CIEL 2014 : Conférence en IngénieriE du Logiciel

Marie-Agnès Peraldi-Frati, Christelle Urtado

Troisième Conférence en IngénieriE du Logiciel

CIEL 2014

Conservatoire National des Arts et Métiers

Paris, du 10 au 12 Juin 2014

Sous la direction de :
Catherine Dubois, Nicole Levy,
Marie-Agnès Peraldi-Frati et Christelle Urtado

Après Rennes en 2012, et Nancy en 2013, la troisième édition de CIEL à lieu du 10 au 12 Juin 2014 au CNAM à Paris.

La conférence CIEL (Conférence en IngénieriE du Logiciel) remplace depuis 2012 les journées IDM (journées sur l'Ingénierie Dirigée par les Modèles) et la conférence LMO (Langages et Modèles à Objets).

Elle reprend donc leurs thématiques, liées aux technologies à objets et à l'ingénierie des modèles dans le domaine des langages, de la représentation des connaissances, des bases de données, du génie logiciel et des systèmes. Partant de ces thématiques historiques, la conférence étend naturellement les thèmes abordés aux agents, aspects, composants, services et lignes de produits logiciels.

Son objectif est de réunir les chercheurs et industriels intéressés par ces différentes thématiques pour faciliter l'échange d'idées, la diffusion de résultats et d'expériences, la structuration de la communauté et l'identification de nouveaux défis.

Pour cette édition 2014, la conférence CIEL se tient conjointement avec les journées nationales du GDR GPL 2014 ainsi qu'avec les conférences AFADL (Approches Formelles dans l'Assistance au Développement de Logiciels) et CAL (Conférence francophone sur les Architectures Logicielles).

Le comité de programme de CIEL 2014 a reçu 23 soumissions parmi lesquelles 7 articles longs, 5 articles courts, 3 résumés de papiers internationaux, 4 résumés de thèse et 4 démonstrations d'outils. Chacun de ces papiers a été évalué par 3 relecteurs. Dans l'optique de permettre une communication et une discussion très large, le comité de programme a accepté la présentation de 19 contributions. A ceci s'ajoutent deux contributions supplémentaires présentées dans la cadre de sessions communes avec CAL 2014.

Les présidentes du comité de programme remercient :

- Les membres du comité de programme et les relecteurs additionnels pour le travail d'évaluation fourni,
- Les auteurs pour leur mobilisation à faire de cet événement le leur,
- Les membres du comité d'organisation,
- Les président et présidentes des comités de,programmes de CAL 2014 et AFADL 2014,
- La directrice du GDR GPL.

Nous vous souhaitons une excellente édition 2014 de la conférence CIEL !

Marie-Agnès Peraldi-Frati et Christelle Urtado
Co-Présidentes de CIEL 2014

Comité de programme

Marie-Agnès Peraldi-Frati, Université Nice Sophia Antipolis – I3S / CNRS — Co-Présidente
Christelle Urtado, LGI2P / Ecole Nationale Supérieure des Mines d'Alès — Co-Présidente

Nicolas Anquetil, Université de Lille 1 - INRIA Lille Nord Europe
Jean-Philippe Babau, Université de Bretagne Occidentale - Lab-STICC
Olivier Barais, Université de Rennes 1 - IRISA
Alexandre Bergel, Université du Chili - DCC - PLEIAD Lab
Antoine Beugnard, Telecom Bretagne - IRISA
Xavier Blanc, Université de Bordeaux 1 - LaBRI
Isabelle Borne, Université de Bretagne Sud - IRISA
Jordi Cabot, École des Mines de Nantes - LINA, INRIA Rennes Bretagne Atlantique
Eric Cariou, Université de Pau – LIUPPA
Sorana Cimpan, LISTIC, Université de Savoie
Philippe Collet, Université Nice Sophia Antipolis - I3S/CNRS
Bernard Coulette, Université de Toulouse 2 Le Mirail - IRIT
Julien DeAntoni, Polytech'Nice - I3S EPC INRIA Aoste
Sophie Dupuy-Chessa, Université de Grenoble - LIG
Vincent Englebert, Université de Namur
Franck Fleurey, SINTEF
Marie-Pierre Gervais, Université Paris Ouest Nanterre La Défense - LIP6
Jérôme Le Noir, Thales Research and Technology
Yves Le Traon, Université du Luxembourg
Jacques Malenfant, Université Pierre et Marie Curie - LIP6
Philippe Merle, INRIA Lille Nord Europe
Naouel Moha, Université du Québec à Montréal
Amédéo Napoli, LORIA – CNRS
Clémentine Nebut, LIRMM, Université de Montpellier 2
Ileana Ober, Université Paul Sabatier - IRIT
Marc Pantel, ENSEEIHT, Université de Toulouse – IRIT
Noël Plouzeau, Université de Rennes 1 - IRISA
Pascal Poizat, Université Paris Ouest Nanterre La Défense - LIP6
Jean-Claude Royer, École des Mines de Nantes - LINA, INRIA Rennes Bretagne Atlantique
Dalila Tamzalit, LINA, IUT de Nantes
Sara Tucci Piergiovanni, CEA LIST
Chouki Tibermacine, Université de Montpellier 2 - LIRMM
Sylvain Vauttier, LGI2P - École des Mines d'Alès
Olivier Zendra, INRIA Nancy Grand Est

Comité d'organisation

Catherine Dubois, ENSIIE - CEDRIC, Evry — Co-Présidente
Nicole Levy, CNAM - CEDRIC, Paris — Co-Présidente

Reda Bendraou, Université Pierre & Marie Curie, LIP6
Tristan Crolard, CNAM – CEDRIC
David Delahaye, CNAM – CEDRIC
Frédéric Gava, Université Paris Est - LACL

Pascal Poizat, Université Paris Ouest Nanterre La Défense - LIP6
Renaud Rioboo, ENSIIE - CEDRIC, Evry

Relecteurs additionnels

Bouchra El Asri, Laboratoire de Génie Informatique - ENSIAS, Rabat, Maroc
Fatma Krichen, Ecole Nationale d'Ingénieurs de Sfax (ENIS), Tunisie

Programme de CIEL 2014

Paris, CNAM, 10 - 12 Juin 2014

Mardi 10 Juin	
10h30-11h	Pause café - accueil
11h00-12h30	Ouverture de la conférence CIEL Conférence invitée CAL/ CIEL, Didier Donsez LIG
12h30-13h00	Session 1 commune CAL / CIEL : Composants *Papier international* *Compatibility Checking for Asynchronously Communicating Software* *Meriem Ouederni, Gwen Salaun and Tevfik Bultan*
13h00-14h00	Repas
14h00-15h30	Session 2 : IDM et validation *Papier international* *Une revue des techniques de vérification formelle pour la transformation de modèles : Une classification tridimensionnelle* *Moussa Amrani, Benoit Combemale, Pierre Kelsen et Yves Le Traon* *Papier long* *Alignement de modèles métiers et applicatifs : Une approche pragmatique par transformations de modèles* *Jonathan Pépin, Pascal André, Christian Attiogbe et Erwan Breton* *Papier long* *Vers la vérification formelle de transformations de modèles orientées objet* *Moussa Amrani, Pierre Kelsen et Yves Le Traon*
15h30-16h	Pause Café
16h-18h	Session 3 commune CAL / CIEL : Cycle de vie des architectures logicielles *Papier international* *Service Identification Based on Quality Metrics - Object-Oriented Legacy System Migration Towards SOA* *Seza Adjoyan, Abdelhak-Djamel Seriai et Anas Shatnawi:* *Papier court* *Un modèle de composants unifié pour l'évolution dynamique des systèmes logiciels* *Salim Kebir et Djamel Meslati* *Papier long* *Génération de métaprogrammes Java à partir de contraintes architecturales OCL* *Sahar Kallel, Chouki Tibermacine, Mohamed Reda Skay, Christophe Dony et Ahmed Hadj Kacem* *Papier court* *Modélisation et vérification formelles en B d'architectures logicielles à trois niveaux d'abstraction* *Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier et Huaxi Yulin Zhang*

Mercredi 11 Juin	
9h-10h30	Ouverture officielle GDR Conférence invitée CAL/CIEL/AFADL/GDR Roland Ducournau LIRMM
10h30-11h	Pause café
11h-13h	Session 4 : Travaux de doctorants *Framework for heterogeneous modeling and composition* *Matias Vara Larsen, Julien Deantoni et Frédéric Mallet* *Graphe de dépendance pour la recontextualisation de modèles* *Paola Vallejo, Mickael Kerboeuf et Jean-Philippe Babau* *Simulation orientée utilisateur des Systèmes d'Information des Smart Grids* *Rachida Seghiri, Frédéric Boulanger, Vincent Godefroy et Claire Lecocq* *Contribution to model verification: operational semantic for System Engineering modeling languages* *Blazo Nastov*
13h-14h	Repas
14h00-15h30	Session 5 : Modélisation de la dynamique *Papier long* *Une sémantique multi-paradigme pour simuler des modèles SysML avec SystemC-AMS* *Daniel Chaves Café, Filipe Vinci Dos Santos, Cécile Hardebolle, Christophe Jacquet et Frédéric Boulanger* *Papier long* *Etendre les patrons de flot de contrôle dynamique avec des dépendances transactionnelles* *Imed Abbassi et Graiet Mohamed*
15h30-16h	Pause café
16h-18h	Session commune Posters/démos CAL/CIEL/AFADL/GDR *ModHel'X, un outil expérimental pour la modélisation multi-paradigmes* *Christophe Jacquet, Cecile Hardebolle et Frédéric Boulanger* *Evaluation de la substituabilité comportementale de composants UML* *Thomas Lambolais, Anne-Lise Courbis et Thanh Liem Phan* *BUT4Reuse Feature identifier: Identifying reusable features on software variants* *Jabier Martinez, Tewfik Ziadi, Jacques Klein et Yves Le Traon* *Recherche de sous-modèles* *Gilles Vanwormhoudt, Bernard Carré, Olivier Caron et Christophe Tombelle*

Jeudi 12 Juin	
9h-10h30	Conférence invitée CIEL/AFADL/GDR Christine Paulin LRI
10h30-11h	Pause café
11h-13h	Session 6 : Modélisation de propriétés non fonctionnelles *Papier international* *ORQA : modélisation de l'énergie et de la qualité de service* *Borjan Tchakaloff, Sébastien Saudrais et Jean-Philippe Babau* *Papier court* *Mise à jour dynamique des applications JavaCard: Une approche pour une mise à jour sûre du tas.* *Razika Lounas, Mohamed Mezghiche et Lanet Jean-Louis* *Papier long* *Challenges in security engineering of systems-of-systems* *Vanea Chiprianov, Laurent Gallon, Manuel Munier, Philippe Aniorte et Vincent Lalanne*
13h-14h	Repas
15h30-16h	Pause café

Vendredi 13 Juin	
9h-10h30	Conférence invitée GDR Gérard Morin Esterel Tech
10h30-11h	Pause café
13h-14h	Repas

Table des matières

Conférences invitées

Conférence invitée du mardi 10 Juin

Didier Donsez

Internet des Choses, Cloud Computing et Big Data
Nouvelles frontières pour le Génie Logiciel

Biographie :

Didier Donsez est Professeur en informatique à l'Université Joseph Fourier, Grenoble, France. Il est membre de l'équipe ERODS du LIG.
Ses recherches sont à l'intersection du génie logiciel, des intergiciels et des systèmes distribués. Il applique plus particulièrement ses recherches dans l'intégration de l'informatique ubiquitaire dans les infrastructures IT des organisations et des entreprises.

Résumé :

L'Internet des Choses, le Cloud Computing et le Big Data Analytics sont des domaines émergents de l'informatique qui révolutionneront nos sociétés et nos économies dans les prochaines années. Leur mise en œuvre reste cependant très complexe pour les développeurs du fait du caractère très concurrentiel des technologies standard ou propriétaires disponibles et de la grande variété de plateformes, de langages et de canevas utilisés et utilisables. Cette présentation donne une brève introduction à ces domaines et liste quelques défis pour le génie logiciel.

Conférence invitée du mercredi 11 Juin

Roland Ducournau

Les talons d'Achille de la programmation par objets

Biographie :

Roland Ducournau est Professeur à l'Université Montpellier 2 depuis 1994. Depuis 1985, il travaille sur la programmation par objets, d'abord dans la SSII Sema Group où il a conçu, développé et appliqué le langage Yafool, puis à l'Université. Il s'est en particulier intéressé, successivement ou simultanément, à l'héritage multiple, aux aspects représentation des connaissances liés au modèle objet, ainsi qu'à l'implémentation efficace des mécanismes objet. Il collabore actuellement avec Jean Privat (UQAM) et Floréal Morandat (LaBRI) autour d'un langage de laboratoire, NIT, conçu à l'origine à Montpellier par J. Privat.

Résumé :

Au niveau des spécifications d'abord. De façon générale, pour chaque grand trait de langage comme la surcharge statique, la généricité, l'héritage multiple (au sens large), on trouve avec difficultés deux langages qui s'accordent sur leurs spécifications. Rien que pour la surcharge statique, sur les 4 langages mainstream que sont C++, Java, C\# et Scala, on trouve 5 spécifications différentes, ce qui fait un peu douter de la pertinence de la notion.
Au niveau de l'implémentation ensuite. Malgré la multitude de systèmes d'exécution de ces langages et leur extrême sophistication, la question des performances et du passage à l'échelle reste posée en cas d'héritage multiple (au sens large) et de chargement (ou édition de liens) dynamique.
L'exposé présentera successivement les éléments de spécification les plus caractéristiques, avec une esquisse de solution, puis la technique d'implémentation à base de hachage parfait qui passe à l'échelle dans un contexte de chargement dynamique et d'héritage multiple.

Conférence invitée du jeudi 12 Juin

Christine Paulin

Preuves formelles d'algorithmes probabilistes

Biographie :

Christine Paulin-Mohring est professeur à l'université Paris-Sud 11 depuis 1997, après avoir été chargée de recherche CNRS au LIP à l'ENS Lyon.
Elle exerce ses activités au LRI dans le cadre du projet commun Inria Toccata. Ses recherches portent sur la théorie des types, les assistants de preuve et leur application au développement de programmes corrects par construction. Elle a contribué au développement de l'assistant de preuve Coq, en particulier en ce qui concerne l'extraction de programmes, les définitions inductives et plus récemment une bibliothèque pour raisonner sur les programmes aléatoires. Elle a coordonné le développement de Coq de 1996 à 2004.
Coq a reçu en 2013 deux prix l'ACM SIGPLAN Programming Languages awardet l'ACM Software System award.
C. Paulin-Mohring a été déléguée scientifique du centre INRIA Saclay Ile-de-France de 2007 à 2011. Elle est responsable depuis 2012 du labex DigiCosme dans le cadre de l'Idex Paris-Saclay. Elle a été directrice de l'ED Informatique Paris-Sud de 2005 à 2012 et dirige actuellement le collège des Ecoles Doctorales de Paris-Sud. Elle préside le département Informatique de Paris-Sud depuis février 2012.

Résumé :

Nous montrons comment utiliser l'assistant de preuves Coq pour raisonner sur des programmes probabilistes.
Notre démarche sera illustrée sur plusieurs exemples dont l'exercice de probabilité du baccalauréat 2013 et l'analyse de programmes qui ne terminent pas toujours.

Conférence invitée du vendredi 13 Juin

Gérard Morin et Yves Guido

SCADE Model-Based Requirements Engineering

Biographie :

Gérard Morin, Directeur des Services Professionnels, a rejoint Esterel Technologies en Octobre 2001 en tant que responsable marketing industriel. Entre 2002 et 2005, il a été directeur du marketing produit pour la ligne de produit SCADE.
Depuis 2005, Il est directeur des Services Professionnels : il dirige ou conseille des projets développés avec SCADE avec nos clients, il manage le développement des formations SCADE (présentiel et e-learning).
Gérard est spécialisé en ingénierie système et logicielle, en particulier dans le domaine des systèmes critiques (standards DO-178C, ARP-4754A, EN50128, et IEC 61508).

Résumé :

Creating a set of complete and correct Requirements is the primary responsibility of Systems Engineers, from the point of view of other teams involved in the construction of a system, even though systems engineering encompasses much more than this The SCADE Model-Based approach, as well as SCADE Data-Based representation, help System Engineers to implement a true Requirements Engineering process. This approach includes the use of the SCADE Rapid Prototyping capability to simulate, early in the development process, the Systems operations. Functional Decomposition, synthesis of Architecture exploration and Interface Control Document are created and maintained through safe iterations, all tightly linked to the set of requirements.

Session 1 commune CAL / CIEL :

Composants

Compatibility Checking for Asynchronously Communicating Software

Meriem Ouederni, Gwen Salaun and Tevfik Bultan

Compatibility is a crucial problem that is encountered while constructing new software by reusing and composing existing components. A set of software components is called compatible if their composition preserves certain properties, such as deadlock freedom. However, checking compatibility for systems communicating asynchronously is an undecidable problem, and asynchronous communication is a common interaction mechanism used in building software systems. A typical approach in analyzing such systems is to bound the state space. In this paper, we take a different approach and do not impose any bounds on the number of participants or the sizes of the message buffers. Instead, we present a sufficient condition for checking compatibility of a set of asynchronously communicating components. Our approach relies on the synchronizability property which identifies systems for which interaction behavior remains the same when asynchronous communication is replaced with synchronous communication. Using the synchronizability property, we can check the compatibility of systems with unbounded message buffers by analyzing only a finite part of their behavior. We have implemented a prototype tool to automate our approach and we have applied it to many examples.

Session 2

IDM et validation

Une Revue des Techniques de Vérification Formelle pour la Transformation de Modèles : Une Classification Tridimensionnelle

Moussa Amrani[1], Benoît Combemale[2], Pierre Kelsen[1] and Yves Le Traon[1]

[1] Université du Luxembourg
{Moussa.Amrani, Pierre.Kelsen, Yves.LeTraon}@uni.lu
[2] Université de Rennes, Institut de Recherche en Informatique et Systèmes Aléatoires
Benoit.Combemale@irisa.fr

Résumé

L'un des enjeux cruciaux dans l'Idm est la validité des transformations, puisque celles-ci sont exécutées sur de larges familles de modèles conformes. Cet article explore la question de la vérification formelle des propriétés des transformations de modèles suivant une approche tridimensionnelle : les transformations impliquées, leurs propriétés de correction, et les techniques de vérification pour les prouver. Cette contribution permet une meilleure compréhension de l'évolution, des tendances et de la pratique actuelle de ce domaine, et facilite l'identification des techniques et des outils pour conduire ces preuves.

L'Idm confère aux transformations une nature duale [4] : vue comme un *modèle de transformation*, l'accent se porte sur la computation qu'elle représente, dont l'expression repose sur un modèle de calcul particulier intégré à un Langage de Transformation (Lt) ; et vue comme une *transformation de modèles*, l'accent se déplace sur les artéfacts (modèles et métamodèles) manipulés. La nature computationelle requiert l'assurance de bonnes propriétés par le Lt sous-jacent (comme la terminaison et le déterminisme), indépendamment de la manipulation de modèles. D'un autre côté, la manipulation de modèles implique l'établissement de propriétés spécifiques relatives aux modèles impliqués, montrant par exemple qu'une transformation produit toujours des modèles de sortie conformes, ou que la sémantique des modèles d'entrée est préservée.

Cet article résume une contribution récente [2, 3] consistant en une revue de la littérature sur la Vérification Formelle des Transformations de Modèles, sur la base d'un cadre tridimensionnel intégrant différents éléments entrant dans ce domaine : *transformations* en Section 1, *propriétés* en Section 2 et *techniques de vérification* en Section 3). En plaçant chaque contribution dans cet espace tridimensionnel (cf. Figure 1), il devient possible de dégager l'évolution du domaine et ses principales tendances, en identifiant par exemple les sous-/sur-représentations en matière de propriétés analysées, ou de techniques utilisées. Outre l'intégration de la plupart des contributions du domaine, cet article introduit la notion nouvelle d'*intention* d'une transformation, permettant une analyse précise des relations entre les différentes dimension de notre espace tridimensionnel. Le lecteur est invité à se référer aux articles originaux pour trouver la liste des contributions : nous résumons ici les notions et arguments principaux de chaque dimension.

1 Dimension 1 : *Transformations*

La Figure 2 schématise l'idée d'une transformation de modèle. Un modèle d'entrée, conforme à un métamodèle source, est transformé en un modèle de sortie, lui-même conforme à un métamodèle cible, par l'exécution d'une spécification de transformation. La spécification est définie sur les métamodèles alors que l'exécution opère sur les modèles. Les métamodèles (ainsi que la spécification de transformation) sont aussi des modèles conformes à leur propre métamodèle : c'est la notion classique de méta-métamodèle, alors que pour les transformations, on parlera de

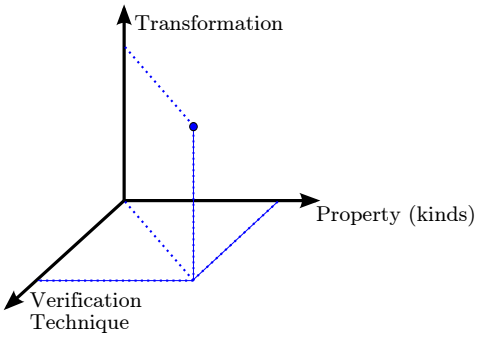


FIGURE 1 – Un espace tridimensionnel comme taxonomie de classification

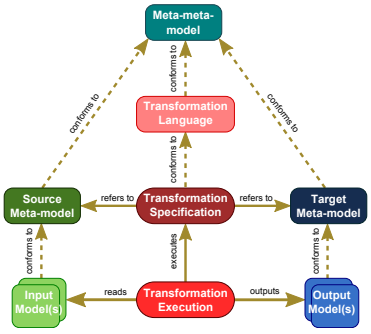


FIGURE 2 – Vision générale de la Transformation de Modèle

langage de transformation, qui permet une spécification consistante. Dans le cas général, une transformation peut manipuler plusieurs modèles en entrée et/ou en sortie.

La première dimension de notre espace tridimensionnel est étudiée suivant deux axes : la *définition* du concept de transformation ; et la *classification de ses caractéristiques*. Dans la littérature, les définitions suivent une progression qui témoigne du décentrage du rôle et de l'utilisation des transformations dans le cycle de développement logiciel. Nous proposons une nouvelle définition qui englobe toutes les précédentes : « *une transformation est la* ***manipulation automatisée*** *et conforme à une* ***spécification****, d'un* ***modèle d'entrée*** *en vue de la production d'un* ***modèle de sortie****, en accord avec une* ***intention spécifique*** » [2, 3]. Cette définition a trois avantages : elle distingue clairement entre spécification et exécution d'une transformation ; elle intègre pleinement la dualité propre aux transformations ; et elle explicite la notion d'intention guidant la spécification.

Deux contributions [5, 6] ont mis en lumière trois caractéristiques propres aux transformations. La première caractéristique concerne les LTs utilisés pour spécifier les transformations, classés en trois catégories, ou « styles », distincts : les langages basés sur un langage de programmation existant (*programming-based*) ; les langages *opérationalisés* (ou *metaprogrammés*) utilisant un langage d'action greffé sur un langage de métamodélisation comme MOF ; et les langages *déclaratifs* basés sur la réécriture de graphes. Ce qui importe pour la vérification formelle est l'existence d'une formalisation de leur sémantique : celle-ci détermine à la fois le type de propriétés qu'il devient nécessaire de prouver (e.g., la terminaison d'un système de réécriture), et le type de techniques de vérification disponibles pour ce faire. Ces deux premières catégories, proches de la programmation traditionnelle, bénéficient directement des avancées en matière de vérification formelle des langages impératifs ou orientés objet. La dernière catégorie trouve ses fondations dans la Théorie des Catégories, pour lesquelles des techniques sont actuellement développées.

La seconde caractéristique des LTs classifiée dans la littérature concerne la *méthode de construction* des LTs (leurs *features*). Un LT est construit sur la base d'unités de transformations, définissant des blocs élémentaires pour les transformations, qui sont ordonnancés de différentes manières pour réaliser une transformation particulière. L'ordonnancement peut se faire de manière implicite (comme dans les langages métaprogrammés, où l'utilisateur n'a aucun contrôle sur le flot d'instructions, défini *a priori*) ou explicite : dans ce cas, l'utilisateur peut bénéficier d'un contrôle partiel, ou dans le meilleur des cas, disposer d'un LMD permettant une définition très fine de l'ordonnancement.

La troisième et dernière caractéristique concerne la *forme des transformations* : comment une transformation manipule ses (méta-)modèles. Si le métamodèle source et cible sont identiques, on parle d'une transformation endogène, et endogène sinon. Si le métamodèle cible rajoute ou ôte des détails sémantiques, changeant ainsi son niveau d'abstraction, la transformation est dite horizontale, et verticale sinon. Si une transformation altère son modèle d'entrée au cours de son exécution, la transformation est destructive (ou *in-place*), et conservatrice (*out-place*) sinon. Enfin, l'arité d'une transformation renseigne sur le nombre de modèles en entrée et en sortie.

Cependant, cette classification est insuffisante pour l'extraction des bonnes propriétés d'une transformation. Par exemple, la classification correspondant à la définition de la sémantique d'un LMD produit deux résultats opposés suivant que la sémantique est définie de manière translationnelle (conservative, exogène et verticale, et d'arité 1 :1) ou opérationnelle (destructive, endogène, horizontale et de même arité), alors que l'on s'attend à devoir prouver des propriétés similaires dans les deux cas, puisque le but sous-jacent de la transformation est le même. Ceci n'est pas une surprise : ces classifications s'intéressent à l'aspect syntaxique, la forme des transformations, quand c'est l'aspect sémantique, le but dans lequel est spécifié une transformation qui importe pour déterminer quelles propriétés prouver pour assurer sa correction. Nous avons partiellement initié ce travail [1] en établissant un catalogue des intentions les plus répandues en IDM : pour plusieurs d'entre elles, nous caractérisons les propriétés intéressantes dont la preuve constitue un argument solide de validité.

2 Dimension 2 : *Propriétés*

Nous identifions deux grandes classes de propriétés reflétant à la nature duale des transformations. La première classe s'intéresse à l'aspect computationnel des transformations, et comprend les propriétés relatives aux LTs : ce sont les classiques propriétés de terminaison et de déterminisme, auxquelles s'ajoutent la bonne formation des transformations (le fait que la spécification se conforme à son LT). Prouver ce type de propriétés se fait souvent avec un compromis : soit le LT conserve son pouvoir d'expression généraliste (i.e. il est Turing-complet), ce qui rend ces propriétés indécidables, nécessitant de définir des conditions suffisantes pour les assurer au cas par cas ; soit l'on réduit le pouvoir d'expression du LT, ce qui permet de prouver ces propriétés par construction. La propriété de bonne formation pour les LTs textuels coïncide avec la correction syntaxique, bien connue en compilation ; elle reste cependant un enjeu pour les LTs visuels afin de faciliter la spécification de transformations.

La seconde classe cible les propriétés propres aux manipulations de modèles. Nous distinguons trois types de propriétés : celles portant sur les *modèles d'entrée et de sortie* (la conformance étant historiquement la première à avoir intéressé les chercheurs, mais aussi toutes les propriétés propres aux transformations n-aires comme les compositions) ; celles établissant une *relation syntaxique* entre éléments du modèle d'entrée et éléments du modèle de sortie (e.g., une classe est toujours transformée en une autre classe comportant un attribut particulier), donnant un moyen simplifié d'exprimer la conservation de structures propres aux modèles manipulés ; et enfin celles établissant une *relation sémantique* et correspondant aux propriétés classiques en vérification (bi-simulation/simulation de modèles, préservation d'invariants ou de propriétés temporelles, etc.), mais qui requièrent un accès explicite à la sémantique des modèles, ou un moyen de la calculer.

3 Dimension 3 : *Techniques de Vérification*

Le problème de la vérification formelle peut être posé comme un *problème de décision* : il s'agit d'établir mathématiquement que toutes les exécutions possibles d'une transformation ne traversent jamais des états interdits (ou erronés). Malheureusement, le calcul explicite de toutes les exécutions est impossible, car infini même pour des transformations simples : la

vérification formelle de transformations est indécidable, i.e. il n'est pas possible d'y répondre sans intervention humaine, en utilisant des ressources finies et sans incertitude. En pratique cependant, on cherche à trouver un équilibre entre les différents critères afin de proposer des analyses pratiques (i.e. dont le temps de calcul n'est pas rédhibitoire) et utilisables (i.e. qui donnent des informations permettant de corriger les problèmes), tout en gérant de manière optimisée le problème inhérant de l'explosion combinatoire.

