réalisée sur le résultat de la composition entre un graphe de contexte, un ensemble d'observateurs et le modèle à valider. S'il existe un état atteint reject d'un observateur de propriété, pour l'un des chemins, alors la propriété est considérée comme n'étant pas vérifiée. En cas d'échec d'une propriété, (détectée comme fausse), dans le cas de TINA, OBP interprète le résultat du model-checking SELT. OBP indique, à l'utilisateur, les séquences d'exécution de l'environnement (chemins), éventuellement filtrés, concernées durant la vérification. Cette indication peut l'aiguiller sur le scénario ayant mis en échec la propriété. Des travaux en cours de développement vise à obtenir des facilités pour restituer des données de plus haut niveau dans le modèle de l'utilisateur lui permettant de constituer son diagnostique.

L'outil OBP a été utilisé dans plusieurs études de cas qui sont décrites au paragraphe suivant.

## 5. Expérimentations et résultats

Nous relatons ici l'application de notre approche à six systèmes dans les domaines de l'avionique embarquée de deux partenaires industriels (dénotés A and B). Pour les cas d'étude ($CS_1$ à $CS_6$), quatre des composants logiciels proviennent du partenaire A et les deux derniers du partenaire B[7]. Pour chaque composant, le partenaire industriel a fourni les documents d'exigences opérationnelles (use cases, propriétés exprimées en langage naturel) ainsi qu'un modèle à valider et exécutable. Les modèles exécutables de composant sont décrits soit en UML2, complété par des programmes ADA ou JAVA, ou soit en langage SDL. Le tableau 1 donne le nombre d'exigences pour chaque composant, et indique la complexité du composant en termes de lignes de code[8].

Nous appliquons et suivons les étapes de la méthode décrite au paragraphe 4 : spécification des propriétés, description des contextes et construction des unités de vérification.

| | $CS_1$ | $CS_2$ | $CS_3$ | $CS_4$ | $CS_5$ | $CS_6$ |
|---|---|---|---|---|---|---|
| Langage de modélisation | SDL | SDL | SDL | SDL | UML2 | UML2 |
| Nombre de lignes de code | 4 000 | 15 000 | 30 000 | 15 000 | 38 000 | 25 000 |
| Nombre d'exigences | 49 | 94 | 136 | 85 | 188 | 151 |

**Tableau 1.** *Classification des cas d'étude.*

### 5.1. *Spécification des propriétés*

Dans cette étape, les exigences décrites en langage naturel sont exprimées sous la forme de propriétés temporelles. Pour créer les modèles CDL en exploitant les patrons de propriétés, nous avons analysé les documents d'exigences textuelles en identifiant les exigences qui pouvaient être traduites dans des automates observateurs. Lors de cette étape, nous avons constaté que la plupart des exigences se déclinaient non pas en une mais en plusieurs propriétés (comme nous l'avons déjà montré dans l'exemple de $S\_CP$). De plus, dans les exigences textuelles, différents niveaux de description, d'abstraction, sont mêlés. Nous avons donc extrait seulement les exigences correspondantes au niveau d'abstraction du modèle à valider. Finalement, nous avons constaté que certaines exigences étaient ambiguës. Nous avons donc du lever ces ambigüités, et pour cela clarifier leur sémantique par un dialogue avec nos partenaires industriels.

Le tableau 2 indique le nombre de propriétés qui ont pu être traduites depuis les exigences. Nous avons classé les exigences qui ont été traitées en trois catégories en

7. $CS_5$ correspond à l'étude de cas $S\_CP$ décrit partiellement en section 2.
8. Pour les cas $CS_1$ à $CS_4$ : le nombre indique le nombre de lignes de code SDL. Pour le cas $CS_5$ (resp. $CS_6$) : il indique le nombre de lignes de code ADA (resp. JAVA).

indiquant, pour chaque cas et pour chaque classe, les pourcentages par rapport à l'ensemble des exigences : les exigences « prouvables » sont celles qui peuvent être capturées avec notre approche et peuvent être converties en observateurs. La technique de vérification peut être appliquée sur un contexte donné sans explosion combinatoire. La catégorie « non calculable » représente les exigences qui peuvent correspondre à un patron, mais ne peuvent pas se traduire par un observateur. Par exemple, les propriétés de vivacité non bornées (liveness) ne peuvent être traduites sous forme d'un observateur, qui ne permet que de capturer les propriétés de vivacité bornée. Nous pourrions cependant traiter ces exigences en générant une formule de logique temporelle qui pourrait alimenter un vérificateur comme TINA. La catégorie non prouvable correspond aux exigences qui ne peuvent pas du tout être interprétées avec notre approche. C'est le cas lorsqu'une propriété fait référence à des événements indétectables (non observables) pour un observateur, tels que l'absence d'un signal.

| | $CS_1$ | $CS_2$ | $CS_3$ | $CS_4$ | $CS_5$ | $CS_6$ | Moyenne |
|---|---|---|---|---|---|---|---|
| Prouvables | 38/49 (78%) | 73/94 (78%) | 72/136 (53%) | 49/85 (58%) | 155/188 (82%) | 41/151 27%) | 428/703 (61%) |
| Non calculables | 0/49 (0%) | 2/94 (2%) | 24/136 (18%) | 2/85 (2%) | 18/188 (10%) | 48/151 (32%) | 94/703 (13%) |
| Non Prouvables | 11/49 (22%) | 19/94 (20%) | 40/136 (29%) | 34/85 (40%) | 15/188 (8%) | 62/151 (41%) | 181/703 (26%) |

**Tableau 2.** *Nombre de propriétés traitées.*

Pour le cas $CS_6$, nous constatons que le pourcentage d'exigences prouvables (27%) est faible. En effet, il était très difficile de ré-exprimer les exigences depuis la documentation dans le cadre des patrons proposés. Nous aurions du consacrer beaucoup plus de temps pour interpréter les exigences avec notre partenaire industriel pour les aligner avec nos patrons. Par contre, pour le $CS_5$, nous notons que le pourcentage (82%) des propriétés prouvables est très élevé. En effet, la majorité des 188 exigences étaient en adéquation avec les patrons. Ceci est du au fait que nous avons eu des discussions très tôt dans le processus de développement avec les ingénieurs en charge de rédiger les exigences, ce qui a permis d'influer sur leur rédaction initiale.

### 5.2. *Exploitation des contextes*

Après que les propriétés aient été exprimées à l'aide de nos patrons, nous avons lié chaque propriété à des scénarios de l'environnement. Ici, le travail a consisté à exprimer les cas d'utilisation dans le format CDL. Pour les différents cas d'études, plusieurs contextes CDL ont été créés en fonction de la complexité des comportements et du nombre des acteurs interagissant avec l'environnement.

Le tableau 3 indique des exemples de performances constatées pour l'étude de cas $CS_5$ correspondant au système $S\_CP$[9]. Les tests sont effectués pour différents modèles CDL de différentes complexité (symbolisée par la valeur $n$). Le tableau indique les temps d'exploration d'un modèle sans et avec la mise en œuvre de contextes.

Lorsque les contextes ne sont pas exploités, il indique le nombre d'états et de transitions générés lors de l'exploration par TINA. Les cases "***" symbolisent l'occurence d'une explosion combinatoire. Lorsque les contextes sont exploités et composés avec le modèle à valider, le tableau indique le nombre de contextes générés par OBP ainsi que le nombre, cumulés pour tous les contextes, d'états et de transitions explorés.

| | **Exploration sans contexte** | | | **Exploration avec contextes** | | | |
|---|---|---|---|---|---|---|---|
| n | Temps (sec) | Nombre d'états explorés | Nombre de trans. explorés | Temps (sec) | Nombre de contextes | Nombre d'états explorés | Nombre de trans. explorés |
| 1 | 10 | 16 766 | 82 541 | 11 | 3 | 16 884 | 82 855 |
| 2 | 25 | 66 137 | 320 388 | 26 | 3 | 66 255 | 320 802 |
| 3 | 91 | 269 977 | 1 297 987 | 92 | 3 | 270 095 | 1 298 401 |
| 4 | 118 | 939 689 | 4 506 637 | 121 | 3 | 939 807 | 4 507 051 |
| 5 | *** | *** | *** | 240 | 3 | 2 616 502 | 12 698 620 |
| 6 | *** | *** | *** | 2161 | 40 | 32 064 058 | 157 361 783 |
| 7 | *** | *** | *** | 4 518 | 55 | 64 746 500 | 322 838 592 |

**Tableau 3.** *Comparaison des performances avec et sans contexte (utilisation de l'explorateur TINA)*

La complexité d'un contexte dépend du nombre d'acteurs mis en jeu et des interactions avec le modèle du système. A chaque modèle, est attaché un ensemble de propriétés correspondant à une phase spécifique ou à des scénarios.

Dans les études de cas, chaque modèle CDL permet de lier un ensemble d'observateurs avec les comportements du contexte. Pour chaque graphe de contexte généré par OBP, un graphe d'accessibilité est généré par l'explorateur du vérificateur utilisé et représente l'ensemble de toutes les exécutions possibles du modèle. Une propriété est dite « non vérifiée » par l'outil si un état « reject » d'un automate observateur correspondant est atteint : OBP affiche la liste des observateurs en erreur et les contre exemples. L'explosion combinatoire peut se produire lors de l'exploration. La création des contextes spécifiques adéquats permet de limiter le nombre des comportements pour un modèle. L'identification des configurations spécifiques qui limitent les espaces d'exploration du modèle doit être opérée avec méthode, comme évoqué précédemment pour que la vérification des propriétés reste fondée.

9. Tests effectués sur une machine Linux 64 bits, 4 Go RAM avec l'outil TINA v.2.9.8 et Frac v.1.4.2

## 6. Discussion et conclusion

Le prototype OBP et le langage CDL ont pu être utilisés pour mener des expérimentations, en contexte industriel. Pour chaque étude de cas, ces vérifications ont été réalisées par les ingénieurs eux-mêmes, avec l'aide de notre équipe, dans les limites des fonctionnalités offertes par l'outil et du niveau d'expression actuel du langage CDL. Cette technique a permis aux utilisateurs d'obtenir un diagnostic pour chaque exigence traitée.

### 6.1. *Bénéfices de l'approche*

**La formalisation du comportement de l'environnement (contextes).** L'apport des modèles CDL, en relation avec le modèle à valider, est d'offrir, d'une part, un cadre pour décrire le comportement des acteurs de l'environnement du modèle. D'autre part, la description de l'environnement permet de restreindre l'exécution du modèle qui interagit avec lui. Parfois, dans des applications industrielles, cette description est informelle et/ou non complète. L'approche modèle permet à l'utilisateur de formaliser cet environnement et de préciser, dans un ensemble de modèles CDL, les cas d'utilisation du composant développé. Cette formalisation ne peut être que bénéfique pour une meilleure conception, même si l'utilisateur ne l'exploite pas, par la suite, pour des analyses formelles. Pour certains modèles, le nombre de modèles CDL peut être important selon la manière dont le système interagit avec les acteurs de l'environnement. C'est suite à une analyse des fonctionnalités du système à développer que l'utilisateur peut décrire exhaustivement l'environnement et les modéliser en CDL. Ceux-ci sont basés sur des diagrammes d'activité « à la UML » et des diagrammes de séquences qui sont d'un accès aisé pour un ingénieur. Conceptuellement, les principes de CDL peuvent être implantés dans d'autres formalismes.

La synthèse des résultats des vérifications, montrée précédemment, fait apparaître la possibilité d'exploiter des ensembles de contextes pour réduire la complexité des modèles à valider. Pour chaque étude de cas, il a été possible de créer des unités de vérification, incluant chacune un modèle CDL, et les fournir en entrée d'OBP pour la génération des ensembles de graphes de contexte. Chaque graphe est composé avec le modèle du composant à valider et les automates observateurs. C'est cette composition qui permet de réduire la taille (finie) du système de transition généré lors de la simulation exhaustive et d'obtenir un résultat de vérification en un temps raisonnable pour l'utilisateur. Dans ce cas, CDL contribue à surmonter l'explosion combinatoire en rendant possible une vérification partielle sur un ensemble restreint de scénarios spécifiés par les automates de contexte.

**La formalisation des exigences.** Les résultats montrent également qu'une grande partie des exigences, devant être traitées, a pu faire l'objet d'une formalisation basée sur les patrons de définition proposés et a pu donner lieu à la génération d'automates observateurs. L'utilisateur peut décrire les exigences, sous une forme textuelle encadrée et non ambiguë, conforme à une grammaire. Celle-ci est suffisamment ex-

pressive pour décrire, sans trop de difficultés pour l'utilisateur, les propriétés de réponse, d'existence et d'absence qui sont des propriétés de sûreté ou de vivacité bornée. Celles-ci sont bien adaptées pour décrire des comportements caractérisant les interactions entre entités ou composants communicants. Elles correspondent, dans ce domaine, à un grand pourcentage des exigences exprimées. Ce mode d'expression permet de lever toute erreur d'interprétation car la sémantique des patrons est clairement définie (sémantique de la logique linéaire (Pnueli, 1977)). Chaque exigence formalisée peut être ensuite traduite automatiquement en un programme formel (automate observateur) qui se compose, lors de la vérification, avec le programme associé au modèle. Il faut noter que dans le domaine des protocoles, ce type de propriété est majoritaire.

**Les résultats de vérification.** Lors de l'étude de l'ensemble des exigences pour chaque cas d'étude, en moyenne 13% de celles-ci (non-calculable) (cf. tableau 2) correspondait à des propriétés de vivacité non traduisibles, avec notre approche, en observateurs. Elles ont donc demandé, après discussion avec les utilisateurs, une adaptation pour les mettre sous la forme de propriétés de vivacité bornée. Les 61% (prouvables) ont été réécrites sans difficultés avec les patrons. Pour le reste des 26% (non prouvables), elles doivent être encore discutées avec les partenaires industriels pour trouver une expression équivalente et prouvable. Dans les études de cas, environ quarante exigences importantes ont été formellement vérifiées avec nos outils. Lors des vérifications, nous avons constaté, dans deux études de cas ($CS_1$ et $CS_5$), une exécution qui n'a pas satisfait aux exigences. Ces deux cas étude correspondent à des systèmes embarqués déjà actuellement opérationnels. Les techniques classiques de simulation n'avaient pas, jusque là, permis de trouver ces erreurs. Une des raisons est que ces erreurs apparaissaient dans des contextes spécifiques ne correspondant pas à des scénarios réellement opérationnels. Ceux-ci n'avaient donc pas été testés lors des simulations antérieures, ni lors des tests sur cible. Mais, aucun document de spécification des deux systèmes ne le précisait explicitement. Ceci montre qu'un manque de formalisation explicite des cas d'utilisation peut laisser un flou dans la spécification lors de la conception.

### 6.2. *L'appropriation du langage CDL*

Dans les cas d'étude, les diagrammes de contexte furent construits, d'une part, à partir de scénarios décrits dans les documents de conception et, d'autre part, à partir des documents d'exigence fournis par les industriels. Deux difficultés majeures sont alors apparues. La première est le manque de description complète et cohérente du comportement de l'environnement. Les cas d'utilisation décrivant les interactions entre un système ($S\_CP$ par exemple) et son environnement sont souvent incomplets. Par exemple, des données concernant les modes d'interaction peuvent être implicites. Le développement des diagrammes CDL requiert alors beaucoup de discussions avec des experts qui ont conçu les modèles afin d'expliciter toutes les hypothèses associées au contexte. La deuxième difficulté concerne la compréhension des exigences. Le problème provient de la difficulté à les formaliser en propriétés formelles. Celles-ci sont

regroupées dans des documents de niveaux d'abstraction différents correspondants aux exigences systèmes ou dérivées. Ces dernières sont des exigences rédigées suite à l'interprétation des exigences système au regard des choix de conception. L'ensemble des exigences analysées étaient rédigées sous forme textuelle et certaines d'entre elles donnaient lieu à des interprétations différentes. D'autres faisaient appel à une connaissance implicite du contexte dans lequel elles doivent être prises en compte. En effet, la plupart des exigences sont à prendre en compte dans une configuration donnée, lorsque le système a atteint une phase opérationnelle. Or les informations, concernant l'historique ayant mené à l'état dans lequel doit se trouver le système, ne sont en général pas explicités dans l'exigence. Elles font partie d'un ensemble d'informations qui doivent se trouver décrites explicitement dans les documents d'analyse du système. Mais parfois, elles ne le sont pas et doivent être alors déduites d'une interprétation sur le comportement du système, pouvant être faite par les experts qui en ont une connaissance parfaite.

Suite à notre proposition, au travers du langage CDL, de mise à disposition d'un cadre de formalisation des contextes et des exigences, les ingénieurs se le sont approprié et y ont trouvé immédiatement un intérêt pour mieux rédiger leurs spécifications. En ce sens, le couplage entre un modèle de type CDL et des ensembles de propriétés est une voie d'étude à poursuivre. CDL invoque un style de spécification très proche d'UML et donc lisible par les ingénieurs. Dans toutes les études de cas, les collaborateurs industriels ont indiqué que les modèles CDL améliorent la communication entre les développeurs possédant différents niveaux d'expérience. Ils les guident dans leur travail de structuration et de formalisation de la description des environnements de leurs systèmes et de leurs exigences. La collaboration avec les ingénieurs responsables de la rédaction des exigences les a motivés pour une prise en compte d'une approche plus formelle, ce qui est une amélioration par rapport à leurs pratiques précédentes. Nous pensons donc que de telles approches contribuent à une meilleure appropriation industrielle des techniques de vérification formelle.

Les contextes et les propriétés sont des artéfacts qu'il faut créer dans le but d'appliquer des techniques de vérification. Ce sont les données qui sont rassemblées dans des unités de vérification en vue d'effectuer les validations des modèles. Elles doivent être capitalisées si les modèles de conception évoluent au cours du cycle de développement. Il semble donc essentiel d'étudier un cadre méthodologique pour aider l'utilisateur à décrire et formaliser les contextes de vérification en tant que composants répertoriés dans un processus déroulant l'activité de vérification.

### 6.3. *Travaux en perspective*

Aujourd'hui, l'approche a été mise en œuvre dans un nombre limité de cas industriels (une dizaine de cas chez divers partenaires). Mais les résultats obtenus dans les expérimentations avec le langage CDL et OBP sont très encourageants et ils ouvrent, bien sûr, tout un ensemble d'axes de travaux de recherche à entreprendre.

Dans le cas de modèles de contextes complexes (acteurs multiples, nombreuses alternatives dans les comportements des acteurs), la traduction des modèles CDL par OBP peut mener à un grand nombre de graphes. Cela multiplie les temps de composition, de simulation et d'analyse d'accessibilité des états d'erreur ou de model-checking. En effet, la complexité d'analyse du modèle a disparue du fait de la technique de restriction des comportements du modèle par composition avec des contextes partiels. Mais cette complexité a été déportée, en amont, lors de la génération des graphes à exploiter. Face à ce problème, un travail est en cours (Dumas *et al.*, 2010; Dumas *et al.*, 2011) pour réduire ces ensembles. Nous cherchons à limiter le nombre de graphes pertinents en identifiant des critères d'équivalence qui permettent d'en éliminer au regard des propriétés à vérifier.

L'affichage des retours de vérification provenant des model-checkers pose également un problème de compréhension pour l'utilisateur. Aujourd'hui OBP affiche les traces d'exécution pour lesquels des observateurs ou des formules ont été falsifiés. Mais ces chemins peuvent être de très grande taille (plusieurs milliers de transitions) et sont donc difficilement interprétables par l'utilisateur car ils décrivent des exécutions au niveau sémantique des systèmes de transitions. Il faut donc concevoir des filtres qui n'affichent que des informations pertinentes et en nombre réduit et les animer dans les modèles de l'utilisateur. Une étude est en cours pour l'implantation d'interpréteurs des données de retours de vérification prenant en compte les intentions de vérification (contexte et exigences) et les relations entre les entités de l'environnement et celles du modèle.

Suite à la mise en œuvre de l'approche et des expérimentations, nous avons la nécessité de formaliser la méthodologie. La manière d'aborder la construction des modèles CDL, inclus dans les unités de vérification, a une implication directe sur la complexité engendrée lors de la vérification. Nous cherchons donc à pouvoir aiguiller l'utilisateur dans sa construction des modèles CDL. Les modèles manipulés par l'utilisateur en phase d'analyse d'un système (cas d'utilisation, diagrammes de séquences, etc.) doivent pouvoir être exploités pour générer automatiquement les modèles CDL et donc les unités de vérification. Ceci doit s'effectuer, d'une manière complète, pour obtenir une couverture complète des cas d'utilisation. Lors de l'utilisation des modèles CDL par les ingénieurs, nous avons constaté qu'il était envisageable de générer l'ensemble des modèles CDL à partir de structures de données de plus haut niveau qui comporteraient toutes les informations nécessaires à cette génération. Ceci présente l'avantage d'offrir à l'utilisateur un formalisme encore plus proche des formalismes qu'il a l'habitude d'utiliser. Un travail en cours (Raji *et al.*, 2010) vise à proposer dans ce but un formalisme nommé User Context Model (UCM) permettant de générer des modèles CDL. Les diagrammes de contexte CDL doivent aussi pouvoir être exploités pour concevoir les tests de l'implantation finale puisqu'ils modélisent les interactions avec le composant à valider.

## 7. Bibliographie


Alur R., Dill D. L., « A Theory of Timed Automata », *Theoretical Computer Science*, vol. 126, p. 183-235, 1994.

Berthomieu B., Ribet P.-O., Verdanat F., « The tool TINA - Construction of Abstract State Spaces for Petri Nets and Time Petri Nets », *International Journal of Production Research*, 2004.

Bozga M., Graf S., Mounier L., « IF2. A validation environment for component-based real-time systems », *CAV'02 : Proceedings of the 15th International Conference on Computer Aided Verification*, Springer-Verlag, Copenhagen, 2002.

Clarke E. M., Emerson E. A., Sistla A. P., « Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications », *ACM Trans. Program. Lang. Syst.*, vol. 8, n° 2, p. 244-263, 1986.

Dhaussy P., Auvray J., Belloy S. D., Boniol F., Landel E., « Using context descriptions and property definition patterns for software formal verification », *Workshop Modevva'08 (hosted by ICST'08)*, Lillehammer, Norway, 2008.

Dhaussy P., Boniol F., « Mise en œuvre de composants MDA pour la validation formelle de modèles de systèmes d'information embarqués », *Revue des Sciences et Techniques Informatiques*, vol. , p. 133-157, 2007.