Nous proposons de classifier les Techniques de Vérification (TVs) en trois types dépendant de deux critères : la spécificité de la technique à une transformation en particulier, et à un modèle d'entrée particulier. Le Type I concerne les TVs indépendantes des transformations et des entrées, et vise souvent les propriétés de la première classe (i.e. relatives aux LTs). Ces propriétés (comme la terminaison) sont souvent prouvées une seule fois, souvent mathématiquement, parce que la preuve outillée requière de traduire dans le langage d'entrée de l'outil de preuve (typiquement, un *theorem-prover*) la sémantique complète du LT analysé. Parfois, ces propriétés sont prouvées par un outil tiers : l'enjeu consiste alors à prouver la correction de la traduction de la sémantique du LT dans le langage de l'outil tiers (e.g., les réseaux de Pétri).

Le Type II recouvre les TVs dépendantes des transformations mais indépendantes des entrées. Les outils classiques sont le plus souvent utilisés : *model-checkers*, *theorem-provers* et *analyseurs statique*. Un large spectre de propriétés sont prouvées de cette manière : des propriétés d'invariants de modèles, de bisimulation entre modèles, de consistance de règles de réécriture, de préservation sémantique, etc. Les outils les plus populaires (SPIN, HOL/Isabelle, Méthode B, Coq, Java PathFinder, etc.) sont d'ailleurs largement réutilisés par les chercheurs dans l'IDM, bien que certains outils comme Groove ou DSLTrans disposent de leurs propres *model-checkers*.

Le Type III concerne les TVs dépendantes à la fois des transformations et des entrées : perçues comme étant plus faciles à mettre en œuvre et à utiliser que les TVs des types précédents, elles ne vérifient pas la correction de la transformation en tant que telle, mais plutôt qu'une exécution se déroule de manière adéquate pour une entrée donnée. Ces techniques sont basées sur des liens de traçabilité entre modèles d'entrée et de sortie, ou utilisent souvent des moteurs de résolution de contraintes intégrant une représentation explicite du modèle en cours d'analyse. Ces techniques ne peuvent donc pas être considérées comme de la vérification formelle.

En conclusion, la recherche résumée dans cet article propose d'aborder la question de la vérification formelle des transformations de modèles à partir d'une approche tridimensionnelle, fournissant une taxonomie intéressante pour classifier les contributions de la littérature. Nous introduisons la notion nouvelle d'*intention* de transformation qui permet de mieux comprendre l'activité de vérification et les relations entre chaque dimension taxonomique.

[1] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a Model Transformation Intent Catalog. In AMT, 2012.

[2] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In VOLT, 2012.

[3] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Survey of Formal Verification Techniques for Model Transformations : A Tridimensional Classification. JoT, 2014.

[4] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations ? Transformation Models ! In MoDELS, 2006.

[5] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. IBM *Systems J.*, **45**(3) :621–645, 2006.

[6] Tom Mens and Pieter Van Gorp. A Taxonomy Of Model Transformation. *Electronic Notes in Theoretical Computer Science (*ENTCS*)*, 152 :125–142, 2006.

Alignement de modèles métiers et applicatifs : une approche pragmatique par transformations de modèle

Jonathan Pepin[1,2], Pascal André[1], Christian Attiogbé[1] and Erwan Breton[2]

[1] AeLoS Team
LINA CNRS UMR 6241 - University of Nantes
{firstname.lastname}@univ-nantes.fr
[2] Mia-Software - Nantes
ebreton@sodifrance.fr

Résumé

La maintenance du système d'informations nécessite de le mettre en phase avec le pilotage de l'entreprise. En pratique, les points de vue métiers et applicatifs s'éloignent à mesure qu'évoluent la stratégie de l'entreprise et la maintenance des applications informatiques. Par conséquent, il est difficile pour les architectes logiciels de mesurer l'impact de l'évolution de celles-ci vis-à-vis des processus métiers ou des technologies utilisées.

Nous proposons une vision pragmatique de rapprochement des deux points de vue, via la rétro-ingénierie du code applicatif et des modèles métiers existants. L'idée est de transformer progressivement de part et d'autre les modèles afin de rapprocher les points de vue. Le résultat de ce processus de transformations permet de tisser les modèles obtenus dans le but d'obtenir un alignement ou de détecter des incohérences. L'approche présentée est mise en œuvre par des transformations de modèles et expérimentée sur un cas concret de taille significative.

Mots-clés : Systèmes d'information - Architecture d'entreprise - Rétro-ingénierie - Alignement - Ingénierie des modèles

1 Introduction

Compte-tenu de l'aspect prégnant de l'informatique dans la gestion des entreprises, l'évolution de leurs systèmes d'information (SI) est devenue un enjeu majeur. Les cycles de décisions (réorganisation, compétitivité, législation) sont courts et nécessitent une forte réactivité du SI. Le SI est aussi assujetti aux progrès technologiques continus tant matériels que logiciels. Les deux systèmes co-évoluent mais pas de manière synchrone. Le cycle de maintenance et de renouvellement du parc applicatif, souvent hétérogène, est plus long du fait de contraintes budgétaires ou organisationnelles. Maintenir le patrimoine applicatif en phase avec l'évolution des métiers (et des technologies) implique des coûts importants.

L'architecture d'entreprise [1] (ou urbanisation en version française) est un domaine qui contribue à proposer des méthodes pour maîtriser les systèmes de l'entreprise et leur évolution. Elle fait apparaître les visions des acteurs de l'entreprise et du système d'information. Ces visions distinctes doivent pouvoir se recouper, c'est l'objectif de l'alignement. L'alignement est un problème complexe du fait de l'éloignement "culturel" des acteurs et des rôles (management, métier, informatique) et les résultats pratiques sont insatisfaisants [5, 2]. En particulier l'alignement Business/IT entre la stratégie d'entreprise (*Business*) et l'informatique d'entreprise (*Information Technology* - IT) est au cœur des préoccupations de l'urbanisation [2, 25]. D'un côté le pilotage raisonne à un niveau stratégique avec des modèles informels et hétérogènes (documents, présentations, feuilles de calcul). De l'autre côté, l'informatique

1. *Enterprise architecture*

s'appuie sur un patrimoine complexe et hétérogène qu'il n'est pas facile de cartographier de manière concise et représentative. En pratique le rapprochement reste général voire approximatif. Notre motivation est de proposer une approche rationnelle de l'alignement pour en évaluer la qualité dans une perspective d'évolution du système d'information.

Les visions stratégie et opérationnelle (ou technique) ne peuvent être alignées directement, du fait du nombre de concepts techniques à intégrer et de la distance sémantique entre les niveaux. Le problème auquel nous faisons face est celui de deux mondes qui ne se comprennent pas : on ne parle pas de la même chose ou alors pas dans les mêmes termes. Plus précisément, il faut pouvoir se comprendre sur un langage commun. Le problème consiste alors à définir ce langage commun et s'exprimer (éventuellement par traduction) dans ce langage commun. Nous proposons une solution constructive pour déterminer le langage commun via des langages intermédiaires. L'idée sous-jacente est de rapprocher les points de vue et que chacun "fasse le pas vers l'autre" : le point de vue métier doit "s'exprimer" plus concrètement et le point de vue métier doit faire preuve d'abstraction pour masquer les nombreux détails. L'*ingénierie des modèles* (IDM) est une réponse technique pour traiter de langages et traductions. Une fois que des modèles "plus compatibles" sont mis en évidence, le second objectif est de déterminer une technique d'alignement outillée pour établir les correspondances entre les concepts dans ces modèles.

Nous proposons ici une approche opérationnelle du rapprochement et de l'alignement des visions métier et applicatives. Nous considérons que chaque point de vue est représenté par un ensemble de modèles. Pour rapprocher les points de vue, nous proposons un processus de rétro-ingénierie qui permet de gagner en abstraction du côté applicatif. Nous définissons ainsi les caractéristiques nécessaires dans les modèles intermédiaires, en nous inspirant des travaux sur l'urbanisation [26, 22, 15] et les architectures logicielles [20]. Ensuite des transformations de modèles permettent de définir le processus d'abstraction des applications en architectures à composants et services. Enfin, nous proposons une solution non-intrusive de tissage pour aligner les modèles métiers et applicatifs. L'approche se veut générique du point de vue des modèles et langages utilisés. Elle est outillée. Une expérimentation a été menée sur un cas concret d'une compagnie d'assurance avec d'un côté des modèles métier en provenance d'un référentiel d'entreprise et de l'autre un code source volumineux Java.

L'article est organisé comme suit. La section 2 situe le problème de l'alignement des points de vue d'un système d'information et présente notre vision du problème. Nous proposons dans la section 3 une approche pragmatique pour aligner une modélisation métier et les applications associées. Cette solution est mise en œuvre par un processus de transformations de modèles réalisant une rétro-ingénierie et un tissage non-intrusif. Nous les détaillons dans la section 4. La section 5 relate l'expérimentation de l'approche. Nous discutons des travaux connexes dans la section 6. La section 7 conclut l'article et trace des perspectives.

2 Aligner les points de vue d'un système d'information

Précisons les données de notre problème.

Le système d'information est un *ensemble organisé de ressources : matériel, logiciel, personnel, données, procédures… permettant d'acquérir, de traiter, de stocker des informations (sous forme de données, textes, images, sons, etc.) dans et entre des organisations* [21]. Les systèmes d'information sont un élément majeur dans le fonctionnement des entreprises car ils constituent le lien entre les différentes parties de l'entreprise, formant les niveaux stratégiques, décisionnels et opérationnels. L'architecture d'entreprise vise à modéliser le fonctionnement de l'entreprise pour en contrôler l'évolution [4]. L'urbanisme cible plus spécifiquement le système d'information et la démarche d'urbanisation vise à en améliorer le fonctionnement en le rationalisant dans un "cercle vertueux de transformation et d'amélioration continue" [26]. Pour Y. Caseau [22], l'urbanisation est en premier lieu une démarche technique

utilisant des principes simples (décomposition, découplage, intermédiation) pour répondre à des objectifs de flexibilité, de mutualisation, de maintenabilité, etc.

Le système d'information est perçu par ses acteurs selon deux aspects complémentaires : l'aspect gestion ou métier (*business*) et l'aspect informatique et technique (*Information Technology IT*). Cette dualité *business/IT* est la caractéristique majeure des systèmes d'information d'entreprise. Elle implique des visions différentes, des méthodes et techniques différentes qu'il faut rapprocher. De plus pour chaque aspect, en fonction des acteurs concernés, plusieurs points de vue se confrontent (des niveaux de préoccupation selon [26]). Par exemple dans l'entreprise on distingue entre autres la vue stratégie (pilotage, gouvernance), la vue métier (organisationnelle, fonctionnelle) et la vue opérationnelle. Du côté informatique, citons la vue applicative (le patrimoine des applications), la vue technique (les composants), la vue physique (infrastructure). Cette classification varie d'une approche à l'autre dans les nombreux cadres d'architecture d'entreprise (*frameworks*) proposés tels que Zachman, DODAF, TOGAF [8].

Les modèles d'architectures présentent souvent ces points de vue en couche d'abstraction où les niveaux bas sont les plus concrets et opérationnels. Par exemple le cadre de Longépé [26], schématisé dans la figure 1 propose cinq couches.

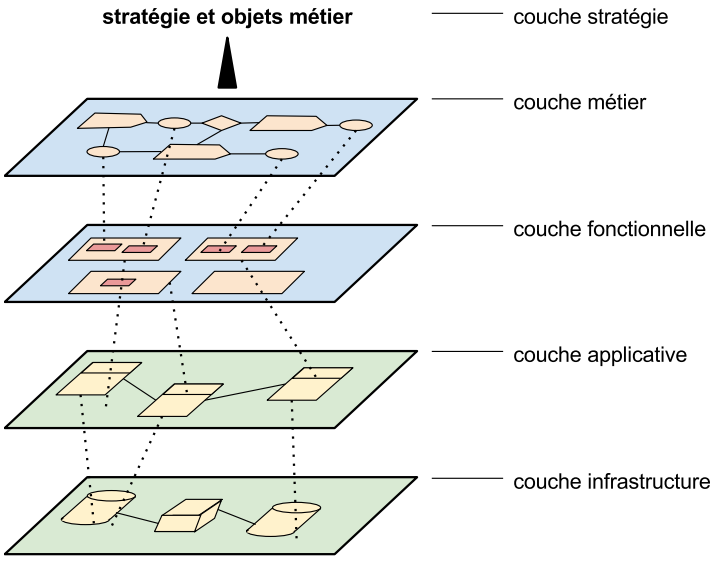


FIGURE 1 – Couches du système d'information et alignement

Stratégie : La couche stratégie donne le positionnement de l'entreprise (marché, organisation, produits) et ses objectifs métier.

Métier : La couche métier prend en charge l'organisation à travers des processus et des activités.

Fonctionnelle : La couche fonctionnelle organise hiérarchiquement les fonctionnalités des activités de la couche métier par exemple en termes de blocs fonctionnels (ex : zone, îlot, quartier).

Applicative : La couche applicative modélise les applications, composants logiciels et services implantant des fonctionnalités, ainsi que leur relations.

Infrastructure : La couche infrastructure représente toutes les ressources nécessaires au stockage, à la communication et à l'exécution des unités logicielles (bases de données, réseau, intergiciels).

L'alignement *business/IT*, instrument d'efficacité, n'est qu'une étape dans la démarche d'architecture d'entreprise et d'urbanisation [25, 2]. En pratique le but est surtout de détecter des incohérences

d'alignement. A travers la vision usuelle de la figure 1, l'alignement est interprété comme une ligne de traçabilité traversant les couches. L'alignement définit alors les concepts à relier entre les couches. Mais le problème est plus complexe [2]. Au sens large, l'alignement couvre différents aspects et d'une méthode à l'autre, on privilégie l'aspect gestion ou bien l'aspect informatique. La méthode GRAAL par exemple distingue trois dimensions : sociale (l'entreprise), physique (infrastructure) et symbolique (logiciel) [15]. L'alignement proposé se fait alors deux à deux. Nous réduisons ici le problème à l'alignement social-symbolique en considérant les deux hypothèses suivantes :

– L'alignement entre l'organisation et l'infrastructure perd de l'importance compte-tenu des architectures réparties et du *cloud*. Il n'est plus nécessaire de rapprocher les personnes des machines.
– Le déploiement est souvent décrit finement et représente explicitement l'alignement entre les programmes (bas niveau des applications) et l'infrastructure.

L'alignement idéal relie la stratégie et les programmes, qui représentent les applications à bas niveau (*cf.* figure 2). Mais le niveau stratégique n'est pas exprimé par des modèles exploitables car informels ; le code lui, contient trop de détails. Le rapprochement des points de vue se fait en concrétisant la stratégie par des modèles de processus métier et en masquant les détails d'implantation (processus d'abstraction ou de rétro-ingénierie) jusqu'à un niveau acceptable pour le "langage commun". Le langage commun est alors défini comme un tissage entre deux langages, une correspondance entre concepts.

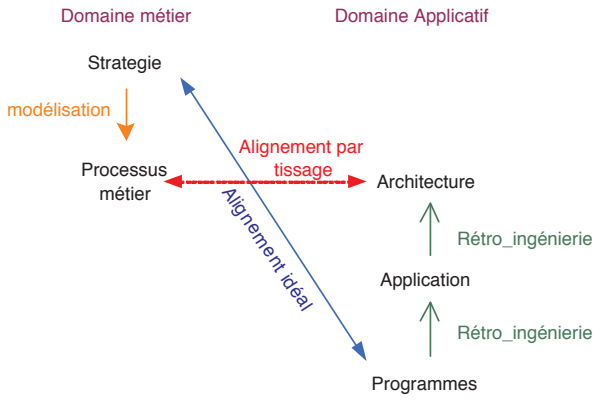


FIGURE 2 – Rapprochement et alignement

La nature des liens d'alignement varie selon les travaux. Dans les cadres TOGAF [13] ou ArchiMate [24], le lien est défini entre les services métiers et les services applicatifs. Dans d'autres travaux [10, 23] le lien est établi entre activités métiers (ou tâches) au sens BPM [3] et des fonctionnalités d'applications. Nous adoptons le point de vue suivant : les propositions d'assistance à l'alignement doivent être génériques et adaptables à différents cadres et différentes notations. L'alignement devient plus simple si on dispose d'une notion de service des deux côtés, mais elle ne s'applique pas naturellement sur le patrimoine d'applications ancien. La section suivante présente le processus de transformation qui supporte notre méthode d'alignement.

2. Un alignement parfait n'est jamais atteint [17] car il y a trop de facteurs de changements tant technologiques que légaux ou encore concurrentiels.
3. Business Process Management

3 Un processus de transformation pour rapprocher les modèles

Nous décrivons les étapes du processus esquissé dans la section 2.

3.1 Abstraction du code

Afin de réaliser un modèle applicatif à partir du code source, trois étapes sont nécessaires :

E1 La découverte du code source afin de réaliser l'analyse du code et de détecter les différentes structures du langage : syntaxe, instruction, type, variable, déclaration, etc.

E2 La transformation vers un modèle pivot pour s'abstraire des spécificités du langage de programmation.

E3 La transformation vers le méta-modèle applicatif comportant les concepts retenus.

Ces étapes peuvent se référer à l'ingénierie dirigée par les modèles (IDM) : la première étape permet d'obtenir le Platfom Specific Model (PSM), il s'agit du modèle correspondant à la plateforme du code source ; la seconde étape s'abstrait de l'architecture logicielle et constitue le Platform Independent Mode (PIM). La définition de ces modèles est habituellement utilisée dans un contexte MDA "*top-down*" ; on part du Computation Independent Model (CIM) qui est un modèle métier indépendant de l'informatisation vers le PSM dans une logique de génération de code. Alors que notre démarche d'abstraction est inverse, on part du PSM pour un tissage avec le CIM. La figure 3 illustre les deux dernières transformations.

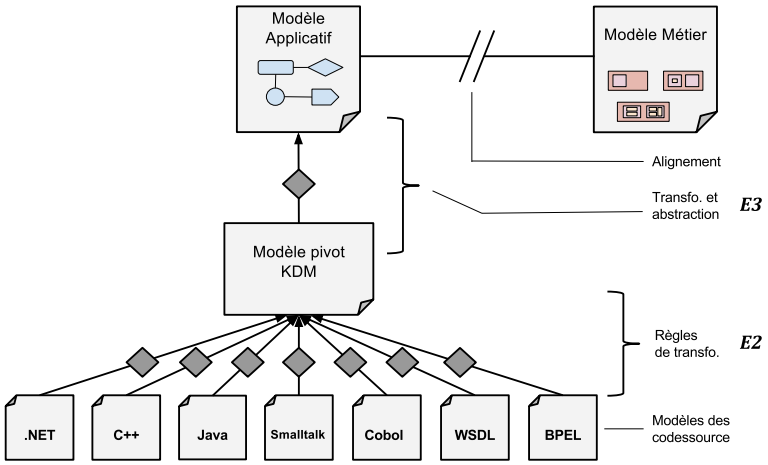


FIGURE 3 – Les étapes de transformation

L'étape *E1* est dépendante du langage et de l'architecture employée, or les systèmes d'information sont hétérogènes et composés de technologies diverses évoluant rapidement dans le temps. Ainsi, il est nécessaire de disposer d'un analyseur syntaxique (*parser*) différent pour chaque technologie rencontrée, nous utilisons un analyseur Java dans notre expérimentation en section 5. Les langages les plus communément rencontrés en entreprise sont Java, C++, .Net, Smalltalk, Cobol…

C'est également la raison pour laquelle la seconde étape (*E2*) de notre méthode est une transformation vers un modèle intermédiaire. Ce dernier est défini par un méta-modèle unique capable de recevoir les concepts des différentes technologies issues de la première étape *E1*. Pour cela nous avons choisi le méta-modèle Knowledge Discovery Metamodel (KDM) qui est un standard spécifié par l'OMG utilisé

dans de nombreux outils informatiques. KDM inclut de nombreuses couches pour stocker les différents aspects des langages de programmation communs [19]. Nous n'utilisons qu'une partie du méta-modèle KDM, notamment le paquetage `Code`.

Enfin, l'étape *E3* consiste en l'abstraction la plus forte pour se séparer de la logique de code programme vers une logique applicative. Ainsi, nous avons défini un méta-modèle, représenté dans la partie en bas à droite sur la figure 4, à partir de la proposition issue de travaux antérieurs [1] mais aussi d'ArchiMate [24] définissant la notion d'application constituée de composants, services, interfaces, fonctions et objets de données. La transformation doit ainsi détecter les aspects contenus dans le modèle KDM, bien que KDM soit un modèle pivot des spécificités peuvent subsister. En effet, la rétro-ingénierie d'un code source complet ne se préoccupe pas de l'utilité des éléments contenus dans le code source, s'il s'agit d'éléments liés à l'aspect métier, utilitaire ou simplement technique. Ainsi, c'est à cette étape de transformation que l'on va tamiser les différents éléments de l'application et ne capter que les concepts utiles pour peupler le modèle applicatif.

3.2 Déclinaison de la stratégie en modèle métier

Cette étape correspond à la phase descendante entre les points de vue stratégie et métier sur la figure 2. Les modèles métiers sont issus de la modélisation du fonctionnement de l'entreprise obtenue par la transcription des connaissances stratégiques et organisationnelles. Il s'agit ainsi d'une phase manuelle de documentation faisant appel à des notions de gestion des entreprises.

Il existe deux grandes représentations métier : le plan d'occupation des sols (POS) ou encore modèle fonctionnelle ou encore lotissement, et la modélisation processus (BPM en anglais). Les deux représentation sont complémentaires, néanmoins les entreprises en fonction de leur niveau d'expertise de leur SI n'utilise qu'une des deux méthodes, et le POS étant la plus simpliste. Le cas de notre étude exposé dans la section 5 utilise uniquement la représentation BPM.

BPM identifie les processus composés d'activités organisées de façon chronologique et représentées par des transitions [18]. Chaque processus peut être décomposé en sous-processus. En fonction du niveau de granularité, les activités peuvent-être décomposées en tâches. Le SI comporte de nombreux processus contenus dans plusieurs modèles. Il existe de nombreux langages de notation BPM : Merise, UML, BPMN... Les concepts essentiels peuvent avoir un nommage différent, mais sont toujours présents : rôle/acteur, processus, activité, tâche, condition, transition, événement.

POS est une structuration du SI en bloc fonctionnels communicants. Elle est la représentation issue de l'urbanisme de la cité[16]. Le découpage présente 3 niveaux de granularité : zone, quartier et îlot. La bonne pratique définit les zones selon les types suivants : échange, opération, données, référentiel, pilotage, ressource. Les quartiers et îlots sont ensuite déterminés par la connaissance de l'entreprise, les îlots étant un niveau de détail supplémentaire spécialisant les quartiers.

Les modélisations processus métier et fonctionnelle peuvent également être représentées dans un référentiel d'entreprise plus vaste où sont également cartographiés des éléments des couches applicative et infrastructure tels que application, base de données, bus, message, etc. Nous verrons dans la section 5 comment les processus métier sont modélisés dans un de ces référentiels.

3.3 Tissage entre les méta-modèles d'architecture d'entreprise

Dans un souci d'adaptation aux démarches d'architecture d'entreprise existantes, nous avons déterminé trois méta-modèles génériques en nous inspirant de divers travaux et standards, ainsi que des travaux de la DISIC [4] et du document de référence [10]. Ainsi nous avons déterminé trois méta-modèles :

4. Direction interministérielle des systèmes d'information et de communication

application (App) comme évoqué dans la section 3.1, fonctionnel (POS) et processus (BPM) permettant la modélisation du métier de l'entreprise comme expliqué dans la section 3.2. Ce trio de méta-modèles est le support "langage" pour l'alignement du SI. Pour réaliser cet alignement, c'est-à-dire lier les fonctionnalités et activités de l'organisation à leur réalisation dans les applications exécutant ces opérations ; des liens sont définis entre les concepts des différents méta-modèles. Ces liens inter-modèles seront créés selon deux axes de description : les traitements ou les données.

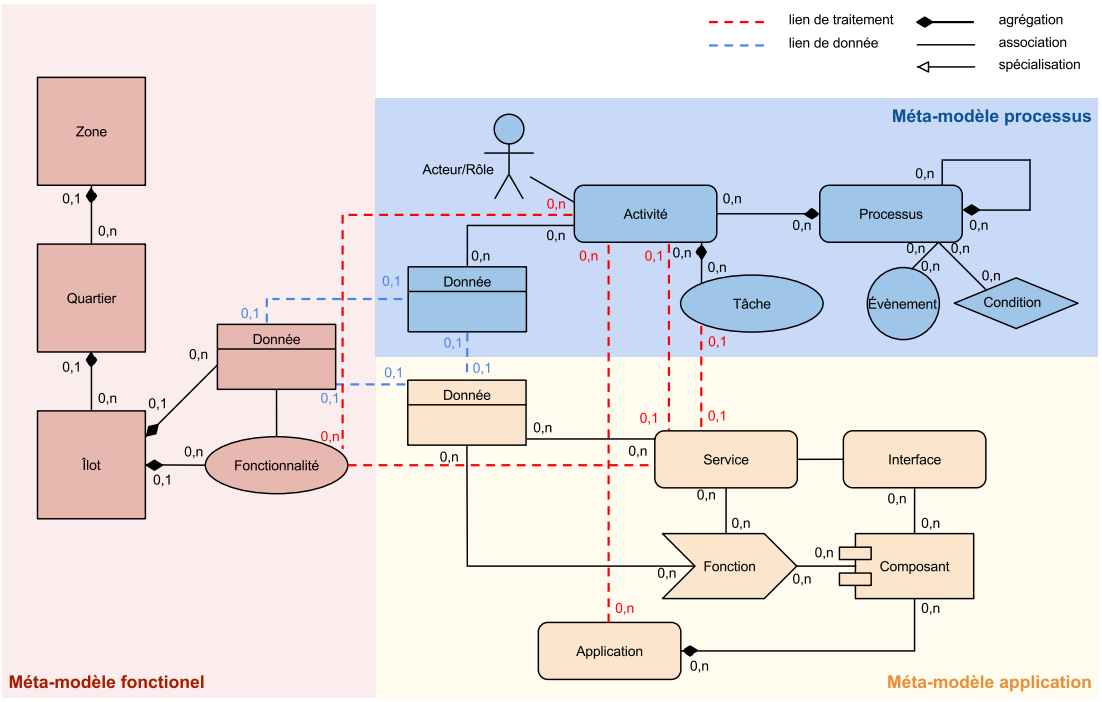


FIGURE 4 – Les méta-modèles d'architecture d'entreprise et leur liens par les traitements et les données

Liens par les traitements : Les fonctionnalités du POS réalisent des activités du BPM ; et sont implémentées par les *Services* dans *App*. En fonction du niveau de granularité de la modélisation BPM : si les tâches sont détaillées elles sont implémentées par les services ; si uniquement les activités sont représentées, ces dernières sont implémentées par les services. De même, de façon macroscopique, les activités peuvent être implémentées directement par une application.

Liens par les données : Chaque méta-modèle possède une classe *Donnée*, un lien existe entre les classes *Donnée* équivalentes.

Les méta-modèles présentés dans la figure 4 sont une représentation simplifiée des méta-modèles que nous avons conçus. La réalité présente toujours des cas complexes. La mise en correspondance dépend du niveau de granularité choisi, de la portée de l'alignement (SI total ou partiel, une ou plusieurs applications), du type des éléments (structure ou comportement), des objectifs de l'alignement…

Cette définition des liens permet d'établir des règles de tissage entre les différents modèles.

4 Mise en œuvre du processus de rapprochement

Notre démarche d'urbanisation est accompagnée d'une solution outillée afin d'automatiser chaque étape du processus de rétro-ingénierie et d'assister le concepteur durant l'étape de tissage.

4.1 Les outils de transformation

Les étapes de transformation employant l'ingénierie des modèles, nous avons choisi d'utiliser la technologie Eclipse EMF qui permet de définir ses propres méta-modèles et de développer des extensions (*plugins*) les manipulant. Nous nous sommes également basé sur un projet *open source* de modernisation des systèmes d'information nommé Eclipse Modisco, ce projet initié par AtlanMod [5] est aujourd'hui porté par l'entreprise Mia-Software [6]. Modisco inclut un mécanisme de découverte pour réaliser l'analyse d'un code source et la transformation de modèle EMF. Modisco inclut également des méta-modèles technologiques pour Java, KDM et SMM.

Vers un modèle du source Java Pour nos expérimentations, relatées en section 5, nous avons choisi le langage Java comme support du code source des programmes. Pour l'étape de rétro-ingénierie *E1*, nous utilisons l'analyseur Java Discovery intégré à Modisco. Il s'appuie sur l'extension JDT (Java Development Toolkit) d'Eclipse pour créer un graphe syntaxique des éléments du langage contenu dans les fichiers de code source. Cette découverte produit un modèle du code Java en sortie.

Vers un modèle KDM La seconde étape *E2* est le passage du modèle de la plateforme source au modèle intermédiaire et indépendant des langages de programmation, ici, le passage de Java vers KDM. Des premiers tests ont été réalisés avec la transformation Java vers KDM écrite en ATL [7] incorporé dans Modisco, mais la transformation s'est révélée inexploitable avec des modèles plus volumineux (> 100 Mo), car elle ne se terminait pas malgré plusieurs dizaines d'heures de traitement. Nous avons tenté de détecter l'erreur et de mettre à jour le moteur d'exécution, mais faute de maitriser l'outil ATL, nous avons décidé d'écrire notre propre transformation avec le logiciel Mia-Transformation. Les transformations de Java vers KDM ont été réécrites en nous inspirant, évidemment, des règles en ATL dans Modisco avec quelques modifications lorsque des améliorations utiles ont été détectées.

Vers un modèle applicatif Pour la troisième étape *E3* de transformation et d'abstraction du modèle KDM vers le modèle applicatif, nous avons codé le méta-modèle Applicatif dans un fichier `.ecore` à l'aide d'EMF, dans lequel nous avons ajouté des règles validation écrites en OCL (listing 1).

Puis nous avons développé la transformation avec Mia-Transformation. Les règles de transformations sont de deux types. Un premier jeu structurel correspond aux langages de programmation orienté objet : classes, interfaces, méthodes, paquetages... Un second jeu sémantique détecte les composants et les objets de données du programme. Cette analyse est la plus complexe. Variable d'un jeu de test à l'autre, elle doit être redéfinie au cas par cas.

Listing 1 – Exemples de règles OCL

```
context DataObject
  inv: self.accessedByFunction.realizes.accesses -> exists(s | s = self)
context Service
  inv: self.realizedBy.assignedFrom.composedOf.assignedTo ->
                                                    exists(s | s = self)
```

5. http://www.emn.fr/z-info/atlanmod/
6. http://www.mia-software.com, Deux auteurs de ce papier font partie de cette entreprise.
7. ATL est un langage et un moteur de transformation *open source* initié par Atlanmod

Après ces trois étapes de transformation, nous obtenons le modèle applicatif stocké dans un fichier XMI, la norme d'échange de modèles de l'OMG.

4.2 Le tissage des modèles

L'utilisation de modèles d'architecture d'entreprise spécifiques à chaque point de vue a pour avantage d'apporter une modularité, néanmoins elle a pour principal inconvénient de nécessiter un moyen de recomposition de modèle s'appuyant sur l'IDM. Ce domaine de recherche est actif et diverses techniques sont proposées [6]. Nous avons choisi la technologie de tissage qui permet de créer un modèle indépendant référençant plusieurs modèles issus de méta-modèles différents et ainsi de créer des liens entre les éléments sources et cibles des différents modèles [12]. Cette technologie est non intrusive puisque les modèles référencés ne sont pas modifiés par le tissage. Des outils ont été développés pour Eclipse EMF, proposant un méta-modèle de tissage et un éditeur graphique notamment : AMW[9] [8] et Virtual EMF[3]. AMW inclut un mécanisme de transformation ATL pour réaliser des tissages de façon automatique. Virtual EMF est plus simple, il propose une interface permettant d'éditer deux modèles pour créer des liens via glisser-déposer. Néanmoins, ces deux éditeurs ne sont plus maintenus depuis plusieurs années et n'ont pas été rendus compatibles avec les nouvelles versions d'Eclipse [9]. Ainsi, nous avons du créer notre propre méta-modèle pour mettre en correspondance différents méta-modèles de nature variable. L'éditeur de lien est inspiré des travaux étudiés en y incluant des améliorations et des fonctionnalités propres.

Le tissage entre différents méta-modèles s'obtient par transformation quand la matière en entrée existe et est suffisamment complète, ou de façon manuelle s'il s'agit de retranscrire de nouvelles informations. Ici les méta-modèles métier et applicatif s'obtiennent à partir de connaissance sur l'organisation de l'entreprise. Cette connaissance est détenue collaborativement par des acteurs informaticiens ou issus du management. Le méta-modèle est exploité selon deux scénarios :

1. Si l'information qui permet d'avoir le lien entre la couche métier et applicative existe, alors il est possible de l'exploiter par rétro-ingénierie et plus précisément par une transformation qui peuple le modèle de tissage.
2. Si l'information n'existe pas, l'assistant permet de créer le tissage avec une approche pratique.

L'éditeur présente une interface en deux parties (*cf.* figure 5). A gauche l'arborescence contient les liens tissés. A droite, les différents modèles chargés et regroupés par méta-modèles. Ainsi, il est possible de tisser autant de modèles et de méta-modèles différents que souhaité. Pour aider à la réalisation du tissage, l'interface dispose d'un champ de recherche qui permet de trouver rapidement les éléments des modèles par leur nom de concept. C'est une fonctionnalité essentielle pour l'architecture d'entreprise, car les volumes des modèles peuvent devenir très importants (*cf.* section 5).

La technologie de tissage nous posait des difficultés car les liens créés n'ont pas de signification, ils peuvent contenir des éléments de n'importe quel méta-modèle. Or dans notre démarche nous avons défini des liens entre les méta-modèles applicatif et métier (*cf.* section 3.3), ainsi nous voulons réaliser des restrictions pour ne pouvoir tisser que des concepts de type particulier : par exemple, *Service* depuis le modèle applicatif et *Activité* depuis le modèle métier. Pour palier à ce problème, nous avons créé un mécanisme de contrainte en réalisant un tissage au niveau méta-modèle. En effet, il suffit d'ouvrir ce même éditeur avec les méta-modèles à tisser et de réaliser le lien avec les types des concepts souhaités. En reprenant notre exemple, il faut charger les méta-modèles applicatif et métier, puis créer un lien appelé *ServiceVersActivité* contenant le type *Service* et le type *Activité*. Lors du tissage au niveau modèle, le nouveau type de lien sera disponible et limitera le tissage à ces types d'éléments. Le tissage étant réalisé de façon manuelle à l'aide de cet éditeur, cette solution de contrainte permet d'éviter les erreurs.

8. Atlas Model Weaver
9. Testé sur Eclipse Juno 4.2

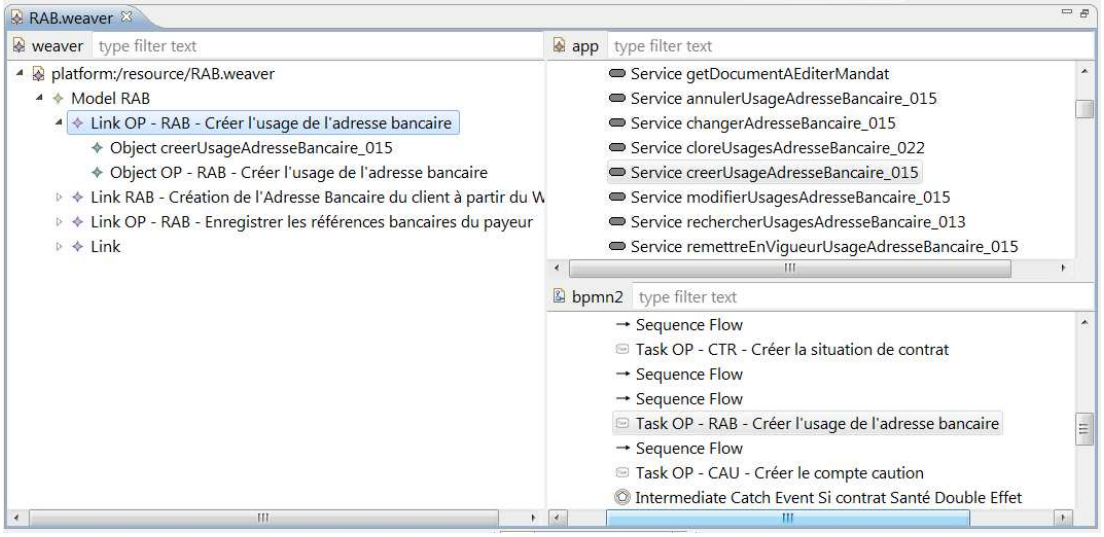


FIGURE 5 – Capture d'écran de notre plugin de tissage

Dans la section suivante, nous présentons une expérimentation à partir d'un cas d'étude proposé par une société d'assurance mutuelle française. L'expérimentation a été réalisée à l'aide de nos outils.

5 Expérimentation

L'expérimentation a pour but d'appliquer notre démarche afin de tester l'automatisation des différentes transformations et la couverture des concepts mis en relation à l'aide du tissage.

Le cas d'étude est issu d'une mutuelle d'assurance française importante et concerne une application gérant les moyens de paiement des contrats et dossiers client. Le logiciel est en production depuis plusieurs années sur le SI de la société d'assurance mutuelle. Il est disponible sur tous les postes des agents en contact avec la clientèle. Le code du cas d'étude est composé *i)* d'un code source complet écrit en Java avec 33 400 classes, environ 3 400 000 lignes de codes ; et *ii)* d'un référentiel d'entreprise sous la forme d'un portail HTML exporté depuis le logiciel MEGA Enterprise Architecture [10]. Le référentiel d'entreprise contient 360 diagrammes de processus métier couvrant la totalité du SI. Le cas d'étude est significatif pour notre problématique de modernisation des systèmes d'informations, le volume de l'application est conséquent et représente un véritable défi pour traiter ce volume d'informations. L'expérimentation se déroule en deux parties : la première consiste à enchaîner les transformations outillées définies dans la section précédente ; la deuxième consiste à réaliser le tissage entre le modèle applicatif obtenu à l'étape précédente et le modèle métier provenant du référentiel d'entreprise.

5.1 La chaîne de transformation automatisée

La première étape de la chaîne de transformation est la découverte avec l'outil Eclipse (*cf.* section 4.1). Cette étape de rétro-ingénierie malgré la taille du code source est très rapide : environ 10 minutes. Alors que le fichier XMI généré en sortie a une taille de 1,4 Go.

10. www.mega.com/fr/solution/business-architecture

La deuxième étape est la transformation du modèle Java vers le modèle KDM avec l'outil Mia-Transformation (*cf.* section 4.1). Les premières tentatives ont échoué, faute de mémoire vive suffisante. Nous avons dû utiliser une machine pourvu de 10 Go de mémoire vive et d'un processeur avec 4 cœurs. Après environ 2 heures de traitement nous obtenons avec succès un modèle KDM, sa taille est de 780 Mo. Comparé au modèle Java, la différence de taille est due au concept de langage qui est moins détaillé dans le paquetage code du métamodèle KDM que dans le métamodèle Java, ainsi toutes les spécificités du langage Java ne sont pas traduites en détail. Néanmoins, les concepts essentiels (classes, interfaces, méthodes, routines, variables, commentaires, etc) sont présents et sont correctement retranscrits.

La dernière étape est la transformation du modèle KDM vers le modèle applicatif (*cf.* section 4.1). Les règles de transformation sémantique ont été adaptées au cas d'étude qui présente une particularité singulière : les concepts d'architecture sont intégrés dans le nom des classes et des interfaces Java via un préfixe. Ainsi, la reconnaissance des services, des interfaces de services et des objets de données est relativement aisée. En sortie de la transformation nous obtenons un fichier XMI de 2 Mo ; cette taille est bien éloignée des modèles précédents. C'est en cela que l'abstraction est intéressante, nous obtenons un nouveau point de vue de l'application plus pratique à lire et à manipuler. Enfin, nous avons validé le modèle obtenu avec les règles OCL incorporées dans le méta-modèle (Listing 1). Les règles sont vérifiées avec succès, aucune erreur n'est remontée. La transformation est non seulement correcte, mais le code source est également complet pour peupler le modèle applicatif.

5.2 Tissage avec le modèle métier

Le référentiel d'entreprise MEGA fourni par la société d'assurance mutuelle n'est pas exploitable directement car il s'agit d'un export HTML statique et les diagrammes sont sous forme d'images. Ainsi, il n'y a pas de modèle XMI à charger ou de rétro-ingénierie possible. Nous avons alors décidé de réaliser une traduction manuelle d'une partie des diagrammes vers le langage de modélisation métier BPMN2. Nous avons choisi BPMN 2.0 [11] car il s'agit d'une norme de l'OMG qui est utilisée dans de nombreux projets. Pour modéliser nos processus métier, nous avons employé le *plugin* Eclipse BPMN2 modeler. Cet outil de modélisation a pour avantage d'avoir un méta-modèle interne qui suit rigoureusement les spécifications de l'OMG.

Concept MEGA	**Concept BPMN2**	**Description**
Acteur	Pool ; Couloir	Élément actif chargé d'une ou plusieurs activités dans le processus
Message	Lien de séquence	Lien orienté entre deux activités
Processus	Processus	Ensemble d'activités liées ayant même finalité
Procédure	Sous-processus	Ensemble de tâches décrivant une vue d'ensemble d'une activité.
Opération	Tâche	Plus petit élément de décomposition d'une activité

TABLE 1 – Règles de traductions des concepts processus métier

Nous avons réduit le périmètre de traduction à une partie de l'activité de la société d'assurance mutuelle appelée *Adresse Bancaire*, car le référentiel d'entreprise MEGA contient 360 diagrammes, le travail complet de traduction serait beaucoup trop long et dénué d'intérêt au vu de nos objectifs de validation de l'approche.

Nous avons chargé les modèles BPMN2 et les modèles applicatifs dans notre éditeur de tissage (*cf.* section 4.2). Nous avons établi des liens entre les tâches provenant du modèle métier et les services provenant du modèle applicatif. Pour cela nous nous sommes servis de la fonction de recherche par nom pour trouver les correspondances. Le travail a été facilité par la présence d'une nomenclature des

11. Business Process Model and Notation

concepts qui se retrouve à différents niveaux et par là même montre de bonnes pratiques de codage, même si des exceptions subsistent.

6 Travaux connexes

La problématique de l'alignement du système d'information (SI) couvre plusieurs aspects, certains d'ordre méthodologique et d'autres d'ordre pratique. Nous avons choisi de traiter trois d'entre eux dans cette section. La première concerne les points de vue à retenir pour l'alignement. La seconde considère l'apport de l'IDM dans le processus d'alignement, La troisième discute des solutions alternatives pour le tissage.

6.1 La méthode d'alignement GRAAL

Une méthode d'alignement de référence est la méthode GRAAL [12] [15] dont la première version a été publiée en 1996 et éprouvée depuis. GRAAL fait penser à la quête du Graal pour les entreprises, l'approche qui réconcilie les points de vue sur le système d'information. La méthode est composée de 4 dimensions, les 3 premières correspondent à des points de vue différents qui permettent l'observation d'un système d'information : *les aspects du système*, *le niveau d'agrégation du système*, et *le cycle de vie du système* ; la quatrième dimension *le niveau de raffinement* concerne le niveau de détail de la description du système.

La première dimension prend en compte des propriétés extérieures observables du système et les classe en deux catégories : les services et les qualités offerts par le système et attendus par l'utilisateur [14]. La deuxième dimension hiérarchise le système en 3 types : le monde physique (ordinateurs, réseaux…), le monde social (processus métier, normes, lois, finance…) et le monde symbolique (logiciels et documents). La notion de temps est abordée par la troisième dimension qui s'occupe des différentes phases dans la vie d'un système d'information : conception, versions successives, fréquences de mise à jour, maintenance, utilisation…

C'est la deuxième dimension qui nous importe le plus ; c'est elle qui va permettre la réalisation de l'alignement. La méthode GRAAL indique que chaque monde (physique, social, symbolique) doit-être aligné aux deux autres. Plus précisément les deux points de vue qui nous intéressent dans le présent article sont les points de vue métier et applicatif. GRAAL énonce le principe d'alignement suivant : *Pour aligner les logiciels (le symbolique) aux processus métier (le social), nous devons aligner les services offerts par le logiciel aux services offerts par les processus* [15].

Retenons donc qu'aligner c'est toujours se ramener à comparer des éléments comparables.

6.2 L'IDM pour l'architecture d'entreprise

L'utilisation de l'ingénierie des modèles appliquée à l'architecture d'entreprise et la transformation de modèles à base de technologie Eclipse EMF n'est pas une approche émergente. D. Castro et al.[7] décrivent une approche "*top-down*" qui vise à générer une architecture web orientée service à partir de modèles métier. Ils détaillent de façon similaire à notre approche les règles de transformation et les concepts qui relient les différentes couches.

Cependant, nous ne partageons pas le même but. Notre point de départ est le patrimoine applicatif existant, quel qu'il soit. Notre méthode doit répondre à l'homogénéité des architectures techniques existantes dans un SI et des différentes représentations métiers possibles de ce SI. La démarche D. Castro est uniquement hétérogène à une architecture orientée service, or une structure homogène est plus facilement traçable entre les couches.

12. Guidelines Regarding Architecture Alignment

Notre méthode ne se prétend pas être universelle avec tous les langages de programmation et toutes les notations de modélisation métiers, une adaptation plus ou moins importante sera nécessaire pour chaque cas rencontré.

6.3 Les différentes techniques de composition des modèles

Plusieurs approches sont envisageables pour composer des modèles entre eux [6]. Nous avons utilisé le tissage car il a l'avantage d'être non intrusif. Néanmoins il existe d'autres technologies.

L'extension de méta-modèles Une méthode possible est d'étendre chaque méta-modèle à mettre en relation pour inclure les concepts opposés à relier. En effet, il est tout à fait possible de réaliser une référence vers le concept d'un autre méta-modèle, notamment un lien d'association, cette méthode à été testée avec EMF. Dans notre cas d'alignement des modèles métiers et applicatif, cela consisterait à modéliser une association dans le méta-modèle métier vers le concept de *Service applicatif* provenant du méta-modèle applicatif, et dans le méta-modèle applicatif de modéliser une association vers le concept de *Tâche* provenant du méta-modèle métier. Néanmoins chaque cas d'étude n'utilisera pas systématiquement le même méta-modèle métier, ainsi il faudra réaliser cette modification pour chaque cas. De plus si un méta-modèle normé est utilisé, sa modification le rend non conforme : par exemple, BPMN.

Définition d'un unique méta-modèle Une autre méthode est de créer un méta-modèle qui inclut à la fois les concepts métier et applicatif, ainsi que les liens entre les concepts des deux couches initiales. Cette méthode a comme désavantage de devoir créer des "super méta-modèles" pour chaque nouveau cas rencontré.

Utilisation de *Facet* Une autre technologie existe pour réaliser une mise en relation non intrusive par l'utilisation du projet *open-source* : Eclipse EMF Facet [13]. Ce projet permet d'étendre un modèle EMF par le mécanisme de facette qui ajoute virtuellement des attributs, des références et des opérations à un concept préexistant d'un méta-modèle. La facette fonctionne grâce à l'implantation de requêtes qui peuvent être codées en Java, en OCL, ou en Javascript. L'inconvénient d'EMF Facet est que les requêtes doivent être exécutées au chargement du modèle ce qui peut avoir un impact négatif sur les performances.

L'étude de différentes méthodes, nous a permis de choisir la méthode la plus souple : le tissage. Elle permet de ne pas redéfinir de nouveau méta-modèle pour chaque cas d'étude. Le méta-modèle de tissage étant lui générique et défini une fois pour toute ; il peut contenir des liens de concepts en provenance de n'importe quel méta-modèle. Nous avons choisi un mécanisme de contraintes défini dans le métamodèle pour maintenir les liens plutôt qu'un croisement des modèles [11] ou une séparation des préoccupations [12].

7 Conclusion et perspectives

Nous avons proposé une approche pragmatique au problème d'alignement des points de vue métier et applicatifs s'insérant dans la démarche d'urbanisation des architectures d'entreprise. Notre solution est basée sur une proposition de modèles intermédiaires génériques, un rapprochement des points de vue et un alignement par tissage de concepts comparables. Le rapprochement est rendu possible par une abstraction progressive du code en architecture applicative à base de composants et services. Concrètement cette abstraction est une rétro-ingénierie basée sur un processus de trois transformations de modèles. L'alignement effectif est implanté à l'aide d'un méta-modèle de tissage et d'un éditeur incluant un mécanisme de contrainte des liens entre méta-modèles.

13. http ://www.eclipse.org/modeling/emft/facet/

Notre approche est instrumentée dans le cadre d'Eclipse EMF et a été expérimenté sur un cas réel d'entreprise, le SI d'une société d'assurance mutuelle. Ce cas d'étude nous a permis d'éprouver notre approche de par son volume. L'application a un source code Java de taille importante, ce qui a été un défi pour que les modèles obtenus par rétro-ingénierie puissent être chargés par nos outils. De bonnes pratiques de codage, notamment pour la nomenclature des noms ont permis de conserver une certaine traçabilité entre les concepts métiers et ceux de l'application. Cette transformation sera nécessairement adaptée pour chaque cas d'étude, et trouver un algorithme générique pour détecter les composants d'architecture reste un problème complexe de génie logiciel [1]. Nous regrettons ne pas avoir accès au source code du référentiel MEGA pour réaliser un tissage par transformation pour éviter une traduction et un tissage manuels. Ce qui nous aurait ensuite permis d'envisager une analyse et des mesures d'alignement.

Notre approche permet à ce stade un alignement des modèles métier et applicatif ; cependant aucune analyse, aucune mesure ne permet encore d'indiquer la qualité de l'alignement. Par conséquent la suite des travaux est de définir des règles pour réaliser différents types d'analyse : couverture, découplage et dépendance. L'analyse permettra d'obtenir une matrice de dépendance (DSM - *Dependency Structure Matrix*) pour obtenir une représentation graphique des relations entre chaque couche. Par exemple les tâches et les services sont mis en correspondance entre les couches processus et applicative ou bien même à l'intérieur d'une couche pour déterminer le couplage. On souhaite ainsi déterminer les adhérences entre composants de la couche applicative. D'autres analyses seront également réalisées pour déterminer où sont disséminées les données dans le SI des objets métier ou pour identifier si le flot d'exécution est conforme à l'enchaînement des tâches d'un processus. L'affichage se ferait par agrandissement (*drill-down* en anglais) afin de se focaliser sur une partie précise en obtenant les détails, par exemple en cliquant sur une tâche obtenir tous les services en relation. Toutes les règles seront exécutées en lot à la demande pour effectuer une analyse qualité régulière sur le SI. La définition de violation par un niveau de seuil permettrait d'obtenir des indicateurs et un ordre de priorité pour procéder à des actions de correction.

Une autre perspective est l'enrichissement du tissage avec plus de concepts pour mieux prendre en compte les points d'évolution du système d'information, pas seulement la structure [23]. En particulier, nous souhaitons pouvoir représenter le couplage entre parties de modèles. Dans cette lignée, nous devons mettre à l'épreuve nos méta-modèles vis-à-vis de pratiques (non automatisées) d'urbanisation et d'alignement.

Références

[1] Nicolas Anquetil, Jean-Claude Royer, Pascal André, Gilles Ardourel, Petr Hnetynka, Tomas Poch, Dragos Petrascu, and Vladiela Petrascu. JavaCompExt : extracting architectural elements from java source code. In *Proceedings of WCRE 2009*, pages 317–318. IEEE, 2009.

[2] L. Aversano, C. Grasso, and M. Tortorella. A literature review of Business/IT alignment strategies. In José Cordeiro, Leszek A. Maciaszek, and Joaquim Filipe, editors, *Enterprise Information Systems*, number 141 in LNBIP, pages 471–488. Springer, January 2013.

[3] Hugo Brunelière and Grégoire Dupé. Virtual EMF - transparent composition, weaving and linking of models. In *EclipseCon Europe 2011*, November 2011.

[4] J. Capirossi. *Architecture d'entreprise*. Collection Management et informatique. Hermes, 2011.

[5] Jae Choi, Derek L. Nazareth, and Hemant K. Jain. The impact of SOA implementation on IT-Business alignment : A system dynamics approach. *ACM Trans. Manage. Inf. Syst.*, 4(1) :3 :1–3 :22, April 2013.

[6] Mickaël Clavreul. *Composition de modèles et de métamodèles : Séparation des correspondances et des interprétations pour unifier les approches de composition existantes*. PhD thesis, Université Rennes 1, December 2011.

[7] V. De Castro, E. Marcos, and Juan M. Vara. Applying CIM-to-PIM model transformations for the service-oriented development of information systems. *Inf. Softw. Technol.*, 53(1) :87–105, January 2011.

[8] Philippe Desfray and Gilbert Raymond. *TOGAF en pratique : Modèles d'architecture d'entreprise*. Management des systèmes d'information. Dunod, 1 edition, 2012.

[9] Marcos Didonet, Del Fabro, Jean Bézivin, and Patrick Valduriez. Weaving models with the eclipse AMW plugin. In *In Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[10] DISIC and Government of France. Cadre commun d'Architecture d'Entreprise applicable au système d'information de l'Etat et à sa transformation, 2012.

[11] Marius Duedahl, Jostein Andersen, and Maung K. Sein. When models cross the border : Adapting IT competencies of business managers. In *Proceedings of the 2005 ACM SIGMIS CPR Conference*, SIGMIS CPR '05, page 40–48, New York, NY, USA, 2005. ACM.

[12] Matthias Galster. Dependencies, traceability and consistency in software architecture : Towards a view-based perspective. In *Proceedings of the 5th European Conference on Software Architecture : Companion Volume*, ECSA '11, page 1 :1–1 :4, New York, USA, 2011. ACM.

[13] The Open Group. *TOGAF Version 9.1*. van Haren Publishing, 10th edition, 2011.

[14] Brahim Lahna, Ounsa Roudiès, and Jean-Pierre Giraudin. Approches par points de vue pour l'ingénierie des systèmes d'information. *e-TI - la revue électronique des technologies d'information*, Numéro 5, 2009.

[15] Marc M. Lankhorst. *Enterprise Architecture at Work - Modelling, Communication and Analysis (3. ed.)*. The Enterprise Engineering Series. Springer, 2013.

[16] Christophe Longépé. *Le projet d'urbanisation du SI : Cas concret d'architecture d'entreprise*. InfoPro. Management des systèmes d'information. Dunod, 4 edition, 2009.

[17] Jerry Luftman, Raymond Papp, and Tom Brier. Enablers and inhibitors of business-IT alignment. *Commun. AIS*, 1(3es), March 1999.

[18] Chantal Morley. *Management d'un projet Système d'Information - Principes, techniques, mise en oeuvre et out : Principes, techniques, mise en oeuvre et outils*. Management des systèmes d'information. Dunod, 7 edition, 2012.

[19] Kestutis Normantas, Sergejus Sosunovas, and Olegas Vasilecas. An overview of the knowledge discovery meta-model. In *Proc. of the 13th International Conference on Computer Systems and Technologies*, CompSysTech '12, page 52–57, NY, USA, 2012. ACM.

[20] Jacques Printz. *Architecture logicielle - Concevoir des applications simples, sûres et adaptables*. Etudes et développement. Dunod, 3 edition, 2012.

[21] Robert Reix. *Systèmes d'information et management des organisations*. Vuibert, Paris, 2011.

[22] Gérard Roucairol and Yves Caseau. *Urbanisation, SOA et BPM : Le point de vue du DSI*. InfoPro, Management des systèmes d'information. Dunod, 4 edition, 2011.

[23] J. Saat, U. Franke, R. Lagerstrom, and Mathias Ekstedt. Enterprise architecture meta models for IT/Business alignment situations. In *Enterprise Distributed Object Computing Conference (EDOC), 2010 14th IEEE International*, pages 14–23, 2010.

[24] The Open Group. *Archimate 2.1 Specification.* Van Haren Pub, 2013.

[25] Azmat Ullah and Richard Lai. A systematic review of business and information technology alignment. *ACM Trans. Manage. Inf. Syst.*, 4(1) :4 :1–4 :30, April 2013.

[26] Club URBA-EA. *Urbanisme des SI et gouvernance : Bonnes pratiques de l'architecture d'entreprise*. InfoPro. Management des systèmes d'information. Dunod, 1 edition, 2010.