Dhaussy P., Pillain P. Y., CDL (Context Description Language) : Syntaxe et sémantique, Technical report, ENSIETA, 2010.

Dhaussy P., Pillain P.-Y., Creff S., Raji A., Traon Y. L., Baudry B., « Evaluating Context Descriptions and Property Definition Patterns for Software Formal Validation », *in* B. S. Andy Schuerr (ed.), *12th IEEE/ACM conf. Model Driven Engineering Languages and Systems (Models'09)*, vol. LNCS 5795, Springer-Verlag, p. 438-452, 2009.

Dumas X., Boniol F., Dhaussy P., Bonnafous E., « Partial Order Application for Software Formal Verification », *European Congress on Embedded Real-Time Software (ERTS'10)*, Toulouse, 2010.

Dumas X., Boniol F., Dhaussy P., Bonnafous E., « Application of partial-order methods for the verification of closed-loop SDL systems », *26th Symposium on Applied Computing SAC'11*, Taïwan, Apr, 2011.

Dwyer M. B., Avrunin G. S., Corbett J. C., « Property specification patterns for finite-state verification », *FMSP*, p. 7-15, 1998.

Dwyer M. B., Avrunin G. S., Corbett J. C., « Patterns in property specifications for finite-state verification », *21st Int. Conf. on Software Engineering*, IEEE Computer Society Press, p. 411-420, 1999.

Farail P., Gaufillet P., Peres F., Bodeveix J.-P., MamounFilali, Berthomieu B., Rodrigo S., Vernadat F., Garavel H., Lang F., « FIACRE : an intermediate language for model verification in the TOPCASED environment », *European Congress on Embedded Real-Time Software (ERTS), Toulouse, 29/01/2008-01/02/2008*, SEE, janvier, 2008.

Feiler P., Gluch D., Hudak J., The Architecture Analysis and Design Language (AADL) : An introduction, Technical report, Society of Automotive Engineers (SAE), 2006.

Fernandez J.-C., Garavel H., Kerbrat A., Mounier L., Mateescu R., Sighireanu M., « CADP : A Protocol Validation and Verification Toolbox », *CAV '96 : Proceedings of the 8th Internatio-*

*nal Conference on Computer Aided Verification*, Springer-Verlag, London, UK, p. 437-440, 1996.

Halbwachs N., Lagnier F., Raymond P., « Synchronous observers and the verification of reactive systems », *in* M. Nivat, C. Rattray, T. Rus, G. Scollo (eds), *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Workshops in Computing, Springer Verlag, Twente, June, 1993.

Haugen O., Husa K. E., Runde R. K., Stolen K., « STAIRS towards formal design with sequence diagrams. », *Software and System Modeling*, vol. 4, n° 4, p. 355-357, 2005.

ITU, « Specification and Description Language (SDL) », *ITU-T Recommendation Z.100*, Geneva, 1992.

ITU, « Message Sequence Chart (MSC) », *ITU-T Recommendation Z.120*, Geneva, 1996.

Janssen W., Mateescu R., Mauw S., Fennema P., Stappen P. V. D., « Model Checking for Managers », *SPIN*, p. 92-107, 1999.

Konrad S., Cheng B., « Real-Time Specification Patterns », *27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA*, 2005.

Pnueli A., « The temporal logic of programs », *SFCS '77 : Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society, Washington, DC, USA, p. 46-57, 1977.

Raji A., Dhaussy P., Aizier B., « Automating Context Description for Software Formal Verification », *Workshop (MoDeVVa'10)*, Oslo, Norway, 2010.

Roger J.-C., Exploitation de contextes et d'observateurs pour la validation formelle de modèles, PhD thesis, ENSIETA, Univ. of Rennes I., December, 2006.

Smith R., Avrunin G., Clarke L., Osterweil L., « Propel : An Approach Supporting Property Elucidation », *24st Int. Conf. on Software Engineering(ICSE02), St Louis, MO, USA*, ACM Press, p. 11-21, 2002.

Whittle J., « Specifying precise use cases with use case charts », *MoDELS'06, Satellite Events*, p. 290-301, 2002.

# Critères de tests pour les automates de modes et application au langage Scade 6

Christophe Junke

Laboratoire LSL (CEA, LIST)
Centre CEA de Saclay, Gif-sur-Yvette, 91191 Cedex
junke.christophe@gmail.com

**Résumé** Les automates de modes de Scade 6 apportent au langage Lustre la notion d'états et de transitions tout en conservant l'approche flots de données du langage synchrone. Nous étudions et adaptons les critères d'automates usuels au cas particulier des automates de modes. Nous proposons une traduction de ces critères en termes Lustre avec horloges, afin de les couvrir avec l'outil de génération de séquences de tests GATeL. Cette traduction pour le moment manuelle repose sur celle des modèles Scade 6 vers Lustre, qui existe déjà par ailleurs. Les critères de couverture structurelle des automates peuvent alors être combinés à des critères fonctionnels exprimés par des objectifs de tests, et servir à la génération automatique de séquences de tests couvrantes dans GATeL.

## 1 Introduction

Les systèmes critiques doivent satisfaire de fortes exigences de fiabilité et de sûreté, ce qui motive l'usage de modèles formels dans leur développement. Nous nous intéressons en particulier aux systèmes modélisés selon l'approche synchrone [1] et à la génération automatique de tests avec GATeL [2,3]. Les langages synchrones se distinguent entre eux par le modèle de calcul qu'ils mettent en avant : un modèle orienté flots de données (Lustre [4], Signal), adapté pour l'expression de lois de commandes, ou un modèle orienté événements (Esterel), plus approprié pour contrôler l'état d'un système. Face aux problèmes complexes pouvant regrouper ces deux aspects, contrôle et commande, des formalismes mixtes existent, comme SyncCharts, Argos et récemment Scade 6, basé sur la notion d'automates de modes [5]. Le langage permet alors de spécifier des états et des transitions : à chaque instant, un automate est dans un unique mode de calcul. Il existe donc une meilleure séparation entre les comportements logiques et les calculs, ce qui simplifie la conception et la validation de programmes réactifs.

La mise en œuvre des automates de modes dans Scade 6 a donné lieu à l'intégration dans GATeL des flots temporisés de Lustre ainsi que des nouveaux opérateurs temporels nécessaires à l'expression de la sémantique des automates [6] ; ceux-ci peuvent en effet être traduits [7] de manière efficace, pour la compilation comme pour la génération de tests, vers un sur-ensemble du langage Lustre avec horloges. Cette intégration nous amène à nous intéresser aux critères de tests propres aux automates de modes dans notre outil de génération de tests. Le test à partir de ces spécifications étant un sujet récent, il n'existe pas à notre connaissance de travaux antérieurs liés à la définition de critères de tests pour les automates de modes. En revanche, des critères de test plus généraux existent pour les machines à états finis [8,9,10,11] et notamment le langage StateCharts [12]. Nous proposons d'adapter ces critères de tests au langage Scade 6, puis d'étendre les outils de sélection de cas de tests disponibles dans GATeL [13]. Un cas de test est spécifié en Lustre dans GATeL à l'aide d'un objectif de test, associé à une spécification du système et de son environnement. Le but de la génération est alors de construire une séquence finie d'entrées et de sorties compatibles avec la spécification, pour laquelle l'objectif est atteint à la fin

de ces séquences. Nous proposons de représenter les critères d'automates en terme d'objectifs de tests dans le langage Lustre avec horloges étendues et montrons comment couvrir de tels critères à l'aide des mécanismes de sélection de GATeL. Ainsi, un critère d'automates peut être traduit dans le langage Lustre en tirant parti de la traduction existante des automates. Ce critère exprime alors directement dans GATeL un ensemble de cas de tests pour lesquels il est possible d'utiliser le moteur de génération de séquences existant. De plus, la couverture structurelle de la spécification d'automate est facilement liée à des critères de couverture fonctionnelle, comme c'est déjà le cas avec les flots de données.

Notre contribution est donc l'adaptation de critères d'automates au modèle Scade 6 ainsi que leur expression sous la forme de termes Lustre directement utilisables par GATeL. L'automatisation de la traduction des critères d'automates est en cours d'expérimentation. La génération de séquences de tests à partir de ces critères traduits est quant à elle automatique, étant donnée l'exploitation des mécanismes de génération et de sélection de GATeL.

Le papier se découpe ainsi : nous commençons par un rappel sur le langage Lustre avec horloges, en définissant notamment les nouveaux opérateurs temporels introduits dans le langage. Nous décrivons ensuite la sémantique des automates de modes, leur traduction à l'aide de flots temporisés et définissons les concepts utiles pour, finalement, décrire et couvrir des critères de tests d'automates à travers la couverture d'objectifs de tests.

## 2 Langage Lustre

### 2.1 Flots de données

Le langage Lustre manipule des flots de données. Une description Lustre d'un système réactif contient un ensemble de déclarations de variables, de la forme `V=Exp`, où `V` désigne les valeurs successives du flot $(v_0, v_1, \ldots)$ et `Exp` est une expression de flot. Une terme Lustre peut exprimer des opérations arithmétiques, logiques et temporelles sur des flots de données. L'ensemble de ces flots évoluent selon une horloge globale commune afin de simplifier les synchronisations entre eux : une référence à une variable de flot $V$ désigne la même valeur de $V$ pour tous les flots. Les opérations usuelles sont étendues sur les flots de données comme suit : soient $X$ et $Y$ deux flots entiers définis par leurs valeurs successives $(x_0, x_1, \ldots)$ et $(y_0, y_1, \ldots)$ alors $X + Y = (x_0 \oplus y_0, x_1 \oplus y_1, \ldots)$. L'opérateur $\oplus$ désigne ici l'addition au niveau des *valeurs* portées par le flot. Afin de pouvoir exprimer des relations temporelles entre différents cycles, le langage Lustre fournit un opérateur de décalage `pre` : la valeur courante de `pre(`$X$`)` est la valeur précédente de $X$. L'opérateur n'étant pas défini au cycle zéro, il est nécessaire de le protéger par un opérateur d'initialisation, noté « `->` » et défini comme suit : $X$ `->` $Y = (x_0, y_1, y_2, \ldots)$. La version la plus récente du langage Scade permet également de manipuler des flots de types énumérés déclarés par l'utilisateur.

### 2.2 Horloges

Afin d'assouplir le principe de synchronisation globale des flots, Lustre propose d'autres opérateurs temporels permettant d'exprimer des décalages relativement à un flot. Pour cela, chaque flot $F$ est associé à une horloge. Celle-ci indique les cycles où le flot est présent. Les horloges classiques de Lustre sont définies à l'aide de flots booléens, mais sont généralisées aux flots de types énumérés dans Scade 6. Si $clk(F)$ est l'horloge de base, $F$ admet une valeur à tous les cycles du système ; sinon, $clk(F)$ est un couple $(id, v)$ et $F$ n'est défini que lorsque le flot $id$ est lui-même présent et vaut la constante $v$. Les identifiants d'horloge sont déclarés à l'aide

de l'opérateur `when` et leur valeur de vérité par `match`.

$$id \ : \ \{u, v, w\} = \texttt{(u,v,u,v,w,v,u)}$$
$$F \ : \ \texttt{int when id match v} = \texttt{(1,2,3)}$$
$$G \ : \ \texttt{int when id match v} = \texttt{(4,5,6)}$$
$$F + G = \texttt{(5,7,9)}$$

Dans l'exemple ci-dessus, on suppose que l'horloge de $id$ est `base`, que $F$ et $G$ sont des flots d'entiers de même horloge $(id, v)$. Alors, les séquences finies décrivent 7 cycles d'exécution du programme, au cours desquels $F$ et $G$ admettent trois valeurs. Les occurences de $v$ pour le flot $id$ déterminent les instants de présence de $F$ et $G$. Celles-ci sont plus courtes que celle de $id$, mais restent synchronisées entre-elles. En particulier, les opérations telles que l'addition, le retard unitaire, etc. s'appliquent de la même manière que précédemment. Les horloges permettent d'avoir des flots de données qui évoluent plus lentement que les cycles d'exécution du système.

Finalement, l'opérateur `merge` permet de recomposer un flot à partir de flots échantillonés sur des horloges complémentaires, c'est-à-dire des horloges $(id, v_i)$ de même identifiant $id$ ayant des valeurs de présence $v_i$ différentes. Ainsi, `merge(`$id$`,`$F_1$`,...,`$F_n$`)` réalise l'entrelacement d'autant de flots $F_1, \ldots, F_n$ que de valeurs du type énuméré $\{v_1, \ldots, v_n\}$ de $id$. Ces flots ont pour horloges respectives $(id, v_1), \ldots, (id, v_n)$. Alors, le flot résultant vaut à chaque cycle où il est défini la valeur courante de l'unique flot $\texttt{F}_j$ tel que la valeur de $id$ soit $v_j$.

*Exemple* Dans la figure 1, le flot $id$ possède un type énuméré admettant trois valeurs, `a`, `b` et `c` ; son horloge $clk(id)$ est présente aux cycles 0, 2, 3 et 4 et absente au cycle 1, ce qui est représenté par + (present) et − (absent) ; ces notations indiquent respectivement que le flot support de l'horloge s'évalue comme sa valeur de présence ou non. Les flots `X`, `Y` et `Z` sont supposés être des entrées de même horloge que $id$. Ces trois flots sont filtrés par l'opérateur `when` selon la valeur de $id$, les flots résultants `Xa`, `Yb` et `Zc` ayant pour horloges respectives $(id, a)$, $(id, b)$ et $(id, c)$. Finalement, ces trois flots sont réunis en un seul flot `M` de même horloge que $id$, à l'aide d'un `merge`.

| Cycles | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| $clk$(id) = $clk$(X) = $clk$(M) | + | − | + | + | + |
| id : {a,b,c} | a | | c | b | a |
| X : int | 1 | | 2 | 3 | 4 |
| Y : int | 11 | | 12 | 13 | 14 |
| Z : int | 5 | | 10 | 15 | 20 |
| Xa = X when id match a | 1 | | | | 4 |
| Yb = Y when id match b | | | | 13 | |
| Zc = Z when id match c | | | 10 | | |
| M = merge(id,Xa,Yb,Zc) | 1 | | 10 | 13 | 4 |

**Figure 1.** Trace d'exécution pour les opérateurs `when` et `merge`.

## 3 Automates de modes

Le langage LUSTRE avec horloges classiques ($when, \ldots$) et primitives temporelles étendues ($merge, \ldots$) nous sert à interpréter les automates de SCADE 6, pour l'encodage des états et des transitions. Nous présentons tout d'abord la sémantique des automates de modes, puis leur traduction vers ce langage.

### 3.1 Sémantique du modèle

Un automate de mode permet de calculer un même ensemble de variables dans différents états. Ces variables sont calculées dans l'unique mode de calcul actif, qui peut changer selon les transitions, mais qui reste implicitement le même tant qu'une transition n'est pas franchissable. Comme les conditions de transitions peuvent être des expressions booléennes Lustre arbitrairement complexes, il est possible que le franchissement d'une transition ne soit possible qu'après une période de calculs de plusieurs cycles dans un même mode. Un automate de mode possède un unique état initial et d'éventuels états finaux.

Il existe deux niveaux de transitions dans un automate (fortes et faibles), ce qui dans le cas général introduit plusieurs étapes de calculs au cours d'un même cycle. Au début d'un cycle de calcul, les conditions des transitions *fortes* sont évaluées, une après l'autre, dans leur ordre de définition. Dès que l'une d'elles est évaluée à vraie, elle est franchie, pouvant faire changer l'état actif de l'automate avant de commencer les calculs sur les sorties. Si aucune transition forte n'est franchissable, l'état actif est l'état courant. Une fois l'état actif connu, les sorties de l'automates sont évaluées, ainsi que les conditions des transitions *faibles*. Ces transitions sont elles aussi évaluées l'une après l'autre, selon leur ordre d'apparition dans la spécification, et déterminent le prochain état de l'automate. Si une transition faible est franchissable, elle est prise et son état d'arrivée définit le prochain état sélectionné de l'automate. Sinon, l'état n'est pas modifié. On peut remarquer que le prochain état sélectionné ne sera pas forcément l'état actif, si au prochain cycle une transition forte existe. Le formalisme possède une limitation importante pour garantir le calcul déterministe, et en un temps statiquement borné, des fonctions de transitions : *à chaque cycle d'exécution, au plus une transition, forte ou faible, peut-être franchie*. Cela signifie que si une transition forte est franchie au début de cycle, aucune transition faible, même si elle franchissable, ne sera empruntée. De même, si une transition forte est franchie, les transitions fortes de l'état cible ne seront pas prises en compte. Cette limitation permet d'éviter les cycles dans l'évaluation de l'état actif : s'il existait une transition forte étiquetée par `e` de $B$ vers $A$ et que `e` valait vrai en entrant dans $A$, le calcul de l'état actif ne terminerait pas.

```
automaton A
initial state A
        unless if e resume B
        let
           y = 1 ;
           z = 0->(pre(Z)+1) ;
        tel
        until  if (not e) resume C

        state B
        unless if e resume C
        let
           y = 2 ;
           z = 0->(pre(Z)+2) ;
        tel

        state C
        let
           y = 3 ;
           z = 0->(pre(Z)+3) ;
        tel
        until if e resume A
returns y,z ;
```

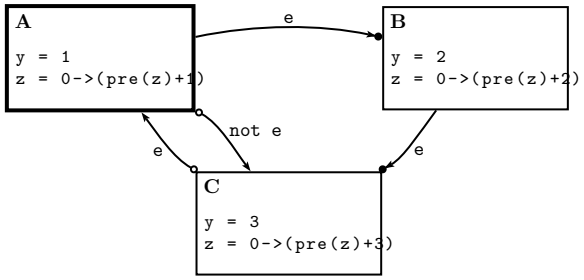



**Figure 2.** Automate de modes $\mathcal{A}$.

*Exemple* La figure 2 montre la spécification d'un automate que l'on appelle $\mathcal{A}$, à la fois sous forme textuelle SCADE 6 et sous la forme d'un graphe orienté. Un tel automate peut apparaître dans un nœud LUSTRE possédant au moins une entrée booléene `e` et définissant deux variables entières `y` et `z`. L'automate définit trois états, dans lesquels les blocs définis par les mots-clés `let` et `tel` contiennent les équations associées aux variables `y` et `z` dans chaque mode. Avant ou après chaque bloc apparaissent respectivement des transitions fortes ou faibles, reconnaissables par les mots-clés `unless` (transitions fortes) et `until` (transitions faibles). Celles-ci sont représentées dans le graphe par des flèches étiquetées, respectivement « $\rightarrow\!\!\bullet$ » et « $\circ\!\!\rightarrow$ ». L'automate $\mathcal{A}$ permet d'illustrer les différentes séquences de transitions, à savoir une transition faible suivie d'une forte (C-A-B), une forte suivie d'une faible (B-C-A) et ainsi de suite.

## 4 Expression des automates et critères à l'aide d'horloges

Nous considérons une traduction existante des automates de modes vers le langage LUSTRE avec horloges [7], implémentée dans le générateur de code de la plateforme Scade. Intuitivement, cette traduction consiste à effectuer les calculs des différents modes d'un automate en les cadençant sur des horloges complémentaires, c'est-à-dire de même flot support, mais pour des valeurs différentes de ce flot. Le flot support est une variable d'état, ses valeurs possibles sont regroupées dans un type énuméré et représentent les différents états de l'automate. Alors, les calculs et les transitions sont effectués à des rythmes différents, puis regroupés sur la même horloge que l'automate à l'aide d'un opérateur `merge`. En général, il faut considérer les deux types de transitions possibles, ce qui donne lieu à plusieurs variables d'états : l'état sélectionné, l'état actif et le prochain état sélectionné. La traduction des critères vers le langage LUSTRE est faite pour le moment de manière manuelle, mais nous pensons que celle-ci peut-être automatisée. En revanche, étant donné un objectif de test et un critère de couverture d'automate traduit en LUSTRE, la génération de séquences de tests pour des modèles LUSTRE est automatique, à travers l'outil GATeL. Nous définissons ici la sémantique des automates de modes à travers des flots temporisés, et introduisons les concepts qui nous seront nécessaires pour exprimer les critères de tests dans GATeL.

*Modèle* Un automate $A = (S, I, Ts, Tw, V)$ est donné par un ensemble d'états $S$, un état initial $I \in S$, des listes triées de transitions fortes $Ts$ et faibles $Tw$, ainsi que l'ensemble des variables de flots $V$ que cet automate doit calculer. On note [] la liste vide et $x :: T$ toute liste construite à partir de la sous-liste $T$ par ajout en-tête de celle-ci d'un élément $x$ quelconque. Par abus de notation, on assimilera une liste à un ensemble lorsque l'ordre des éléments ne sera pas important. Une transition est un tuple de la forme $(X, c, Y)$, avec $X \in S$ un état source, $c$ la condition de transition, une expression LUSTRE de type `bool`, et $Y$ l'état cible de la transition. Les trois composantes d'une transition $t$ sont notées respectivement $t.src$, $t.cond$ et $t.dst$. Avec ces notations, l'automate $\mathcal{A}$ de la figure 2 s'exprime ainsi :

$$\mathcal{A} = (S, \mathtt{A}, T, V)$$
$$S = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$$
$$V = \{\mathtt{y}, \mathtt{z}\}$$
$$Ts = \{(\mathtt{A}, \mathtt{e}, \mathtt{B}), (\mathtt{B}, \mathtt{e}, \mathtt{C})\}$$
$$Tw = \{(\mathtt{C}, \mathtt{e}, \mathtt{A}), (\mathtt{A}, \mathtt{(not\ e)}, \mathtt{C})\}$$

*Fonctions de transition* Les listes de transitions sont supposés triés selon l'ordre dans lequel elles sont spécifiées. Soient $s$ et $s'$ deux états de $S$ tels que $s \neq s'$, et $T$ une liste de transitions. On définit alors la fonction de transition $trans(s, T)$ de la manière suivante :

$$\begin{aligned} trans(s, []) &= s \\ trans(s, (s', c, t) :: T) &= trans(s, T) \\ trans(s, (s, c, t) :: T) &= \texttt{if } c \texttt{ then } t \texttt{ else } trans(s, T) \end{aligned}$$

La fonction retourne le prochain état de l'automate, étant donnés l'état courant et l'ensemble ordonné de transitions. Il est possible que le prochain état soit le même que l'état courant, bien qu'une transition ait été prise. On définit $fired(s, T)$ et $fired(T)$, pour tout état $s$ et pour un ensemble quelconque $T$ de transitions (l'ordre des transitions n'a pas d'importance ici).

$$\begin{aligned} fired(\{t_1, \ldots, t_n\}) &= t_1.cond \texttt{ or } \ldots \texttt{ or } t_n.cond \\ fired(s, T) &= fired(\{t \in T \mid t.src = s\}) \end{aligned}$$

Le prédicat $fired(T)$ indique si au moins une des conditions de franchissement de l'ensemble $T$ de transitions est réalisée. Il est restreint aux transitions sortantes d'un état à l'aide du prédicat $fired(s, T)$, qui nous permet de savoir, lors du calcul du prochain état, si une transition forte a été franchie au cycle courant depuis l'état sélectionné. En effet, s'il existe une transition forte franchissable depuis l'état sélectionné, $fired$ s'évalue comme `true` et la transition est nécessairement prise. Le prédicat $fired$ nous permet de garantir la propriété d'une seule transition prise par cycle, car il conditionne le franchissement des transitions faibles.