Vers la Vérification Formelle de Transformations de Modèles Orientées Objet

Moussa Amrani, Pierre Kelsen and Yves Le Traon

Université du Luxembourg
{Moussa.Amrani, Pierre.Kelsen, Yves.LeTraon}@uni.lu

Résumé

Cette Thèse contribue au domaine de la Vérification Formelle de Transformations de Modèles suivant deux perspectives. *Méthodologiquement*, elle propose un état de l'art complet du domaine, et définit un Cadre de Description aidant les ingénieurs à comprendre quelles propriétés doivent être vérifiées pour garantir la correction de leurs transformations. *Pratiquement*, elle propose un prototype pour la vérification de transformations écrites en Kermeta, un moteur de transformation orienté objet aligné sur Mof.

1 Introduction

Cet article décrit les résultats produits dans le cadre notre thèse [2] qui portait sur deux sujets traditionnellement séparés : l'Ingénierie des Modèles et l'Analyse Logicielle par Vérification Formelle. Nous introduisons d'abord succintement ces domaines, présentons nos contributions, avant de proposer un aperçu plus détaillé de chacune d'elles.

1.1 Ingénierie des Modèles

Apparue à la fin des années 90 avec Uml, l'Ingénierie des Modèles (Idm) est une méthodologie de développement logiciel qui poursuit l'effort pour repousser les limites propres à ce domaine, afin de réduire les coûts de développement et de maintenance des logiciels, d'améliorer leurs performances et leurs fiabilité, et de mieux maîtriser leur évolution. Dans les langages de programmation classiques, un programme était d'abord constitué d'un ensemble de procédures et de fonctions, puis d'un ensemble d'objets en interaction pour les deuxième et troisième générations. Vue comme la quatrième génération, l'Idm considère l'interaction avec la machine à travers un ensemble de modèles et de transformations pour toutes les étapes du cycle de développement.

Le principe de la transformation de modèles suit le schéma décrit dans la Figure 1 : on souhaite *transformer* un (ou plusieurs) modèle(s) d'entrée en un (ou plusieurs) modèle(s) de sortie, mais de façon contrôlée, rigoureuse et automatisée. Pour cela, il faut définir la structure d'un modèle au travers d'un *métamodèle* qui définit quelles constructions sont permises au sein d'un modèle : ce dernier est dit *conforme* au métamodèle s'il en respecte la syntaxe. On distingue deux aspects dans le processus de transformation : la *spécification*, définie par l'ingénieur à partir des métamodèles, décrit les *exécution*s pour modifier le(s) modèle(s) d'entrée afin d'obtenir le(s) modèles de sortie. Les métamodèles sont eux-mêmes modélisés : leur syntaxe est définie par un *méta-métamodèle*, qui a souvent la capacité de représenter sa propre syntaxe (ce qui permet de clore la chaîne, ou pyramide [] de métamodélisation). De manière similaire, la spécification des transformations se conforme à une syntaxe définie par un métamodèle, plus souvent appelé *langage de transformation*, dont la syntaxe est aussi définie par un méta-métamodèle. On distingue traditionnellement deux paradigmes de transformation : les *langages de transformation de graphes*, permettant des spécifications visuelles, trouvent leurs fondements

dans la théorie des catégorie ; et les *langages métaprogrammés*, qui sont très proches des langages de programmation usuels et en partagent donc les fondements théoriques.

Parmi les approches existantes de l'IDM, la Modélisation Dédiée consiste à représenter les concepts d'un domaine d'expertise, ainsi que ses règles internes, au sein d'un modèle, pour permettre aux ingénieurs de travailler au niveau d'abstraction dont ils ont l'habitude dans leur domaine, parce qu'ils manipulent ainsi les constructions avec lesquelles ils sont familiers au lieu de travailler avec des concepts provenant des solutions techniques envisagées. La sensation est renforcée par la possibilité d'adjoindre des syntaxes visuelles reprenant les codes et pictogrammes d'un domaine : ainsi, un banquier pourra définir de nouveaux produits bancaires pour lesquels la plateforme de modélisation, au courant des contraintes pesant sur l'ensemble des produits, pourra en signaler les conflits, au lieu d'implémenter ces produits à l'aide de classes implémentant une interface assortie d'invariants codant ces conflits. Des Langages de Modélisation Dédiée (LMD), outils de base de la Modélisation Dédiée, permettent ainsi d'atteindre un fort niveau d'automatisation dans les tâches répétitives que rencontrent les ingénieurs, leur permettant de focaliser leurs efforts cognitifs sur les problèmes métier plutôt que sur leurs solutions techniques. Ainsi, il devient possible de manipuler ces modèles pour les traduire dans d'autres langages ou générer du code exécuté sur plusieurs plateformes tout en tenant compte des spécificités de chacune, ou encore extraire des informations pertinentes, analyser ces modèles avant même de les exécuter, les combiner entre eux pour réaliser d'autres tâches, etc.

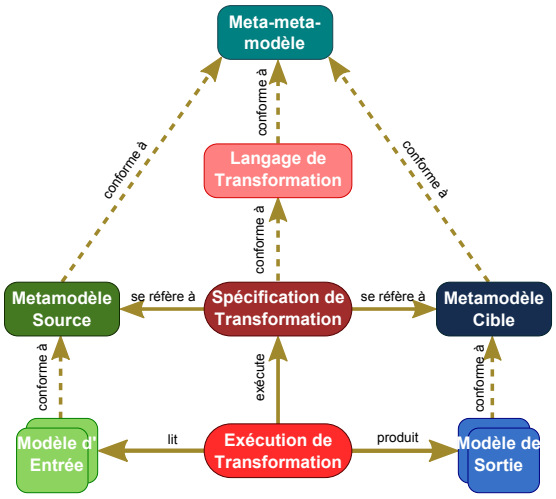


FIGURE 1 – Schéma général pour la Transformation de Modèles [4, 15]

1.2 Vérification de Logiciels

La haute automatisation permise par la transformation de modèle, ainsi que la versatilité des domaines représentés par les modèles permettent à l'IDM d'incarner une réponse possible aux défis de certains domaines industriels qui exigent de maîtriser la complexité logicielle, de produire de nombreuses variantes d'un même système pour différentes cibles ou différents contextes, ou de concilier des systèmes fortement hétérogènes nécessitant de concilier différents modèles de calcul au sein d'une même application. C'est ainsi que des secteurs comme l'aéronautique, l'automobile ou l'aérospatial commencent à adopter les principes de l'IDM afin de répondre à des marchés toujours plus exigeants.

Bien souvent, le Test ne suffit pas pour répondre aux standards de certification de ces secteurs industriels : les dysfonctionnements dans un logiciel embarqué ou critique se traduit par de lourdes pertes, soit financières, soit humaines, et même parfois écologiques. Le recours à des méthodes plus lourdes, mais plus puissantes, est alors nécessaire : la Vérification Formelle consiste à analyser mathématiquement la correction d'un système exhaustivement (i.e. pour toute exécution possible) et statiquement (i.e. sans devoir l'exécuter), sur la base d'un modèle du système et de la donnée de propriétés de correction. L'inconvénient principal de la Vérifi-

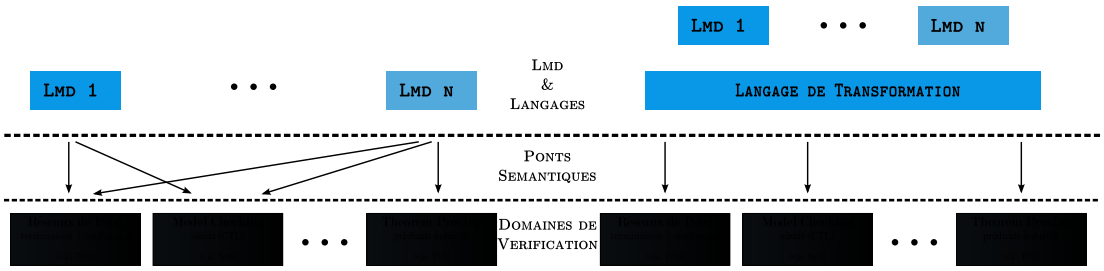


Figure 2 – Analyse Formelle de Lmds. À gauche, l'approche classique consiste à définir un pont sémantique pour chaque Lmd vers chaque domaine de vérification nécessaire à la vérification d'un certain type de propriétés. À droite, notre proposition consiste à définir un tel pont à partir du langage de transformation, réduisant ainsi la complexité générale de la tâche.

cation Formelle est le problème dit d'explosion combinatoire : explorer toutes les exécutions possibles en un temps fini est impossible pour des systèmes infinis. Plusieurs techniques et outils existent déjà, développés par la communauté travaillant sur la vérification assistée par ordinateur (*Comuter Assisted Verification*), mais ceux-ci ont été développé pour des langages de plus bas niveau.

1.3 Contributions

Au fur et à mesure de l'adoption de l'Idm pour le développement d'applications embarqués, critiques, sécurisées, mais aussi pour d'autres secteurs moins exigeants, l'enjeu crucial de la vérification formelle de l'Idm va se poser : comment mettre à disposition cette technique d'analyse ? Comment faire bénéficier de ces résultats pour faciliter le travail quotidien des ingénieurs en Idm ?

Passer par l'adaptation au contexte de l'Idm des techniques d'analyse formelle est une idée naturelle qui se heurte rapidement à la complexité des artefacts manipulés : les modèles peuvent représenter des structures à la complexité arbitraire (de simples structures de données à des domaines d'expertise entiers, en passant par des transformations), ce qui complique leur analyse et celle des transformations associées. En outre, les ingénieurs en métamodélisation et transformation ne sont généralement pas des experts de la vérification, et vont donc avoir du mal à concilier analyse formelle et expertise métier, d'autant que chaque activité suit des cycles antagonistes : l'Idm procède par évolutions itératives et incrémentales lors de la mise au point des modèles et des transformations, tandis que la vérification formelle nécessite que l'ensemble de ces éléments soient définis avant de pouvoir utiliser les outils correspondants. Voilà un défi de taille : *comment aider les ingénieurs en* Idm *à prouver la correction de leurs transformations ?*

En termes d'analyse formelle, il est illusoire d'espérer pouvoir utiliser un seul outil pour prouver les propriétés de correction de toutes les transformations (l'approche dite « *on-fits-all* »), parce que la limite théorique imposée par l'indécidabilité de ces techniques, et par celle plus pratique d'explosion combinatoire. De nombreuses contributions ont suivi le chemin suivant : pour chaque Lmd, un pont sémantique vers un domaine de vérification adéquat pour prouver certaines propriétés est défini. Comme l'illustre la Figure 2 (à gauche), cette étape doit se répéter pour chaque domaine capable de prouver de nouvelles propriétés, mais surtout pour chaque nouveau Lmd, augmentant la complexité de la tâche de manière exponentielle et la rendant fastidieuse face à la multiplication des Lmds dans tous les domaines. Le second défi peut donc se formuler ainsi : *comment faciliter la vérification de* Lmd *dans le cadre général ?*

Notre thèse répond à ces deux défis en proposant un cadre méthodologique guidant les ingénieurs dans leurs tâches de vérification, et une proposition pratique pour réduire la complexité liée au nombre croissant de LMDs :

1. Méthodologiquement, nous proposons un Cadre Descriptif visant à servir de support de travail pour les ingénieurs impliqués dans la vérification formelle de transformations. Ce Cadre les aide à identifier précisément le but de leurs transformations, ce que nous appelons *intention*, et propose une classification précise des propriétés à prouver pour chacune de ces intentions.
2. Pratiquement, nous proposons de définir les ponts sémantiques nécessaires à la vérification de LMD directement au niveau des langages de transformation, comme le montre la Figure 2 (à droite). Ainsi, tout LMD écrit dans un langage particulier peut directement utiliser les analyses proposées dans chaque domaine où un pont existe déjà, et chaque nouveau pont bénéficie à l'ensemble des LMDs, présents ou futurs. Afin de réaliser cette vision, il est nécessaire de définir une sémantique de référence du langage de transformation en question, i.e. une sémantique spécifiée indépendamment de leur plateforme d'exécution, ou des domaines de vérification potentiels. Cette sémantique permet de documenter précisément les abstractions opérées lors de chaque pont sémantique sans restreindre la définition originale, permettant ainsi d'étendre les possibilités d'analyse liées à chaque langage de transformation.

Nous détaillons notre Cadre Descriptif pour la vérification de transformations de modèles dans la Section 2. Nous avons appliqué notre proposition pratique à un langage de transformation populaire, Kermeta : nous donnons des éléments de compréhension de la sémantique de référence de Kermeta en Section 3, et décrivons notre prototype KMV en Section 4, incarnant un domaine de vérification permettant le model-checking et le theorem-proving de transformations Kermeta.

2 Un Cadre Descriptif pour Vérifier les Transformations

De nombreux aspects de l'IDM liés à la transformation de modèles ont été étudiés : comment construire de manière raisonnée et efficace des familles de langages de transformation assurant de bonnes propriétés [21] ; comment appliquer les transformations adéquates suivant le contexte d'application [20]. Peu de recherches, en revanche, ont été conduites dans le but d'étudier les différentes intentions d'une transformation, en particulier pour guider leur développement ou leur validation. Plusieurs classifications des transformations ont vu le jour [11, 16], mais elles traitent du point de vue de l'ingénierie des langages, alors que du point de vue de notre premier défi, c'est le point de vue de l'analyse formelle qui importe. Une question naturelle serait donc de savoir *comment les chercheurs et les ingénieurs mènent-ils actuellement leur travail lorsqu'il s'agit de formellement vérifier des* LMDs. Nous avons conduit une revue de la littérature pour y répondre [5, 6, 3], et avons mis en évidence la centralité des dimensions entrant en jeu dans l'activité de vérification des transformations : la transformation en cours d'analyse, les différentes propriétés impliquées, et les techniques de vérification pour les prouver. Cette étude a aussi montré que les classifications existantes ne suffisent pas à extraire les propriétés pertinentes pour prouver la correction des transformations : ces classifications sont syntaxiques, attachées à la forme des transformations, alors même qu'elles devraient s'intéresser au but de la manipulation décrite par la transformation, leur *intention*.

Notre Cadre Descriptif répond partiellement à ce manquement, en proposant deux contributions majeures : d'abord, il propose un Catalogue des intentions permettant de regrouper les transformations partageant le même type de manipulation de modèles ; il les associe ensuite

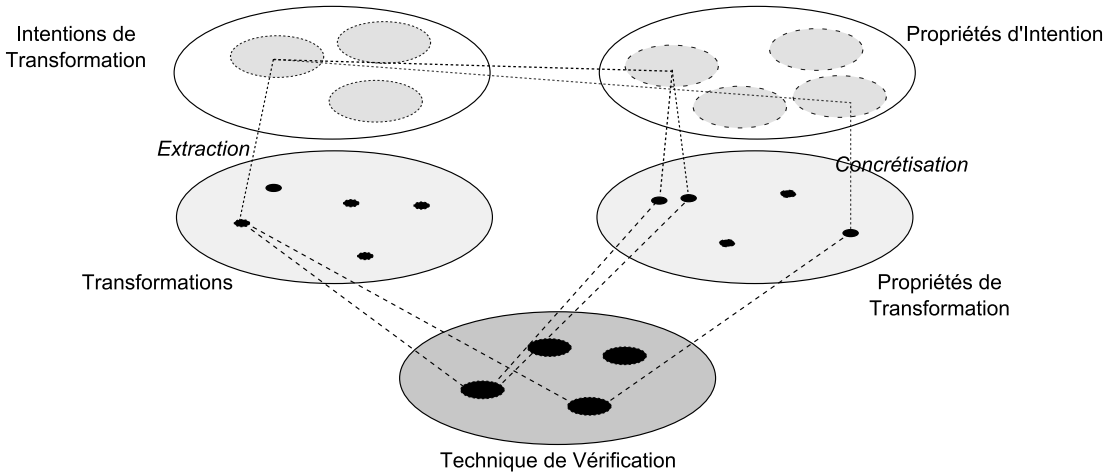


FIGURE 3 – Schéma général illustrant le Cadre Descriptif, qui propose de travailler à un plus haut niveau d'abstraction, pour associer aux intentions de transformation des propriétés d'intention, concrétisées en des propriétés de transformation assurant la correction des transformations de modèle à l'aide d'une technique de vérification.

avec des *propriétés d'intention*, comme la terminaison, la traçabilité, la préservation d'aspects structurels ou sémantiques, etc. dont la preuve conduit à montrer formellement la correction des transformations. Ce Cadre peut être utilisé dans de nombreux scénarii, en particulier pour identifier l'intention liée à une transformation, dériver les propriétés nécessaires pour prouver la correction d'une transformation. Nous détaillons le Cadre Descriptif dans ses aspects méthodologique (Section 2.1) et opérationnel (Section 2.2), avant d'énoncer les principaux résultats pour valider cette contribution.

2.1 Description Méthodologique

Traditionnellement, un ingénieur met en œuvre une technique de vérification sur la base de la transformation dont il veut prouver la correction, et la donnée de propriétés pertinentes, comme le montre le bas de la Figure 3. Malheureusement, transformation et propriétés ne sont pas liées : est-ce que les propriétés capturent adéquatement la notion de correction liée à cette transformation ? Bien souvent, ce lien n'est établi qu'informellement sur la base de l'expérience et le savoir-faire de l'ingénieur.

Notre Cadre Descriptif propose une approche systématique pour pouvoir extraire ces propriétés en travaillant à un niveau plus abstrait. Au lieu de considérer les aspects syntaxiques d'une transformation, notre Cadre se base sur la sémantique sous-jacente, l'*intention de la transformation*, pour associer un ensemble de *propriétés d'intention* qui caractérise toute transformation possédant cette intention, en oubliant les détails de réalisation d'une transformation (son langage de spécification, son caractère exogène, conservatif, son arité, etc.) Afin d'obtenir des propriétés prouvables, il est nécessaire de passer par une étape de *concrétisation* qui réintroduit les détails spécifiques à la transformation au sein des propriétés d'intention afin d'obtenir des propriétés de transformation. En outre, cette association intention/propriétés d'intention définit une série de critères dont l'application permet d'*extraire* l'intention liée à une transformation.

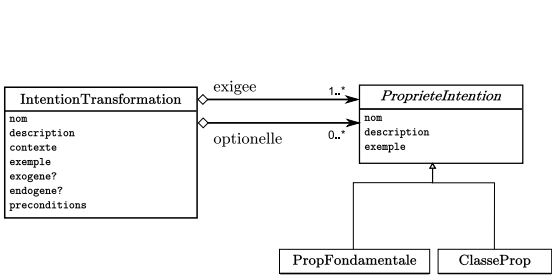


FIGURE 4 – Métamodèle du Cadre Descriptif.

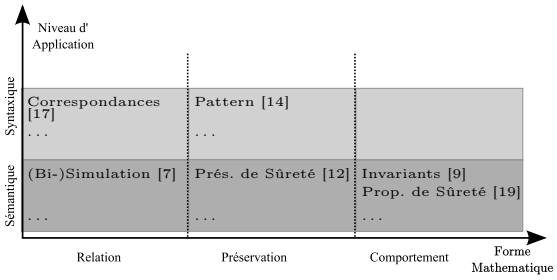


FIGURE 5 – Classes de Propriétés relatives aux intentions étudiées, illustrées par une contribution typique (cf. [5, 15] pour les détails).

2.2 Vue Opérationnelle

Concrètement, notre Cadre Descriptif prend la forme du métamodèle décrit en Figure 4. La classe IntentionTransformation décrit une intention à la manière d'un patron de conception objet : une intention possède un nom, est décrite de manière générale par description, placée dans ses différents contextes d'usage, tandis qu'un ensemble de preconditions permet de circonscrire les transformations éligibles. Enfin, un ou plusieurs exemples documente chaque intention à l'aide d'une transformation typique extraite de la littérature.

La classe ProprieteIntention décrit une propriété qui peut être vue comme un *template* à trous décrivant soit des spécificités d'une transformation (sa spécification, ou certains éléments qui y sont liés comme le métamodèle cible), soit des caractéristiques d'une propriété elle-même, caractéristique à toute transformation possédant une intention particulière. Une telle propriété possède aussi un nom ; une spécification mathématique et un exemple permettent une description détaillée. La précision et le nombre de "trous" rend ces propriétés plus ou moins précises, nécessitant en contrepartie un travail plus ou moins important lors de la concrétisation. Bézivin rappelle qu'une transformation peut être considérée de deux points de vue duaux : comme un modèle de transformation (*transformation model*), l'accent se porte sur le calcul opéré par la transformation ; et comme une transformation de modèle (*model transformation*), l'accent sur porte sur les modèles manipulés par la transformation [8]. Similairement, nous considérons deux classes de propriétés : les *propriétés fondamentales* caractérisent le calcul (avec comme instances particulières de ces propriétés, la terminaison, le déterminisme, le typage) tandis que les *classes de propriétés* capturent les manipulations de modèle spécifiques à une intention [**?**].

D'un point de vue pratique, les classes de propriétés seront concrétisées par les ingénieurs pour prouver la correction de leurs transformations. Une classe de propriété est caractérisée à la fois par son niveau d'application, syntaxique ou sémantique, et par sa forme mathématique, qui décrit la manière dont la correction de la manipulation de modèle s'exprime. Dans le cadre de notre thèse, nous avons étudié trois formes différentes, comme décrit en Figure 5 :

— Une *Relation* construit une relation mathématique (au sens de la théorie des ensembles) entre artéfacts d'entrée et de sortie : lorsqu'elle est *syntaxique*, une Relation est souvent appelée *correspondance*, et relie des éléments de métamodèle et de modèle (classes, références, attributs, instances, etc.) ; lorsqu'elle est *sémantique*, elle relie des états des domaines sémantiques de chaque modèle.

— Une *Préservation* exprime le fait que quelque chose sur le modèle d'entrée est préservé dans le modèle de sortie : *syntaxiquement*, une préservation s'exprime souvent à l'aide d'un langage de *patterns* qui décrit un agencement particulier des éléments du modèle

d'entrée que l'on aimerait retrouver, sous une forme ou une autre, dans le modèle de sortie ; *sémantiquement*, ce sont souvent des logiques temporelles qui expriment des propriétés de sûreté que le modèle de sortie doit aussi satisfaire.
— Une *Propriété Comportementale* caractérise l'exécution de la transformation au lieu de chercher des liens entre les modèles d'entrée et de sortie (comme le font les deux formes précédentes), exprimées au travers d'invariants ou de formules de logiques temporelles.

Nous distinguons deux types de propriétés. Les propriétés exigees décrivent les propriétés *nécessaires* à une transformation pour être qualifiée avec une intention particulière, bien qu'elles ne soient pas *suffisantes* au sens mathématique, puisque des intentions similaires partagent une partie de leurs propriétés caractéristiques : il faut alors recourir aux autres attributs dans IntentionTransformation afin de désambiguïser l'intention associée. Les propriétés optionnelles collectent d'autres propriétés désirables des transformations appartenant à une intention, bien qu'elles ne soient pas nécessaires, mais permettent de mieux guider soit la procédure de concrétisation, soit la preuve elle-même.

2.3 Résultats

Notre Cadre Descriptif rassemble plusieurs résultats intéressants qui ouvrent la voie à une véritable ingénierie de la vérification des transformations [4, 15].

1. À partir d'une large étude de la littérature, nous construisons un Catalogue identifiant 21 intentions de transformations, partiellement décrites comme des instances de la classe IntentionTransformation, et organisées en catégories traduisant les différents types de manipuation. Nous avons soumis ce Catalogue à une validation empirique auprès d'une cinquantaine d'ingénieurs et chercheurs, sur la base d'un questionnaire demandant d'identifier l'intention de transformations typiques à partir de notre description d'intentions.

2. Nous décrivons l'association entre cinq intentions (choisies parmi les plus répandues dans la littérature) et leurs propriétés de transformations : la Simulation, la Translation, l'Analyse, la Requête et le Raffinement. Ces intentions guident la spécification des classes de propriétés proposées en Figure 5 : nous pensons que d'autres classes seront nécessaires pour décrire les propriétés des intentions appartenant à d'autres catégories.

3. Nous validons notre Cadre à l'aide d'une étude de cas industrielle : celle-ci représente une chaîne automatisée d'une trentaine de transformations manipulant 22 métamodèles différents, et décrit la génération d'un code validé exécuté sur une infrastructure pilotant les fenêtres électriques d'une voiture. Nous avons identifié en utilisant notre méthodologie l'intention de l'ensemble de ces transformations.

4. Nous montrons comment concrétiser les propriétés d'intention en les appliquant à deux transformations choisies dans l'étude de cas : à partir de leurs spécifications, nous tirons les propriétés de transformation nécessaires à l'utilisation dans un outil d'analyse.

La méthodologie décrite ici pose les bases de réflexion pour une véritable ingénierie de la vérification au sein de l'IDM en présentant un cadre d'étude des transformations systématique, extensible, et répétable. Il peut par exemple servir à développer de nouvelles plateformes de developpement dédiées à la spécification de transformations focalisées sur une intention particulière, afin d'assurer par construction les bonnes propriétés qui leur sont associées.

3 Une Sémantique Formelle de Référence pour Kermeta

Kermeta est une boîte à outils conçue pour répondre aux exigences technologiques de l'Idm tout en respectant les standards en vigueur de l'Omg pour les différentes activités en Idm : métamodélisation, contraintes sur les modèles, transformations et sémantique des langages (cf. Figure 6). Trois langages constituent le noyau dur de Kermeta : le Langage Structurel, aligné sur Mof, permet de définir la structure des modèles ; le Langage de Contraintes, aligné sur Ocl, permet de définir des contraintes sur ces structures (vue comme une sémantique statique) et ainsi d'exclure des constructions qui ne représentent pas la réalité ; et le Langage d'Action, basé sur des constructions orientées objet afin de manipuler ces modèles pour définir des transformations cohérentes [13, 22].

Dans le cadre de cette thèse, nous avons complètement défini la sémantique de ces langages. Plutôt que d'entrer dans les détails mathématiques, assez techniques, nous choisissons ici d'insister sur les principes déterminant la construction mathématique en insistant sur les difficultés rencontrées : nous détaillons d'abord les restrictions langagières opérées, puis détaillons la construction de la spécification mathématique [1].

3.1 Restrictions

Kermeta est une riche plateforme de métamodélisation et de transformation, couvrant de multiples activités : ingénierie des besoins, gestion de traçabilité, de contrôle d'accès, mais aussi composition de modèles. Formaliser autant de facettes n'était pas raisonnable dans le temps d'une thèse ; aussi avons-nous préféré nous concentrer sur un noyau de métamodélisation et de transformation représentatif de l'ensemble de ces activités, étant entendu que ces activités sont au final elles-mêmes exprimées à l'aide de transformations.

Pour représenter ses modèles, Kermeta utilise une extension conservatrice de Mof, le standard de l'Omg pour la métamodélisation : Kermeta sait importer n'importe quel modèle Mof, et un modèle Kermeta reste compatible avec Mof. Dans notre travail, nous avons volontairement restreint la portée de notre formalisation à ce standard, excluant de fait des caractéristiques traitées par Kermeta comme la généricité (i.e. l'équivalent des classes paramétrées de Java) pour définir des structures de données concises et réutilisables, et les *types de modèle* qui permettent d'étendre la notion de type aux modèles afin de garantir le bon comportement des transformations sur les modèles pris en entrée. De plus, la *modélisation par aspects* est facilitée en Kermeta pour gérer les différentes facettes couvertes par un modèle, et de les transformer en tenant compte du tissage. Nous n'avons pas formalisé le Langage de Contraintes, utilisé à la fois pour restreindre les modèles considérés comme valides, et pour la spécification par contrats : la plupart de ces constructions sont déjà représentés dans le Langage d'Action que nous avons isolé.

3.2 Langage Structurel

Le Langage Structurel de Kermeta, ainsi que notre formalisation mathématique, suit la définition en deux parties de l' Omg pour Mof : eMof (pour *Essential* Mof) couvre la définition des métamodèles tandis que cMof (pour *Complete* Mof) fournit la possibilité de définir des modèles. Notre formalisation se base sur la Théorie des Ensembles : elle utilise les noms des entités composant un modèle ou un métamodèle (les énumérations, les packages, les classes et leurs propriétés et opérations), et traduit les relations entre ces entités par des ensembles et des fonctions partielles, conformément à la manière dont ils sont définis par Mof. La Figure 7

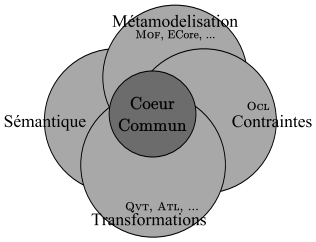


FIGURE 6 – Kermeta est conçu comme un noyau couvrant différentes activités liées à l'IDM : métamodélisation, contraintes, transformation et sémantique.

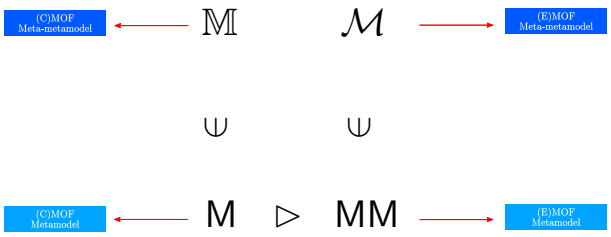


FIGURE 7 – Procédé de construction formelle pour représenter les métamodèles et modèles. Deux langages $\mathcal{M}$ et $\mathbb{M}$ capturent les définitions des langages eMOF (pour les métamodèles) et cMOF (pour les modèles). La relation $\triangleright$ exprime qu'un modèle $\texttt{M} \in \mathbb{M}$ est conforme à son métamodèle $\texttt{MM} \in \mathcal{M}$.

traduit les relations entre la formalisation mathématique et les spécifications techniques de MOF :

- l'ensemble $\mathcal{M}$ capture la syntaxe des métamodèles, i.e. les noms d'entités composant un métamodèle particulier et les types auxquels ces noms sont associés (par exemple, une référence entre deux classes possède le type de la classe sur laquelle elle pointe), conformément à la typologie spécifiée par eMOF ;
- l'ensemble $\mathbb{M}$ capture la syntaxe des modèles, i.e. l'association entre ces noms et les valeurs qu'ils portent au sein d'un modèle particulier, avec les restrictions définies dans cMOF ;
- une relation $\triangleright$ traduisant la conformité d'un modèle par rapport à son métamodèle, sous la forme d'un prédicat vérifiant que les valeurs des entités d'un modèle sont valides par rapport aux déclarations de type de ces entités au sein du métamodèle.

Les ensembles $\mathcal{M}$ et $\mathbb{M}$ peuvent être vus comme la représentation mathématique des méta-métamodèles décrits par eMOF et cMOF, respectivement. La Figure 8 donne une représentation visuelle du méta-métamodèle utilisé par Kermeta, similaire à eMOF. Un métamodèle est un ensemble de packages contenant éventuellement des sous-packages. Chaque package définit un ensemble de types : un types de données représente une entité d'un des types primitifs usuels (booléen, entiers, etc.) ; une énumération définissant une liste de litéraux ; ou enfin une classe éventuellement abstraite et pouvant hériter d'autres classes, possédant un nom, et des composantes structurelles (*Features*). Une composante structurelle de classe possède un nom et un type et peut être soit une propriété (une référence ou un attribut), soit une opération, potentiellement abstraite, avec un type de retour, une liste de paramètres et un corps.

MOF tire sa définition d'UML, lui-même directement inspiré par la programmation orienté objet, et en particulier Java. Il n'est donc pas étonnant de retrouver le même vocabulaire et les mêmes notions (comme les notions de package et de classe, qui ont le même sens). Cependant, plusieurs différences méritent d'être soulignées :

- MOF autorise l'héritage multiple, une possibilité offerte en Java uniquement au niveau des interfaces et pas des classes ;
- Le concept de propriété structurelle remplace la notion de champ Java, à la différence qu'il n'est pas possible de spécifier une portée de visibilité (en utilisant les mots-clefs `public/private/protected` en Java), ni une notion de globalité (pas d'utilisation de `static`), puisque toute propriété structurelle en MOF devient disponible pour toute

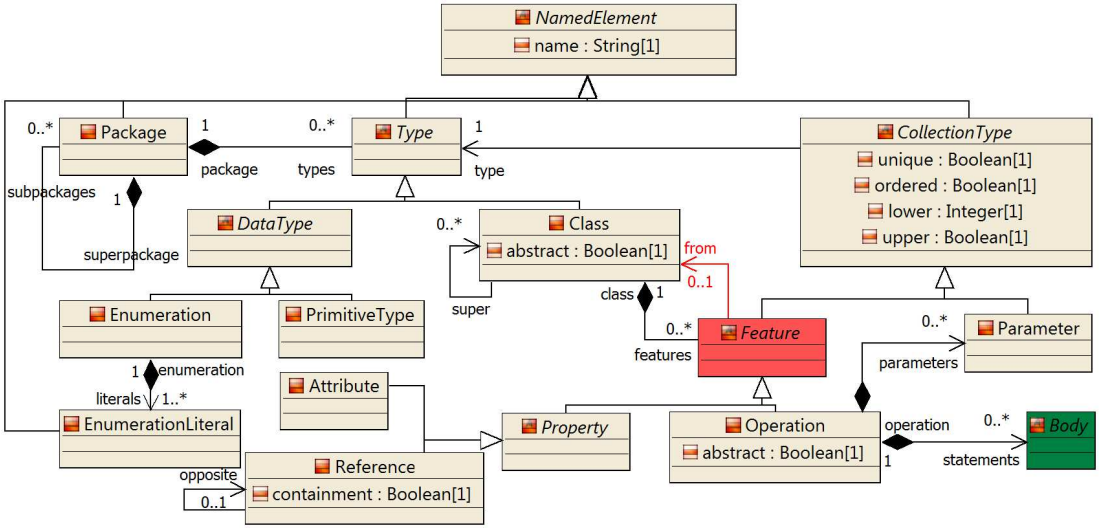


FIGURE 8 – Méta-métamodèle simplifié de Kermeta [13], compatible avec le standard de l'OMG eMOF [18]. En vert, la classe *Body* capture dans le méta-métamodèle les instructions du Langage d'Action servant à définir le comportement des *Opérations*. En rouge, la classe *Feature*, portant la référence *from*, est ajoutée au méta-métamodèle originel pour permettre de gérer les ambigüités de navigation et d'invocation d'opérations.

instance d'une sous-classe ;

— Les notions de multiplicité (inexistance en Java) et de collection sont native en MOF, alors qu'elles ne sont accessible qu'au travers d'une API en Java ;

— La surcharge d'opération (définir des opérations portant le même nom, mais avec des paramètres différents) n'est pas permise en Kermeta, alors que la redéfinition reste naturellement permise pour favoriser le comportement polymorphique. L'interdiction de la surcharge provient du fait qu'en présence d'héritage multiple, la résolution dynamique d'instance se trouve grandement compliquée et nécessite une résolution trop laborieuse.

— De même, l'héritage multiple suppose que l'on puisse gérer le cas où une propriété structurelle `prop` est définie le long de deux chemins d'héritage différents. À la base de ces chemins, une instruction de navigation comme `c.prop` est ambigüe. Pour résoudre ce problème tout en conservant les mécanismes de désambiguisation propres à Kermeta, nous avons introduit une nouvelle classe *Feature* dans le méta-métamodèle Kermeta, qui est la seule modification par rapport à eMOF.

Si ces choix semblent restrictifs par rapport à Kermeta, ils nous permettent au contraire de généraliser notre approche à tous les langages compatibles avec MOF (en particulier, les diagrammes de classes d'UML), conférant à notre formalisation un caractère plus général.

3.3 Langage d'Action

Nous avons formalisé la sémantique du Langage d'Action de Kermeta de manière classique : nous avons défini une grammaire BNF préservant au maximum la syntaxe concrète de Kermeta, pour laquelle nous avons spécifié un système de typage, défini inductivement, permettant d'assurer une correction statique des instructions, puis nous avons défini l'effet des instructions sur

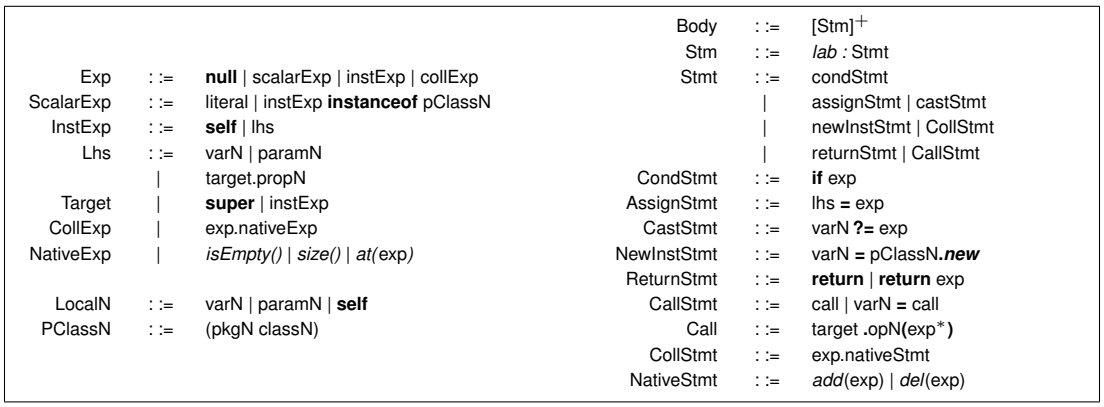


FIGURE 9 – Grammaire du Langage d'Action : expressions (à gauche) et Instructions (à droite).

les modèles grâce à une sémantique opérationnelle structurelle de type petit-pas [23].

Notre Langage d'Action ne suit pas strictement celui de Kermeta, mais définit plutôt une représentation intermédiaire simplifiée pour les besoins de la spécification formelle. Nous avons opéré plusieurs simplifications à différents niveaux :

— Nous avons d'abord supprimé toute instruction en rapport avec les restrictions opérées sur le Langage Structurel (généricité et type de modèle, essentiellement), puisqu'elles n'ont plus de sens ;
— Nous ne traitons pas des constructions relatives au traitement des exceptions, et des appels à Java, qui représentent des dimensions orthogonales à la transformation (les exceptions permettant de contrôler les cas d'erreur, les appels Java permettant d'utiliser des bibliothèques de traitement sans rapport avec la modélisation, par exemple pour faire du tri, ou de la cryptographie) ;
— Nous avons supprimé la structure de bloc pour les instructions, ne conservant que le bloc principal représenté par le corps d'une opération. Cela permet de simplifier la spécification formelle en ne s'encombrant pas des complications liées à la portée des variables : celles-ci sont toutes déclarées au début des opérations et portent sur tout le corps, quitte à les renommer en cas de conflit.
— Enfin, nous avons aplati les structures de branchement (conditionnelles et boucles) afin d'avoir une représentation uniforme des instructions, ce qui permet de séparer le contrôle de flot de la définition de l'effet des instructions.

Ces simplifications sont d'ordre purement syntaxique : elles n'affectent pas directement le comportement du langage, ni ne restreignent les possibilités offertes à l'ingénieur pour définir des transformations. La Figure 9 décrit la grammaire considérée : on remarque que le langage est séparé en deux groupes syntaxiques : les instructions Stm, exprimant des modifications dans le modèle, sont construites à partir d'expressions Exp qui n'ont pas d'effet de bord.

Nous choisissons de ne pas détailler le système de typage. La construction de la sémantique opérationnelle structurelle suit le schéma classique : nous définissons un système de réécriture agissant sur des configurations $\gamma \in \Gamma$ dont le déclenchement est conditionné par l'instruction en cours et par un ensemble de conditions sur la configuration courante. Une configuration γ doit gérer trois éléments : l'*instruction en cours d'exécution* (identifiée par son label $\mathsf{lab} \in \mathsf{Lab}$), un *tas* permettant de conserver les informations nécessaires en cas d'invocation d'opération, et un domaine sémantique modélisant mathématiquement les éléments sur lesquels agissent les

transformations. En Kermeta, celui-ci comprend naturellement le modèle $\mathsf{M} \in \mathbb{M}$ impliqué, mais aussi un environnement local $l \in \mathbb{L}$ gérant les valeurs des variables locales à l'opération en cours d'exécution.

Les instructions du Langage d'Action ressemblent à celles d'un langage orienté objet : les conditionnelles CondStmt (recouvrant ici les instructions conditionnelles et les itératives), la création d'instance NewInstStmt, le casting CastStmt partagent la même sémantique que les instructions correspondantes en Java. Deux instructions ont une sémantique particulière au sein de Kermeta : l'affectation AssignStmt et l'invocation d'opération callStmt. L'affectation doit affecter une expression à une partie gauche tout en préservant la cohérence d'un modèle (en particulier vis-à-vis des références opposées et des valeurs collection). L'invocation d'opération est traitée de manière particulière : il s'agit de retrouver dynamiquement l'opération à invoquer sur une instance en gérant l'héritage multiple (i.e si une opération n'est pas disponible dans une classe, elle est définie dans l'une des classes dont elle hérite). En Kermeta, l'absence de surcharge d'opération permet de s'assurer qu'un nom d'opération est unique sur chaque chemin d'héritage.

4 Kmv : Un Outil de Vérification pour Kermeta

À partir de la sémantique de référence présentée en Section 3, nous définissons un pont sémantique vers Maude [10], un langage de programmation généraliste basé sur la réécriture de spécifications algébriques permettant deux types d'analyses formelles (*model-checking* et *theorem-proving*). Nous détaillons dans cette Section les raisons qui nous ont poussé à un tel choix, avant de discuter deux études de cas réalisées dans le cadre de cette thèse.

4.1 Maude comme Domaine de Vérification

Fruit d'une trentaine d'années de recherches fondamentales, Maude tire partie de l'expérience acquise dans le développement de langages de programmation basés sur le paradigme de la réécriture. Il possède donc une maturité et une stabilité certaine, de même qu'une large base communautaire d'utilisateurs et de projets industriels. Ses fondements logiques (logique d'appartenance équationnelle et logique de réécriture) reposent sur les mêmes bases mathématiques que les outils utilisés pour notre formalisation sémantique (théorie des ensembles et sémantique opérationnelle structurelle), ce qui permet de réduire l'effort de définition des abstractions opérant au niveau du pont sémantique : ainsi, notre implémentation suit de près la spécification formelle, définissant exactement une règle de réécriture en Maude pour chaque alternative de règle de réécriture opérationnelle.

Ces éléments nous ont convaincu du bien-fondé de notre choix. Maude a déjà été utilisé pour spécifier formellement de nombreux langages aux paradigmes variés (Haskell, Prolog, Ml, Lotos, Csp), montrant ainsi d'intéressantes capacités d'adaptation. En outre, ce qui fait la force de Maude est la possibilité, à partir d'une même spécification, d'utiliser à la fois un moteur de *model-checking* capable de rivaliser avec les outils à la pointe de l'état de l'art, et d'un outil de *theorem-proving* travaillant sur les spécifications algébriques.

Enfin, Maude semble être un bon candidat pour de futures évolutions. En particulier, il serait intéressant d'intégrer la généricité, caractéristique présente nativement pour les spécifications algébriques en Maude. De plus, comme nous l'expliquons dans les travaux futurs, gérer des transformations temps-réel pourrait se faire en utilisant Rt-Maude, l'extension temps-réel de Maude, tout en réutilisant le travail déjà existant.

4.2 Le Prototype Kmv

Notre prototype, Kmv (pour *Kermeta* in *Maude Verification*), exploite l'Arbre Syntaxique Abstrait résultant du compilateur Kermeta pour traduire les transformations Kermeta en Maude. Comme le décrit la Figure 10, nous réutilisons ainsi le travail réalisé par le compilateur (résolution des références externes, instanciation correcte des classes génériques, tissage des aspects) pour obtenir une définition complète, contournant ainsi certaines des restrictions imposées par notre spécification sémantique, simplifiant ainsi le travail de traduction en évitant de refaire des vérifications directement au niveau des fichiers Kermeta originaux.

Kmv est construit sur la base de deux autres spécifications Maude qui ont déjà montré leur efficacité : mOdCl définit un système de typage et de valeurs, ainsi qu'une syntaxe minimale permettant l'évaluation d'expressions Ocl sur des modèles Uml qui passe les benchmarks standards pour Ocl ; Maudeling étend et affine les spécifications de mOdCl avec les notions de packages, de multiplicité, de collection, ainsi qu'une définition fidèle de la conformance, permettant une représentation fidèle, précise et facilement manipulable des métamodèles et modèles Mof. Enfin, Kmv rajoute les spécificités relatives à Kermeta : la définition des opérations et de leur corps, qui utilise le Langage d'Action que nous avons défini à la Figure 9.

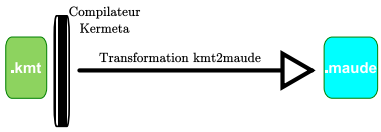


Figure 10 – Kmv travaille sur l'arbre syntaxique produit par le compilateur Kermeta pour simplifier le pont sémantique vers Maude.

4.3 Études de Cas

Nous avons vérifié l'absence d'interblocage dans des processus xSpem, une mouture simplifiée et exécutable du Lmd standardisé Spem pour la définition de procédés de développement. Notre premier modèle, constitué d'une dizaine de processus, décrit un cycle de développement logiciel avec les tâches connexes : production de documentation, versionnement, soumission de déliverables au clients, etc. tandis que notre second modèle, comportant aussi une dizaines de processus, décrit l'inscription d'étudiants dans une université le long des différentes étapes administratives (inscription pédagogique, administrative, et sportive, assurance sociale, délivrance des documents, etc.) xSpem permet de décrire précisément l'ordonnancement de tâches, i.e. si une tâche peut démarrer ou s'arrêter avant/après les tâches qui lui sont reliées. Nous avons montré deux invariants sur chacun des modèles : si chaque tâche est arrêtée, alors elle n'est pas bloquée ; et si chaque tâche n'est pas bloquée, alors éventuellement elle s'arrêtera. La conjonction de ces deux invariants prouve l'absence d'interblocage pour l'ensemble du processus de développement propre à chaque modèle.

Nous avons vérifié la correction des modèles de conception haut niveau obtenu après l'étape de spécification des besoins dans le cadre de notre étude de cas utilisée pour la validation de notre Cadre Descriptif (Section 2). Plusieurs Lmds capturent les différents aspects du fonctionnement d'une fenêtre (les boutons de chaque passager et leurs effets sur les fenêtres, les options de blocage, l'intégration avec l'énergie de la voiture pour arrêter leurs actions lorsque le contact est coupé, etc.). Les modèles issus de ces Lmds sont combinés et transformés en un Réseau de Petri qui traduit la sémantique de ces Lmds, i.e. comment les différents modèles interagissent entre eux pour simuler le comportement des fenêtres. On vérifie alors le Réseau : par exemple, qu'une action sur le bouton de vérouillage empêchait effectivement d'actionner les vitres côté passager ou à l'arrière. La sémantique des Réseaux de Petri, ainsi que les transformations combinant les différents modèles, étaient écrit à l'aide de transformation de graphes ; nous les avons réimplémentées en Kermeta et nous avons utilisé Kmv pour prouver les mêmes propriétés.

5 Conclusion

Notre thèse vise à répondre à deux défis contemporains de l'IDM : comment aider les ingénieurs en charge de la spécification de transformations de modèle à mettre en œuvre des techniques de vérification formelle pour assurer leur correction ; et comment réduire la complexité de la vérification des LMDs (dont la sémantique est exprimée à l'aide d'une transformation). Pour répondre au premier défi, nous avons proposé un Cadre Descriptif qui introduit la notion nouvelle d'intention de transformation, associée à un ensemble de propriétés d'intention caractérisant précisément chaque intention du point de vue de la vérification. Ce Cadre est une contribution intéressante visant à poser les bases d'une véritable ingénierie de la vérification dans le cadre de l'IDM. Pour répondre au second défi, nous proposons de réduire la complexité de la vérification de LMDs par la définition de ponts sémantiques au niveau des langages de transformation au lieu des LMDs. Nous avons montré la faisabilité de l'approche en appliquant cette proposition à Kermeta, un langage métaprogrammé de transformation, et en construisant un prototype permettant le *model-checking* et le *theorem-proving* de transformations écrites dans ce langage.

Chaque contribution a été validée par plusieurs études de cas et applications pratiques, mais il reste encore plusieurs points à améliorer. Nous sommes actuellement en train de valider empiriquement, de manière plus rigoureuse, notre catalogue d'intentions afin d'évaluer la portée de son pouvoir de classification, et nous allons documenter davantage d'intentions en sus de celles déjà décrites dans la thèse. En outre, nous allons chercher à automatiser au maximum le processus de concrétisation de propriétés d'intentions en propriétés de transformations. Nous améliorons aussi KMV afin de gagner en performance d'exécution et de vérification, et de mieux exploiter les diagnostics retournés par Maude afin de refléter les erreurs au niveau de la spécification en Kermeta, et de l'intégrer de manière plus transparente à l'éditeur natif de Kermeta.

Notre thèse à ouvert plusieurs perspectives de travail, tant du point de vue méthodologique que pratique. Nous concentrerons nos efforts à l'extension du Langage d'Action servant de représentation intermédiaire à Kermeta : celui-ci ne lui est pas spécifique, mais peut servir de pivot langagier à de nombreux langages de transformation orientés objet (par exemple, Epsilon ou la partie impérative de fUML), ouvrant la possibilité de fournir des outils d'analyse formelle à plusieurs langages de transformation. En outre, tout nouveau domaine sémantique pourra être réutilisé par l'ensemble de ces langages au prix d'une compilation vers cette représentation intermédiaire. En outre, il serait intéressant d'enrichir Kermeta avec des constructions temps-réel, pour pouvoir définir la sémantique de LMDs temporisés. Notre représentation intermédiaire va refléter ces constructions, ouvrant à nouveau la possibilité d'analyser d'autres langages de transformation temps-réel, et notamment les standards SysML ou MARTE.

Références

[1] Moussa Amrani. A Formal Semantics of Kermeta. In Marjan Mernik, editor, *Formal and Practical Aspects of Domain-Specific Languages : Recent Developments*, chapter 10, pages 270–309. IGI Global, Hershey, PA (USA), September 2013.

[2] Moussa Amrani. *Towards The Formal Verification of Model Transformations — An Application to Kermeta*. PhD thesis, University of Luxembourg, 2013.

[3] Moussa Amrani, Benoît Combemale, Pierre Kelsen, and Yves Le Traon. Une Revue des Techniques de Vérification Formelle pour la Transformation de Modèles : Une Classification Tridimensionnelle. In *Actes de la Conférence en Ingénierie du Logiciel*, 2014.

[4] Moussa Amrani, Jürgen Dingel, Leen Lambers, Levi Lúcio, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Towards a Model Transformation Intent Catalog. In *Analysis of Model Transformation Workshop*, 2012.

[5] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Verification of Model Transformations Workshop*, 2012.

[6] Moussa Amrani, Levi Lúcio, Gehan Selim, Benoît Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Survey of Formal Verification Techniques for Model Transformations : A Tridimensional Classification. *Journal of Technology*, 2014.

[7] Benoit Combemale, Xavier Crégut, Pierre-Loïc Garoche, and Xavier Thirioux. Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. *Journal of Software*, 4(9) :943–958, November 2009.

[8] Jean Bézivin, Fabian Büttner, Martin Gogolla, Frédéric Jouault, Ivan Kurtev, and Arne Lindow. Model Transformations ? Transformation Models ! In *Conference on Model Driven Engineering Languages and Systems*, 2006.

[9] Jordi Cabot, Robert Clarisó, Esther Guerra, and Juan de Lara. Verification and Validation of Declarative Model-to-Model Transformations Through Invariants. *Journal of Systems and Software*, **83**(2) :283–302, 2010.

[10] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, Narciso Marti Oliet, Jose Meseguer, and Carolyn Talcott. *All About* Maude. *A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science (*Lncs*)*. Springer, July 2007.

[11] Krzysztof Czarnecki and Simon Helsen. Feature-Based Survey of Model Transformation Approaches. Ibm *Systems Journal*, **45**(3) :621–645, 2006.

[12] Dániel Varró and András Pataricza. Automated Formal Verification of Model Transformations. In *Workshop on Critical Systems Development in UML*, pages 63–78, 2003.

[13] Zoé Drey, Cyril Faucher, Franck Fleurey, Vincent Mahé, and Didier Vojtisek. *Kermeta Language — Reference Manual*. University of Rennes, Triskell Team, April 2009.

[14] Levi Lúcio, Bruno Barroca, and Vasco Amaral. A Technique for Automatic Validation of Model Transformations. In *Conference on Model Driven Engineering Languages and Systems*, 2010.

[15] Levi Lúcio, Moussa Amrani, Jürgen Dingel, Leen Lambers, Rick Salay, Gehan Selim, Eugene Syriani, and Manuel Wimmer. Model Transformation Intents and Their Properties. *Journal of Software And Systems*, 2014.

[16] Tom Mens and Pieter Van Gorp. A Taxonomy Of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152 :125–142, 2006.

[17] Anantha Narayanan and Gabor Karsai. Verifying Model Transformation By Structural Correspondence. *Workshop on Graph Transformation and Visual Modeling Techniques*, 10 :15–29, 2008.

[18] Object Management Group. Meta-Object Facility 2.0 Core Specification (06-01-01). Technical report, Object Management Group, 2006.

[19] José E. Rivera, Francisco Durán, and Antonio Vallecillo. Formal Specification and Analysis of Domain-Specific Models Using Maude. *Simulation : Transactions of the Society for Modeling and Simulation International*, 85(11/12) :778–792, November 2009.

[20] Jesús Sánchez Cuadrado, Esther Guerra, and Juan de Lara. Generic Model Transformations : *Write Once, Reuse Everywhere*. In *International Conference on Theory and Practice of Model Transformations*, Lecture Notes in Computer Science. Springer, 2011.

[21] Eugene Syriani and Hans Vangheluwe. De-/Re-constructing Model Transformation Languages. *Workshop on Graph Transformation and Visual Modeling Techniques*, 29 :1–14, 2010.

[22] Triskell Team. The Kermeta Website `http://www.kermeta.org`.

[23] Glynn Winskel. *The Formal Semantics of Programming Languages : An Introduction (Foundations of Computing)*. MIT Press, Cambridge, MA (USA), February 1993.

Session 3 commune CAL / CIEL

Cycle de vie des architectures logicielles

Service Identification Based on Quality Metrics Object-Oriented Legacy System Migration Towards SOA

Seza Adjoyan, Abdelhak Seriai and Anas Shatnawi

Migrating towards Service Oriented Architecture SOA has become a major topic of interest during the recent years. Since emerging service technologies have forced non-service based software systems to become legacy, many efforts and researches have been carried out to enable these legacy systems to survive. In this context, several service identification solutions have been proposed. These approaches are either manual, thus considered expensive, or rely on ad-hoc criteria that fail to identify relevant services. In this paper, within the big picture of migrating object-oriented legacy systems towards SOA, we address automatic service identification from source code based on service quality characteristics. To achieve this, we propose a quality measurement model where characteristics of services are refined to measurable metrics. The approach was validated via two realistic case studies and has produced satisfying results.

Un modèle de composants unifié pour l'évolution dynamique des systèmes logiciels

Salim Kebir[1,2,3] and Djamel Meslati[3]

[1] Ecole préparatoire aux sciences et techniques, Annaba, Algérie
[2] Ecole Nationale Supérieure d'Informatique, Alger, Algérie
[3] Laboratoire d'ingénierie des systèmes complexes, Université d'Annaba, Algérie
s_kebir@esi.dz djamel_meslati@yahoo.fr

Résumé

Dans ce papier nous proposons un modèle de composants appelé *composant universel* pour l'évolution dynamique des systèmes logiciels ainsi qu'une plateforme d'exécution d'applications qui s'appuie sur ce modèle. Pour cela, nous partons de l'hypothèse que n'importe quelle constituante d'un système, quelle que soit sa granularité, fournit une donnée et/ou exécute un traitement et peut ainsi être abstraite en composant logiciel. Par conséquent, l'évolution dynamique d'un système peut être exprimée en terme d'ajout/suppression/remplacement de composants/connexions.

Mots-clés : Evolution dynamique des logiciels, composants logiciels, modèle de composant, composant universel.

Abstract

In this paper, we propose a component model called *universal component* for the dynamic software evolution together with an execution platform that relies on this model. To achieve this, we assume that any part of a software system, whatever its granularity, can provide data and/or perform an action and can thus be described as a software component. Therefore, dynamic software evolution can be expressed in term of adding/removing/replacing components and connections.

Keywords : Dynamic software evolution, software components, component model, universal component.

1 Introduction

Mentionnée pour la première fois durant les années 1970, l'expression *évolution des logiciels* est confondue jusqu'à nos jours avec l'expression *maintenance des logiciels* [1][2] D'après [1], ces deux disciplines du génie logiciel - même si de prime abord elles ont les mêmes impacts économiques, organisationnels et techniques - sont de nature différente : la maintenance des logiciels a une connotation négative qui indique que le logiciel est en train de vieillir et de se détériorer tandis que l'évolution a une connotation positive qui indique que le logiciel s'adapte, s'améliore, s'octroie de nouvelles fonctionnalités et évolue vers une nouvelle version [3].