Finalement, pour une liste de transitions $T$ et pour toute transition $x$ présente dans $T$, on définit le prédicat $Cond(x)$ comme étant la condition de franchissement de $x$ ; en effet, $x$ n'est franchissable que si $x.cond$ est vraie alors que les conditions de transition apparaissant avant dans la spécification ne sont pas réalisées. On pose $Cond(x) = Cond(x, T)$, qui se traduit en Lustre de la manière suivante, pour toute liste $L$ et toutes transitions $x$ et $y$ avec $x \neq y$ :

$$\begin{aligned} Cond(x, x :: L) &= x.cond \\ Cond(x, y :: L) &= \texttt{not(}y.cond\texttt{) and } Cond(x, L) \end{aligned}$$

*Évaluation dans le contexte d'un mode* Le mécanisme général de la traduction des automates vers le langage avec horloges consiste à calculer les sorties et les transitions de chaque état d'un automate à l'aide de flots sur des horloges complémentaires (une horloge par état), par filtrages (opérateur *when*), puis à entrelacer les différents calculs sur une seule et même horloge par un opérateur *merge*. Ainsi, l'unique état actif est représenté par une unique horloge présente à un cycle, et seuls les calculs dans cet état sont effectivement réalisés et remontés vers les sorties du système réactif. Or, le ralentissement à l'aide d'un opérateur *when* n'a l'effet escompté que s'il est appliqué aux entrées du système. En effet, lorsque l'on applique un opérateur *when* autour d'un expression $E$, les calculs de $E$ sont réalisés, puis filtrés. En revanche, une fois une expression filtrée, les calculs sur cette expression ne sont effectués que lorsque l'horloge est présente. C'est pourquoi nous introduisons la fonction de réécriture $[\![Exp]\!]_v^{id}$, où $Exp$ est une expression Lustre et $(id, v)$ une horloge. Cette fonction est définie récursivement sur les termes du langage et permet de ralentir une expression sur une horloge donnée, en filtrant les entrées dont elle dépend. Sur tous les termes composés du langage, elle s'applique aux sous-termes directement. Pour tout terme $T$, où $T$ est une constante ou une variable d'entrée, elle ralentit $T$ sur l'horloge $(id, v)$. Pour toute variable calculée $O$ associée à une expression de définition $Exp_O$, la fonction introduit la variable intermédiaire $O_v$, dont l'expression est le ralentissement de $Exp_O$ selon l'horloge $(id, v)$.

Soient $A$, $B$, $C$ et $E$ des expressions quelconques et $op$ un opérateur binaire LUSTRE ; La définition (partielle) de la fonction est la suivante :

$$\begin{aligned}
[\![A \ op \ B]\!]_v^{id} &= [\![A]\!]_v^{id} \ op \ [\![B]\!]_v^{id} \\
[\![\texttt{pre(}E\texttt{)}]\!]_v^{id} &= \texttt{pre(}[\![E]\!]_v^{id}\texttt{)} \\
&\vdots \\
[\![T]\!]_v^{id} &= T \ \texttt{when id match v} \\
[\![O]\!]_v^{id} &= O_v, \text{ avec } O_v = [\![Exp_O]\!]_v^{id}
\end{aligned}$$

Cette réécriture nous permet d'exprimer les calculs effectués dans chaque état d'un automate, en remplaçant dans les expressions de flots de chaque état les occurences des entrées du systèmes par des variables intermédiaires.

*Sémantique d'évaluation d'un automate* Les modes de calculs d'un automate sont traduits en calculs sur des flots ralentis selon des horloges propres à chaque mode. Pour tout automate $A = (S, Init, Ts, Tw, V)$, le type énuméré $ST_A$ représente par définition le domaine de valeurs $S = \{s_1, \ldots, s_n\}$, l'ensemble des états. Il existe alors trois variables de flots par automates, de type $ST_A$ : le flot $act_A$ représente à chaque cycle l'état actif de l'automate $A$, celui dans lequel les flots de $V$ sont calculés, et à partir duquel on détermine le prochain état sélectionné de l'automate. L'état sélectionné est représenté par le flot $sel_A$ et correspond à un état intermédiaire de l'automate sur lequel est basé le calcul de l'état actif. Les gardes des transitions fortes sont évaluées dans l'état sélectionné, et si l'une des transitions est franchie, alors l'état actif $act_A$ devient l'état cible de la transition. Sinon, aucune transition forte n'est franchissable et l'état actif est égal à l'état sélectionné $sel_A$. La variable booléenne $fired_A$ vaut alors `false`, ce qui autorise le calcul des transitions faibles à ce cycle. Lorsque les sorties de l'automates sont calculées, dans l'état actif $act_A$, les transitions faibles sont potentiellement évaluées et permettent de déterminer le prochain état sélectionné, $nxt_A$. Ainsi, selon qu'une transition faible est franchissable ou non, la valeur de $nxt_A$ vaudra l'état cible de cette transition faible ou $act_A$. Avec $V = \{v_1, \ldots, v_m\}$ l'ensemble des variables calculées, la sémantique de l'automate s'exprime de la manière suivante :

$$\begin{aligned}
sel_A &= Init\texttt{->}\ \texttt{pre(}nxt_A\texttt{)}\ \texttt{;} \\
act_A &= \texttt{merge(}sel_A, [\![trans(s_1, Tw)]\!]_{s_1}^{sel_A}, \ldots, [\![trans(s_n, Tw)]\!]_{s_n}^{sel_A}\texttt{)}\texttt{;} \\
fired_A &= \texttt{merge(}sel_A, [\![fired(s_1, Tw)]\!]_{s_1}^{sel_A}, \ldots, [\![fired(s_n, Tw)]\!]_{s_n}^{sel_A}\texttt{)}\texttt{;} \\
nxt_A &= \texttt{if (}fired_A\texttt{) then } act_A \\
&\qquad \texttt{else merge(}act_A, [\![trans(s_1, Ts)]\!]_{s_1}^{act_A}, \ldots, [\![trans(s_n, Ts)]\!]_{s_n}^{act_A}\texttt{)}\texttt{;} \\
v_1 &= \texttt{merge(}act_A, [\![v_1]\!]_{s_1}^{act_A}, \ldots, [\![v_1]\!]_{s_n}^{act_A}\texttt{)}\texttt{;} \\
&\vdots \\
v_m &= \texttt{merge(}act_A, [\![v_m]\!]_{s_1}^{act_A}, \ldots, [\![v_m]\!]_{s_n}^{act_A}\texttt{)}\texttt{;}
\end{aligned}$$

La figure 3 détaille les variables internes et les transitions de l'automate $\mathcal{A}$ pour une séquence arbitraire de valeurs pour le flot `e`.

## 5 Critères de tests

Nous présentons les mécanismes de GATeL pour décrire et couvrir des critères de tests, et montrons comment les utiliser pour couvrir des automates de modes selon des critères usuels, comme la couverture des états, des transitions et des paires de transitions.

| Cycles | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| e | t | f | t | t | f | t | t |
| $sel_{\mathcal{A}}$ | A | B | B | C | A | C | A |
| $act_{\mathcal{A}}$ | B | B | C | C | A | C | B |
| $nxt_{\mathcal{A}}$ | B | B | C | A | C | A | B |
| $z_A$ | | | | | 0 | | |
| $z_B$ | 0 | 2 | | | | | 4 |
| $z_C$ | | | 0 | 3 | | 6 | |
| z | 0 | 2 | 0 | 3 | 0 | 6 | 4 |

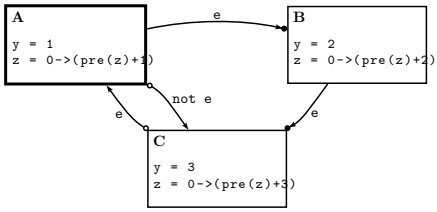



**Figure 3.** Trace d'exécution de l'automate $\mathcal{A}$.

### 5.1 Expression des critères dans GATeL

GATeL est guidé par une spécification Lustre d'un système, de son environnement et d'un objectif de test. Cela signifie qu'à la fin d'une séquence de test générée avec GATeL, l'objectif doit s'évaluer à *true*, et que toutes les valeurs d'entrées et de sorties générées doivent respecter la sémantique donnée par la spécification. Un objectif est spécifié sous la forme « `reach` *Exp* », où *Exp* est une expression logique Lustre. Il est possible de décrire des objectifs impliquant une chaîne d'événements passés, pour spécifier des scénarios de test guidées par une spécification de haut-niveau, comme des cas d'utilisations. Les critères fonctionnels sont ainsi facilement exprimables à travers différents objectifs de tests. Par exemple, si un système peut déclencher deux alarmes différentes, et que l'on souhaite générer une séquence où au moins l'une d'elle est vraie, on peut écrire « `reach (a1 or a2)` ». Il est possible de raffiner le cas de test à l'aide d'une exploration structurelle de la spécification, comme nous allons le voir. Cette exploration est liée à l'approche paresseuse de génération adoptée par GATeL, exposée dans le paragraphe suivant.

*Propagation de contraintes* GATeL interprète la génération de séquences de tests sous la forme d'un problème de satisfaction de contraintes. Les séquences sont produites en partant de la fin, où l'objectif de test doit s'évaluer à *true*, et en remontant dans le passé jusqu'à atteindre le cycle initial d'une séquence de tests. Seuls les flots nécessaires à la réalisation de l'objectif de tests sont parcourus, de manière paresseuse, selon une sémantique en arrière du langage synchrone [14]. Les valeurs des flots aux différents cycles ainsi parcourus sont représentées par des variables logiques, liées entre elles à travers une traduction des opérateurs Lustre à l'aide de contraintes. Le système de contraintes évolue dynamiquement : de nouvelles variables et contraintes sont ajoutées en cours de propagation. Le mécanisme de propagation de contraintes ne descend toutefois pas systématiquement dans chaque branche d'une expression Lustre : lorsque, par exemple, un opérateur `if` est propagé à un cycle donné sans qu'il soit possible de déterminer, ni la valeur de la condition à ce cycle, ni quelle branche du `if` s'unifie exclusivement avec le résultat, alors la propagation est suspendue. Lorsqu'un système de contraintes atteint un point fixe par propagation avec des contraintes en attente, la phase de résolution, c'est-à-dire la recherche exhaustive de solutions guidée par heuristiques, peut tenter dans ce cas une valuation aléatoire de la condition du `if`. Cependant, avant de réaliser une résolution du système, et donc de générer une séquence de test, l'utilisateur a la possibilité de raffiner le cas de test donné par l'objectif en sélectionnant un cas de test aux endroits où la propagation de contraintes a été suspendue.

*Dépliage structurel* Dans l'exemple précédent, l'opérateur `if`, apparaît comme étant *dépliable* dans l'interface utilisateur au cycle où la condition est indéterminée : la sélection de cet opérateur produit deux sous-cas de tests, un dans lequel la condition est vraie et un autre dans lequel la condition est fausse. GATeL se souvient de l'ensemble des cas de tests ainsi créés, sous la forme

d'un arbre de décisions. Un testeur peut mettre en œuvre un critère de couverture structurel de la spécification grâce au dépliage des opérateurs, comme par exemple la couverture des conditions, des décisions, ... Ainsi, pour l'exemple précédent « `reach (a1 or a2)` », le dépliage de l'opérateur `or` permet de distinguer le cas où l'un ou l'autre terme est supposé vrai ; deux séquences peuvent donc être générées, couvrant l'objectif selon les deux issues principales du scénario envisagé.

*Dépliage fonctionnel* Nous montrons ici comment couvrir un objectif de test selon un critère fonctionnel, à l'aide de l'opérateur `split` de GATeL. En effet, on peut souhaiter couvrir un objectif de test selon différents cas qui ne sont pas exprimés dans le modèle LUSTRE. Par exemple, le nœud LUSTRE `ResetArrow` suivant combine la flèche de LUSTRE sur les entiers avec un flot booléen de redémarrage, et l'on souhaite introduire un critère de couverture pour cet opérateur :

```
node ResetArrow(A,B:int; R:bool) returns (I: int);
let
    I = A -> (if (R) then A else B);
    /*! split I with [ R; (not R); (I=A); (I=B) ] !*/
tel;
```

À chaque instant où `R` vaut vrai, ainsi qu'à l'instant initial, le flot `I` est égal au flot `A` ; sinon, il est égal au flot `B`. Le critère de couverture arbitraire mis en œuvre ici permet d'obtenir quatre cas de test à chaque fois qu'une instance de `ResetArrow` est propagée dans le système de contraintes. En effet, une expression de la forme « `split` $V$ `with [`$E_1$`,...,`$E_n$`]` » introduit des des nouveaux cas de tests : à chaque cycle où la variable LUSTRE $V$ est ajoutée dans le système de contraintes, GATeL propose de séparer le cas de test courant en $n$ cas, où chaque cas $i \in \{1, \ldots, n\}$ est tel que le flot booléen $E_i$ est vrai au cycle considéré.

### 5.2 Couverture des états actifs

La couverture des états d'un automate de mode est interprétée comme la couverture des états actifs. Pour cela, nous définissons le critère $all\_modes(A)$ relatif à tout automate $A$. On note $S = \{s_1, \ldots, s_n\}$ l'ensemble des états de $A$. On suppose qu'il existe un objectif de test dépendant de $Obj$, une variable booléenne quelconque. Le critère s'exprime alors de la manière suivante :

$$all\_modes(A, Obj) = \texttt{split } Obj \texttt{ with [ } (act_A \texttt{ = } s_1) \texttt{ ; ... ; } (act_A \texttt{ = } s_n) \texttt{ ]}$$

L'opérateur `split..with` introduit la séparation en sous-cas de tests à chaque introdution de $Obj$ dans le système de contraintes. Avec un objectif de test valant simplement $true$, les séquences de tests générées montrent l'accessibilité des différents modes de calculs, mais on peut aussi par exemple générer des séquences pour un objectif tout en sélectionnant le mode de calcul de l'automate au cours des cycles. Pour l'automate $\mathcal{A}$, le critère suivant cherche à atteindre un des modes de calcul et à y rester jusqu'à ce que `z` vaille 6.

```
Obj = (z = 6) and
      (true -> (if (z > 0) then (act_A = pre(act_A)) else true))) ;
/*! reach Obj !*/
/*! split Obj with [ (act_A = s_1) ; ... ; (act_A = s_n) ] !*/
```

Pour le mode $B$, par exemple, la séquence d'entrée générée est $e = (t, f, f, f)$, faisant passer l'automate de l'état initial $A$ à l'état actif $B$ dès le premier cycle, produisant ainsi $z = (0, 2, 4, 6)$. De même, la séquence d'entrée pour le cas $C$ est $e = (f, f, f, f)$, faisant passer l'automate de

l'état initial à l'état $C$ par la transition faible « `not e` » ; on a alors $z = (0, 0, 3, 6)$. On constate qu'il n'existe pas de séquence où l'automate reste dans le mode $A$ pour calculer `z`, car pour toute valeur de `e`, celle-ci déclenche l'une des transitions sortant de $A$ (et parce qu'il n'existe pas de transition forte de $C$ vers $A$).

### 5.3 Couverture d'une transition

*Transition forte* Soit $t_s \in Ts$ une transition forte définie dans un automate quelconque $A$, et réutilise les notations de la section 4. La couverture de la transition est définie par $tr(t_s)$ :

$$tr(t_s) = (sel_A = t_s.src) \texttt{ and } (act_A = t_s.dst) \texttt{ and } [\![Cond(t_s)]\!]^{sel_A}_{t_s.src}$$

Le prédicat signifie que l'état sélectionné doit être l'état source de la transition, que l'état actif doit être son état cible, et que l'évaluation de sa condition dans l'état source doit être vraie. On remarque que l'expression obtenue est mal formée, dans le sens où l'horloge est plus lente dans le dernier terme de la conjonction ; cependant, pour éviter de surcharger les notations, et surtout parce que l'on sait qu'il existe nécessairement un instant de présence pour ce terme (étant donné $sel_A = t_s.src$), on suppose qu'il existe une projection implicite du dernier terme sur l'horloge de l'automate.

*Transition faible* La couverture d'une transition faible se fait de manière similaire ; nous surchargeons pour cela le prédicat $tr$. Soit $t_w \in Tw$ une transition faible d'un automate $A$. La couverture de cette transition est :

$$tr(t_w) = (\texttt{not } fired_A) \texttt{ and } (act_A = t_w.src) \texttt{ and } (nxt_A = t_w.dst) \texttt{ and } [\![Cond(t_w)]\!]^{act_A}_{t_w.src}$$

On impose ici dans le critère qu'aucune transition forte n'ait été franchie au même cycle, à l'aide du terme « `not` $fired_A$ ».

### 5.4 Couverture de toutes les transitions

Le prédicat *all_transitions* défini ci-après permet d'obtenir autant de cas de tests qu'il y a de transitions fortes et faibles dans un automate quelconque $A$. On note pour cela $T = Tw \cup Ts$, tel que $T = \{t_1, \ldots, t_m\}$, avec $m$ le nombre total de transitions dans $A$. On définit *all_transitions* pour toute variable *Obj* définissant l'objectif de test :

$$all\_transitions(A, Obj) = \texttt{split } Obj \texttt{ with [ } tr(t_1) \texttt{ ; } \ldots \texttt{ ; } tr(t_m) \texttt{ ]}$$

Dans notre exemple, la couverture de toutes les transitions s'exprime donc par :

```
Obj = true ;
/*! reach Obj !*/
/*! split Obj with [
  (sel_A = A) and (act_A = B) and e ;
  (sel_A = B) and (act_A = C) and e ;
  not(fired_A) and (act_A = C) and (nxt_A = A) and e ;
  not(fired_A) and (act_A = A) and (nxt_A = C) and (not e) ] !*/
```

### 5.5 Couverture des paires de transitions

La couverture des paires de transitions est un critère dans lequel on souhaite franchir deux transitions, une d'un état $A$ à un état $B$, puis une autre de l'état $B$ à l'état $C$. Dans un automate de mode, les transitions ne sont pas forcément toutes applicables d'un cycle à un autre, puisque les conditions peuvent être complexes et demander à l'automate de rester dans un mode pendant un certain nombre de cycles avant de pouvoir en sortir. Le critère que nous mettons en œuvre consiste à avoir un objectif dans lequel la première transition doit avoir été vraie, puis qui est passé en un ou plusieurs cycles dans l'état $C$ à l'aide de la deuxième transition. On note $t_{AB}$ et $t_{BC}$ les transitions respectives entre les états $A$, $B$ et $C$ d'un automate quelconque. On rappelle que $T$ est la concaténation des listes $Tw$ et $Ts$. L'objectif LUSTRE de couverture $TP(t_{AB}, t_{BC})$ s'exprime ainsi :

$TP(t_{AB}, t_{BC})$ `= transAB and transBC;`
`transAB =` $tr(t_{AB})$ `or ( false -> pre(transAB) ) ;`
`transBC = transAB and` $tr(t_{BC})$ `and (true -> pre(`$NoTrFrom(B, T)$`)) ;`

Le prédicat $NoTrFrom(B, T)$ représente l'absence de transition depuis l'état $B$, étant donné la liste de transitions $T$. Il se réécrit en une expression LUSTRE de la manière suivante, pour toute liste $L$, pour tout état $s$ et toute transition $t$.

$$NoTrFrom(s, []) = true$$
$$NoTrFrom(s, t :: L) = \begin{cases} \texttt{not(}tr(t)\texttt{) and } NoTrFrom(s, L) & \text{si } t.src = s \\ NoTrFrom(s, L) & \text{sinon} \end{cases}$$

L'équation de `transAB` devient et reste vraie dès que la transition de $A$ à $B$ est franchie. La variable `transAB` conditionne la valeur de vérité de $TP(t_{AB}, t_{BC})$ : la valeur ne peut être vraie que si `transBC` est vraie alors que `transAB` est ou a déjà été vraie. On garantit alors que les exécutions sont observées dans le bon ordre : d'abord $t_{AB}$, puis $t_{BC}$. Le flot porté par `transBC` est vrai lorsque la transition de $B$ vers $C$ est réalisé et ne l'a jamais été depuis que `transAB` est vraie. Plus précisément, on garantit qu'aucune transition depuis $B$ n'a été effectuée depuis que l'on est dans l'état $B$, à l'aide de $NoTrFrom$ (on impose que les autres transitions ne sont pas franchies). La couverture de toutes les paires de transitions est réalisée comme précédemment, à l'aide d'un `split` pour l'ensemble des couples de transitions de la spécification.

*Exemple* L'automate $\mathcal{A}$ précédent admet quatre transitions, que l'on note $t_{AB}$, $t_{BC}$, $t_{AC}$ et $t_{CA}$, ainsi que quatre paires de transitions possibles : $p1 = (t_{AB}, t_{BC})$, $p2 = (t_{BC}, t_{CA})$, $p3 = (t_{CA}, t_{AB})$ et $p4 = (t_{CA}, t_{AC})$. La couverture d'un objectif $Obj$ `= true` avec :

`split` $Obj$ `with [` $TP(p_1)$`;` $TP(p_2)$`;` $TP(p_3)$`;` $TP(p_4)$ `]`

donne les quatre séquences possibles suivantes :

- $e = (t, t)$ : franchit les deux transitions fortes, de $A$ vers $B$ puis de $B$ vers $C$.
- $e = (t, t, t)$ : va dans $B$ depuis l'état initial, franchit la transition forte vers $C$ puis la faible vers $A$.
- $e = (f, t, t)$ : va dans l'état $C$ avec `not(e)`, prend la transition faible de $C$ vers $A$ puis la forte vers $B$.
- $e = (f, t, f)$ : va dans l'état $C$ depuis l'état initial, puis emprunte les deux transitions faibles $t_{CA}$ et $t_{AC}$.

## 6 Conclusion

Nous avons présenté une adaptation de critères de tests usuels pour automates finis au modèle d'automates de modes de Scade 6. Cette adaptation est basée sur une transformation des automates dans le langage Lustre avec horloges, d'une part, et sur la traduction des critères de tests en objectifs de tests interprétables par GATeL, d'autre part. Les expressions des critères d'automates en objectifs de tests Lustre peuvent être couvertes structurellement par notre outil de génération de séquences de tests, couvrant ainsi les critères d'automates modélisés. Le mélange des critères fonctionnels et des critères de couverture structurelle de la spécification d'automate permet une expression fine des tests que l'on souhaite générer. Nous expérimentons l'implémentation de la traduction et son intégration dans GATeL, ainsi que le développement de mécanismes de sélections plus appropriés aux modèles d'automates dans notre outil. Il est possible d'étendre les tests générés pour prendre en compte des modèles de fautes spécifiques aux automates. En particulier, la génération de séquences discriminantes [10] nous semble intéressante pour la vérification des états parcourus en fonction des entrées/sorties des implémentations sous test.

## Références


1. Benveniste, A., Berry, G. : The Synchronous Approach To Reactive And Real-Time Systems. Proceedings Of The IEEE **79**(9) (Sep 1991) 1270–1282
2. Marre, B., Arnould, A. : Test sequences generation from LUSTRE descriptions : GATEL. In : Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering. Volume 0., Los Alamitos, CA, USA, IEEE Computer Society (2000) 229
3. Blanc, B., Marre, B. : Test Selection Strategies for Lustre Descriptions in GATeL. ENTCS **111** (January 2005) 93–111
4. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J. : Lustre : A Declarative Language for Programming Synchronous Systems. In : POPL. (1987) 178–188
5. Maraninchi, F., Remond, Y. : Mode-automata : About modes and states for reactive systems. In : ESOP. Volume 1381. (1998) 185–199
6. Blanc, B., Junke, C., Marre, B., Le Gall, P., Andrieu, O. : Handling State-Machines Specifications with GATeL. In : Proceedings of the Workshop on Model Based Testing 2010. (2010)
7. Colaço, J.L., Pagano, B., Pouzet, M. : A Conservative Extension of Synchronous Data-flow with State Machines. In : EMSOFT. (2005) 173–182
8. Burton, S., Clark, J., McDermid, J. : Automatic generation of tests from Statechart specifications. (2009) 31
9. Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. : Model-based testing of reactive systems : advanced lectures. Springer-Verlag New York Inc (2005)
10. Lee, D., Yannakakis, M. : Principles and methods of testing finite state machines - A survey. Proceedings Of The IEEE **84** (1996) 1090–1123
11. Offutt, J., Liu, S., Abdurazik, A., Ammann, P. : Generating test data from state-based specifications. Software Testing, Verification and Reliability **13**(1) (March 2003) 25
12. Hong, H.S., Kim, Y.G., Cha, S.D., Bae, D.H., Ural, H. : A test sequence selection method for statecharts. Softw. Test. Verif. Reliab **10** (2000) 203–227
13. Laurent, O., Seguin, C., Wiels, V. : A methodology for automated test generation guided by functional coverage constraints at specification level. ASE 2006 : 21st IEEE International Conference on Automated Software Engineering, Proceedings (2006) 285–288
14. Junke, C., Blanc, B. : CSP dynamiques pour la génération de tests de systèmes réactifs. Journées Francophones de Programmation par Contraintes (2009)

# Session du groupe de travail MTV$^2$

Méthodes de test pour la validation et la vérification

# Uniform Monte-Carlo Model Checking

Johan Oudinet[1,2], Alain Denise[1,2,3], Marie-Claude Gaudel[1,2], Richard Lassaigne[4,5], and Sylvain Peyronnet[1,2,3]

[1] Univ Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405;
[2] CNRS, Orsay, F-91405;
[3] INRIA Saclay - Île-de-France, F-91893 Orsay cedex;
[4] Univ. Paris VII, Equipe de Logique Mathématique, UMR7056;
[5] CNRS, Paris-Centre, F-75000

**Abstract.** Grosu and Smolka have proposed a randomised Monte-Carlo algorithm for LTL model-checking. Their method is based on random exploration of the intersection of the model and of the Büchi automaton that represents the property to be checked. The targets of this exploration are so-called *lassos*, i.e. elementary paths followed by elementary circuits. During this exploration outgoing transitions are chosen uniformly at random.
Grosu and Smolka note that, depending on the topology, the uniform choice of outgoing transitions may lead to very low probabilities of some lassos. In such cases, very big numbers of random walks are required to reach an acceptable coverage of lassos, and thus a good probability either of satisfaction of the property or of discovery of a counter-example. In this paper, we propose an alternative sampling strategy for lassos in the line of the uniform exploration of models presented in some previous work.
The problem of finding all elementary cycles in a directed graph is known to be difficult: there is no hope for a polynomial time algorithm. Therefore, we consider a well-known sub-class of directed graphs, namely the reducible flow graphs, which correspond to well-structured programs and most control-command systems.
We propose an efficient algorithm for counting and generating uniformly lassos in reducible flowgraphs. This algorithm has been implemented and experimented on a pathological example. We compare the lasso coverages obtained with our new uniform method and with uniform choice among the outgoing transitions.

## 1 Introduction

Random exploration of large models is one of the ways of fighting the state explosion problem. In [11], Grosu and Smolka have proposed a randomized Monte-Carlo algorithm for LTL model-checking together with an implementation called $MC^2$. Given a finite model $M$ and an LTL formula $\Phi$, their algorithm performs random walks ending by a cycle, (the resulting paths are called *lassos*) in the Büchi automaton $B = B_M \times B_{\neg\,\Phi}$ to decide whether $L(B) = \emptyset$ with a

probability which depends on the number of performed random walks. More precisely, their algorithm samples lassos in the automaton $B$ until an *accepting* lasso is found or a fixed bound on the number of sampled lassos is reached. If $MC^2$ find an accepting lasso, it means that the target property is false (by construction of $B$, an accepting lasso is a counterexample to the property). On the contrary, if the algorithm stops without finding an accepting lasso, then the probability that the formula is true is high. The advantage of a tool such as $MC^2$ is that it is fast, memory-efficient, and scalable.

Random exploration of a model is a classical approach in simulation [3] and testing (see for instance [26,6]) and more recently in model-checking [21,8,20,1]. A usual way to explore a model at random is to use isotropic random walks: given the states of the model and their successors, an isotropic random walk is a succession of states where at each step, the next state is drawn uniformly at random among the successors, or, as in [11] the next transition is drawn uniformly at random among the outgoing transitions. This approach is easy to implement and only requires local knowledge of the model.

However, as noted in [21] and [19], isotropic exploration may lead to bad coverage of the model in case of irregular topology of the underlying transition graph. Moreover, for the same reason, it is generally not possible to get any estimation of the coverage obtained after one or several random walks: it would require some complex global analysis of the topology of the model.

Not surprisingly, it is also the case when trying to cover lassos: Figure 1 from [11], shows a Büchi automaton with $q+1$ lassos: $l_0, l_1, \ldots, l_q$ where $l_i$ is the lasso $s_0 s_1 \ldots s_i s_0$. With an isotropic random walk, $l_q$ has probability $1/2^q$ to be traversed.

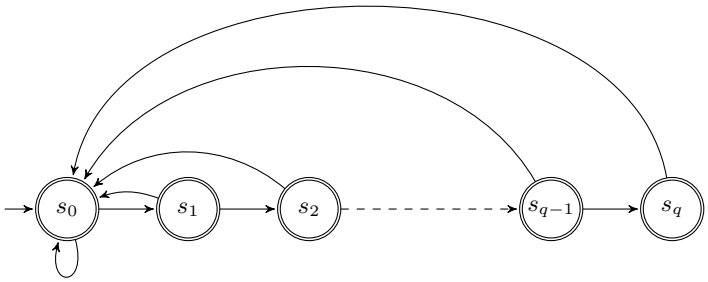



Fig. 1: A pathological example for Büchi automata

In this paper, we propose an alternative sampling strategy for lassos in the line of the uniform exploration of models presented in [7,17] and [10]. In the example above, the low probability of $l_q$ comes from the choice done by the random walk at each state. It has to choose between a state that leads to a single lasso and a state that leads to an exponential number of lassos. In the case of an

isotropic random walk, those two states have the same probability. If the number of lassos that start from each state is known, the choice of the successors can be guided to balance the probability of lassos so as to get a uniform distribution and to avoid lassos with a too small probability. Coming back to Figure 1, with a uniform distribution on lassos, $l_q$ has probability $1/q$ to be traversed instead of $1/2^q$.

However, the problem of counting and finding all elementary cycles in a directed graph is known to be difficult. We briefly recall in Section 2 why there is no hope of polynomial time algorithms for this problem. Therefore, we consider a well-known sub-class of directed graphs, namely the reducible flow graphs [12], which correspond to well-structured programs and most control-command systems. In Section 3, we show that the set of lassos in such graphs is exactly the set of paths that start from the initial state and that end just after the first back edge encountered during a depth-first search. Then on the basis of the methods for counting and generating paths uniformly at random, presented in [7,17] and [10], we give an algorithm for counting the number of lassos in a reducible flowgraph and uniformly generating random lassos in reducible flowgraphs. Section 4 presents how this algorithm can be used for LTL model-checking. Section 5 reports how this algorithm has been implemented and how it has been experimented on an example similar to the pathological example in Figure 1. We compare the lasso coverages obtained with our new uniform method and with uniform choice among the successors or the outgoing transitions.

## 2 Counting and Generating Lassos in Directed Graphs

A lasso in a graph is a finite path followed by an elementary, or simple, cycle. There are two enumeration problems for elementary cycles in a graph. The first one is counting and the second one is finding all such cycles. These two problems are hard to compute: let $FP$ be the class of functions computable by a Turing machine running in polynomial time and $\sharp CYCLE(G)$ be the number of elementary cycles in a graph $G$; one can prove that $\sharp CYCLE(G) \in FP$ implies $P = NP$ by reducing the Hamiltonian circuit problem to decide if the number of elementary cycles in a graph is large.

For the problem of finding all elementary cycles in a directed graph there is no hope for a polynomial time algorithm. For example, the number of elementary cycles in a complete directed graph grow faster than the exponential of the number of vertices. Several algorithms were designed for the finding problem. In the algorithms of Tiernan [23] and Weinblatt [25] time exponential in the size of the graph may elapse between the output of a cycle and the next. However, one can obtain enumeration algorithms with a polynomial delay between the output of two consecutive cycles.

Let $G$ be a graph with $n$ vertices, $e$ edges and $c$ elementary cycles. Tarjan [22] presented a variation of Tiernan's algorithm in which at most $O(n.e)$ time elapses between the output of two cycles in sequence, giving a bound of $O(n.e(c+1))$ for the running time of the algorithm. To our knowledge, the best algorithm for the

finding problem is Johnson's [14], in which time consumed between the output of two consecutive cycles as well as before the first and after the last cycle never exceeds the size of the graph, resulting in bounds $O((n+e).(c+1))$ for time and $O(n+e)$ for space.
Because of the complexity of these problems in general graphs, we consider in this paper a well-known sub-class of directed graphs, namely the reducible flow graphs [12]. Control graphs of well-structured programs are reducible. Most data-flow analysis algorithms assume that the analysed programs satisfy this property, plus the fact that there is a unique final vertex reachable from any other vertex. Similarly, well-structured control-command systems correspond to reducible dataflow graphs. But in their case, any vertex is considered as final: this makes it possible the generalisation of data-flow analysis and slicing techniques to such systems [15]. Informally, the requirement on reducible graphs is that any cycle has a unique entry vertex.

It means that a large class of critical systems correspond to such flowgraphs. However, arbitrary multi-threaded programs don't. But in many cases where there are some constraints on synchronisations, for instance in cycle-driven systems, reducibility is satisfied.

The precise definition of reducibility is given below, as well as a method for counting and uniformly generating lassos in such graphs.

## 3 Uniform random generation of lassos in reducible flowgraphs

A *flowgraph* $G = (V, E)$ is a graph where any vertex of $G$ is reachable from a particular vertex of the graph called the source (we denote this vertex by $s$ in the following). From a flowgraph $G$ one can extract a spanning subgraph (e.g. a directed rooted tree whose vertex set is also $V$) with $s$ as root. This spanning subgraph is known as *directed rooted spanning tree* (DRST). In the specific case where a depth-first search on the flowgraph and its DRST lead to the same order over the set of vertices, we call the DRST a *depth-first search tree* (DFST). The set of back edges is denotes by $B_E$. Given a DFST of $G$, we call *back edge* any edge of $G$ that goes from a vertex to one of its ancestors in the DFST. And we say that a vertex $u$ *dominates* a vertex $v$ if every path from $s$ to $v$ in $G$ crosses $u$.

Here we focus on *reducible flowgraphs*. The intuition for reducible flowgraphs is that any loop of such a flowgraph has a unique entry, that is a unique edge from a vertex exterior to the loop to a vertex in the loop. This notion has been extensively studied (see for instance [12]). The following proposition summarizes equivalent definitions of reducible flowgraphs.

**Proposition 1.** *All the following items are equivalent :*

1. $G = (V, E, s)$ *is a reducible flowgraph.*
2. *Every DFS on* $G$ *starting at* $s$ *determines the same back edge set.*
3. *The directed acyclic graph (DAG)* $dag(G) = (V, E - B_E, s)$ *is unique.*
4. *For every* $(u, v) \in B_E$, $v$ *dominates* $u$.

5. *Every cycle of $G$ has a vertex which dominates the other vertices of the cycle.*

We now recall the precise definition of lassos:

**Definition 1 (lassos).** *Given a flowgraph $G = (V, E, s)$, a path in $G$ is a final sequence of vertices $s_0, s_1, \ldots, s_n$ such that $s_0 = s$ and $\forall 0 \leq i < n \quad (s_i, s_{i+1}) \in E$. An elementary path is a path such that the vertices are pairwise distinct. A lasso is a path such that $s_0, \ldots, s_{n-1}$ is an elementary path and $s_n = s_i$ for some $0 \leq i < n$.*

We can now state our first result on lassos in reducible flowgraphs.

**Proposition 2.** *Any lasso of a reducible flowgraph ends with a back edge, and any back edge is the last edge of a lasso.*

*Proof.* Suppose a traversal starting from the source vertex goes through a lasso and finds a back edge before the end of the lasso. Since the graph is a reducible flowgraph this means that we have a domination, thus we see two times the same vertex in the lasso, which is a contradiction with the fact that a lasso is elementary (remember a lasso is an elementary path).

Suppose now that the traversal never sees a back edge. Then no vertex has been seen twice, thus the traversal is not a lasso.

Using this proposition, we can now design a simple algorithm for

- counting the number of lassos in a reducible flowgraph,
- uniformly generating random lassos in a reducible flowgraph.

Here uniformly means equiprobably. In other word any lasso has the same probability to be generated as the others.

Following the previous proposition, the set of lassos is exactly the set of paths that start from the source and that end just after the first back edge encountered. At first, we change the problem of generating lassos into a problem of generating paths of a given size $n$, by slightly changing the graph: we add a new vertex $s_0$ with a loop, that is an edge from itself to itself, and edge from $s_0$ to $s$. And we give $n$ a value that is an upper bound of the length of the lassos in the new graph. A straightforward such value is the depth of the depth-first search tree plus one.

Now for any vertex $u$, let us denote $f_u(k)$ the number of paths of length $k$ starting from vertex $u$ and finish just after the first back edge encountered. The edge $(s_0, s_0)$ is not considered as a back edge because it does not finish a lasso in the initial graph $G$. The number of lassos is given by $f_{s_0}(n)$. And we have the following recurrence:

- $f_{s_0}(1) = 0$.
- $f_u(1) =$ number of back edges from $u$ if $u \neq s_0$.
- $f_u(k) = \sum_{(u,v)\in E'} f_v(k-1)$ for $k > 1$ where $E'$ is the set of edges of the graph, except the back edges.

The principle of the generation process is simple: starting from $s$, draw a path step by step. At each step, the process consists in choosing a successor of the current vertex and going to it. The problem is to proceed in such a way that only (and all) lassos of length $n$ can be generated, and that they are equiprobably distributed. This is done by choosing successors with suitable probabilities. Suppose that, at one given step of the generation, we are on state $u$, which has $k$ successors denoted $v_1$, $v_2$, ..., $v_k$. In addition, suppose that $m > 0$ transitions remain to be crossed in order to get a lasso of length $n$. Then the condition for uniformity is that the probability of choosing state $v_i$ $(1 \leq i \leq k)$ equals $f_{v_i}(m-1)/f_u(m)$. In other words, the probability to go to any successor of $u$ must be proportional to the number of lassos of suitable length from this successor.

Computing the numbers $f_u(i)$ for any $0 \leq i \leq n$ and any state $u$ of the graph can be done by using the recurrence rules above.

Table 1 presents the recurrence rules which correspond to the automaton of Figure 2.

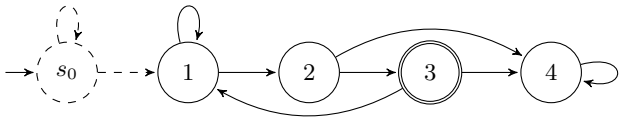



Fig. 2: An example of a Büchi automaton, from [11]. We have added the vertex $s_0$ and its incident edges, according to our procedure for generating lassos of length $\leq n$ for any given $n$.

$$
\begin{array}{lll}
f_{s_0}(1) = f_2(1) = 0 & & \\
f_1(1) \;\; = f_3(1) = f_4(1) = 1 & & \\
f_{s_0}(k) = f_{s_0}(k-1) \;+\; f_1(k-1) & & (k>1) \\
f_1(k) \;\; = f_2(k-1) & & (k>1) \\
f_2(k) \;\; = f_3(k-1) \;+\; f_4(k-1) & & (k>1) \\
f_3(k) \;\; = f_4(k-1) & & (k>1) \\
f_4(k) \;\; = 0 & & (k>1)
\end{array}
$$

**Table 1.** Recurrences for the $f_i(k)$.

Now a primitive generation scheme is as follows:

- Preprocessing stage: Compute a table of the $f_u(i)$'s for all $0 \leq i \leq n$ and any state $u$.
- Generation stage: Draw the lassos according to the scheme seen above.

Note that the preprocessing stage must be done only once, whatever the number of lassos to be generated. Easy computations show that the memory space requirement is $O(n \times |V|)$ integer numbers, where $|V|$ denotes the number of vertices in $G$. The number of arithmetic operations needed for the preprocessing stage is in the worst case in $O(nd|V|)$, where $d$ stands for the maximum number of transitions from a state; and the generation stage is $O(nd)$ [13]. However, the memory required is too large for very long lassos. In [18] we presented an improved version, named Dichopile, which avoids to have the whole table in memory, leading to a space requirement in $O(|V| \log n)$, at the price of a time requirement in $O(nd|V| \log n)$.

## 4 Application to LTL Model-Checking

This section shows how to use this generation algorithm (and its variants, see [18]) for LTL randomised model-checking. We note $M$ and $\Phi$ the considered model and LTL formula. We propose a randomised method to check whether $M \models \Phi$. We call $B_M$ a Büchi automaton corresponding to $M$: the transitions are labeled, the underlying graph is assumed to be reducible, and all the states are accepting (but this condition can be relaxed). We call $B_{\neg\,\Phi}$ the Büchi automaton corresponding to the negation of $\Phi$. The problem is to check that $L(B) = \emptyset$ where $B = B_M \times B_{\neg\,\Phi}$.

It is possible to avoid the construction of the product $B = B_M \times B_{\neg\,\Phi}$. The idea is to exploit the fact that this product is the result of a total synchronisation between $B_M$ and $B_{\neg\,\Phi}$: as explained in [4], the behaviours of $B$ are exactly those behaviours of $B_M$ accepted by $B_{\neg\,\Phi}$. It means that a lasso in $B$ corresponds to a lasso in $B_M$ (but not the reverse). Therefore, it is possible to draw lassos from $B_M$ and, using $B_{\neg\,\Phi}$ as an observer, to reject those lassos that are not in $B$. It is well-known that such rejections preserve uniformity in the considered subset [16].

### 4.1 Drawing lassos in $B$

There is a pre-processing phase that contains the one described in Section 3, namely:

**(pre-DFS)** the collection of the set $B_E$ of back edges via a DFS in $B_M$, and computing $n$, the depth of the DFS + 1;

**(pre-vector)** the construction of the vector of the $|V|$ values $f_u(1)$, i.e. the numbers of lassos of length 1 starting for every vertex $u$;

The two steps above are only needed once for each model. They are independent of the properties to be checked. The third step below is dependent on the property:

**(pre-construction of the negation automaton)** the construction of the Büchi automaton $B_{\neg\,\Phi}$.

Lassos are drawn from $B_M$ using the Dichopile algorithm [18] and then observed with $B_{\neg\,\Phi}$ to check whether they are lassos of $B$. Moreover, it is also

checked whether an acceptance state of $B_{\neg\,\Phi}$ is traversed during the cycle. The observation may yield three possible results:

- the lasso is not a lasso in $B_M \times B_{\neg\,\Phi}$;
- the lasso is an accepting lasso in $B_M \times B_{\neg\,\Phi}$;
- the lasso is a non-accepting lasso in $B_M \times B_{\neg\,\Phi}$.

**Observation of lassos**

The principle of the observation algorithm is: given a lasso of $B_M$, and the $B_{\neg\,\Phi}$ automaton, the algorithm explores $B_{\neg\,\Phi}$ guided by the lasso: since $B_{\neg\,\Phi}$ is generally non deterministic, the algorithm performs a traversal of a tree made of prefixes of the lasso.

When progressing in $B_{\neg\,\Phi}$ along paths of this tree, the algorithm notes whether the state where the cycle of the lasso starts and comes back has been traversed; when it has been done, it notes whether an accepting state of the automaton is met.