L'évolution dans les systèmes logiciels critiques ou de très large échelle se heurte à un problème majeur. D'un coté les systèmes logiciels dits critiques (e.g. Systèmes bancaires, serveurs web, SGBDs) doivent fonctionner de manière continue et inintérruptibles. D'un autre côté, dans les systèmes logiciels de très large échelle, l'arrêt, la modification, la recompilation et la rééxecution peuvent conduire à des coûts considérables. Dans de tels systèmes, l'évolution doit agir de manière dynamique sans perturber l'exécution.

Dans ce papier, nous proposons un modèle de composants appelé *composant universel* qui prend en compte l'évolution dynamique des logiciels. Dans la section 2, nous présentons notre modèle de composants. La section 3 illustre l'utilisation de notre modèle à travers un exemple réel. La section 4 discute des travaux connexes. Enfin, la section 5 conclut le papier.

2 Approche

2.1 Hypothèses

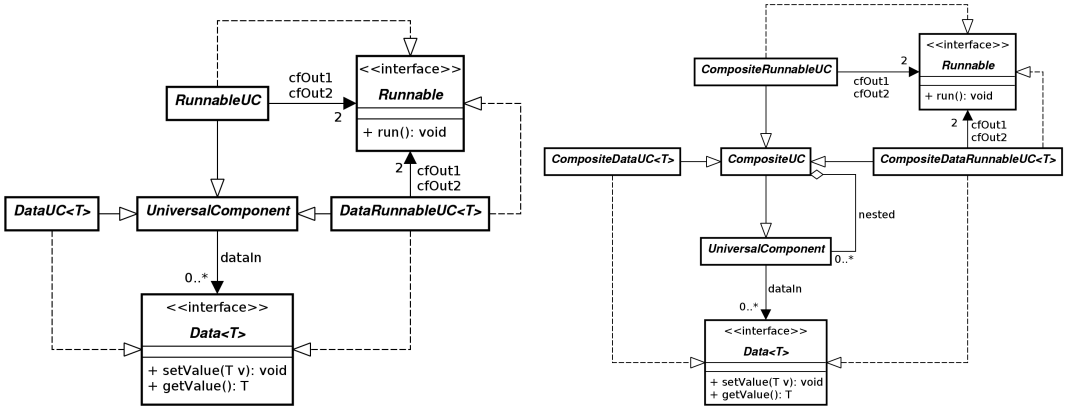
Nous partons de l'hypothèse que n'importe quelle application peut être construite en assemblant des composants logiciels. Cette hypothèse s'appuie sur les définitions d'un composant logiciel les plus couramment admises dans la littérature. Schmidt [4] définit un composant comme *une instance autonome d'un type de donnée qui peut être assemblée avec d'autres composants pour former une application.* Cette définition souligne qu'un composant peut abstraire une donnée et que d'autres composants de l'application peuvent accéder à ou modifier cette donnée. Dans [5], un composant est défini comme *une entité logicielle exécutable qui peut être assemblée dynamiquement.* Cette définition considère qu'un composant, une fois déployé et assemblé, est amené à exécuter un traitement. Enfin, Szyperski [6] définit un composant logiciel comme *une unité de composition avec des interfaces contractuellement spécifiées et des dépendances explicites au contexte. Il peut être déployé indépendamment et peut être sujet d'assemblage par un tiers.* La première partie de la définition indique que les interfaces fournies et requises sont le moyen par lequel les composants sont assemblés. La seconde partie de la définition insiste sur le caractère composite des composants qui une fois assemblés, peuvent donner naissance à une application ou à de nouveaux composants.

En combinant et en raffinant les éléments communs à ces définitions, nous proposons la définition suivante d'un composant logiciel : **Un composant est une entité logicielle qui peut encapsuler une donnée et/ou exécuter un traitement. Il peut être assemblé dynamiquement avec d'autres composants au moyen d'interfaces clairement définies au préalable pour donner naissance à un composant composite ou à une application.**

2.2 Le composant universel

Le composant universel est un modèle de composants basé sur la définition que nous avons donnée dans la section précédente. Les instances de ce modèle sont appelées "composants universels" (UC). Un composant universel peut fournir une donnée, accéder aux données d'autres composants et exécuter un traitement. Grâce à ces trois propriétés, un composant universel peut encapsuler n'importe quelle constituante d'un système, quelle que soit sa granularité. Le modèle de composant universel repose sur le méta-modèle exprimé par un diagramme de classes UML dans la figure 1a.

Dans la deuxième partie de la définition que nous avons donnée dans la section précédente, nous avons dit qu'un composant peut former avec d'autres composants une structure plus complexe appelée composant composite. Le modèle de composant universel permet de créer de telles entités. Pour atteindre cela, nous avons étendu le méta-modèle de la figure 1a pour prendre en charge les composants composites. Cette extension se traduit par l'utilisation du patron de conception composite (figure 1b) où les instances de la classe *CompositeUC* peuvent contenir des instances de la classe *UniversalComponent* qui peuvent à leur tour être des instances de la classe *CompositeUC*.



(a) Composants universels simples (b) Extension pour les composites

Figure 1: Méta-modèle du composant universel

Un composant universel possède des interfaces fournies et requises pour les données (Data) et l'exécution (Runnable). Ces interfaces correspondent aux entrées et sorties de part et d'autre des flux de données et d'exécution. La figure 2 donne une vue abstraite d'un composant universel. À l'image d'un composant électronique, un composant universel peut être assemblé avec d'autres composants à l'aide des ports DIn,$Dout$,$CFIn$ et $CFOut$.

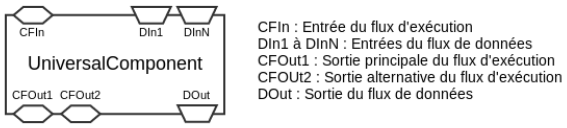


Figure 2: Vue abstraite du composant universel

2.3 Implémentation

Nous avons implémenté un prototype du composant universel en nous basant sur le modèle de composant JavaBeans. Ce prototype fournit une instance du méta-modèle de la figure 1 et permet la création de composants données et/ou exécutable simples ou composites. La correspondance entre un composant universel et un JavaBean n'est pas triviale, pour cela nous avons établi une correspondance entre les entités du modèle de composants universels et les entités du modèle de composants JavaBeans comme suit :

— **D'un point de vue structurel**, un composant universel simple correspond à un objet JavaBean et ses ports correspondent à des interfaces fournies et requises. Notre implémentation permet de traiter de manière homogène les composants simples et composites en utilisant le patron de conception composite.

— **D'un point de vue comportemental**, d'un coté, la transmission du flux de données correspond à des appels de méthodes entre des objets JavaBeans. D'un autre côté, la transmission du flux d'exécution correspond au déclenchement d'une action suite à l'arrivée d'un évènement. Nous avons implémenté ce mécanisme en utilisant le patron

de conception observer, où chaque composant joue le rôle à la fois d'un observateur et d'un émétteur d'évènements, un évènement étant le flux d'exécution.

— **D'un point de vue créationnel**, l'installation (resp. la désinstallation) d'un composant correspond au chargement (resp. déchargement) dynamique d'une classe Java et l'appel de son constructeur (resp. destructeur).

2.4 Granularité d'un composant universel

La granularité d'un composant universel est très fine. Elle peut aller d'une simple variable à un ensemble de fonctions. Afin d'illustrer cela, nous avons developpé une librairie de composants réutilisables pour abstraire les constituantes les plus couramment utilisées d'une système, à savoir les variables, les opérations arithmétiques et logiques, les structures conditionnelles, les fonctions, les blocs d'instructions. La figure 3 présente l'utilisation des composants universels pour construire une boucle *while* qui affiche la valeur d'une variable et l'incrémente jusqu'à ce qu'elle atteigne la valeur 10.

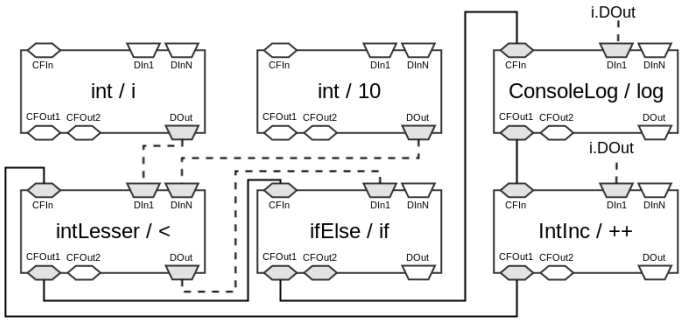


Figure 3: Boucle while represéntée par des composants universels

2.5 Support pour l'évolution dynamique

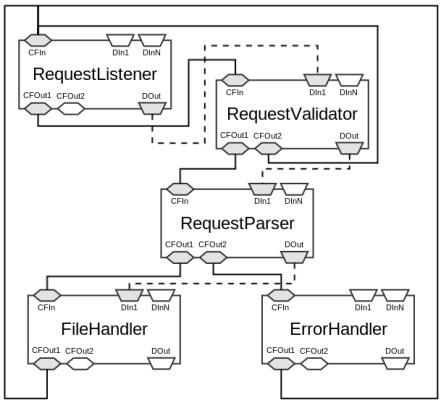
Afin de faire évoluer dynamiquement un programme conçu avec des composants universels, nous proposons la possibilité d'exécuter des actions sur ses composants. Nous appelons l'ensemble des actions exécutées lors d'une évolution dynamique un *patch*. Un patch est composé d'opérations élémentaires permettant d'installer, désinstaller, créer, supprimer, connecter et déconnecter les composants. Afin de s'assurer qu'un patch s'exécute correctement, notre approche vérifie les deux propriétés suivantes :

— **La complétude du patch** : la propriété de complétude s'assure que le patch s'exécute au complet ou pas du tout. Pour cela, nous utilisons un mécanisme de journalisation et de rollbacks des actions qui se sont produites sur le système. Ce mécanisme permet d'annuler l'effet d'une action si une erreur se produit pendant l'exécution du patch.

— **La consistance du patch** : la propriété de consistance s'assure que l'état du système soit consistent. Pour cela, nous vérifions que tous les composants concernés par le patch sont dans un état passif, c'est-à-dire que le flux de contrôle et de données n'est détenu par aucun d'entre eux. Si c'est le cas, l'utilisateur peut retarder ou annuler le patch.

3 Exemple d'une evolution dynamique

L'application exemple que nous avons choisie pour illustrer le déroulement d'une évolution dynamique est un serveur web que nous avons développé en utilisant le modèle de composants universels. Il est composé de 5 composants universels dont le rôle est décrit dans la figure 4.



RequestListener (rl) : Ecoute le port 80, et transmet le flux d'exécution dès qu'une requête arrive.

RequestValidator (rv) : S'assure que la requête est bien une requête HTTP valide et transmet le flux d'execution sur CFOut1 si c'est le cas.

RequestParser (rp) : Extrait le nom du fichier demandé par la requête. Si le fichier existe, il transmet le flux d'exécution sur CFOut1 sinon, il le transmet sur CFOut2.

FileHandler (fh) : Lit le fichier à partir de la mémoire secondaire et l'envoie au client

ErrorHandler (eh) : Envoie un message d'erreur au client

Figure 4: Application exemple

Afin d'illustrer un exemple d'évolution dynamique sur ce système, nous décidons d'introduire un nouveau composant universel qui met en cache les requêtes les plus fréquentes pour éviter de faire des lectures en mémoire secondaire. Le patch qui permet de faire cette évolution est le suivant :

```
1. install FileCacheHandler.jar
2. create FileCacheHandler fch
3. disconnect fh.cfin
4. disconnect fh.din1
5. connect fch.cfin rp.cfout1
6. connect fch.din1 rp.dout
7. connect fh.cfin fch.cfout1
8. connect fh.din1 fch.dout
```

Ce patch charge la classe du nouveau composant à partir d'un fichier jar et le crée (1 et 2). Puis déconnecte les composants FileHandler et RequestParser (3 et 4). Ensuite connecte le nouveau composant avec RequestParser (5 et 6) et FileHandler (7 et 8).

4 Travaux connexes

Plusieurs travaux se sont intéressés aux approches basées composants logiciels pour l'évolution dynamique des systèmes. Wang et. al. [7] proposent une approche basée sur un serveur d'applications qui supporte la mise à jour dynamique à travers l'interception des messages échangés entre composants. Dowling et al. [8] proposent le modèle de composant K-Component. Ce modèle permet de faire évoluer dynamiquement un système via des transformations de graphes. Les approches proposées dans [9], [10] et [11] considère l'évolution dynamique comme une préoccupation transversale et utilisent la programmation orientée aspects pour la supporter.

En comparaison avec les approches [7][10] et [11], notre approche offre une granularité variable de la notion de composant (e.g. un composant peut aller d'une simple instruction à un serveur web) qui permet de diminuer l'impact de changement. Même si nous avons implémenté notre approche avec JavaBeans, le modèle que nous proposons est une spécification générique contrairement aux approches [9][8]. Par ailleurs, dans notre approche, les composants ne peuvent communiquer qu'à travers les ports DIn,$Dout$,$CFIn$ et $CFOut$, ce qui offre un faible degré de couplage et un haut degré d'homogénéité. Enfin, nous avons montré que le modèle de composants universels est facilement extensible à travers un mécanisme d'héritage et d'implémentation.

5 Conclusion

Dans ce papier, nous avons présenté une approche d'évolution dynamique basée sur les composants logiciels. Dans un premier temps, nous avons présenté la spécification et l'implémentation d'un modèle de composants générique appelé "composant universel". Ensuite, nous avons présenté un exemple qui illustre comment le modèle de composant universel permet l'évolution dynamique à travers l'ajout et la suppression de composants et de connexions. Dans le futur, nous prévoyons d'enrichir la bibliothèque des composants sur étagères offerts par notre approche et de concevoir une approche de réingénierie d'applications patrimoniales en composants universels.

Références

[1] T. Mens. Introduction and roadmap : History and challenges of software evolution. In *Software Evolution*, pages 1–11. Springer Berlin Heidelberg, 2008.

[2] M.M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9) :1060–1076, Sept 1980.

[3] M.W. Godfrey and D.M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 129–138, Sept 2008.

[4] D.C. Schmidt. Why software reuse has failed and how to make it work for you. *C++ Report*, 11(1) :1999, 1999.

[5] S. De Cesare, M. Lycett, and R.D. Macredie. *Development Of Component-based Information Systems (Advances in Management Information Systems).* M. E. Sharpe, Inc., Armonk, NY, USA, 2005.

[6] C. Szyperski. *Component Software : Beyond Object-Oriented Programming.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[7] Q. Wang, J. Shen, X. Wang, and H. Mei. A component-based approach to online software evolution : Research articles. *J. Softw. Maint. Evol.*, 18(3) :181–205, May 2006.

[8] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns*, REFLECTION '01, pages 81–88, London, UK, UK, 2001. Springer-Verlag.

[9] C. Costa-Soria, J. Perez, and J.A. Carsi. Handling the dynamic reconfiguration of software architectures using aspects. In *Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on*, pages 263–266, March 2009.

[10] P.C. David and T. Ledoux. An aspect-oriented approach for developing self-adaptive fractal components. In *Software Composition*, pages 82–97, 2006.

[11] N. Pessemier, L. Seinturier, L. Duchien, and T. Coupaye. A component-based and aspect-oriented model for software evolution. *IJCAT*, 31(1/2) :94–105, 2008.

Génération de métaprogrammes Java à partir de contraintes architecturales OCL

Sahar Kallel[1,2], Chouki Tibermacine[1], Mohamed Reda Skay[1], Christophe Dony[1] and Ahmed Hadj Kacem[2]

[1] LiRMM, CNRS, Université Montpellier 2, France
Chouki.Tibermacine@lirmm.fr
dony@lirmm.fr
mohamedreda.skay@gmail.com
[2] ReDCAD, Université de Sfax, Tunisie
sahar.kallel@lirmm.fr
ahmed.hadjkacem@fsegs.rnu.tn

Résumé

Afin d'expliciter et de rendre tangibles leurs choix de conception, les concepteurs des logiciels intègrent aux modèles des leurs applications, des contraintes que ces modèles puis leurs implantations devront vérifier. Différents environnements existants permettent la vérification de contraintes durant la phase de modélisation, mais dans la plupart des cas, ils ne considèrent pas la génération de code qui permettrait de continuer à vérifier ces contraintes durant la phase d'implémentation. Hors il s'avère que c'est possible dans un certain nombre de cas. Les environnements qui offrent cette fonctionnalité l'offrent uniquement pour les contraintes fonctionnelles (portant sur l'état des objets de l'application) et pas pour les contraintes architecturales (portant sur la structure de l'application). Ayant constaté ces limitations, nous décrivons dans cet article un système qui associe à un modèle UML d'une application, des contraintes architecturales, écrites en OCL, puis de générer un métaprogramme permettant la vérification de ces contraintes à l'exécution. Les contraintes architecturales font références aux éléments du métamodèle UML utilisés pour décrire les modèles des applications.
Mots clés : architecture logicielle, contrainte d'architecture, OCL, Java Reflect

1 Introduction

L'architecture est la base de la conception d'une application [5]. Elle nous donne un aperçu général de l'organisation de l'application qui nous aide à vérifier certaines propriétés, comme les attributs de qualité. Dans ce contexte, les langages de description d'architectures ont été créés pour spécifier, décrire et vérifier de telles architectures d'applications sans qu'il soit nécessaire de se préoccuper, dans un premier temps, de l'implémentation de leurs fonctionnalités. La vérification se base notamment sur les contraintes que ces langages permettant d'associer aux architectures.

OCL [20] est un langage d'expression de contraintes connu, bien outillé et facile à apprendre et à utiliser [8]. En effet, il est le standard de l'OMG (Object Management Group) et il a pour objectif de spécifier les contraintes fonctionnelles et architecturales en phase de conception, les rendant ainsi vérifiables sur les modèles d'une application. Les contraintes OCL sont écrites relativement à un contexte. Ce contexte est un élément d'un modèle UML, le plus souvent une classe, une opération ou une association présente dans le diagramme. Nous pouvons distinguer deux catégories de contraintes OCL, les contraintes fonctionnelles qui sont appliquées sur un contexte d'un modèle précis et les contraintes d'architecture [24] qui sont écrites sur

un métamodèle donné. Par exemple, dans un diagramme de classe, où nous retrouvons une classe *Employé*, ayant comme attribut *age* de type entier, une contrainte fonctionnelle OCL représentant un invariant sur cette classe peut tester les valeurs de cet attribut pour qu'ils soient toujours compris dans l'intervalle 16 à 70. Cette contrainte sera vérifiée sur toutes les instances du modèle UML, et donc sur toutes les instances de la classe *Employé*. Dans ce qui suit, nous allons nous concentrer sur les contraintes architecturales. Ces contraintes sont écrites non pas sur un modèle mais sur un métamodèle. Elles permettent de spécifier de façon formelle ou informelle certaines décisions de conception comme le choix d'un patron ou d'un style architectural. Nous pouvons dire que ces contraintes ne relèvent pas du niveau fonctionnel (contraintes sur les valeurs des attributs, par exemple), elles relèvent plutôt du niveau structurel, et donc architectural. La spécification de ce type de contraintes en phase d'implémentation peut se faire manuellement. Cependant, celles-ci peuvent être également obtenues automatiquement en transformant les contraintes spécifiées lors de la phase de conception. En effet, dans ces deux phases de développement ces contraintes sont écrites avec deux formalismes différents, qui sont interprétables indépendamment, alors qu'elles ont la même sémantique, et qu'elles peuvent être dérivées les unes des autres. Dans l'ingénierie logicielle, la plupart des générateurs de code font des transformations de modèles en code source sans prendre en compte les contraintes spécifiées sur ces modèles. Il existe un nombre limité d'outils qui ont été proposés pour générer du code à partir de contraintes OCL. Ils sont tous utilisés pour des contraintes fonctionnelles et non architecturales. Les travaux sur les contraintes d'architecture [24] sont récents et portent surtout sur l'évolution de l'application.

L'objectif de ce papier est de traduire les contraintes architecturales écrites en OCL en métaprogrammes Java. Nous désignons par un métaprogramme un programme qui utilise le mécanisme d'introspection (pas nécessairement d'intercession) du langage de programmation pour implémenter une contrainte d'architecture. En utilisant ce mécanisme, les contraintes d'architecture seront évaluées à l'exécution des programmes sur lesquels les contraintes sont vérifiées. Ceci n'est pas nécessaire pour certaines catégories de contraintes d'architecture dans lesquelles une analyse statique de l'architecture suffit. Dans certains cas, la contrainte doit vérifier les types des instances créées à l'exécution et leurs interconnexions, elle doit donc être évaluée dynamiquement.

Dans la suite de cet article, la section 2 sera consacrée à un exemple illustratif de notre travail. La section 3 portera sur l'approche générale que nous avons adoptée pour atteindre notre objectif. La section 4 présente les métamodèles UML et Java sur lesquels nous pouvons exprimer les contraintes architecturales et connaitre le principe de raffinement et de transformation qui seront détaillés respectivement dans les sections 5 et 6. La section 7 détaille la partie génération de métaprogrammes. Enfin, nous présenterons les travaux connexes dans la section 8 et dans la section 9 nous conclurons et discuterons les perspectives.

2 Exemple illustratif

Nous prenons l'exemple des contraintes représentant l'architecture du patron MVC (Model-View-Controller) [23]. Ces contraintes peuvent être traduites ainsi:

- Le modèle ne doit pas connaître ses vues; en pratique ceci revient à dire qu'il ne doit avoir aucune connaissance de la façon dont ses vues lui sont associées.

- Le modèle ne possède pas de références du contrôleur. Ceci permet d'avoir plusieurs contrôleurs possibles pour le modèle.

Une modélisation possible du patron MVC dans une application donnée consiste à utiliser les stéréotypes. Nous supposons donc que nous disposons de trois stéréotypes nous permettant de désigner les classes dans une application qui représentent la vue (*View*), le modèle (*Model*) et le contrôleur(*Controller*).

Les contraintes d'architecture peuvent être traduites en OCL de la manière suivante :

```
1 context Class inv :
2 self.package.profileApplication.appliedProfile.ownedStereotype
3    -> exists(s:Stereotype | s.name='Model')
4 implies
5      self.supplierDependency.client
6    -> forAll(t: Type | not(t.oclAsType(Class)
7       .package.profileApplication.appliedProfile.ownedStereotype
8       ->exists(s:Stereotype | s.name='View' or s.name='Controller')
9                          )
10             )
```

Listing 1: Contraintes du patron MVC en OCL

La première ligne du listing 1 déclare le contexte de la contrainte. Elle indique que la contrainte s'applique à toute classe de l'application. Les lignes 2 et 3 récupèrent l'ensemble de classes qui représentent le modèle (annotées par *@Model*) en utilisant la navigation *package.profileApplication.appliedProfile.ownedStereotype*[1]. La ligne 5 représente l'ensemble des classes ayant une dépendance directe avec le contexte en utilisant *self.supplierDependency.client*. Le reste de la contrainte permet de parcourir ce dernier ensemble et vérifier qu'il ne comporte pas des classes annotées *View* ou *Controller*.

Notre objectif est d'obtenir un métaprogramme traduit automatiquement à partir des contraintes OCL. Les contraintes du patron MVC peuvent être exprimées en code Java de la façon suivante :

```
1 public boolean invariant(Class<?> uneClasse) {
2       if(uneClasse.isAnnotationPresent(Model.class)) {
3           Field [ ] attributs = uneClasse.getDeclaredFields();
4           for(Field unAttribut : attributs) {
5               if(unAttribut.getType().isAnnotationPresent(View.class)
6               || unAttribut.getType().isAnnotationPresent(Controller.class))
7                   return false;
8          }
9      }
10   return true;
11 }
```

Listing 2: Contrainte du patron MVC en Java

Le listing 2 représente une méthode Java qui utilise la librairie *Reflect*. Nous utilisons *getDeclaredFields()* (ligne 3) pour collecter l'ensemble des attributs et *isAnnotationPresent(..)* pour vérifier si l'attribut possède une annotation particulière.

[1]Le métamodèle UML nous permet d'appliquer une stéréotype uniquement au niveau package et non au niveau entité. Le problème est résolu dans certains outils comme RSA-IBM dans lequel il dispose d'une méthode *getAppliedStereotypes()*, elle s'applique directement à un élément du modèle et non à son package.

3 Approche générale

Le but de notre travail est de proposer une méthode qui permet la génération de code à partir de contraintes architecturales. Dans ce contexte nous avons choisi UML comme le langage de modélisation et Java comme langage de programmation. Des contraintes écrites en OCL sur le métamodèle UML est le point de départ de notre méthode. Le résultat voulu est un métaprogramme Java qui spécifie la contrainte de départ et utilise la librairie *Reflect* de Java. La figure 1 décrit l'approche générale proposée dans cet article.

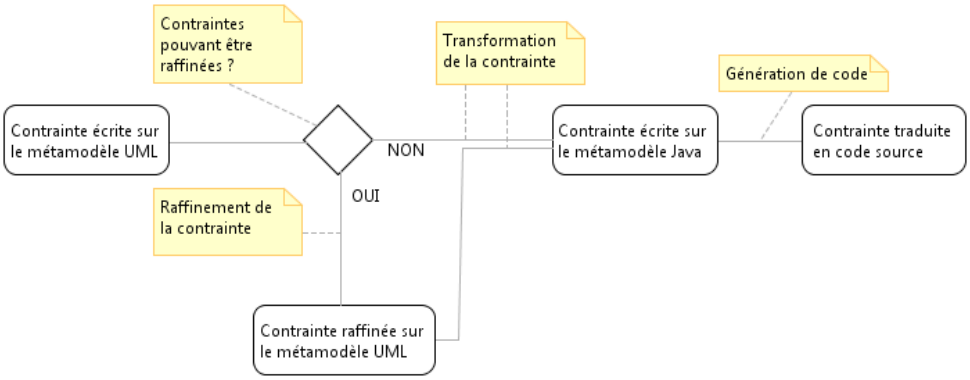


Figure 1: Description du travail

La figure 1 montre que notre travail englobe trois étapes. Si la contrainte nécessite un raffinement[2] alors la première étape consiste à la transformer tout en restant dans le métamodèle UML (par exemple, une contrainte possède une navigation vers la métaclasse *Dependency* dans le métamodèle UML doit être raffinée afin de présenter les différents niveaux de dépendances). Sinon, une étape de transformation des contraintes est mise en place afin d'arriver à la fin à la partie de la traduction syntaxique c'est à dire la génération de code. En effet, notre approche nécessite tout d'abord une projection des abstractions de niveau conception vers des abstractions de niveau implémentation (projeter les abstractions dans le métamodèle UML vers le métamodèle Java) et par la suite traduire la syntaxe.

4 Métamodèles utilisés dans les transformations

Pour effectuer le raffinement et la transformation de contraintes, nous avons besoin de décrire les métamodèles UML et Java afin de fixer par la suite les règles de transformation des contraintes d'un métamodèle à l'autre.

4.1 Métamodèle UML

Dans cet article nous nous sommes intéressés à la partie des classes (Chapitre 7 de la spécification la superstructure du langage UML, version 2.4.1) du métamodèle UML défini

[2]Le terme raffinement désigne une spécialisation d'un niveau abstrait vers un autre concret.

dans la spécification UML [22]. Cependant il existe des entités de ce métamodèle dont nous pouvons nous passer dans notre spécification de contraintes d'architecture. Ainsi nous avons décidé de le reprendre et supprimer les entités dispensables dans notre travail afin de rendre les contraintes moins verbeuses. La figure 2 décrit le métamodèle simplifié d'UML.