The algorithm terminates either when it reaches the end of the lasso or when it fails to reach it: it is blocked after having explored all the strict prefixes of the lasso present in $B_{\neg\,\Phi}$. The first case means that the lasso is also a lasso of $B_M \times B_{\neg\,\Phi}$; then if an accepting state of $B_{\neg\,\Phi}$ has been seen in the cycle of the lasso, it is an accepting lasso. Otherwise it is a non-accepting lasso. The second case means that the lasso is not a lasso in $B_M \times B_{\neg\,\Phi}$.

As soon as an accepting lasso is found, the drawing is stopped.

### 4.2 Complexities

Given a formula $\Phi$ and a model $M$, let $|\Phi|$ the length of formula $\Phi$, $|V|$ the number of states of $B_M$ and $|E|$ its number of transitions, the complexities of the pre-processing treatments are the following:

**(pre-DFS)** this DFS is performed in $B_M$; it is $O(|E|)$ in time and $O(|V|)$ in space in the worst case;

**(pre-vector)** the construction of the vector of the $f_u(1)$ is $\Theta(|V|)$ in time and space;

**(pre-construction of the negation automaton)** the construction of $B_{\neg\,\Phi}$ is $O(2^{|\Phi|.\log|\Phi|})$ in time and space in the worst case [24].

The drawing of one lasso of length $n$ in $B_M$ using the Dichopile algorithm is $O(n.d.|V|.\log n)$ in time and $O(|V|.\log n)$ in space; then its observation is a DFS of maximum depth $n$ in a graph whose size can reach $O(2^{|\Phi|)}$. Let $d_{B_{\neg\,\Phi}}$ the maximal out-degree of $B_{\neg\,\Phi}$, the worst case time complexity is $min(d^n_{B_{\neg\,\Phi}}, 2^{|\Phi|})$, and the space complexity is $O(n)$.

The main motivation for randomised model-checking is gain in space. With this respect, using isotropic random walks as in [11] is quite satisfactory since a local knowledge of the model is sufficient. However, it may lead to bad coverage of the model. For instance, [11] reports the case of the Needham-Schroeder protocol where a counter-exemple has a very low probability to be covered by

isotropic random walk. The solution presented here avoids this problem, since it ensures a uniform drawing of lassos, but it requires more memory: $\Theta(|V|)$ for the pre-processing and $O(|V|.\log n)$ for the generation.

We can also compare the complexity of our approach to the complexity of practical algorithm for the model checking of LTL. The NDFS algorithm [5] has a space complexity of $O(r \log |V|)$ for the main (randomly accessed) memory, where $r$ is the number of reachable states. This complexity is obtained thanks to the use of hash tables. Information accessed in a more structured way (e.g. sequentially) are stored on an external memory and retrieved using prediction and cache to lower the access cost.

In this case, uniform sampling of lassos, with a space complexity of $O(|V| \log n)$ is close to the $O(r \log |V|)$ of the classic NDFS algorithm, without being exhaustive. Nevertheless, a randomized search has no bias in the sequence of nodes it traverses (as is NDFS), and may lead faster to a counterexample in practice.

### 4.3 Probabilities

Let $lassos_B$ the number of lassos in B, and $lassos_{B_M}$ the number of lassos in $B_M$, i. e. $lassos_{B_M} = f_s(n)$, we have $lassos_B \leq f_s(n)$. The probability for a lasso to be rejected by the observation is $lassos_B / f_s(n)$, where the value of $lassos_B$ is dependent on the considered LTL formula. Thus, the average time complexity for drawing $N$ lassos in $B$ is the complexity of the pre-processing, which is $O(|E|) + O(2^{|\Phi|}))$ plus $N \times f_s(n)/lassos_B$ [16] times the complexity of drawing one lasso in $B_M$ and observing it, which is $O(n \log n|V|) + min(d^n_{B_{\neg\ \Phi}}, 2^{|\Phi|})$.

Since the drawing in $B_M$ is uniform, and $f_s(n)$ is the number of lassos the probability to draw any lasso in $B_M$ is $1/f_s(n)$. Since there are less lassos in $B$, and rejection preserves uniformity [16], the probability to draw any lasso in $B$ is $1/lassos_B$ which is greater or equal to $1/f_s(n)$. It is true for any lasso in $B$, the accepting or the non accepting ones. Thus there is no accepting lasso with too low probability as it was the case in the Needham-Schroeder example in [11].

Moreover, the probability $\rho$ that $M \models \Phi$ after $N$ drawings of non-accepting lassos is greater than:

$$1 - \big(1 - 1/f_s(n)\big)^N$$

Increasing $N$ may lead to high probabilities. Conversely, the choice of $N$ may be determined by a target probability $\rho$:

$$N \geq \frac{\log(1-\rho)}{\log(1 - 1/f_s(n))} \tag{1}$$

**Remark:** A natural improvement of the method is to use the fact that during the preliminary DFS some lassos of $B_M$ are discovered, namely one by back edge. These lassos can be checked as above for early discovery of accepting lassos in $B_M \times B_{\neg\phi}$. However, it is difficult to state general results on the gain in time and space, since this gain is highly dependent on the topology of the graph.

## 5 Experimental Results

In this section, we report results about first experiments, which use the algorithm presented in Section 4 to verify if an LTL property holds on some models. We first chose a model in which it is difficult to find a counter-example with isotropic random walks. The idea is to evaluate the cost-effectiveness of our algorithm: does it find a counter-example in a reasonable amount of time and memory? How many lassos have to be checked before we get a high probability that $M \models \Phi$?

### 5.1 Implementation and methodology

This algorithm has been implemented using the RUKIA library[6]. This C++ library proposes several algorithms to generate uniformly at random paths in automata. In particular, the Dichopile algorithm that is mentioned in Section 3. We also use several tools that we mention here: The Boost Graph Library (BGL) for classical graph algorithms like the depth-first search algorithm; The GNU Multiple Precision (GMP) and Boost.Random libraries for generating random numbers; The LTL2BA software [9] to build a Büchi automaton from a LTL formula.

We did all our experiments on a dedicated server whose hardware is composed of an Intel Xeon 2.80GHz processor with 16GB memory. Each experiment was performed 5 times. The two extreme values are discarded and the three remaining values are averaged.

### 5.2 Description of the model and the formula

Figure 3 shows $B_M$, the Büchi automaton of the model. It has $q$ states and every state, except $s_q$, has two transitions labeled by the action a: stay in the current state or move to the next state. The state $s_q$ can only do the action ¬a.

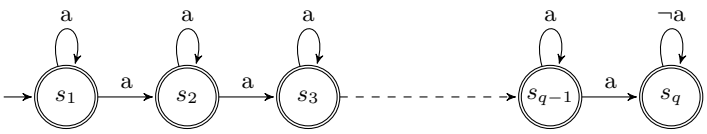



Fig. 3: The Büchi automaton, $B_M$, of the model. Its transitions are labeled by a or ¬a, to indicate if the action a occurs or not

The property that we want to check on this model is: "an action a should occur infinitely often". In LTL, this property can be expressed as

$$\phi = GFa,$$

[6] http://rukia.lri.fr

where the operator G means "for all" and the operator F means "in the future".

It is clear that $M \not\models \Phi$ because if $s_q$ is reached, then no $a$ will occur. Hence, there is a behavior in $M$ where the action $a$ will not occur infinitely often. However, it is difficult for an isotropic random walk to find the lasso that traverses $s_q$. In the next section, we measure the difficulty to find this lasso with our algorithm.

### 5.3 LTL model-checking with uniform generation of lassos

The first step of the algorithm is to build a Büchi automaton that represents the formula $\neg\phi$ (here, $\neg\phi = FG\neg a$). We used the tool LTL2BA and got the automaton in Figure 4.

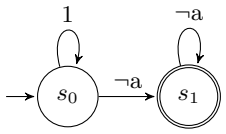



Fig. 4: The Büchi automaton, $B_{\neg\phi}$, of the formula $\neg\phi$. The label 1 means that both a and ¬a are accepted

Then, we can apply our algorithm to generate lassos in $B_M$ until we find an accepting lasso in $B_M \cap B_{\neg\phi}$, by observing the lasso of $B_M$ with the automaton $B_{\neg\phi}$. Table 2 shows the time needed to our algorithm to find the counter-example in two versions of the automaton in Figure 3, one with $q = 100$ states and the other with $q = 1000$ states. We also tried to find this counter-example with isotropic random walks. It did not work, even after the generation of 2 billions lassos. Thus, using uniform generation of lassos provides a better detection power of counter-examples.

**Table 2.** $M \not\models \Phi$: elapsed time, used memory and numbers of lassos generated in $B_M$ by the algorithm of Section 4 to find the counter-example of Section 5.2

| # states | Time (s) | Mem (MiB) | Nb lassos |
|---|---|---|---|
| 100 | 0.38 | 49 | 70 |
| 1000 | 546 | 50 | 680 |

As we know the probability to find the counter-example in $B$ with both isotropic random walks and uniform random generation (i.e., $1/2^q$ for an isotropic random walk and $1/q$ for a uniform random generation), we can compute the number of lassos required to achieve a target probability $\rho$. Table 3 describes those numbers for some target probabilities and for the two previous versions of the automaton in Figure 3.

**Table 3.** $M \models \Phi$: numbers N of lassos to be generated with isotropic random walks (resp. uniform random generation) in order to ensure a probability $\rho$ that $M \models \Phi$. The symbol $\infty$ means a huge number, which cannot be computed with a calculator

| | | N | |
|---|---|---|---|
| # states | $\rho$ | isotropic | uniform |
| 100 | 0.9 | $10^{30}$ | 227 |
| | 0.99 | $\infty$ | 454 |
| | 0.999 | $\infty$ | 681 |
| 1000 | 0.9 | $\infty$ | 2300 |
| | 0.99 | $\infty$ | 4599 |
| 10000 | 0.9 | $\infty$ | 23022 |

**Note:** In general, the minimal probability to find a counter example is unknown. In the case of uniform random generation of lassos, we have a lower bound of this probability. Thus, the maximal number of lassos to be generated for a given probability can always be determined with Formula 1, which it is not possible with isotropic random walks.

## 6 Conclusion

We have presented a randomised approach to LTL model checking that ensures a uniform distribution on the lassos in the product $B = B_M \times B_{\neg \Phi}$. Thus, there is no accepting lasso with too low probability to be traversed, whatever the topology of the underlying graph.

As presented here, the proposed algorithm still needs an exhaustive traversal of the state graph during the pre-processing stage. This could seriously limit its applicability. A first improvement of the method would be to use on the fly techniques to avoid the storage in memory of the whole model during the DFS. A second improvement would be to avoid the complete storage of the vector, parts of it being computed during the generation stage, when needed. However, it will somewhat increase the time complexity of drawing. Another possibility would be approximate lasso counting, thus approximate uniformity, under the condition that the approximation error can be taken into account in the estimation of the satisfaction probability, which is an open issue.

First experiments, on examples known to be pathological, show that the method leads to a much better detection power of counter-examples, and that the drawing time is acceptable. In Section 5, we report a case with long counter-examples difficult to reach by isotropic random walks. A counter-example is discovered after a reasonable number of drawings, where isotropic exploration would require prohibitive numbers of them. The method needs to be validated on some more realistic example. We plan to embed it in an existing model-checker in order to check LTL properties on available case studies.

The method is applicable to models where the underlying graph is a reducible flow graph: we give a method for counting lassos and drawing them at random in this class of graphs, after recalling that it is a hard problem in general. Reducible data flow graphs correspond to well-structured programs and control-command systems (i.e., the steam boiler [2]). A perspective of improvement would be to alleviate the requirement of reducibility. It seems feasible: for instance, some data flow analysis algorithms have been generalised to communicating automata in [15]. Similarly, we plan to study the properties and numbers of lassos in product of reducible automata, in order to consider multi-threaded programs.

## References


1. Nazha Abed, Stavros Tripakis, and Jean-Marc Vincent. Resource-aware verification using randomized exploration of large state spaces. In *Model Checking Software, 15th International SPIN Workshop*, volume 5156 of *Lecture Notes in Computer Science*, pages 214–231, 2008.
2. Jean-Raymond Abrial, Egon Börger, and Hans Langmaack, editors. *Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995)*, volume 1165 of *Lecture Notes in Computer Science*. Springer, 1996.
3. David Aldous. An introduction to covering problems for random walks on graphs. *J. Theoret Probab.*, 4:197–211, 1991.
4. Béatrice Bérard, Michel Bidoit, Alain Finkel, François Laroussinie, Antoine Petit, Laure Petrucci, and Philippe Schnoebelen. *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer, 2001.
5. Costas Courcoubetis, Moshe Vardi, Pierre Wolper, and Mihalis Yannakakis. Memory-efficient algorithms for the verification of temporal properties. *Formal methods in system design*, 1(2):275–288, 1992.
6. Alain Denise, Marie-Claude Gaudel, and Sandrine-Dominique Gouraud. A generic method for statistical testing. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 25–34. IEEE Computer Society, 2004.
7. Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, and Sylvain Peyronnet. Uniform random sampling of traces in very large models. In *1st International ACM Workshop on Random Testing*, pages 10 –19, July 2006.
8. Mattew B. Dwyer, Sebastian G. Elbaum, Suzette Person, and Rahul Purandare. Parallel randomized state-space search. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 3–12, 2007.
9. Paul Gastin and Denis Oddoux. Fast LTL to Büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Proceedings of the 13th International Conference on Computer Aided Verification (CAV'01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 53–65, Paris, France, July 2001. Springer.
10. Marie-Claude Gaudel, Alain Denise, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased random exploration of large models. In *4th ETAPS Workshop on Model Based Testing*, volume 220, Issue 1, 10 of *Electronic Notes in Theoretical Computer Science*, pages 3–14, 2008. invited lecture.

11. Radu Grosu and Scott A. Smolka. Monte Carlo model checking. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.
12. Matthew S. Hecht and Jeffrey D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.
13. Tim Hickey and Jacques Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4):645–655, 1983.
14. Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1):77–84, 1975.
15. Sébastien Labbé and Jean-Pierre Gallois. Slicing communicating automata specifications: polynomial algorithms for model reduction. *Formal Asp. Comput.*, 20(6):563–595, 2008.
16. Luc Devroye. *Non-Uniform Random Variate Generation*. Springer-Verlag, 1986.
17. Johan Oudinet. Uniform random walks in very large models. In *RT '07: Proceedings of the 2nd international workshop on Random testing*, pages 26–29, Atlanta, GA, USA, November 2007. ACM Press.
18. Johan Oudinet, Alain Denise, and Marie-Claude Gaudel. A new dichotomic algorithm for the uniform random generation of words in regular languages. In *Conference on random and exhaustive generation of combinatorial objects (GASCom)*, Montreal, Canada, September 2010. 10 pages.
19. Radek Pelánek, Tomáš Hanžl, Ivana Černá, and Luboš Brim. Enhancing random walk state space exploration. In *Proc. of Formal Methods for Industrial Critical Systems (FMICS'05)*, pages 98–105, Lisbon, Portugal, 2005. ACM Press.
20. Neha Rungta and Eric G. Mercer. Generating counter-examples through randomized guided search. In *SPIN Workshop*, volume 4595 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2007.
21. Hemanthkumar Sivaraj and Ganesh Gopalakrishnan. Random walk based heuristic algorithms for distributed memory model checking. *Electr. Notes Theor. Comput. Sci. - Proc. of Parallel and Distributed Model Checking (PDMC'03)*, 89(1), 2003.
22. Robert E. Tarjan. Enumeration of the elementary circuits of a directed graph. *SIAM J. Comput.*, 2(3):211–216, 1973.
23. James C. Tiernan. An efficient search algorithm to find the elementary circuits of a graph. *Commun. ACM*, 13(12):722–726, 1970.
24. Moshe Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Logics for Concurrency: Structure versus Automata, volume 1043 of Lecture Notes in Computer Science*, pages 238–266. Springer-Verlag, 1996.
25. Herbert Weinblatt. A new search algorithm for finding the simple cycles of a finite directed graph. *J. ACM*, 19(1):43–56, 1972.
26. Colin H. West. Protocol validation in complex systems. In *SIGCOMM '89: Symposium proceedings on Communications architectures & protocols*, pages 303–312, New York, NY, USA, 1989. ACM.

# Refinement-based CFG Reconstruction from Executables ⋆,⋆⋆

Sébastien Bardin, Philippe Herrmann, and Franck Védrine

CEA, LIST,
Gif-sur-Yvette CEDEX, 91191 France
first.name@cea.fr

**Abstract.** We address the issue of recovering a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. The problem is tackled in an original way, with a refinement-based static analysis working over finite sets of constant values. Requirement propagation allows the analysis to automatically adjust the domain precision only where it is needed, resulting in precise CFG recovery at moderate cost. First experiments, including an industrial case study, show that the method outperforms standard analyses in terms of precision, efficiency or robustness.

**Motivation.** Automatic analysis of programs from their executable files has many potential applications in safety and security, for example: automatic analysis of mobile code and malware, security testing or worst case execution time estimation. We address the problem of (safe) CFG reconstruction, i.e. constructing a both safe and precise approximation of the Control Flow Graph (CFG) of a program given as an executable file. CFG reconstruction is a cornerstone of safe binary-level analysis: if the recovery is unsafe, subsequent analyses will be unsafe too; if it is too rough, they will be blurred by too many unfeasible branches and instructions.

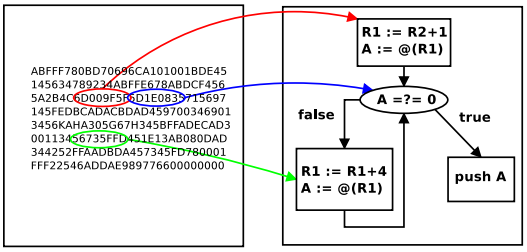



**Fig. 1.** CFG reconstruction from an executable file

**Challenges.** Such an approximation is difficult to obtain mainly because of dynamic jumps, i.e. jump instructions whose target expression is resolved at run-time and may

⋆ Work partially funded by ANR (grants ANR-05-RNTL-02606 and ANR-08-SEGI-006).
⋆⋆ The material presented here is taken from a preliminary version of the VMCAI'11 paper [3].

vary from one execution to the other. Dynamic jumps are very sensitive instructions and a small loss in precision on target expressions may affect dramatically the quality of the subsequent analysis, leading to vicious circles between value analysis and CFG reconstruction. Moreover, there is no reason why all valid targets of a dynamic jump should follow a nice regular pattern. Indeed they are just addresses in the executable code, often arbitrarily assigned by a compiler. Hence any analysis based on popular domains (i.e. convex domains possibly enhanced with congruence information) will introduce many false targets. For example, consider an instruction `cgoto(x)` with `x` $\in \{1355, 1356, 2126\}$: such an analysis cannot recover better than `x` $\in [1355..2126]$, reporting 99% of false targets.

Note that, unfortunately, dynamic jumps are ubiquitous in native code programs: they are introduced at compile-time either for efficiency (`switch` in C) or by necessity (return statements, function pointers in C, virtual methods in C++, etc.).

**Related approaches.** Industrial tools like IDA PRO [10] or AIT [9] usually rely on linear sweep decoding (brute force decoding of all code addresses) or recursive traversal (recursive decoding until a dynamic jump is encountered), enhanced with limited constant propagation, pattern matching techniques based on the knowledge of the compiling chain process and user annotations. These techniques are unsafe on general programs, missing many legal targets and branches. The only safe techniques are those by Reps *et al.* [4, 5] - based mainly on stride intervals propagation, and by Kinder and Veith [7, 8] - based on k-set (sets of bounded cardinality) propagation. Experiments reported by the authors show that while each approach performs much better than current industrial tools, both techniques still recover many false targets. Especially, stride intervals cannot capture precisely sets of jump targets, and k-sets are too sensitive to their cardinality bound, potentially leading to either imprecise or expensive analyses.

**Our approach.** We propose an original refinement-based procedure to solve CFG reconstruction [3]. The procedure is built on two main steps: a forward k-set propagation with local cardinality bounds (ranging from $0$ up to a given parameter $Kmax$), and a refinement step controlling these cardinality bounds.

The forward propagation is mostly a standard one, enhanced with a few original mechanisms: (1) abstract values are downcast according to local cardinality bounds, permitting to lose information and increase efficiency; (2) $\top$ values (i.e. abstract values denoting the whole domain) are tagged with additional information recording their origin (for example $\top_{\langle 1,3,12\rangle}$ denotes the abstraction to $\top$ of the k-set $\{1, 3, 12\}$), allowing to pinpoint the *initial sources of precision loss* (ispl) and give clue for correction (cf. refinement); (3) alias, jump targets and branches that have been fired during propagation are recorded into a *journal* (cf. refinement).

Refinement is lazy and on-demand. When a jump expression evaluates to $\top$, the refinement mechanism takes place, trying to find out ispls responsible for the violation (guided by backward data dependencies and journal information) and to correct them by locally improving the domain precisions (using $\top$-flags).

**Results.** From a theoretical point of view, the procedure is sound and runs in polynomial-time. Moreover it is as precise as standard k-set propagation on a class of non-trivial programs, including dynamic jumps and alias [3]. From a practical point of view, the

procedure has been implemented and evaluated on an industrial safety-critical program (32 kloc) and on small handcrafted programs. It appears to be reasonably efficient (taking less than 5 minutes for the industrial case study), very precise (only 7% of false targets, beating standard approaches based on convex domains by several orders of magnitude), and very robust: the procedure does need an initial parameter, but its exact value does not seem to matter.

## References


1. Balakrishnan, G., Gruian, R., Reps, T. W., Teitelbaum, T.: CodeSurfer/x86-A Platform for Analyzing x86 Executables. In: CC 2005. Springer, Heidelberg (2005)
2. Bardin, S., Herrmann, P.: Structural Testing of Executables. In: IEEE ICST 2008. IEEE Computer Society, Los Alamitos (2008)
3. Bardin, S., Herrmann, P., Védrine, F.: Refinement-based CFG reconstruction from Unstructured Programs. In: VMCAI 2011. Springer, Heidelberg (2011)
4. Balakrishnan, G., Reps, T. W.: Analyzing memory accesses in x86 executables. In: CC 2004. Springer, Heidelberg (2004)
5. Balakrishnan, G., Reps, T. W.: Analyzing Stripped Device-Driver Executables. In: TACAS 2008. Springer, Heidelberg (2008)
6. Godefroid, P., Levin, M. Y., Molnar, D.: Automated Whitebox Fuzz Testing. In: NDSS 2008. The Internet Society (2008)
7. Kinder, J., Veith, H.: Jakstab: A Static Analysis Platform for Binaries. In: CAV 2008. Springer, Heidelberg (2008)
8. Kinder, J., Zuleger, F., Veith, H.: An Abstract Interpretation-Based Framework for Control Flow Reconstruction from Binaries. In: VMCAI 2009. Springer, Heidelberg (2009)
9. Absint homepage `http://www.absint.com/`
10. IDA Pro homepage `http://www.hex-rays.com/idapro`

# Génération de tests à partir de mutations de protocoles de sécurité en HLPSL

Frédéric Dadeau[1], Pierre-Cyrille Héam[1] and Rafik Kheddam[2]

[1] LIFC / INRIA CASSIS Project – 16 route de Gray - 25030 Besançon cedex, France
{frederic.dadeau,pierre-cyrille.heam}@lifc.univ-fcomte.fr
[2] LCIS – 50, rue Barthélémy de Laffemas BP54 – 26902 Valence cedex 09, France
rafik.kheddam@lcis.grenoble-inp.fr

**Résumé** Nous présentons dans cet article l'application de techniques de génération de tests pour des protocoles de sécurité utilisant la mutation de modèles. Les protocoles sont écrits en HLPSL (High-Level Protocol Specification Language) et sont modifiés dans l'objectif d'introduire des fautes ou des choix d'implantation du protocole. Les mutants sont ensuite analysés par un outil de vérification de protocoles qui conclura soit à la sûreté du protocole (i.e. le protocole est insensible à la mutation considérée) soit à la présence d'une faille en renvoyant un trace d'attaque qui pourra être utilisée comme cas de test pour une implantation du protocole.

## 1 Motivations et présentation de l'approche

Nous nous intéressons dans cet article [3] à la validation de protocoles de sécurité. Ceux-ci représentent des échanges de messages (qui peuvent être chiffrés, hashés, signés, etc.) entre agents. Ces protocoles visent à établir des moyens de communication sécurisés entre les acteurs, soit pour les authentifier, soit pour leur permettre d'échanger des informations sensibles. Le projet européen AVISPA [4] [2] a permis de mettre au point un langage de modélisation de protocoles de sécurité, nommé HLPSL (High-Level Protocol Specification Language) [4], accompagné de plusieurs outils dédiés à la vérification des protocoles, ainsi que d'un ensemble de modèles de protocoles de sécurité issus d'exemples classiques ou industriels. Si l'étape de vérification permet de s'assurer que le modèle du protocole préserve bien certaines propriétés de sécurité (autentification, secret, etc.), cette vérification reste au niveau du modèle et ne fournit aucune garantie qu'une implantation du protocole sera réalisée conformément au modèle.

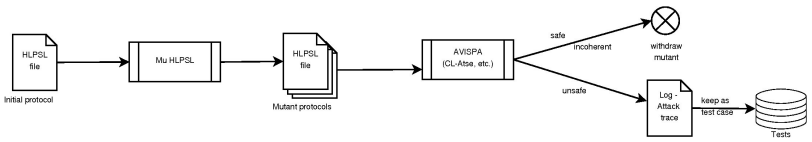



**Figure 1.** Approche de génération de test à partir de mutations de protocoles

Nous proposons une approche de validation par génération de tests de conformité au modèle à partir de mutations [1] de celui-ci. Les mutations proposées traduisent non seulement des erreurs, mais également des choix d'implantation qui peuvent être effectués par un développeur du protocole. La démarche est résumée par le schéma proposé en Figure 1. Un fichier HLPSL, décrivant un protocole initialement vérifié comme sûr, est donné en entrée d'un générateur de mutants qui applique les opérateurs de mutation proposés. Les mutants sont ensuite analysés par l'outil CL-AtSe [3]. Trois résultats sont alors possibles : (*i*) le protocole muté reste sûr, ce qui signifie que le protocole initial n'est pas sensible au type d'erreur introduit, (*ii*) le protocole devient incohérent, ce qui signifie que la mutation a introduit un comportement erratique d'un des agents qui a pour

3. Une version étendue de cet article a été publiée à la conférence ICST'2011 [5]
4. http://avispa-project.org

conséquence de bloquer l'exécution du protocole, (*iii*) le protocole est devenu non-sûr et une trace d'attaque est retournée, qui constituera un cas de test pour une implantation.

## 2 Opérateurs de mutation

Les opérateurs de mutations proposés sont au nombre de 7. Ils traduisent soit un choix d'implantation soit une erreur potentiellement réalisable par un développeur, qui peut avoir des conséquences sur la sécurité d'un protocole :

- Ou-exclusif : force les schémas de chiffrement à utiliser l'opérateur ou-exclusif.
- Exponentiel : force les schémas de chiffrement à utiliser l'opérateur exponentiel.
- Homomorphisme : remplace le chiffrement d'un message concaténé par la concaténation de messages chiffrés.
- Clé publiques : force plusieurs participants à partager la même clé publique.
- Substitution : remplace une partie d'un message par une constante arbitraire.
- Hash functions : supprime l'utilisation de fonctions de hash à l'intérieur d'un protocole.
- Permutation : permute les blocs à l'intérieur des messages.

Ces mutations sont traduites au niveau de la spécification HLPSL, qui est ensuite analysée par le back-end CL-AtSe de AVISPA.

## 3 Expérimentations

Pour nos expérimentations, nous avons développé un outil appliquant un ensemble de mutations, sélectionnées via une interface graphique, à un protocole HLPSL [5]. Nous avons appliqué nos opérateurs de mutation sur un ensemble de 50 protocoles sûrs, issus du projet AVISPA, et disponibles sur le site internet du projet. L'objectif était d'évaluer la pertinence des opérateurs de mutation proposés.

Les résultats de cette expérience sont donnés dans le tableau disponible en Figure 2. Nous constatons que certains opérateurs de mutation sont propices à introduire des failles dans les protocoles considérés (ou-exclusif, homomorphisme, hash functions). D'autres mutations sont moins efficaces, mais arrivent tout de même à trouver leur application dans des cas précis (clé publiques, substitution, permutation). Même si le nombre de mutants finalement produits reste modeste (6%), il est intéressant de constater que beaucoup de mutations proposées sont sans effets sur la sûreté du protocole. Ceci constitue un autre résultat intéressant concernant la sensibilité des protocoles de sécurité à de potentielles erreurs ou des choix d'implantation.

| **Mutation / Resultat** | **Sûr** | **Non-sûr** | **Incohérent** | **Total** |
|---|---|---|---|---|
| Exponentiel | 30 | 0 | 3 | 33 |
| Ou-exclusif | 17 | 13 | 3 | 33 |
| Homomorphisme | 18 | 15 | 0 | 33 |
| Clé publique | 45 | 2 | 0 | 47 |
| Substitution | 286 | 5 | 134 | 425 |
| Hash Functions | 47 | 24 | 8 | 79 |
| Permutation | 420 | 1 | 4 | 425 |
| **Total** | **863** | **60** | **152** | **1075** |

**Figure 2.** Results of Mutant Filtering

## Références


1. R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection : Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
2. A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma, P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron, "The avispa tool for the automated validation of internet security protocols and applications," in *CAV*, ser. Lecture Notes in Computer Science, K. Etessami and S. K. Rajamani, Eds., vol. 3576. Springer, 2005, pp. 281–285.
3. M. Turuani, "The CL-Atse Protocol Analyser," in *Term Rewriting and Applications - Proc. of RTA*, ser. Lecture Notes in Computer Science, vol. 4098, Seattle, WA, USA, 2006, pp. 277–286.
4. *The High Level Protocol Specification Language*, AVISPA project, Delivrable 2.1, 2003, `http://www.avispa-project.org/delivs/2.1/d2-1.pdf`.
5. F. Dadeau, P.-C. Héam and R. Kheddam. *Mutation-Based Test Generation from Security Protocols in HLPSL.*, In proc. of the 4th International Conference on Software Testing, Verification and Validation (ICST'2011). Berlin, Germany, IEEE Computer Society Press, 2011.


5. Cet outil est disponible à l'adresse : `http://lifc.univ-fcomte.fr/~fdadeau/tools/jMuHLPSL.jar`

# Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

# Software Maintenance Analysis and Understanding of the Software Structure

Jannik Laval, Usman Bhatti, Nicolas Anquetil, and Stéphane Ducasse

RMoD Project-Team
INRIA - Lille Nord Europe
USTL - CNRS UMR 8022, Lille, France
http://rmod.lille.inria.fr/

## 1 Context

### 1.1 Maintenance Problematic

Most of the effort during a software system lifecycle is spent in supporting its evolution[12]. Maintainers have to deal with large source code systems and have to spend a large part of their time understanding the system. Corbi[3] estimates the portion of time invested in program comprehension to be between 50 and 60%.

Software evolves over time with the modification, addition and removal of new classes, methods, functions, and dependencies. A consequence is that some classes may not be placed in the right packages and the software modularization is broken[5, 6]. As a consequence, software modularization must be maintained. In that respect, it is then important to understand, to assess and to optimize the concrete organization of packages and their relationships.

### 1.2 Package Granularity

A package is a unit of reuse and deployment: it is built, tested, and released as a whole as soon as one of its classes is changed, or used elsewhere[11]. We name software modularization the organization of classes into multiple packages.

A package provides and requires services from other packages. They can play central roles or peripheral[4]: some packages act as reference hubs, others as authorities. Packages have different usage patterns, often depending on the clients that use them[1]. These multiple views of packages do not ease the understanding and the maintenance.

The remodularization task is to make a better package organization after a structure deterioration. Reengineers have to: (i) understand the structure at package level and at class level and assess its quality; (ii) understand where are structural problems; and (iii) take decisions and verify the impact of these decisions.

Existing approaches lack of (i) a deep understanding fine-grained package structure and dependencies; (ii) an identification of package dependency problems; and (iii) an analyze of the impact of a change on the package structure.

## 2 Understanding Package Structure

### 2.1 Identifying cycle causes

We propose eDSM[9], an approach to enrich Dependency Structural Matrix with eCell, a view displaying the internals of a package dependency. We adapt eDSM for software reengineering with contextual information about (i) the type of dependencies (inheritance, class reference, ...); (ii) the proportion of referencing entities; and (iii) the proportion of referenced entities. We highlight strongly connected component (SCC) and stress potentially simple fixes for cycles using coloring information.

CycleTable[8] presents (i) a heuristic to focus on *shared* dependencies between cycles in Strongly Connected Components; and (ii) a visualization highlighting dependencies to efficiently remove cycles in the system. This visualization is completed with eCell (small views displaying the internals of a dependency).

### 2.2 Package Layered Structure Identification in presence of Cycles

We propose oZone[7] an approach which provides (i) a strategy to highlight dependencies which break Acyclic Dependency Principle; and (ii) an organization of package in multiple layers even in presence of cycles. While our approach can be run automatically, it also supports human inputs and constraints.

### 2.3 Supporting Simultaneous Versions for Software Evolution Assessment

We propose Orion[10], an interactive prototyping tool for reengineering to simulate changes and compare their impact on multiple versions of software source code models. Our approach offers an interactive simulation of changes, reuses existing assessment tools, and has the ability to hold multiple and branching versions simultaneously in memory. Specifically, we devise an infrastructure which optimizes memory usage of multiple versions for large models. This infrastructure uses an extension of the FAMIX source code meta-model but it is not limited to source code analysis tools since it can be applied to models in general.

### 2.4 Metrics analysis

In[2], we study a real structuring case (on the Eclipse platform) to try to better understand if (some) existing metrics would have helped the software engineers in the task. Results show that the cohesion and coupling metrics used in the experiment did not behave as expected and would probably not have helped the maintainers reach there goal.

Currently, we are working on a dependency analysis of packages to measure their relation with their framework. The results show that framework dependencies form a considerable portion of the overall package dependencies. This means that low cohesive packages should not considered systematically as package of low quality.

## References


1. H. Abdeen, I. Alloui, S. Ducasse, D. Pollet, and M. Suen. Package reference fingerprint: a rich and compact visualization to understand package relationships. In *European Conference on Software Maintenance and Reengineering (CSMR)*, pages 213–222. IEEE Computer Society Press, 2008.
2. N. Anquetil and J. Laval. Legacy software restructuring: Analyzing a concrete case. In *CSMR 2011: Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, Germany, 2011.
3. T. A. Corbi. Program understanding: Challenge for the 1990's. *IBM Systems Journal*, 28(2):294–306, 1989.
4. S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.
5. S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.
6. W. G. Griswold and D. Notkin. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol.*, 2(3):228–269, 1993.
7. J. Laval, N. Anquetil, and S. Ducasse. Ozone: Package layered structure identification in presence of cycles. In *Proceedings of the 9th edition of the Workshop BElgian-NEtherlands software eVOLution seminar, BENEVOL 2010*, Lille, France, 2010.
8. J. Laval, S. Denier, and S. Ducasse. Cycles assessment with cycletable. Technical report, INRIA, 2010.
9. J. Laval, S. Denier, S. Ducasse, and A. Bergel. Identifying cycle causes with enriched dependency structural matrix. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, Lille, France, 2009.
10. J. Laval, S. Denier, S. Ducasse, and J.-R. Falleri. Supporting simultaneous versions for software evolution assessment. *Journal of Science of Computer Programming (SCP)*, May 2010.
11. R. C. Martin. Design principles and design patterns, 2000. www.objectmentor.com.
12. I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.

# D'une application orientée objet vers une application à base de composants via une architecture à base de composant

Simon Allier[1,2] , Salah Sadou[1] , Houari A. Sahraoui[2] , Régis Fleurquin[1]

[1] VALORIA, Université de Bretagne Sud, Vannes, France
[2] DIRO, Université de Montréal, Canada
{Simon.Allier, Salah.Sadou, Regis.Fleurquin}@univ-ubs.fr, sahraouh@iro.umontreal.ca

## 1 Introduction

Un système, décrit avec un grand nombre d'éléments fortement interdépendants, est complexe, difficile à comprendre et à maintenir. Ainsi, une application orientée objet (OO) est souvent complexe, car elle contient des centaines de classes avec de nombreuses dépendances plus ou moins explicites. Par ailleurs, une même application, utilisant le paradigme composant, contiendrait un plus petit nombre d'éléments, faiblement couplés entre eux et avec des interdépendances clairement définies. Il est clairement établi que le paradigme composant fournit une bonne représentation de haut niveau des systèmes complexes. Ainsi, il peut représenter un "espace de projection" des systèmes OO complexes. Une telle projection peut faciliter l'étape de compréhension d'un système, un pré-requis nécessaire avant toute activité de maintenance.

Dans ce cas de figure, l'architecture à base de composants n'est qu'une représentation "contemplative" utilisable uniquement par le concepteur. Or il est possible d'utiliser cette représentation, comme un modèle, pour effectuer une restructuration complète d'une application OO opérationnelle vers une application équivalente à base de composants tout aussi opérationnelle. La nouvelle application bénéficiant, ainsi, de toutes les bonnes propriétés associées au paradigme composants.

Pour atteindre cet objectif, nous devons résoudre trois problèmes: i) identifier les ensembles de classes qui devraient être regroupées pour former des composants "abstraits" ; ii) identifier les interfaces fournies et requises de ces composants. Ainsi, nous obtenons une représentation architecturale orientée composants de l'application OO ; iii) et enfin, projeter la représentation architecturale vers un modèle concret de composants.

Il existe plusieurs travaux qui traitent, partiellement, le premier problème [10,8,7] (une synthèse est donnée dans [5]). Le deuxième problème n'est traite que par les contributions [4,9]. Ce que nous proposons est une solution globale et homogène aux trois problèmes cités ci-dessus. Une description plus détaillée de ce travail est donnée dans [2].

## 2 Identification des composants

Deux étapes sont nécessaires pour produire une architecture à base de composants à partir d'une application OO : i) identifier les composants "abstraits" ; ii) identifier les interfaces fournies et requises de ces composants.

*Identification des composants :* Dans notre cas, un composant est défini comme un groupe de classes collaborant entre elles pour fournir une fonctionnalité de haut niveau de l'application. Ainsi, pour construire une vision orientée composant de l'application, nous devons définir une partition de ces classes. Chaque ensemble de cette partition étant un composant. L'identification des composants est basée sur l'extension de la méthode que nous avons présentée dans [3], elle est divisée en trois sous étapes :

– Dans la première étape, nous utilisons des traces obtenues en exécutant les scénarios correspondant aux cas d'utilisation de l'application afin d'identifier les "core" composants (constitués de classes présentes dans les traces). Pour cela, nous utilisons deux méta-heuristique : un algorithme génétique [6], suivi de l'algorithme *simulated annealing* [**?**]. En sortie nous obtenons

une partition qui minimise le couplage entre ensembles de classe (composants) et maximise la cohésion dans ces ensembles.
- Dans la deuxième étape, nous utilisons un graphe d'appels statiques afin d'ajouter dans les "core" composants les classes manquantes. En effet, toutes les classes de l'application ne sont pas nécessairement couvertes par les traces d'exécution. Cette étape utilise les même méta-heuristique que l'étape précédente.
- Enfin, la dernière étape consiste à affiner manuellement la partition obtenue. Dans cette étape, l'utilisateur bénéficie de certaines informations fournies par l'outil sur la solution obtenue afin de la modifier s'il le juge nécessaire.

*Identification des interfaces:* Elle est effectuée en deux sous-étapes: identification des services, puis organisation de ces services dans des interfaces.

Dans notre cas, les services fournis et requis d'un composant correspondent, respectivement, aux appels de méthodes entrants et sortants de ce composant. L'identification des services se fait à partir de graphe d'appels d'un système construit à partir de donnée statistique et dynamique afin d'être le plus complet possible. Pour obtenir les interfaces requises et fournies d'un composant, nous avons besoin de grouper ses services fournis, respectivement requis, en sous-ensembles cohérents, selon le domaine d'application. L'application étant construite en utilisant le paradigme OO, nous utilisons ce paradigme pour identifier ces sous-ensembles. Ainsi, les méthodes (services) définies dans la même classe appartiendront à la même interface.

## 3 D'une architecture à base de composants vers une application orienté composants

Pour restructurer l'application OO opérationnelle vers une application à base de composants opérationnelle, nous utilisons les concepts orientés objets pour implémenter les interfaces fournies et requises des composants, puis nous projetons les composants identifiés vers un modèle de composants concrets. Pour illustrer notre approche, nous avons choisi d'utiliser le modèle de composants OSGi.

*Construction des interfaces :* Les interfaces identifiées dans l'étape précédente n'ont pas d'existence en tant que telle dans l'application OO. En conséquence, pour rendre ces interfaces opérationnelles, nous devons les construire. De plus, la construction des interfaces est soumise à deux contraintes: le code source de l'application ne doit pas être modifié et seuls les services (méthode) des interfaces doivent être accessibles depuis l'extérieur d'un composant. Pour cela, nous utilisons les patrons de conception *Adaptator* et *Facade*.

*Projection des composants vers un modèle concret :* Nous avons utilisé le modèle de composant OSGil [1] qui nous permet de déployer facilement l'architecture orientée composants identifiée. Dans le *framework* OSGi, un composant (appelé *bundle*) correspond à un ensemble de classes. A l'aide d'un manifeste il est possible de définir les *packages* visibles depuis l'extérieur du *bundle* (ses interfaces fournies du composants). De même, il est possible de définir les *packages* nécessaires au *bundle* (ses interfaces requises). Ainsi, nos composants sont organisés sous la forme de *bundles*. leurs interfaces fournies et requises sont placées dans des *packages* explicites pour pouvoir les déclarer en tant que telles dans le manifeste.

## 4 Conclusion

La solution proposée est complète. En effet, elle restructure une application OO opérationnelle en son équivalente orientée composants opérationnelle. La solution pour l'identification de l'architecture à base de composants est générique. Elle n'est basée que sur des notions générales du paradigmes objets et composants et peut, ainsi, s'appliquer à tout langage OO et tout modèle de composants.

La projection vers un modèle concret dépend, naturellement, d'un langage OO particulier et d'un modèle de composants (dans notre cas Java et OSGi). Mais nous pensons que l'architecture orientée composants, obtenue après la deuxième étape, facilite grandement la projection vers d'autres modèles de composants concrets.

Une des limites de notre approche est que les composants identifiés sont dépendants de l'applications. En effet, les traces d'exécution utilisés sont dictés par la logique de l'application. Ces composants peuvent être utilisés pour restructurer l'application à des fin de maintenabilité, mais nous ne pouvons prétendre qu'ils sont réutilisables dans d'autres contextes. Une des améliorations possibles de notre approche concerne la définition des interfaces des composants. En effet, celle-ci est basée sur la définition individuelle des classes du composant. Dans nos travaux futurs, nous voulons capturer plus de sémantique sur les composants identifiés (définition d'un ensemble de classes) afin d'obtenir une meilleure répartition des services dans les interfaces.

## References


1. OSGI Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.2. Technical report, OSGI Alliance, September 2009.
2. Simon Allier, Salah Sadou, Houari A. Sahraoui, and Régis Fleurquin. From object-oriented applications to component-oriented applications via component-oriented architecture. In *WISCA*, 2011.
3. Simon Allier, Houari A. Sahraoui, Salah Sadou, and Stéphane Vaucher. Restructuring object-oriented applications into component-oriented applications by using consistency with execution traces. In *CBSE*, pages 216–231, 2010.
4. Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit, and Mourad Oussalah. Quality-driven extraction of a component-based aachitecture from an object-oriented system. In *CSMR*, pages 269–273, 2008.
5. Stephane Ducasse and Damien Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35:573–591, 2009.
6. J. Holland. *Adaptation in natural and artificial systems.* University of Michigan Press, 1975.
7. Soo Dong Kim and Soo Ho Chang. A systematic method to identify software components. In *APSEC*, pages 538–545, Washington, DC, USA, 2004. IEEE Computer Society.
8. Jong Kook Lee, Seung Jae Seung, Soo Dong Kim, Woo Hyun, and Dong Han Han. Component identification method with coupling and cohesion. In *APSEC*, pages 79–86, Washington, DC, USA, 2001. IEEE Computer Society.
9. Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Eng.*, 13(2):225–256, 2006.
10. Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1):77–93, 2008.