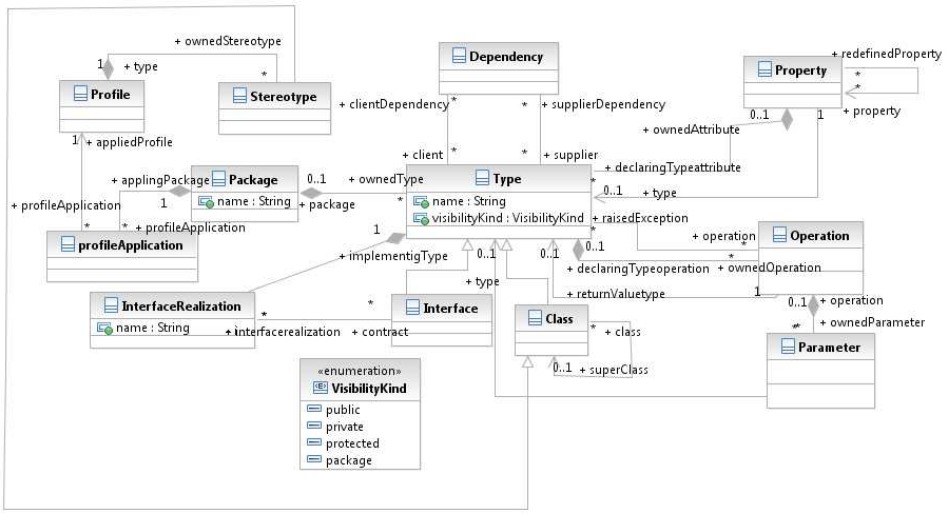


Figure 2: Extrait du métamodèle UML modifié

Ce métamodèle se focalise sur la description des classes et notamment sur la description des packages, des attributs, des méthodes, des dépendances et des profils. Un package est composé d'un certain nombre de types. Un type peut avoir la capacité de participer dans des dépendances. A droite de la figure, il est indiqué qu'un type (*Class* ou *Interface*) peut déclarer des attributs qui sont des instances de *Property* et des méthodes qui sont des instances d'*Operation*. Ces opérations peuvent avoir des paramètres et des types de retour. La partie gauche de la figure illustre le fait que nous pouvons appliquer un profile à un package, et qu'un profile est constitué d'un certain nombre de stéréotypes.

4.2 Métamodèle Java

Puisque le but est de générer du code source Java, il a fallu créer un métamodèle Java. Le langage Java fournit le mécanisme d'introspection, nous nous sommes alors appuyés sur la librairie *Reflect* [2] pour créer un métamodèle Java simplifié. Nous aurions pu définir notre métamodèle en s'appuyant sur une autre source documentaire (à partir de la spécification du langage Java par exemple). Mais nous avons volontairement choisi la librairie *Reflect* parce qu'elle nous donne accès au niveau méta du langage et aussi parce qu'elle reflète exactement ce que nous pouvons faire dans le code Java généré. Nous nous sommes contentés des entités permettant l'écriture des contraintes d'architecture. La figure 3 décrit le métamodèle Java que nous avons défini.

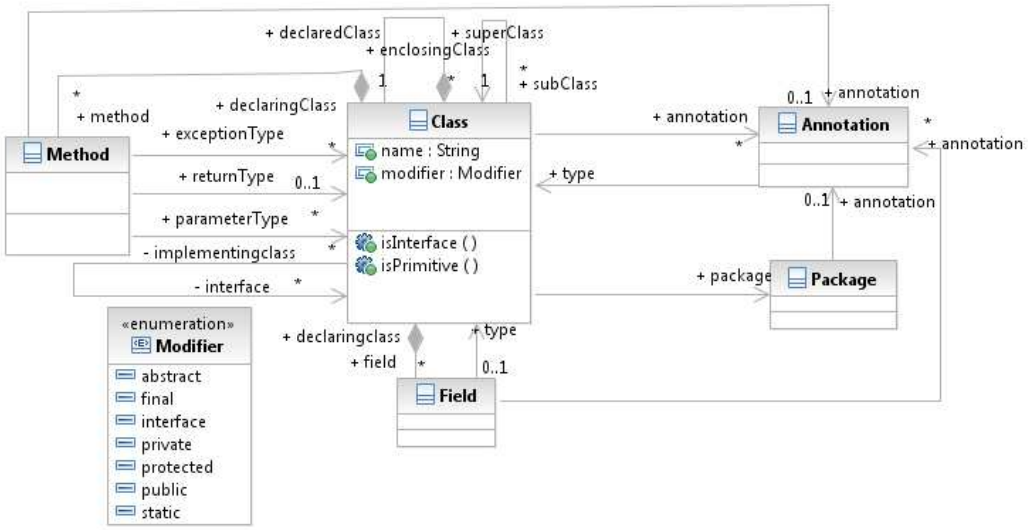


Figure 3: Extrait d'un métamodèle Java

Dans la figure 3, il est indiqué qu'une classe peut être définie dans un package. De plus, une classe peut contenir des attributs, qui sont des instances de la métaclasse *Field* et des méthodes qui sont des instances de la métaclasse *Method*. Nous pouvons remarquer également que *Class*, *Field* et *Method* peuvent être annotés par plusieurs annotations, qui sont des instances de la métaclasse *Annotation*.

Après avoir étudié les métamodèles UML et Java, nous allons par la suite expliquer le principe de raffinement des contraintes d'architecture exprimées en OCL. Ce type de transformations est dit endogène puisque les contraintes sources et cibles du raffinement sont écrites sur le même métamodèle UML.

5 Raffinement des contraintes

Dans une contrainte OCL exprimée dans le métamodèle UML (voir figure 2), nous pouvons récupérer tous les types (classe, interface) qui ont une dépendance à un autre type spécifique en utilisant par exemple "*supplierDependency.client*". Cette expression n'a pas d'équivalence directe dans Java et par conséquent, il faut raffiner la contrainte sur le métamodèle UML en exprimant les différentes catégories de dépendances pour pouvoir la transformer d'une façon précise.

Souvent une dépendance entre deux classes se traduit par la déclaration dans la première classe d'au moins un attribut ayant comme type la deuxième classe. Nous pouvons également retrouver des paramètres dans les opérations de la première classe qui ont comme type la deuxième classe. Ou encore, dans une opération de la première classe, le type de retour est la deuxième classe.

Si nous reprenons notre exemple de la section 2, le listing 1 peut être raffiné de la façon suivante :

```
1 context Class inv :
2  self.package.profileApplication.appliedProfile.ownedStereotype
3  -> exists(s:Stereotype|s.name='Model') implies
4   self.ownedAttribute.type
5   -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
6     appliedProfile.ownedStereotype
7     ->exists(s:Stereotype | s.name='View' or s.name='Controller')))
8  and self.ownedOperation.returnValuetype
9   -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
10    appliedProfile.ownedStereotype
11    ->exists(s:Stereotype | s.name='View' or s.name='Controller')))
12 and self.ownedOperation.ownedParameter.type
13  -> forAll(t: Type | not(t.oclAsType(Class).package.profileApplication.
14    appliedProfile.ownedStereotype
15    ->exists(s:Stereotype | s.name='View' or s.name='Controller')))
```

Listing 3: Contrainte du MVC raffinée

Nous obtenons en premier lieu l'ensemble des attributs de la classe stéréotypée *Model* en utilisant *self.ownedAttribute.type*. Sur cet ensemble, nous vérifions par la suite le reste de la contrainte (lignes 5-7). Le même mécanisme sera effectué pour les ensembles des types des valeurs de retours (ligne 8) et des paramètres des méthodes (ligne 12).

Nous avons opté pour le raffinement des contraintes, le même principe utilisé dans l'étape de transformation qui sera détaillé dans la section suivante. La seule différence est que nous n'utilisons que le métamodèle UML dans l'étape de raffinement.

6 Transformation des contraintes

L'objectif de la transformation est de pouvoir remplacer dans une contrainte d'architecture le vocabulaire du métamodèle UML par le vocabulaire du métamodèle Java. Il a fallu alors faire la correspondance entre les termes du métamodèle UML et ceux du métamodèle Java en les répartissant en 3 catégories : les métaclasses, les rôles et les navigations.
Dans le tableau 1 nous avons présenté pour chaque métaclasse, rôle et navigation d'UML son équivalent en Java.

Nous proposons la définition des projections d'abstractions sous la forme des *mappings* dans lequel nous désignons pour chaque entité du métamodèle UML sa correspondante dans le métamodèle Java. Nous avons opté pour une spécification de ces mappings en XML et une transformation effectuée par un programme ad-hoc au lieu d'utiliser un langage de transformation existant (Acceleo [3], Kermeta [17], ATL [16]). En effet, nous considérons que les contraintes ne sont pas des modèles. Nous aurions pu générer pour les contraintes, des modèles sur lesquels nous appliquons des transformations; mais le problème ici est la lourdeur dans la mise en place de la solution : nécessite de transformer le texte de la contrainte en modèle, transformer ensuite ce modèle et puis générer du texte de la nouvelle contrainte à partir de son modèle. Nous avons opté pour une solution simple qui consiste à exploiter un compilateur OCL

	UML	**Java**
Métaclasse	**Class**	**Class**
Rôle	ownedAttribute	field
	ownedOperation	method
	superClass	superClass
	nestedType	declaringClass
	interfaceRealization	interface
	package	package
Navigation	package.profileApplication	
	.appliedProfile.ownedStereotype	annotation
Métaclasse	**Property**	**Field**
Rôle	type	type
	declaringTypeattribute	declaringClass
Métaclasse	**Operation**	**Method**
Rôle	returnValuetype	returnType
	declaringTypeoperation	declaringClass
	ownedParameter	parameterType
	raisedException	exceptionType
Métaclasse	**Stereotype**	**Annotation**
Métaclasse	**Package**	**Package**

Table 1: Correspondance UML-Java(Métaclasse, Rôle, Navigation)

qui permet de générer un arbre syntaxique (AST) à partir du texte de la contrainte. Cette AST nous permet d'appliquer facilement les différents traitements. Nous appliquons alors le tableau de correspondance (tableau 1) sur l'AST généré afin d'obtenir une contrainte exprimée sur le métamodèle Java.

Après application de cette méthode de transformation sur notre exemple (Section 2), le listing 3 devient comme suit :

```
1 context Class inv :
2 self.annotation -> exists(s:Annotation|s.name='Model')
3 implies
4 self.field.type->forAll(t: Class | not(t.oclAsType(Class).annotation
5 ->exists(s:Annotation | s.name='View' or s.name='Controller')))
6 and
7 self.method.returnType->forAll(t: Class| not(t.oclAsType Class).annotation
8 ->exists(s:Stereotype| s.name='View' or s.name='Controller')))
9 and
10 self.method.parameterType->forAll(t: Class | not(t.oclAsType(Class).annotation
11->exists(s:Annotation | s.name='View' or s.name='Controller')))
```

Listing 4: Contrainte du patron MVC après la transformation

Comme il est indiqué dans le listing 4, nous avons remplacé, entre autres, *package.profileApplication.appliedProfile.ownedStereotype* par *annotation*, *ownedOperation* par *method* etc, en respectant les *mappings* présentés précédemment. Ces *mappings* sont appliqués en commençant tout d'abord par les navigations, ensuite les rôles et enfin les métaclasses.

L'usage des *mappings* déclaratifs nous donne la possibilité lorsque les métamodèles évoluent de modifier facilement les entités changées. En plus ceci nous permet de proposer une méthode plus générique et qui ne dépend pas des métamodèles sur lesquels nous travaillons.

7 Génération de code

La génération de code consiste à traduire la contrainte transformée, exprimée donc sur le métamodèle Java en métaprogramme Java. Pour générer ce code, nous avons suivi les étapes suivantes. Premièrement, nous générons l'arbre syntaxique AST à partir de la contrainte écrite sur le métamodèle Java. Ensuite, nous effectuons un parcours hiérarchique sur cet arbre (de haut en bas et de gauche à la droite) et nous générons du code Java en nous basant sur les règles ci-dessous. Il faut mentionner tout d'abord que la première règle est appliquée une seule fois dans la génération de code d'une contrainte. Les autres règles sont appliquées le long de l'analyse de la contrainte selon le type du nœud de l'AST. En fait, si c'est un rôle ou une navigation alors nous devons appliquer la règle 2. S'il s'agit d'un quantificateur, la règle 3 est alors appliquée et ainsi de suite.

1. Il faut considérer tout d'abord qu'une contrainte est représentée par une méthode Java qui retourne un booléen et qui prend en paramètre un objet de type la métaclasse sur laquelle la contrainte s'applique. Cette méthode se trouve dans une classe Java et fait appel si nécessaire à d'autres méthodes implémentées lors de la génération du code.

2. Chaque rôle et navigation dans le métamodèle Java sera remplacé par la méthode accesseur dans Java *Reflect* (*getter*) de Java. Par exemple si nous naviguons vers *Field* nous appliquons *getDeclaredFields()*[3]. Si nous voulons accéder au type de retour d'une méthode nous appelons *getReturnType()*.

3. Concernant les quantificateurs et les opérations ensemblistes, nous avons défini pour chacune une template Java. Des exemples sont représentés dans le tableau 2. La méthode *select(..)* présentée dans la dernière ligne du tableau 2 peut être appliquée sur des différents types OCL comme *Set*, *Collection* ou *Sequence*. Lors de la génération de code, chaque type OCL sera remplacé par son équivalent en Java.

OCL	Java

`forAll(ex:OclExpression): Boolean`	`private boolean forAll(Collection c) {`
	`  for(Iterator i = c.iterator(); c.hasNext();) {`
	`  if(!exInJava) return false;`
	`  }`
	`  return true;`
	`}`

[3]Nous utilisons *getDeclaredField()* au lieu de *getFields()* pour récupérer tous les attributs (privés et publiques). Pour ceux que nous héritons, nous devons les spécifier dans la contrainte OCL en utilisant le rôle *superClass*.

exists(ex:OclExpression):Boolean	private boolean exists(Collection c) {
	for (Iterator i = c.iterator(); c.hasNext();) {
	ElementType e = (ElementType) i.next();
	if(exInJava) return true;
	}
	return false;
	}

includes(o:Object):Boolean	contains(o:Object):Boolean
includesAll(o:Object):Boolean	containsall(o:Object):Boolean

select(ex:OclExpression):Sequence	List result = new ...();
	private list select(Collection c) {
	for (Iterator i = c.iterator(); c.hasNext();){
	ElementType e = (ElementType) i.next();
	if (exInJava) {
	result.add(e);
	}
	}
	return result;
	}

Table 2: Génération de code des quantificateurs et opérations ensemblistes d'OCL en Java

4. Dans chaque quantificateur ou opération ensembliste, nous parcourons de façon récursive l'expression évaluée comme s'il s'agit d'une sous contrainte et nous générons de nouveau les correspondances : si nous rencontrons un rôle ou une navigation dans le métamodèle Java nous re-appliquons la règle 2. Dans le cas ou le quantificateur est imbriqué, nous re-appliquons la règle 3.

5. Pour les opérateurs logiques (*and*, *not*, ...), nous avons défini aussi des méthodes. Ces méthodes sont implémentées dans une classe nommée *OpLogiques*. Si la contrainte contient un opérateur logique alors cette classe va être déclarée comme une classe mère de la classe générée dans laquelle se trouve la méthode invariant.

6. Les opérations arithmétiques ($>$, $<$, $=$, ...) et les types (*Integer*, *Real*, *String*, ...) sont les mêmes dans le code généré de la contrainte.

7. Les opérations sur les chaînes de caractères comme *substring(), concat()* sont changées par les méthodes Java équivalentes.

Pour mieux comprendre le déroulement de la génération de code, le tableau 3 illustre un exemple de génération de code d'une contrainte OCL exprimée dans le métamodèle Java. Il s'agit de la première contrainte du patron MVC présenté dans la section 2. Pour des raisons de simplicité et de limitation de places nous raffinons ici la dépendance entre deux classes par

la déclaration dans la première classe d'au moins un retour de méthodes ayant comme type la deuxième classe.

Contrainte	Métaprogramme Java

`context Class inv :`	`Boolean result=true;`
	`public Boolean invariant(Class<?> uneClasse){`
	`  //code`
	`  return result;`
	`}`

`self.annotation`	`Annotation[] annotations=` `  uneClasse.getAnnotations();`
`->exists(a:Annotation|a.name='Model')`	`Boolean resultexists1= exists1(annotations);`
`implies`	`if(resultexists1){` `}`
`self.method`	`Method[] methods= uneClasse.getDeclaredMethods();`

`->forAll(m:method|not`
`(m.returnType.annotation`
`->exists(a:Annotation|`
`  a.name='View')))`
	`Boolean resultforAll1 = forAll1(methods);`

Table 3: Exemple de génération de code de la première contrainte du patron MVC

Nous avons présenté dans le tableau 3 pour chaque partie de la contrainte son code équivalent en Java tout en respectant les règles expliquées précédemment. Le listing 5 montre un extrait du métaprogramme généré à partir de cette contrainte.

```
1 public class Contrainte extends OpLogiques{
2 //...
3   public Boolean invariant(Class<?> uneClasse){
4      Annotation [] annotations = uneClasse.getAnnotations();
5      resultexists1 = exists1(annotations);
6      if(resultexists1){
7            Method [] methods = uneClasse.getDeclaredMethods();
8            Boolean resultforAll1 = forAll1(methods);
9    }
10 return resultforAll1;
11  }
12  private Boolean exists1(Annotation [] annotations){
13     for(Annotation annota: annotations){
14           Class a = annota.annotationType();
15           if(a.equals(Model.class))
16                   return true;
17     }
18  return false;
19  }
```

```
20  private Boolean forAll1(Method[] methods){
21     for(Method m : methods){
22        Type type = m.getReturnType();
23        Annotation[] annotations = type.getAnnotations();
24        resultexist2 = not(exists2(annotations));
25      if(!resultexist2)
26          return false;
27     }
28  return true;
29  }
30  private Boolean exists2(..){..}
31 }
```

Listing 5: Extrait de métaprogramme généré à partir de la première contrainte du MVC

Comme nous le remarquons, ce code est différent syntaxiquement au code optimal voulu (voir listing 2) mais ils ont la même sémantique. Il est évident que la traduction automatique ne permet pas d'obtenir un code ayant une complexité optimale. Toutefois, celle-ci apporte une aide précieuse aux développeurs qui devront plutôt se focaliser sur l'implémentation de la logique métier de leur application.

8 Travaux connexes

Notre travail repose sur deux grandes parties: la transformation des contraintes OCL d'un métamodèle à un autre et la génération de code à partir de ces contraintes.

Hassam et al. [12] ont proposé une méthode de transformation de contraintes OCL lors du *refactoring* des modèles UML en utilisant la transformation des modèles. Leur approche consiste à préparer un ensemble ordonné des opérations pour le *refactoring* des contraintes OCL après le *refactoring* du modèle UML et pour cela ils doivent premièrement utiliser une méthode d'annotation du modèle UML source pour obtenir un modèle cible annoté aussi en utilisant une transformation de modèles. Par la suite ils prennent les deux annotations pour former une table de *mapping* qui sera utilisée finalement avec Roclet [15] pour transformer les contraintes OCL du modèle source en des contraintes OCL conformes au modèle cible obtenu. Nous trouvons que notre idée d'utiliser un compilateur OCL existant est plus simple. Leur solution de transformation des contraintes est difficile à mettre en place comme nous avons mentionné dans la section 6 et nécessite des connaissances sur les outils et les langages de transformation de modèles. D'ailleurs Roclet avec lequel ils font la transformation utilise des résultats de Dresden OCL comme analyseur des contraintes [15].

Cabot et al. [9] ont travaillé sur la transformation des modèles UML/OCL en CSP (Constraint Satisfaction Problem) afin de vérifier des propriétés de qualité sur le modèle. Cette approche consiste en premier lieu à transformer le modèle en CSP, les propriétés à vérifier sont traduites comme des contraintes CSP et par la suite utiliser ECLiPSe [4] pour résoudre le CSP. Le résultat est une instance du diagramme de classe du modèle utilisé qui approuve ou désapprouve les propriétés à vérifier. La première partie de cette approche est similaire à notre transformation du système parce que la transformation en CSP se fait à l'aide de l'outil Dresden OCL comme un compilateur OCL pour pouvoir analyser les contraintes à transformer

et appliquer les règles qui conviennent. Cependant, nous avons implémenté en plus de la transformation une méthode pour générer du code source interprétable directement sur le code de l'application et dans le même environnement d'exécution sans besoin d'utiliser un outil externe d'interprétation des contraintes sur l'application.

Les auteurs dans [11] ont proposé une méthode pour simplifier les contraintes OCL. Ils identifient des règles nécessaires pour la simplification des contraintes OCL en les comparant par les systèmes de réécriture habituelles. Similairement, les auteurs dans [25] ont utilisé un langage appelé ACL qui présente un noyau commun d'expression des contraintes et des profiles différents. Chaque profil est défini sur un métamodèle, qui représente les abstractions architecturales manipulées à chaque étape dans le processus de développement. La simplification des contraintes est complémentaire à note approche. Nous pouvons procéder par une simplification des contraintes lors de la phase de transformation avant la génération de code.

Dans la pratique de l'ingénierie des modèles, des outils comme Eclipse OCL [1], Octopus [19] et Dresden OCL [10] ont contribué à la traduction des contraintes OCL en Java. Les contraintes sont des contraintes fonctionnelles et non architecturales. Le code généré par Dresden OCL [10, 13, 18] est difficilement compréhensible. En fait, il est vrai que Dresden OCL est le premier outil développé dans ce domaine mais il utilise du vocabulaire proposé uniquement dans l'API Dresden OCL. Ce code est normalement adressé aux développeurs qui ont déjà vu Dresden OCL contrairement à notre objectif qui est de s'adresser à tout développeur voulant transformer des contraintes et ayant des connaissances en Java mais pas obligatoirement connaissant l'API Dresden OCL. En plus, il faut obligatoirement créer au préalable les classes du modèle pour pouvoir générer la contrainte. Nous remarquons ainsi que cet outil est pratique uniquement pour les contraintes fonctionnelles et non pas pour les contraintes architecturales. Ces raisons expliquent pourquoi nous avons choisi de ne pas travailler avec le générateur de code proposé par Dresden OCL.

Un autre outil existe est intitulé OCL2j [7] permettant la génération des contraintes fonctionnelles en Java en utilisant la programmation par aspects. Cet outil ne supporte pas la transformation des contraintes d'architecture et n'utilise pas une librairie standard pour Java comme la librairie *Reflect*. L'indisponibilité de cet outil ne nous a pas permis de l'examiner.

9 Conclusion et perspectives

Nous avons présenté dans cet article une méthode pour automatiquement transformer des contraintes d'architecture OCL en métaprogrammes Java en utilisant l'introspection fournie par ce langage. Notre méthode consiste tout d'abord à écrire les contraintes sur un métamodèle UML, les raffiner si nécessaire, les transformer par la suite en contraintes exprimées sur un métamodèle Java et enfin générer du code source Java en se basant sur des règles de génération de code précises. Le mécanisme de réflexion utilisé est un standard dans Java. Par ailleurs, nous pouvons utiliser des librairies d'analyse statique comme JDT [14] ou de *ByteCode* comme BCEL [6] mais l'objectif, comme c'est fixé, est d'utiliser ce qui est standard dans Java et de ne pas recourir à des librairies externes.

Dans notre méthode proposée, la couverture d'OCL n'est pas totale, nous avons développé des prototypes qui ne prennent pas en compte certaines constructions comme *Any*, *Union*..

Nous pouvons énoncer comme perspectives : la mise en œuvre de ces contraintes dans un environnement de développement intégré. Une méthode peut être développée pour vérifier ces contraintes sur le code de l'application (les injecter à la fin des constructeurs des classes, les placer dans une libraire qui sera appelée à la demande). Nous pouvons encore penser à un niveau d'abstraction plus élevé de façon que la spécification de contraintes devient indépendante d'un paradigme donné en utilisant un métamodèle de graphes (une description d'architecture étant considérée comme un graphe avec des nœuds et arrêtes), puis leur transformation vers les différents paradigmes (Objet, composant et services). Nous pouvons encore nous concentrer sur la spécification et l'interprétation des contraintes architecturales sur les applications à base de services. En fait, nous pouvons tout d'abord exprimer les contraintes d'architecture sur les patrons SOA (Service Oriented Architecture) modélisées par SoaML [21] et par la suite les interpréter dans l'implémentation de ces patrons en utilisant des frameworks comme OSGI.

References

[1] Eclipse ocl. `http://www.eclipse.org/modeling/mdt/?project=ocl`.

[2] Java reflect. `http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/packagesummary.html`.

[3] Acceleo. Implementation of mof to text language. `http://www.omg.org/news/meetings/tc/mn/specialevents/ecl/Juliot-Acceleo.pdf`.

[4] Krzysztof R Apt and Mark Wallace. *Constraint logic programming using ECLiPSe*. Cambridge University Press, 2007.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley, 2012.

[6] BCEL. The byte code engineering library. `http://commons.apache.org/proper/commons-bcel/`.

[7] Lionel C Briand, Wojciech Dzidek, and Yvan Labiche. Using aspect-oriented programming to instrument ocl contracts in java. *Technischer Report, Carlton University, Kanada*, 2004.

[8] Lionel C. Briand, Yvan Labiche, Massimiliano Di Penta, and Han (Daphne) Yan-Bondoc. An experimental investigation of formality in uml-based development, 2005.

[9] Jordi Cabot, Robert Clarisó, and Daniel Riera. Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 547–548. ACM, 2007.

[10] Birgit Demuth. The dresden ocl toolkit and its role in information systems development. In *Proc. of the 13th International Conference on Information Systems Development (ISD'2004)*, 2004.

[11] Martin Giese and Daniel Larsson. Simplifying transformations of ocl constraints. In *Model Driven Engineering Languages and Systems*, pages 309–323. Springer, 2005.

[12] Kahina Hassam, Salah Sadou, Régis Fleurquin, et al. Adapting ocl constraints after a refactoring of their model using an mde process. In *BElgian-NEtherlands software eVOLution seminar (BENEVOL 2010)*, pages 16–27, 2010.

[13] Heinrich Hussmann, Birgit Demuth, and Frank Finger. Modular architecture for a toolset supporting ocl. In *UML 2000—The Unified Modeling Language*, pages 278–293. Springer, 2000.

[14] JDT. Java development tools. `http://www.eclipse.org/jdt/`.

[15] Cédric Jeanneret, Leander Eyer, S Markovic, and Thomas Baar. Roclet: Refactoring ocl expressions by transformations. In *Software & Systems Engineering and their Applications, 19th International Conference, ICSSEA*, volume 2006, 2006.

[16] Frédéric Jouault and Ivan Kurtev. Transforming models with atl. In *Satellite Events at the MoDELS 2005 Conference*, pages 128–138. Springer, 2006.

[17] Kermeta. Kermeta. http://www.kermeta.org.

[18] LCI. Object constraint language environnement. http://lci.cs.ubbcluj.ro/ocle/.

[19] Octopus. Ocl tool for precise uml specifications. http://octopus.sourceforge.net.

[20] OMG. Object constraint language, version 2.3.1, document formal/2012-01-01. http://www.omg.org/spec/OCL/2.3.1/PDF/.

[21] OMG. Service oriented architecture modeling language (soaml)specification. http://www.omg.org/spec/SoaML/1.0.1/PDF.

[22] OMG. Unified modeling language superstructure, version 2.4.1, specification document formal/2011-08-06. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[23] T Reenskaug. Thing-model-view editor an example from a planning system, xerox parc technical note (may 1979).

[24] Chouki Tibermacine. chapter Architectures logicielles : contraintes d'architecture.

[25] Chouki Tibermacine, Régis Fleurquin, and Salah Sadou. Simplifying transformation of software architecture constraints. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, SAC '06, pages 1240–1244, New York, NY, USA, 2006. ACM.

Abstract

In order to clarify and make tangible their design choices, software designers integrate in their applications' models, constraints that their models and their implemetations should verify. Various existing environments enable constraint checking during the modeling phase, but in most cases they do not generate code that would continue to check these constraints during the implementation phase. It turns out that this is possible in a number of cases. Environments that provide this functionality only offer it for the functional constraints (related to the application objects' states) and not for the architectural ones (related to the application structure). Considering these limitations, we describe in this paper a system that associates architectural constraints, written in OCL, to an UML model and then generates a metaprogram allowing the verification of these constraints at runtime. Architectural constraints make references to UML metamodel's elements used to describe applications' models.

Keywords: Software architecture, architectural constraint, OCL, Java Reflect

Modélisation et vérification formelles en B d'architectures logicielles à trois niveaux

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

[1] EMA/LGI2P, Nîmes, France
{abderrahman.mokni, christelle.urtado, sylvain.vauttier}@mines-ales.fr
[2] LIRMM, CNRS et Université de Montpellier 2, Montpellier, France
marianne.huchard@lirmm.fr
[3] ENS/INRIA, Lyon, France
yulin88@gmail.com

Résumé

La réutilisation est une notion centrale dans le développement à base de composants. Elle permet de construire des logiciels à grande échelle de meilleure qualité et à moindre coût. Afin d'intensifier la réutilisation dans les processus de développement, un ADL à trois dimensions, nommé Dedal, a été proposé. Dedal permet de décrire la spécification, l'implémentation et le déploiement d'une architecture. Chaque définition doit être cohérente, complète et, réutilisant la définition de niveau supérieur, conforme à celle-ci. Cet article présente des règles formelles permettant de préserver et de vérifier ces trois propriétés dans des définitions d'architectures décrites en Dedal. Les règles sont exprimées avec le langage formel B afin d'automatiser leur vérification.