# Etudes empiriques sur la qualité d'un logiciel lors de son évolution – l'approche MoCQA

Tom Mens[1], Benoît Vanderose[2], and Leandro Doctors[1] and Flora Kamseu[2]

[1] Université de Mons, Belgique
tom.mens@umons.ac.be
[2] Université de Namur (FUNDP), Belgique
benoit.vanderose@fundp.ac.be

**Résumé** Des études empiriques sont nécessaires pour mieux comprendre l'impact de l'évolution et de la maintenance sur la qualité des systèmes logiciels. Plus spécifiquement, elles permettent de comprendre quelles sont les caractéristiques principales affectant la qualité d'un système logiciel lors de son évolution (p.ex., la complétude, la cohérence, la fiabilité, la présence de la documentation et des tests, le respect des conventions de codage, la présence de "design patterns" et l'absence de "antipatterns"). Certaines de ces caractéristiques peuvent être mesurées de manière objective (c'est-à-dire, en utilisant des méthodes et outils de mesure logicielle), alors que d'autres peuvent seulement être estimées de manière subjective (par une analyse ou inspection humaine).

Nous proposons l'utilisation du framework MoCQA, pour « Model-Centric Quality Assessment », permettant de générer des modèles de qualité opérationnels, adaptés aux besoins spécifiques des projets à évaluer, et permettant de combiner et d'aligner des mesures objectives avec des évaluations subjectives. Nous utiliserons ce framework pour effectuer des études empiriques sur des groupes d'étudiants en informatique dans plusieurs universités en Belgique, en France et ailleurs. La réplication d'une même étude permettra d'avoir un échantillon suffisamment grand pour en tirer des conclusions statistiquement significatives.

**Mot-clés :** qualité logicielle, mesure logicielle, évolution logicielle, étude empirique

## 1 Introduction

L'impact de l'évolution et de la maintenance sur les divers aspects de la qualité des systèmes logiciels reste peu connu. L'évolution et la maintenance logicielle peuvent affecter plusieurs aspects d'un système logiciel. Parmi ces aspects, on peut nommer : la complétude, la cohérence, la fiabilité, le respect des conventions de codage et de nommage, et l'utilisation d'un bon style de programmation.

Des études empiriques sont nécessaires pour mieux comprendre la relation entre l'évolution et la qualité d'un système logiciel. Cependant, ce type d'étude est difficile à réaliser pour deux raisons principales :

1. La divergence entre méthodes et outils de mesure : Certaines caractéristiques logicielles peuvent être mesurées de manière objective, en utilisant des méthodes et outils de mesure logicielle, alors que d'autres peuvent seulement être estimées de manière subjective par une analyse ou inspection humaine.
2. Le manque d'accès aux groupes de développeurs travaillant en entreprise et prêt à s'investir dans une étude empirique.

Nous proposons de pallier ces deux problèmes en utilisant le framework de qualité MoCQA (présenté en section 2) pour effectuer des études empiriques sur des groupes d'étudiants en informatique (présenté en section 3). Une première étude a déjà été effectuée, une deuxième étude plus poussée est actuellement en cours.

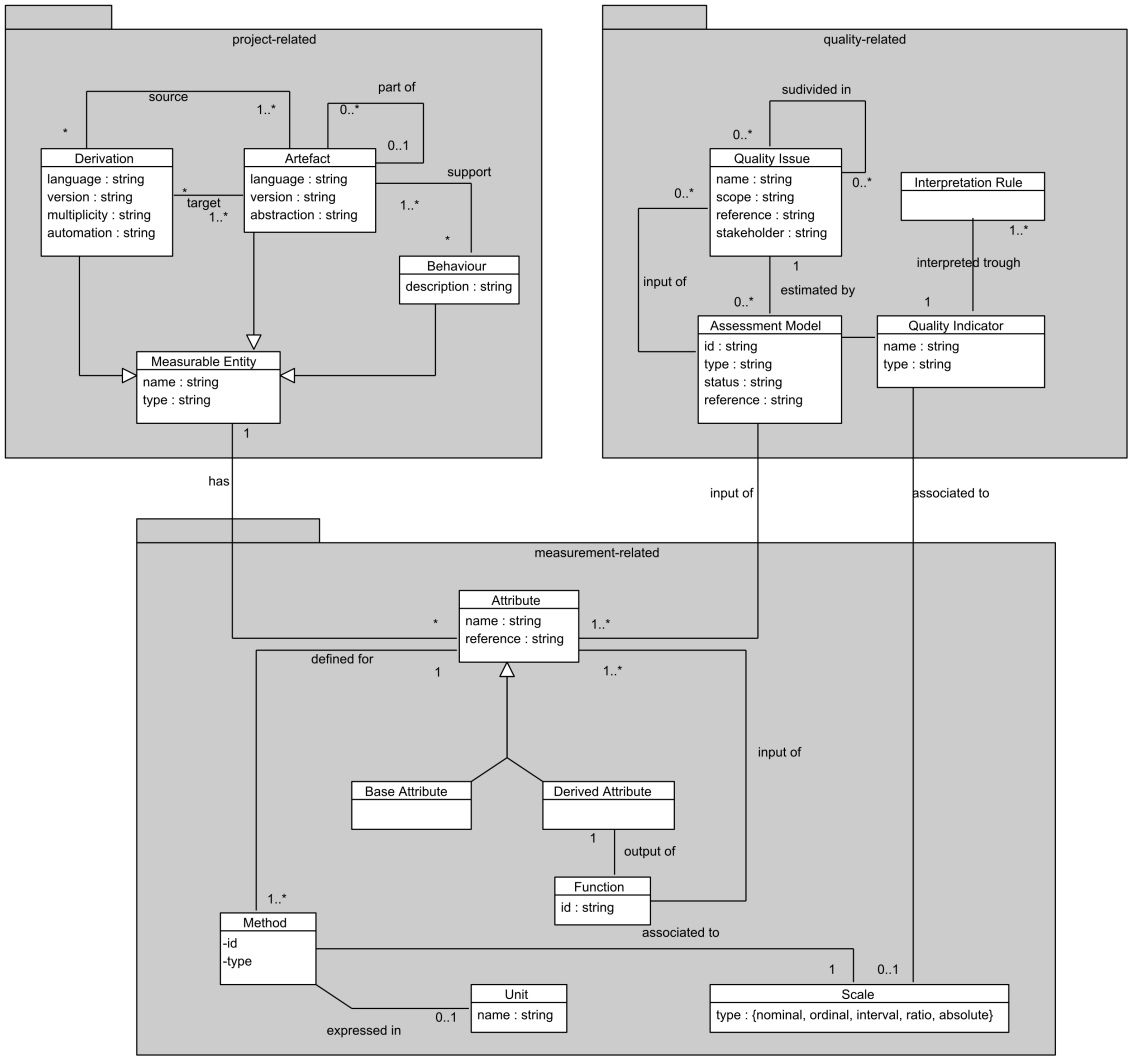



**Figure 1.** Le métamodèle de MoCQA

## 2 Le framework MoCQA

Le framework MoCQA (Model-Centric Quality Assessment) [5] [6] est un framework théorique conçu pour aider à planifier et à assister une assurance qualité ciblée tout au long du cycle de vie logiciel. La particularité de MoCQA est de s'appuyer sur un méta-modèle de qualité, illustré dans la figure 1. Ce méta-modèle de qualité capture tous les concepts des modèles de qualité traditionnels (tels que ceux de McCall [4] ou issu de la norme ISO/IEC 9126 [3]) ainsi que les principaux concepts propres à la mesure logicielle [2,1]. De plus, le méta-modèle de qualité offre un groupe de concepts dédié à la définition explicite et détaillée des entités mesurables impliquées dans le processus d'assurance qualité.

En s'appuyant sur ces trois volets, le framework permet de générer des modèles de qualité opérationnels et taillés sur mesure (modèle MoCQA) qui définissent une hiérarchie de questions sur la qualité d'un projet, ainsi que les méthodes destinées à l'évaluer, et la description des ressources (et des liens entre elles) nécessaires à l'évaluation des questions sur la qualité.

De plus, comme illustré à la figure 2, le niveau d'abstraction du méta-modèle de qualité permet l'intégration, au sein d'un même modèle MoCQA, de modèles de qualité et de méthodes d'évaluations hétérogènes (c-à-d, objectives ou subjectives) issus de différentes sources.

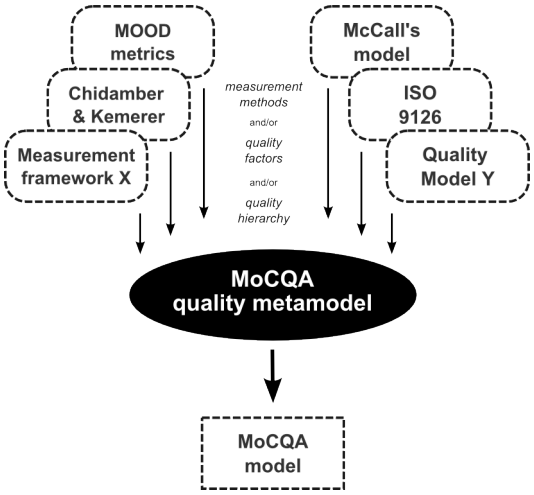



**FIGURE 2.** Intégration de différents modèles de qualité

## 3 Expérimentations

Au moyen du framework MoCQA, nous sommes en mesure de mener des études empiriques portant sur la relation entre l'évolution et la qualité d'un projet de développement logiciel. La méthodologie suivie consiste à générer un modèle MoCQA adapté au contexte, et puis d'appliquer ce modèle sur plusieurs projets d'étudiants afin de collecter de l'information sur la qualité de ceux-ci. Cette information peut alors être analysée en vue de trouver des impacts significatifs de la maintenance sur la qualité (ou inversement).

Dans ce cadre, les avantages fournis par l'utilisation du framework MoCQA sont multiples. D'une part, l'existence d'un modèle MoCQA permet d'assurer la répétabilité de l'expérience. En effet, le modèle MoCQA constitue une carte de l'assurance qualité qui rend très aisé le fait de reproduire une même évaluation dans différents environnements. D'autre part, la flexibilité du framework permet d'intégrer des méthodes d'évaluation subjectives et de les aligner sur des méthodes (c-à-d, comparer les méthodes sur base des attributs qu'elles visent). Enfin, MoCQA permet d'évaluer la qualité d'un projet en se basant sur plusieurs artéfacts mis en relation (p.ex., mettre en relation un diagramme de classe avec une partie du code source, ou mettre les tests en relation avec le code exécutable).

Une première expérience concernant la complétude d'un projet sur base de la complétude du plan de test et de l'analyse des exigences a été menée à l'Université de Namur, sur des projets développés par des étudiants de 1ère année de master, dans le cadre de leur cours de Méthodologie de Développement Logiciel. Cette expérience a mis en évidence qu'une évaluation de la qualité basée sur les relations entre les artefacts s'avérait plus pointue et offrait un meilleur potentiel pour guider la maintenance du projet.

Une deuxième expérience plus poussée est actuellement en cours de réalisation à l'Université de Mons sur des projets d'étudiants de 3e année de bachelier en sciences informatiques dans le cadre de leur projet de génie logiciel en UML et Java. L'étude concerne plusieurs aspects de qualité (complétude, correction, respect de conventions de codage). L'expérience porte sur deux aspects principaux : la possibilité de substituer des méthodes d'évaluation subjectives (inspection humaine) par des mesures objectives (outils automatisés) et la possibilité de prédire un impact de la maintenance sur la qualité d'un projet sur base de celles-ci. La mise en oeuvre consiste à mener, pour chaque projet d'étudiants, deux évaluations indépendantes (l'une basée sur des méthodes subjectives, l'autre sur des mesures objectives) qui seront alignées à travers un modèle MoCQA. Les résultats obtenus seront comparés au travers des projets évalués. Si les résultats sont concluants, l'expérience sera répétée dans d'autres universités afin d'avoir un échantillon suffisamment large pour en tirer des conclusions statistiquement significatives.

*Remerciements.* Cette recherche a été effectuée dans le cadre du Centre d'Expertise en Ingénierie et Qualité des Systèmes (CE-IQS) du portefeuille TIC, co-financé par le Fonds Européen de Développement Régional (FEDER) et Wallonia, Belgique.

## Références

1. Alain Abran. *Software Metrics and Software Metrology.* John Wiley & Sons Interscience and IEEE-CS Press, 2010.
2. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics : A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 1998.
3. ISO/IEC. 9126-1, software engineering - product quality - part 1 : Quality model, 2001.
4. Jim A. Mccall, Paul K. Richards, and Gene F. Walters. Factors in software quality. volume i. concepts and definitions of software quality. Technical Report ADA049014, GENERAL ELECTRIC CO SUNNYVALE CALIF, November 1977.
5. Tom Mens, Leandro Doctors, Naji Habra, Benoit Vanderose, and Flora Kamseu. Qualgen : Modeling and analysing the quality of evolving software systems. In Tom Mens, Yiannis Kanellopoulos, and Andreas Winter, editors, *CSMR*, pages 351–354. IEEE Computer Society, 2011.
6. Benoît Vanderose, Flora Kamseu, and Naji Habra. Towards a model-centric quality assessment. In *Proc. 20th Int'l Workshop on Software Measurement (IWSM2010)*, 2010.

# Table ronde autour du logiciel libre, tendances, enjeux et impacts pour la recherche et l'économie numérique

## Participants

Franck Barbier (Président de session)
Roberto Di Cosmo, Laboratoire PPS, Université de Paris VII
François Letellier, Consultant indépendant - logiciel open source et innovation ouverte
François Mérand, Microsoft Alliance, National Practice Leader

**Résumé** :
Au-delà d'une réalité technologique, le logiciel libre et open source est une réalité économique. Il devient un enjeu stratégique (peut-être bientôt de souveraineté ?) pour les Etats vu le caractère de plus en plus diffusant du numérique dans les objets et systèmes du quotidien. Ce débat vise tout d'abord à re-préciser les définitions de "libre", "open source", donner les types de licence connues... Après un état des lieux (qu'est-ce qui existe à l'échelle planétaire ? quels sont les acteurs et "moins acteurs"...), les intervenants de la table ronde donneront leur opinion et leur vision sur des questions ouvertes comme : Quels (nouveaux ?) business models ? Tout logiciel produit doit-il être open source et/ou libre ? Qui finance leur production ? Quels schémas d'évolution pour le libre/open source, des organisations (fondations) aux bases de code à versionner, moderniser, porter... Quelles assurances, quelles garanties, quelles économies d'échelle, quelle pérennité... pour les (grands) utilisateurs ? On n'oubliera pas les Etats qui, sur les sujets de défense, mènent une réflexion aujourd'hui poussée sur "sécurité numérique et libre/open source".

# Posters et démonstrations

# COSTO / Kmelia : a Platform to Specify and Verify Component and Service Software

**AeLoS Team** LINA - UMR 6241 CNRS - Université de Nantes – EMN - France

## SPECIFICATION

## VERIFICATION

## Overview

The COSTO toolbox is an Eclipse platform that consists of:

- a core module with an ANTLR-based parser and an API to access the Kmelia (internal) model,
- several verification and exportation modules,
- a set of eclipse plugins.

*Figure 1 shows a sample of the kind of errors (typing, observability, incompleteness of the mapping) that are detected. Besides standard completion, the editor supports smart completion in the case of assembly links.*

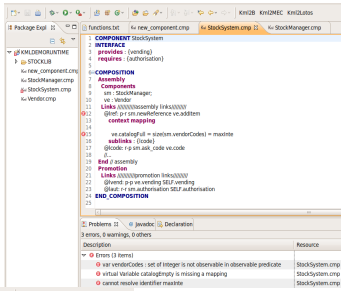
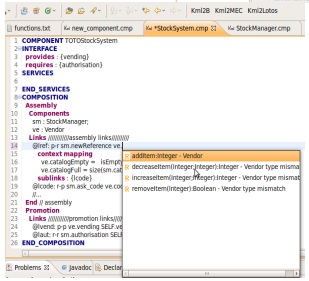
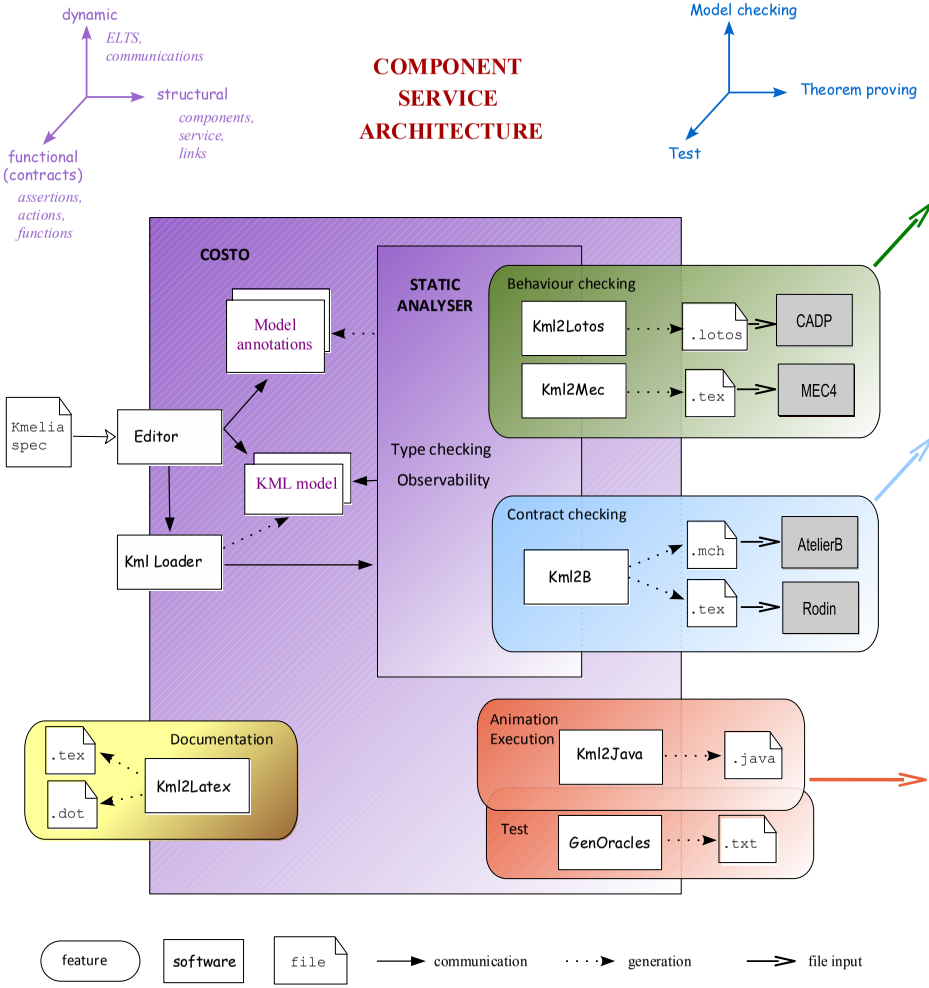


**Figure 1.** Kmelia editor in Eclipse



*In Figure 2, only required services defined in the `Vendor` component type are proposed and the user is warned that some of them do not match the exact signature of the provided service `new_reference` which is defined in the `StockManager` component type.*

**Figure 2.** Smart completion




## Behaviour checking

Using the exportation buttons on the top of the editor while selecting an assembly link generate models in MEC, LOTOS to be verified in external tools.

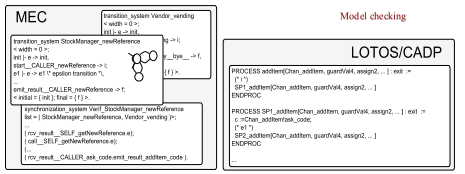
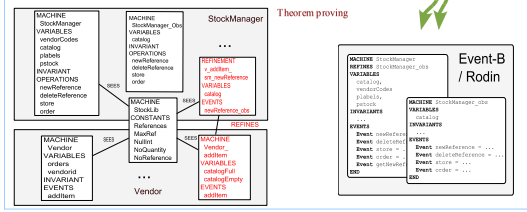


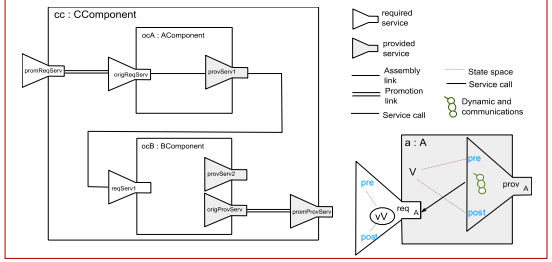

**Figure 3.** Exportations to MEC/LOTOS

## Contract checking

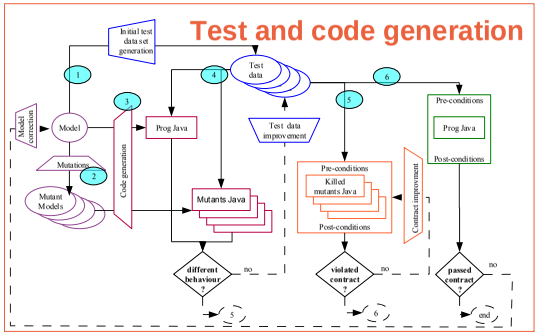
| service | component | assembly | composite |
|---|---|---|---|
| behavioural consistency | component consistency | service signature compatibility (ssic) | ssic |
| functional correctness | protocol correctness | service structure consistency | sstc |
| | service accessibility | service compliance | sco |
| | | behavioural compatibility | bhc |




## A Kmelia architecture




## Test and code generation




## Some Publications


- Christian Attiogbé, Pascal André, and Gilles Ardourel. Checking Component Composability. In 5th International Symposium on Software Composition, SC'06, volume 4089 of LNCS. Springer, 2006.
- Pascal André, Gilles Ardourel, and Christian Attiogbé. Dening Component Protocols with Service Composition: Illustration with the Kmelia Model. In 6th International Symposium on Software Composition, SC'07, volume 4829 of LNCS. Springer, 2007.
- Pascal André, Gilles Ardourel, Christian Attiogbé, and Arnaud Lanoix. Using assertions to enhance the correctness of kmelia components and their assemblies. Electronic Notes in Theoretical Computer Science, 263:5 – 30, 2010. Proceedings of the 6th International Workshop on Formal Aspects of Component Software (FACS 2009).
- P. André, G. Ardourel, C. Attiogbé, and A. Lanoix. Contract-based Verification of KmeliaComponent Assemblies using Event-B. In Proceedings of the Formal Foundations of Embedded Software and Component-Based Software Architectures (FESCA 2010), 2010.
- M. Messabihi, P. André, C. Attiogbé. Multi-levels Use of Contracts for Trusted Components. In Workshop on Component and Service Interoperability, 2010. (WCSI 2010) – EPTCS http://arxiv.org/pdf/1010.2827.
- P. André, G. Ardourel, M. Messabihi. Component Service Promotion: Contracts, Mechanisms and Safety. Proceedings of the 7th International Workshop on Formal Aspects of Component Software (FACS 2010) – LNCS Springer.


## Key features

- Component and Service
- Required service context
- Contract and Protocols
- Assembly and mappings
- Composite and promotion
- Syntax and type checking
- Model-checking
- Theorem proving
- Test and mutation

## For further information

Please contact coloss@univ-nantes.fr

More information on this and related projects can be obtained at http://www.lina.sciences.univ-nantes.fr/coloss/index_en.php.

A PDF-version of the poster is available at http://www.lina.sciences.univ-nantes.fr/coloss/download/posterCosto.pdf

# Program Transformation based Views for Modular Maintenance

Akram AJOULI, Julien COHEN, and Rémi DOUENCE

ASCOLA group: EMN-INRIA, LINA

## 1 Context

Since modular programming is a practical solution for separation of concerns, it participates in reducing development time and it favors maintainability and reuse [5]. Despite of these benefits, modularity does not resolve the classical expression problem [7] and more generally the tyranny of the dominant decomposition [6]: evolutions are modular only on the principal axis of decomposition.

This is illustrated by the two possible structures for a same program given in Fig. 1. The architecture given in Fig. 1(a) is modular with respect to functions (it is modular to add or modify a function), its dominant decomposition is function-oriented. The architecture given in Fig. 1(b) is modular with respect to data constructors (when the data-structure is extended or adapted, it is modular to extend or update the corresponding functions), its dominant decomposition is data-oriented. In object oriented programming, the same duality is observed between the Visitor and the Composite design patterns.

```
module Expr  where

data Expr =
     Const Int
   | Add (Expr,Expr)
```

```
module Eval where
import Expr

eval(Const i)     = i
eval(Add (e1,e2)) = eval e1 + eval e2
```

```
module ToString where
import Expr

toString(Const i)     = show i
toString(Add (e1,e2)) =
    toString e1 ++ "+" ++ toString e2
```

(a) Functional decomposition.

```
module Expr  where

data Expr =
    Const Int
  | Add (Expr,Expr)

fold fc fa (Const i) = fc i
fold fc fa (Add (e1,e2))=
  fa (fold fc fa e1) (fold fc fa e2)
```

```
module Const where
 eval i     = i
 toString i = show i
```

```
module Add where
 eval     x y = x + y
 toString x y = x ++ "+" ++ y
```

```
module Client where
import Expr
import Const
import Add

toString x =
 fold Const.toString Add.toString x

eval x = fold Const.eval Add.eval x
```

(b) Constructor decomposition.

**Fig. 1.** Two alternative architectures for a same Haskell program.

Whatever structure is chosen, some evolutions will be cross-cutting (non modular). In particular, adapting a function in the data-oriented architecture is cross-cutting and vice versa.

To solve this problem, a practical solution would be to be able to choose the architecture each time one has an evolution to implement. This corresponds to a concept of views of programs where each view expresses the whole program, but with a different code structure (as in [1]). With such a solution, a programmer can implement an evolution in the view which is the most convenient. The practicability of this solution depends on the availability of a tool (possibly based on program transformations) to be able to pass from one view to another (from one code structure to another).

## 2 Contributions

We provide a prototype tool for the Haskell language to support the concept of views described above [2]. Our tool allows to build transformations to switch Haskell programs from one view to another. We do this by providing high-level interfaces to a refactoring tool for Haskell (HaRe [4]): transformations are built by chaining elementary operations of refactoring. Moreover, the refactoring tool ensures that the transformations preserve the semantics of the programs: since each elementary refactoring preserves the semantics, the whole transformation also does.

## 3 Challenges

To be practicable, our proposal must respond to the following challenges:

**Automatic inference of program transformations.** A disadvantage of our proposal is that the transformation has to be defined imperatively, which is tedious. To solve this, we must provide automatic transformation inference based on constraints expressed by the user. This would also avoid to have to maintain the transformations as the programs evolve.

**Invertibility of program transformations.** Once a first transformation is set up (automatically or not), the inverse transformation could be computed from the first one. However, all refactoring operations are not invertible and we cannot tell now what architecture transformations can be built with invertible elementary transformations.

**Minimum precondition.** As static type checking has advantages over dynamic typing with respect to assistance for development, a minimal precondition analysis performed on transformations can help to set up transformations and check their properties [3]. Thus, one could have the guaranty that a transformation will succeed before performing it.

## References

1. Andrew P. Black and Mark P. Jones. The case for multiple views. In *ICSE 2004 Workshop on Directions in Software Engineering Environments*, 2004.
2. Julien Cohen and Rémi Douence. Views, Program Transformations, and the Evolutivity Problem in a Functional Language. 19 pages, Research Report hal-00481941, version 2, January 2011.
3. Günter Kniesel and Helge Koch. Static composition of refactorings. *Science of Computer Programming*, 52(Issues 1-3):9–51, Aug. 2004. Special Issue on Program Transformation.
4. Huiqing Li and Simon Thompson. Tool Support for Refactoring Functional Programs. In *Partial Evaluation and Program Manipulation*, San Francisco, CA, USA, Jan. 2008.
5. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, December 1972.
6. Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Jr. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, ICSE '99, pages 107–119, New York, NY, USA, 1999. ACM.
7. Philip Wadler. The expression problem. Note to Java Genericity mailing list, Nov. 1998.

# Cloud Components for Highly Distributed Environments

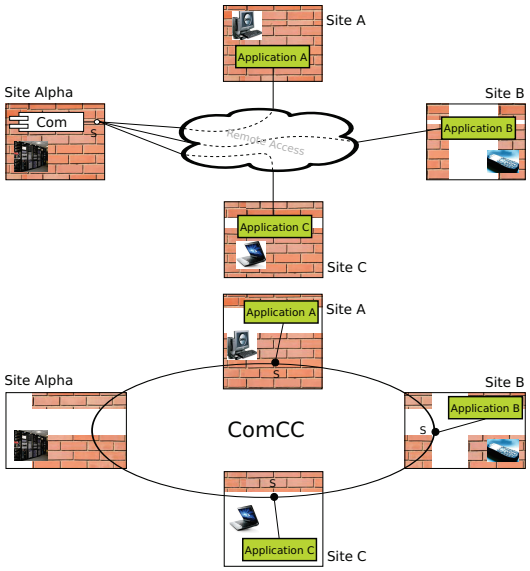
**Antoine Beugnard, Ali Hassan,**
Telecom Bretagne - France

## Scientific Challenge

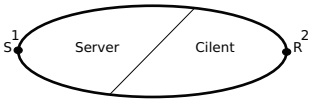
- There is excessive and increasing need to build complex mobile and pervasive systems for entertainment and professional uses.
- The fundamental software engineering techniques **available now** are inherited from stable distributed environments.
- We propose a solution for this gap.

## Paradigm Shift

- We propose software engineering shift from **distribution transparency** to **localization acknowledgment**.
- All communication and networking related issues are **migrated** to be internal to the component border.
- Any device that needs to access the services provided by the component needs to instantiate a role of that component **locally.**
- The role implementation that is instantiated over a host is tailored to that host (it passes a formal check before instantiation).
- As result **we guarantee high level of confidence of the quality of service on the point where the user is,** not at the source of the service.




## Cloud Component Definition




- Set of roles (interfaces). $\overline{\Lambda}$
- Each role has multiplicity. $\overline{\mu}$
- The expected deployment environment where the CC will be deployed is defined clearly (location). $\overline{L}$, $Z$

$\Omega com \equiv (\overline{\Lambda},\ \overline{\mu},\ \overline{L},\ Z)$ where:

$\overline{\Lambda} = \{\Lambda S,\ \Lambda R\}$

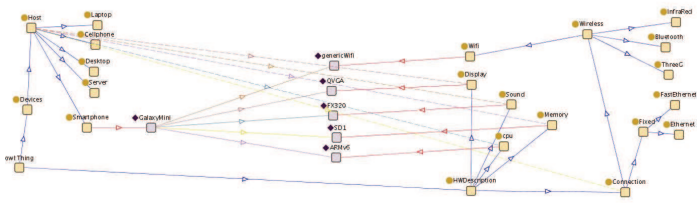
$\overline{\mu} = \{(\Lambda S,\ 1),\ (\Lambda R,\ 2)\}$

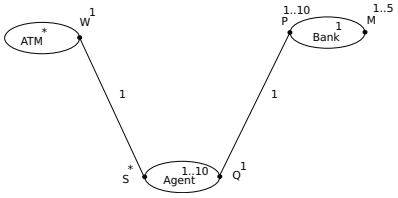
$\overline{L} = \{\{TServer,\ TClient\_1,\ TClient\_2\}\}$

$Z:\ \ \Lambda S \downarrow TServer,\ \ \Lambda R \downarrow TClient\_1,\ \ \Lambda R \downarrow TClient\_2$

## Location Modeling - CC Assembly Checker

We model location using object logic and Ontology






Automatic formal connectivity checker increases the level of confidence

antoine.beugnard@telecom-bretagne.eu, ali.hassan@telecom-bretagne.eu

# Towards a unified formal model for service orchestration and choreography

Diana Allam - PhD Student
Supervised by: Herve Grall and Jean-Claude Royer

ASCOLA group; EMN-INRIA, LINA
4 rue Alfred Kastler, 44307 NANTES Cedex 3, France
{diana.allam, herve.grall, jean-claude.royer}@emn.fr

The growth of Internet has extended the scope of software applications, leading to network-based architectures. The main characteristic of these architectures is that they restrict the communication between remote components to message passing. Service-oriented computing is a solution to organise the exchange of messages in a network-based architecture, by using services as primitive components. Thus, each component can be a client, a server or both. Since a service-oriented application typically spans a number of different organizations, its executions is subject to stringent security requirements. That is the reason why the partners involved generally define a contract at the global level in order to enforce some security policy. From the contract, each partner deduces by projection a specification of the security functionalities that it must locally implement. Of course, in order to be useful, all these projections must ensure that the local functionalities effectively collaborate to realize the global contract.

The ANR project CESSA [1] essentially aims at tackling this problem: in the context of service-oriented computing, it will provide means to locally enforce a security policy globally defined, by resorting to aspect-oriented programming. The core of the project is a formal model. Generalizing the top-down approach described above for security, the formal model is based on two stages.

**Description stage** Aiming at expressing global properties and their local projections, it will lead to a choreography language and an interface language for local processes.

**Realization stage** Aiming at expressing local processes and their collaboration, it will lead to a process orchestration language and a collaboration infrastructure.

The stages are linked, since the formal model must satisfy the fundamental projection property: given a choreography, if each local process realizes the corresponding projection of the choreography, then their collaboration realizes the choreography.

In this abstract, we will focus on the realization stage, already achieved. We will also shortly describe some future steps for fulfilling the description stage and for linking the two stages.

**Context: choreography and orchestration of web services** A choreography defines a contract between partners that collaborate by exchanging messages: it corresponds to the protocol used to exchange messages. Locally, each partner controls processes that invoke or provide services. At the process level, an orchestration describes the interactions of one service with other services. Contrary to the choreography where no central coordinator exists, an orchestration makes explicit a central coordinator which is responsible for invoking and combining the services realizing the composition.

There are languages not only for the orchestration of web services like the standard business process execution language (BPEL), but also for choreography like the web services choreography description language (WS-CDL) [7]. There are also languages that offer the possibility of modeling these two levels like the standard business process model and notation (BPMN 2.0).

The formal model that we now present can be considered as an abstraction of BPEL and of BPMN 2.0.

**Collaboration infrastructure** A collaboration involves a set of processes that execute in a concurrent way and interact each other via message exchanges in a completely asynchronous way. Each process has an address, like a uniform resource locator. When a process executes, it can receive messages from other

[1] CESSA (Compositional Evolution of Secure Services using Aspects) is an ANR project in partnership with SAP labs France, EURECOM and IS2T, http://cessa.gforge.inria.fr

processes that knows its address and send messages to processes the address of which it knows. The topology of the collaboration network can evolve as addresses can be exchanged. For example, in a simple client-server interaction, the client can send a request because he knows the server address. The server's knowledge of the network increases at the reception of the request message which contains the client location. By this way, the server can reply the client while he did not know its address before the request.

**Process orchestration language** A wide variety of formal models of web services exists for the orchestration level. Amongst the candidate models we consider [1,5,6]. These models are essentially based on the π-calculus, since this process calculus allows channels to be communicated, a fundamental feature as seen above and in [2]. [5] introduces an extension of the π-calculus with a notion of transaction. Viroli [6] introduces also a π-calculus based semantics but focusing on correlation sets. It details the session management in BPEL. Another notable work is the one from Abouzaid and Mullins [1]. The author proposes a formal BP-language close to the π-calculus and inspired from the work of Lucchi and Mazzara for handlers and of Viroli for correlation sets. Following and extending these studies, we also propose a process language as an extension of the π-calculus. The three most important differences between our model and the π-calculus are (i) the language has state variables and expressions to denote the values assigned to the variables while in the π-calculus, there are no state variables (but they can be encoded) (ii) whereas a channel is represented in our process model by an ordered pair, a location and a name for the message, there is no way to define a new location, contrary to the π-calculus, and (iii) channels are completely asynchronous to be closer to real network (it is not the case for synchronous channels as they are defined in the π-calculus). One more important contribution in our model is the use of correlation sets in a more general form than the one defined in BPEL or in its formalization. Correlation allows a process to hold conversations with partners involved in a collaboration. It is equivalent to the creation of sessions to maintain a connection between two partners. In the standard semantics of the π-calculus, when different processes are waiting for an incoming message, the process fired is randomly chosen. With correlation, it is the process whose state is correlated with the incoming message.

**Extensions and future work** As an extension of the formal model, we expect to add exceptions and handlers so to have a common abstraction of what is defined in BPEL and BPMN. Our goal is to reduce the complexity and the ambiguity that we see in the BPEL standard and to have a more understandable and simpler model closer to BPMN 2.0. Beyond the extension, we will complete the formal model by the description stage. We aim to follow the top-down approach, from the global choreography to the local orchestration. Starting from a choreography described as collaboration types, inspired from session types [3], we will deduce its projection over each process involved. Then each partner will develop its processes realizing the projections, in order to finally produce a collaboration realizing the global choreography. We are more particularly interested in projection theories [4], specially for analyzing and enforcing some security properties. We aim to extend these theories to other interaction modes, not only to the synchronous mode. At the present, there is no study of the purely asynchronous case for collaborations, as they are defined in our framework.

## References


1. Faisal Abouzaid and John Mullins. Formal specification of correlation in WS orchestrations using BP-calculus. *Electr. Notes Theor. Comput. Sci*, pages 3–24, 2010.
2. Michele Boreale and al. SCC: A service centered calculus. In *Web Services and Formal Methods, Third International Workshop, WS-FM, Proceedings*, Lecture Notes in Computer Science, pages 38–57. Springer, 2006.
3. Marco Carbone, Kohei Honda, and Nobuko Yoshida. A calculus of global interaction based on session types. *Electronical Notes in Theoretical Computer Science*, 171:127–151, June 2007.
4. Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. pages 2–17. Springer, 2007.
5. Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program*, pages 96–118, 2007.
6. Mirko Viroli. A core calculus for correlation in orchestration languages. *J. Log. Algebr. Program*, 2007.
7. Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standards - the case of REST vs. SOAP. *Decision Support Systems*, 40:9–29, 2005.

# Construction collaborative de services de télécommunications: un processus dirigé par les modèles pour la génération d'outils


## Auteur

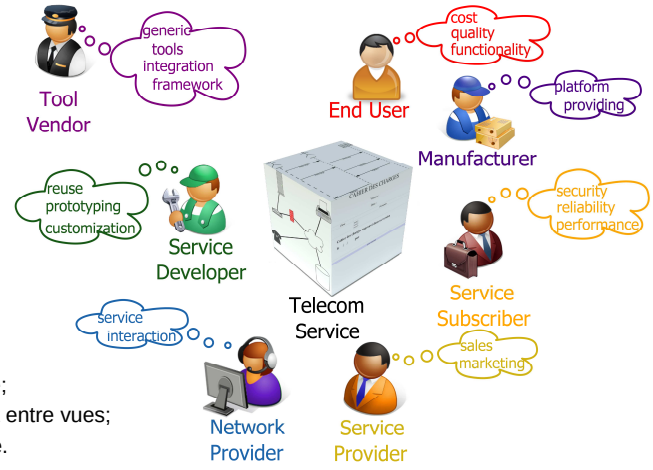
Vanea CHIPRIANOV
Doctorant 3ème année
Dépt. LUSSI

Sous la direction de:
Yvon KERMARREC
Dépt. LUSSI
Siegfried ROUVRAIS
Dépt. INFO


## Partenaires

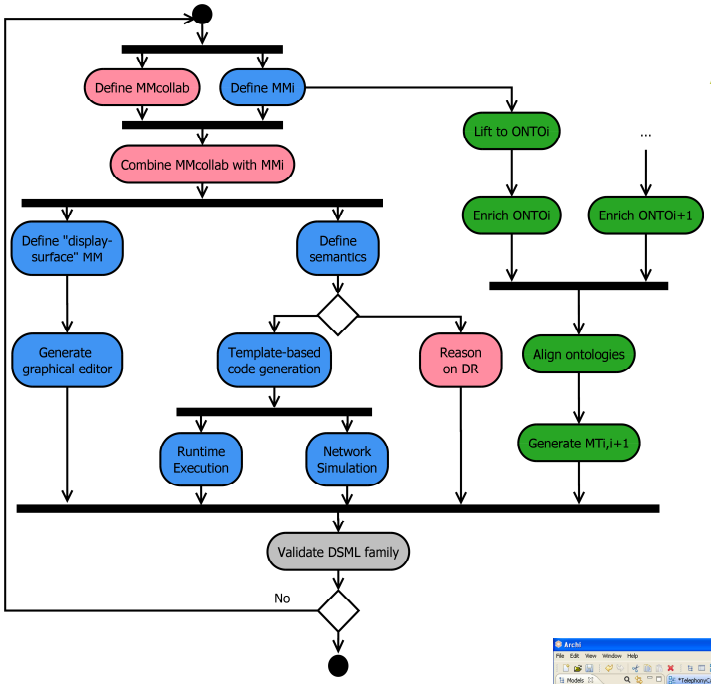
## Contexte

- Systèmes télécoms complexes, s'appuyant sur plusieurs domaines et entreprises, avec des contraintes spécifiques par métier;
- Chaque métier a des habitudes, termes, langages, standards, outils dédiés;
- La complexité peut être mieux gérée en vues (e.g. 3 en ArchiMate), dédies à un ou plusieurs métier + 1 métier (i.e. *Tool Vendor*) pour fournir les outils;
- Travail collaboratif: N designers par vue.




## Problèmes

- Construction de modèles spécifiques à chaque vue;
- Collaboration des équipes de designers, par vue et entre vues;
- Interopérabilité des outils spécifiques à chaque vue.



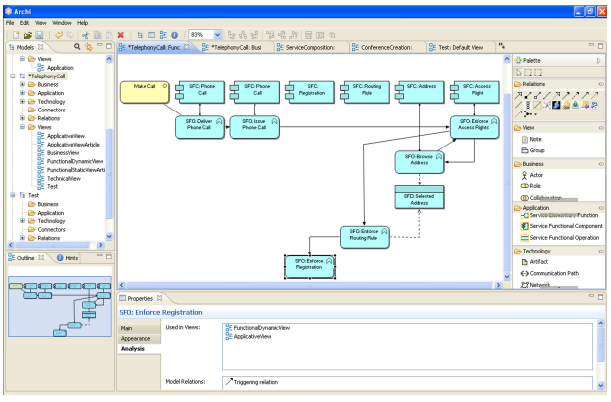

## Approche

Un processus dédié au *Tool Vendor* pour:

- La conception de langages (i.e. *Domain Specific Modeling Language*) et outils dédiés à chaque vue;
- La conception d'un langage permettant aux designers de collaborer en indiquant et capitalisant les choix des conception (i.e. *Design Rationale);*
- Une approche basée sur l'application de transformations des modèles et ontologies pour assurer l'interopérabilité entre les outils de deux vues voisines dans le cycle de vie.
- L'intégration des retours des designers, qui valident la famille de langages.

## Résultats

- Définition de 3 langages dédiés, comme extensions de ArchiMate (cf. TOGAF), pour ses 3 vues;
- Fourniture d'un éditeur graphique intégré pour les 3 langages;
- Définition d'un langage pour les choix des conception et son intégration dans l'éditeur graphique;
- Transformations vers des outils existants, spécifiques au domaine des télécommunications (e.g. NS-2, OPNET).



## Pour en savoir plus


- Chiprianov, Kermarrec, Rouvrais: *Towards semantic interoperability of graphical DSMLs for telecommunications service design*. In 2nd Intl. Conf. on Models and Ontology-based Design of Protocols, Architectures and Services (**MOPAS 2011**), pp. 21-24.
- Chiprianov, Kermarrec, Rouvrais: *Meta-tools for Software Language Engineering: A Flexible Collaborative Modeling Language for Efficient Telecommunications Service Design*. In: **FlexiTools** WS at the 32nd ACM/IEEE Int. Conf. on Software Engineering (**ICSE 2010**).
- Chiprianov, Kermarrec, Alff, P.: *A Model-Driven Approach for Telecommunications Network Services Definition*. In: Proc. of the 15th Open European Summer School and IFIP TC6. 6 WS on The Internet of the Future (**EUNICE 2009**), pp. 199-207.
- Chiprianov, Kermarrec: *Model-based DSL Frameworks: A Simple Graphical Telecommunications Specific Modeling Language*. In: French Colloq. on Model Driven Eng. (**IDM 2009**), pp. 179–186.
- Chiprianov, V., Kermarrec, Y.: *An Approach for Constructing a Domain Definition Metamodel with ATL*. In: Model Transformation with ATL (**MtATL 2009**), 1st Int. WS, pp. 18-33.



Contact vanea.chiprianov@telecom-bretagne.eu - http://perso.telecom-bretagne.eu/vaneachiprianov/

## Résumé

Ce document contient les actes des troisièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées à l'Université de Lille I du 8 au 10 Juin 2011.

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.