1 Introduction

L'ingénierie du logiciel a beaucoup évolué durant les deux dernières décennies. En effet, des nouveaux besoins tels que la construction et la maintenance de systèmes larges et complexes, la diminution des coûts et temps de développement et la sûreté des logiciels, sont apparus. Le développement à base de composants, discipline du génie logiciel [8], semble être la solution la plus adéquate pour répondre à tous ces besoins. Elle propose une démarche de construction de logiciels à large maille par assemblage de blocs de composants préexistants et suffisamment découplés pour être utilisés dans de multiples contextes. Le logiciel ainsi produit est décrit sous la forme d'une architecture logicielle comprenant les composants et les connexions devant les relier. Plusieurs langages connus sous le nom d'ADL (Architecture Description Language) ont été proposés afin de décrire les architectures logicielles. Certes, la plupart des ADLs, tels que C2 [9], Wright [2] et Darwin [5], permettent de modéliser les composants, les connecteurs et les configurations mais aucun ADL ne couvre toutes les étapes de cycle de vie d'une architecture.

Dans nos travaux précédents [10, 11], un ADL à trois niveaux, nommé Dedal, a été proposé. La particularité de Dedal est de représenter explicitement trois niveaux d'abstraction dans la définition d'une architecture : la spécification, la configuration et l'assemblage. Ces niveaux correspondent respectivement aux étapes de conception, implémentation et déploiement d'un système. L'objectif de Dedal est de favoriser la réutilisation dans les processus de développement à base de composants et de supporter une évolution contrôlée des architectures logicielles. Néanmoins, Dedal manque du formalisme nécessaire pour gérer ces processus d'une manière automatique et pour vérifier la consistance des définitions d'architectures dans chaque niveau et leur conformité par rapport au niveau supérieur afin d'éviter les problèmes d'érosion et de dérive [7].

L'objectif dans cet article est de palier à ce manque en proposant une formalisation des trois niveaux de Dedal ainsi que les règles de consistance et de conformité régissant ces définitions. La formalisation est exprimée en B [1], un langage de modélisation formelle fondé sur la théorie des ensembles et la logique des prédicats.

La suite de l'article est organisée de la manière suivante. La section 2 présente un aperçu du modèle Dedal. La section 3 définit une formalisation en B des trois niveaux d'architectures de Dedal. La

section 4 propose les règles de consistance et de conformité que doivent vérifier les définitions décrites en Dedal. La section 5 clôture le papier donnant une conclusion et des perspectives à ce travail.

2 Dedal, le modèle d'architectures à trois niveaux

Pour illustrer les concepts de Dedal, on propose un exemple d'application à la domotique. Il s'agit d'un système d'orchestration de scénarios de confort à domicile (Home Automation Software HAS). L'objectif est de pouvoir gérer la luminosité et la température du bâtiment en fonction du temps et de la température ambiante. Pour celà, on propose une architecture avec un composant orchestrateur qui interagit avec les équipements adéquats afin de réaliser le scénario de confort souhaité.

2.1 Le niveau spécification

La spécification est le premier niveau de description d'une architecture. Elle permet de définir une architecture idéale répondant aux exigences du cahier des charges. La spécification est constituée de composants rôles, de leurs connexions ainsi que du comportement global de l'architecture. Chaque composant rôle remplit des fonctionnalités requises dans le système. Défini comme un types de composant abstrait, sa description permet de guider la recherche de classes de composants concrets dans les bibliothèques afin de trouver une implémentation de l'architecture. La figure 1-a montre une spécification de l'architecture du HAS composée des rôles orchestrateur (*HomeOrchestrator*), lumière (*Light*), temps (*Time*), thermomètre (*Thermometer*) et climatiseur (*CoolerHeater*).

2.2 Le niveau configuration

La configuration est le deuxième niveau de description d'une architecture. Elle représente une implémentation de l'architecture par des classes de composants concrets et est dérivée de sa spécification en utilisant la description des rôles comme critère de recherche et de sélection dans une bibliothèque de composants [3]. Les types de composants sélectionnés doivent correspondre aux rôles identifiés dans le niveau spécification. En effet, chaque classe de composant possède un nom et des attributs et implémente un type de composant qui décrit ses interfaces ainsi que son comportement. En Dedal, les classes de composants peuvent être primitives ou composites. Ainsi, un rôle peut être réalisé par une composition de classes de composants et inversement, une classe de composant peut réaliser plusieurs rôles à la fois. La figure 1-b illustre l'architecture de HAS au niveau configuration ainsi qu'un exemple de composant composite (*AirConditioner*) qui réalise en même temps les rôles *Thermometer* et *CoolerHeater*. Les classes *AndroidOrchestrator*, *Lamp*, *Clock* réalisent respectivement les rôles *HomeOrchestrator*, *Light* et *Time*.

2.3 Le niveau assemblage

L'assemblage est le troisième niveau de description d'une architecture. Il correspond au déploiement de l'architecture et à son exécution. En Dedal, l'assemblage est décrit par des instances de composants et éventuellement des contraintes telles que le nombre maximum d'instances d'une classe de composant. Les instances de composants peuvent posséder des attributs valués qui représentent leurs états initial et courant. Ceci permet de décrire des paramétrages spécifiques des états de composants permettant différentes utilisations de l'architecture. La figure 1-c présente l'architecture du HAS au niveau assemblage. On note qu'une classe de composant peut avoir plusieurs instances comme montré dans l'exemple par les deux instances *lamp1* et *lamp2* de *Lamp*.

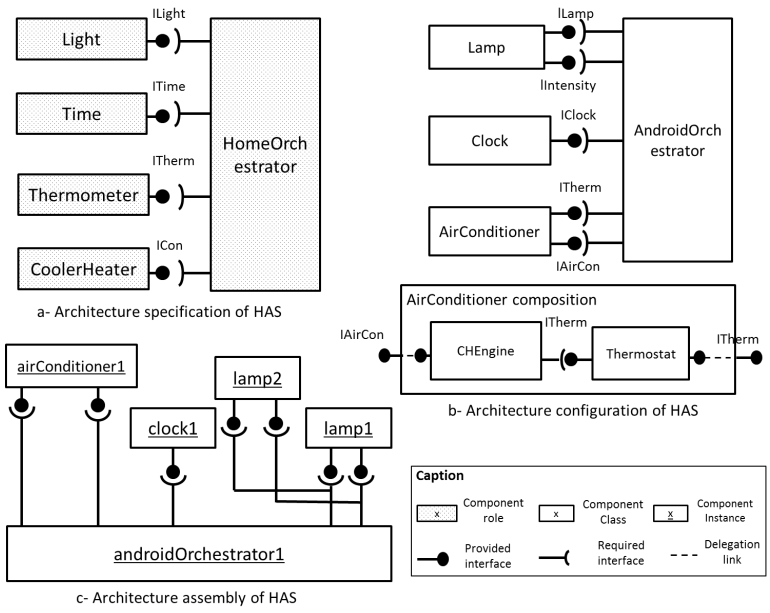


FIGURE 1 – Les trois niveaux d'architectures de Dedal

2.4 Les relations entre les trois niveaux d'architectures

La figure 2 illustre les relations entre les composants dans les trois niveaux d'architectures de Dedal. Une classe de composant réalise (*realizes*) un rôle défini dans la spécification et implémente (*implements*) un type de composant. La description de ce dernier doit correspondre à (*matches*) celle du rôle réalisé. Une classe de composant peut avoir plusieurs instances (*instantiates*) dans le niveau assemblage.

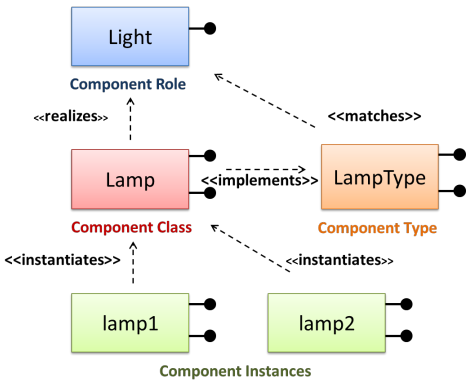


FIGURE 2 – Les relations entre les composants dans Dedal

Afin de pouvoir vérifier la consistance d'une architecture dans chaque niveau ainsi que la conformité d'un niveau d'abstraction par rapport au niveau supérieur, il est indispensable de définir formellement chaque niveau ainsi que les règles qui régissent ses relations avec le niveau supérieur. Cet objectif fait l'objet de notre proposition que nous présentons dans les sections suivantes.

3 Modélisation formelle en B de Dedal

Cette section présente la formalisation en B des concepts de Dedal. Le choix de B est motivé par le souhait d'étendre Dedal avec des définitions formelles exprimées dans un langage de prédicats et pouvant être vérifiées automatiquement moyennant des outils existants.
La formalisation comprend deux types de définitions. Des définitions génériques liées aux concepts communs à tous les ADLs à savoir les composants, les connexions et les architectures et des définitions spécifiques liées aux concepts de l'ADL Dedal. Le modèle formel générique peut servir de base pour d'autres ADL.
Vu la limite de l'espace, nous ne présentons que la partie de la formalisation que nous jugeons indispensable pour comprendre la contribution proposée (certaines déclarations d'ensembles et de variables sont donc omises dans cet article). La formalisation complète est présentée dans un travail précédent [6] qui propose les règles de substituabilité et compatibilité ainsi que les règles entre les composants de différents niveaux. Dans cet article, nous complétons les règles précédemment formulées par des règles de consistance et de conformité d'architectures.
Le modèle générique *Arch_concepts*. La table 1 présente la formalisation des concepts d'architecture, composant et connexion.

MACHINE *Arch_concepts* **INCLUDES** *Basic_concepts*
SETS *ARCHS ; ARCH_NAMES ; COMPS ; COMP_NAMES*
VARIABLES
architecture, arch_components, arch_connections, component, ...
INVARIANT
/* Un composant possède un nom et un ensemble d'interfaces */
$component \subseteq COMPS \wedge comp\_name \in component \rightarrow COMP\_NAMES \wedge$
$comp\_interfaces \in component \rightarrowtail \mathcal{P}\ (interface) \wedge$
/* Un client (repect. serveur) est un couple de composant et une interface */
$client \in component \leftrightarrow interface \wedge server \in component \leftrightarrow interface \wedge$
/* Une connexion est une bijection entre un client et un serveur */
$connection \in client \rightarrowtail\!\!\!\!\!\rightarrow server \wedge$
/* Une architecture est composée de composants et de connexions*/
$architecture \subseteq ARCHS \wedge arch\_components \in architecture \rightarrow \mathcal{P}_1(component) \wedge$
$arch\_connections \in architecture \rightarrow \mathcal{P}(connection)$

Notations spécifiques en B :
$\leftrightarrow$: relation $\rightarrowtail$: injection $\rightarrowtail\!\!\!\!\!\rightarrow$: bijection
$\mathcal{P}$(<set>) : ensemble des parties de <set> $\mathcal{P}_1$ (<set>) = $\mathcal{P}$ (<set>) - $\varnothing$

TABLE 1 – Spécification formelle des concepts génériques

Arch_concepts inclut un autre modèle nommé *Basic_concepts* qui contient la formalisation détaillée de la notion d'interface et ses éléments (type, direction et signature) ainsi que les règles de substituabilité et de compatibilité entre interfaces.
Formalisation des concepts de Dedal. Les niveaux spécification et configuration reprennent pratiquement les même définitions que celles qui sont proposées dans le modèle générique. En effet, les composants rôles (niveau spécification) et les types de composants (niveau configuration) partagent les mêmes caractéristiques (nom et liste d'interfaces) et héritent donc du concept de composant. Les classes de composants ont une description différente du fait qu'ils peuvent avoir des attributs supplémentaires et que leurs interfaces sont déduites du type de composant implémenté. Les connexions sont décrites de la même façon dans tous les niveaux (*i.e* : un client avec interface requise doit être connecté à un serveur avec une interface fournie compatible). La table 2 montre une partie de la formalisation des trois niveaux de Dedal chacun dans un modèle B dédié (*arch_specification*, *arch_configuration* et *arch_assembly*).
Les règles intra et inter-niveaux dans Dedal. Conformément aux concepts définis précédemment, il existe deux types de règles dans Dedal : des règles intra-niveau qui sont appliquées sur les concepts d'un même niveau d'abstraction et qui définissent notamment la substituabilité et la compatibilité entre les composants et des règles inter-niveau qui définissent les relations entre deux niveaux d'abtsraction différents (voir figure 2). Nous citons à titre d'exemple la règle de réalisation entre un rôle et une classe de composant qui sera utilisée par la suite pour établier la règle de conformité :

MACHINE *Arch_specification* **USES** *Arch_concepts* ... **PROPERTIES** **/* Les composants rôles (resp. les spécifications) héritent du concept de composant (resp. architectures) */** $COMP\_ROLES \subseteq COMPS \wedge ARCH\_SPEC \subseteq ARCHS$
MACHINE*Arch_configuration* ... **INVARIANT** **/* chaque composant classe possède un nom, une liste d'attributs et implémente un composant type*/** $compClass \subseteq COMP\_CLASS \wedge class\_name \in compClass \rightarrow CLASS\_NAME \wedge$ $class\_attributes \in compClass \rightarrow \mathcal{P}(attribute) \wedge$ $class\_implements \in compClass \rightarrow compType \wedge$ **/* Un composant composite est un composant classe et est décrit par une configuration** $compositeComp \subseteq compClass \wedge composite\_uses \in compositeComp \rightarrow config \wedge$ **/* Une délégation est une relation entre une interface déléguée et une interface interne** $delegation \in delegatedInterface \rightarrow interface \wedge$ **/* Une configuration est composée d'au moins un composant classe et de connexions */** $config \subseteq CONFIGURATIONS \wedge config\_components \in config \rightarrow \mathcal{P}_1(compClass)$ $config\_connections \in config \rightarrow \mathcal{P}(connection)$
MACHINE *Arch_assembly* ... **INVARIANT** **/* Un composant instance est désigné par un nom et possède un état initial et un état courant*/** $compInstance \subseteq COMP\_INSTANCES \wedge compInstance\_name \in compInstance \rightarrow INSTANCE\_NAME \wedge$ $initiation\_state \in compInstance \rightarrow \mathcal{P}\ (attribute\_value) \wedge$ $current\_state \in compInstance \rightarrow \mathcal{P}\ (attribute\_value) \wedge$ **/* Un assemblage est constitué d'instances et de connexions*/** $assm \subseteq ASSEMBLIES \wedge assm\_components \in assm \rightarrow \mathcal{P}_1(compInstance) \wedge$ $assm\_connection \in assm \rightarrow \mathcal{P}(connection)$

Table 2 – Formalisation des trois niveaux de Dedal

Règle de réalisation : Une classe de composant CL réalise un rôle CR si le type de CL est conforme à (*matches*) CR. La description du type de CL doit contenir au moins syntaxiquement les même interfaces que le rôle CR ou des spécialisations des interfaces de CR. Ceci est énoncé formellement comme suit :

$$
\begin{array}{l}
realizes \in compClass \leftrightarrow compRole \wedge \\
\forall\ (CL,CR).(CL \in compClass \wedge CR \in compRole \\
\Rightarrow \\
\quad ((CL,CR) \in realizes \\
\qquad \Leftrightarrow \\
\quad \exists\ CT.(CT \in compType \wedge (CT\ ,\ CR) \in matches \wedge (CL,CT) \in class\_implements)) \\
)
\end{array}
$$

$$
\begin{array}{l}
matches \in compType \leftrightarrow compRole \wedge \\
\forall\ (CT,CR).(CT \in comType \wedge CR \in compRole \\
\Rightarrow \\
((CT,CR) \in matches \\
\quad \Leftrightarrow \\
\exists (inj).(inj \in comp\_interfaces(CR) \rightarrowtail\!\!\!\!\!\!\twoheadrightarrow comp\_interfaces(CT) \wedge \\
\quad \forall\ (int).(int \in interface \\
\qquad \Rightarrow \\
\quad inj(int) \in int\_substitution[\{int\}]) \\
)))
\end{array}
$$

La formalisation des trois niveaux de Dedal est un pas essentiel pour établir les règles de consistance et de conformité entre les niveaux. Ces règles constituent la deuxième proposition de ce papier que nous présentons dans la section suivante.

4 Vérification des architectures à trois niveaux

A l'instar de la formalisation du modèle Dedal, cette section présente des règles génériques pouvant être appliquées sur une architecture en général à savoir les règles de consistance et des règles spécifiques au modèle Dedal qui définissent les relations entre chaque niveau en vue de vérifier la conformité d'un niveau par rapport à un autre.

4.1 Règles de consistance d'une architecture

Une architecture est dite consistante si elle répond à toutes les fonctionnalités décrites dans le cahier des charges (complétude) et si tous ses composants sont correctement connectés (exactitude).
Complétude : Une architecture *arch* est dite complète si pour chacun de ses clients *cl*, il existe une connexion *conn* tel que *cl* est dans l'une des extrémités de *conn*. Formellement :

$$
\begin{array}{l}
\forall\ (arch,\ cl).\\
\quad (arch \in architecture \wedge cl \in client \wedge cl \in arch\_clients(arch)\\
\qquad \Rightarrow\\
\quad \exists\ conn.\\
\quad\ (conn \in connection \wedge conn \in arch\_connections(arch) \wedge\\
\quad\ cl \in \mathbf{dom}(\{conn\})))
\end{array}
$$

La règle d'exactitude complète la règle précédente en imposant que toutes les connexions soient correctes (*i.e* : entre un client et un serveur compatibles).
Exactitude : Pour tout client *cl* et tout serveur *se*, si *cl* et *se* sont connectés, alors leurs interfaces *int1* et *int2* sont compatibles. Formellement :

$$
\begin{array}{l}
\forall\ (cl,\ se).\\
\quad (cl \in client \wedge se \in server\\
\qquad \Rightarrow\\
\quad\ ((cl,\ se) \in connection\\
\qquad \Rightarrow\\
\quad\ \exists\ (C1,\ C2,\ int1,\ int2).\\
\quad\ (C1 \in component \wedge C2 \in component \wedge C1 \neq C2 \wedge int1 \in interface \wedge int2 \in interface \wedge\\
\quad\ cl = (C1,\ int1) \wedge se = (C2,\ int2) \wedge (int1,\ int2) \in int\_compatible)))
\end{array}
$$

4.2 Les règles de conformité dans Dedal

Les règles de conformité sont basées sur les relations entre les niveaux d'architectures en Dedal (*cf.* Figure 2). Les règles relatives à ces relations sont détaillées dans [6].
Conformité d'une configuration par rapport à sa spécification : Une configuration *Conf* implémente une spécification *Spec* si pour tout rôle *CR* de *Spec*, il existe au moins une classe de composant *CL* dans *Conf* qui le réalise. Formellement :

$$
\begin{array}{l}
implements \in configuration \leftrightarrow specification \wedge\\
\forall\ (Conf,\ Spec).(Conf \in configuration \wedge Spec \in specification\\
\quad \Rightarrow\\
\qquad (Conf,\ Spec) \in implements\\
\qquad\quad \Leftrightarrow\\
\quad \forall\ CR.(CR \in compRole \wedge CR \in spec\_components(Spec) \Rightarrow\\
\qquad \exists\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge\\
\qquad\quad (CL,CR) \in realizes)))
\end{array}
$$

On note qu'un composant rôle peut être réalisé à travers un composant composite qui n'existe pas initialement dans la bibliothèque à composants mais dont la composition est calculée à travers d'autres classes existantes.
Conformité d'un assemblage par rapport à une configuration : Un assemblage *Asm* instancie une configuration *Conf* si toutes les classes de composants de *Conf* possèdent au moins une instance dans *Asm* et toutes les instances qui sont dans *Asm* sont des instances de classes de composants qui sont dans *Conf*. Formellement :

$$
\begin{array}{l}
instantiates \in assembly \rightarrow configuration\\
\forall\ (Asm,Conf).(Asm \in assembly \wedge Conf \in configuration\\
\Rightarrow\\
((Asm,Config) \in instantiates\\
\quad \Leftrightarrow\\
\forall\ CL.(CL \in compClass \wedge CL \in config\_components(Conf)\\
\quad \Rightarrow\\
\quad \exists\ CI.(CI \in compInstance \wedge CI \in assm\_components(Asm) \wedge\\
(CI,CL) \in comp\_instantiates)) \wedge\\
\forall\ CI.(CI \in compInstance \wedge CI \in assm\_components(Asm)\\
\quad \Rightarrow\\
\quad \exists\ CL.(CL \in compClass \wedge CL \in config\_components(Conf) \wedge\\
(CI,CL) \in comp\_instantiates))\ ))
\end{array}
$$

5 Conclusion et perspectives

Dedal est un modèle architectural qui permet une représentation explicite et distincte des spécifications, configurations et assemblages constituant les trois niveaux d'une architecture logicielle. Cet article a proposé des règles formelles en B définissant la consistance d'architectures logicielles ainsi que la conformité entre les trois niveaux de Dedal. Les règles ont été validées grâce à l'animateur ProB [4] et en réalisant des vérifications sur des instances de modèles B de Dedal relatives à l'exemple HAS. L'un des futurs objectifs est d'automatiser la transformation en modèles formels de Dedal de leurs descriptions textuelles ou graphiques (tel que UML) afin de permettre une vérification automatique des architectures logicielles. Une perspective de court terme consiste ausssi à établir des règles d'évolution permettant (1) de faire évoluer une architecture à un niveau d'abstraction donné tout en préservant sa consistance et (2) de propager l'impact du changement vers les autres niveaux afin de préserver la conformité et d'éviter les problèmes de dérive et d'érosion.
Sur le plan pratique, nous envisageons de développer un outil intégré autour d'Eclipse permettant l'édition (textuelle et graphique) et la vérification des descriptions d'architectures en Dedal et la gestion automatique de l'évolution architecturale.

Références

[1] Jean-Raymond Abrial. *The B-book : Assigning Programs to Meanings*. Cambridge University Press, New York, USA, 1996.

[2] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM TOSEM*, 6(3) :213–249, July 1997.

[3] Gabriela Arévalo, Nicolas Desnos, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. FCA-based service classification to dynamically build efficient software component directories. *International Journal of General Systems*, pages 427–453, 2008.

[4] Michael Leuschel and Michael Butler. ProB : An automated analysis toolset for the B method. *International Journal on Software Tools for Technology Transfer*, 10(2) :185–203, February 2008.

[5] Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1996.

[6] Abderrahman Mokni, Marianne Huchard, Christelle Urtado, Sylvain Vauttier, and Huaxi (Yulin) Zhang. Formal modeling of software architectures at three abstraction levels. *Rapport technique*, 2014.

[7] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4) :40–52, October 1992.

[8] Ian Sommerville. *Software engineering (9th edition)*. Addison-Wesley, 2010.

[9] Richard N. Taylor, Nenad Medvidovic, Kenneth M. Anderson, E. James Whitehead, Jr., and Jason E. Robbins. A component- and message-based architectural style for GUI software. In *Proceedings of the 17th ICSE*, pages 295–304, New York, USA, 1995. ACM.

[10] Huaxi (Yulin) Zhang, Christelle Urtado, and Sylvain Vauttier. Architecture-centric component-based development needs a three-level ADL. In *Proceedings of the 4th ECSA*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, August 2010. Springer.

[11] Huaxi (Yulin) Zhang, Lei Zhang, Christelle Urtado, Sylvain Vauttier, and Marianne Huchard. A three-level component model in component-based software development. In *Proceedings of the 11th GPCE*, pages 70–79, Dresden, Germany, September 2012. ACM.

Session 4

Travaux de doctorants

Framework for Heterogeneous Modeling and Composition *

Matias Ezequiel Vara Larsen[1]
, Julien DeAntoni[1]
and Frédéric Mallet[1]

Université Nice Sophia Antipolis,
I3S, INRIA, AOSTE
varalars@i3s.unice.fr, julien.deantoni@polytech.unice.fr, frederic.mallet@unice.fr

Abstract

Embedded and cyber-physical systems are becoming more and more complex. They are often split into subsystems and each subsystem can be addressed by a Domain Specific Modeling Language (DSML). A DSML efficiently specifies the domain concepts as well as their behavioral semantics. For a single system, several models conforming to different DSMLs are then developed and the system specification is consequently heterogeneous, *i.e.*, it is specified by using heterogeneous languages. The behaviors of these models have to be coordinated to provide simulation and/or verification of the overall system. This thesis studies this coordination problem with the objective of enabling the execution and the verification of the heterogeneous systems, by relying on the behavioral composition of DSMLs. We are currently proposing a language named B-COoL (Behavioral Composition Operator Language) to specify the composition of DSMLs behavior. The aim of B-COoL is two-fold: to capitalize the coordination at the language level, and to enable the automatic generation of the coordination between models conforming to composed DSMLs.

1 Introduction

The development of embedded and cyber-physical systems is becoming more and more complex. Such systems involve the interactions of very different subsystems, *e.g.*, DSP, Sensors, Analog-Digital devices. The requirements of each subsystem are usually captured by using a Domain Specific Modeling Language (DSML). An heterogeneous system is then specified using several DSMLs. To simulate and verify heterogeneous systems, models conforming to different DSMLs have to be coordinated.

Existing approaches [3, 1] allow the user to hierarchically coordinate heterogeneous models. These approaches provide a unique abstract syntax and the behavioral semantics is brought by using a specific Model of Computation (MoC). For instance, in [5], Ptolemy [3] has been used to embed hierarchical finite state machines within different concurrent models of computations. Although these approaches are good for experimenting, they hide the coordination semantics in the tool. This limits the study and the tuning of the coordination. More importantly, they do not provide any specific language to help in the specification of the coordination.

Differently, coordination languages [7] allow the user to manually specify the coordination between models. Thus, the user can explicitly define the coordination between two models written in different languages. There are two kinds of coordination languages: *endogenous* and *exogenous*. Endogenous languages provide primitives that must be incorporated within a model

*This work is partially supported by the ANR INS Project GEMOC (ANR-12-INSE-0011), and the CNRS PICS Project MBSAR.

for its coordination. In exogenous languages such primitives are without a model. In any case, since the coordination depends on the coordinated models, it has to be redone for each pair of models.

There is a need to *capitalize* the specification of the coordination between models at the language level by defining a language dedicated to the composition of language behavior. Current structural composition languages [4, 6] allow the user to specify composition rules at the language level. These rules are then applied between models. Such composition rules rely on two major steps: *matching* and *merging*. The matching decides *what* elements of the syntax are selected and *when* (*i.e.*, under which condition) the merging is applied. The merging specifies *how* the selected elements are combined. Similarly, a behavioral composition language relies in a matching step that selects behavioral elements of models. However, the merging step is replaced by a *coordination step* that specifies how the selected elements must be coordinated.

We propose the Behavioral Composition Operator Language (B-COoL) to specify the behavioral composition of languages. B-COoL allows the user to define operators that define composition rules between elements of the semantics of one or more language(s). This specification can be used to automatically generate the coordination between two (or more) models. Currently, B-COoL relies on a behavioral interface of the languages in terms of Events as proposed in [2]. First experiments of coordination have been based on the use of CCSL as illustrated by the coordination of a Finite State Machine and a fUML model [8].

References

[1] F. Boulanger and C. Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *Proceedings of ICST'08*, pages 318–327. IEEE Comp. Soc., 2008.

[2] Benoit Combemale, Julien Deantoni, Matias Vara Larsen, Frédéric Mallet, Olivier Barais, Benoit Baudry, and Robert France. Reifying Concurrency for Executable Metamodeling. In Richard F. Paige Martin Erwig and Eric van Wyk, editors, *6th International Conference on Software Language Engineering (SLE 2013)*, Lecture Notes in Computer Science, Indianapolis, Etas-Unis, 2013. Springer-Verlag.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.

[4] Franck Fleurey, Benoit Baudry, Robert France, and Sudipto Ghosh. Models in software engineering. chapter A Generic Approach for Automatic Model Composition, pages 7–15. Springer-Verlag, Berlin, Heidelberg, 2008.

[5] A. Girault, Bilung Lee, and E.A. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 18(6):742–760, 1999.

[6] Dimitrios S. Kolovos, Richard F. Paige, and Fiona A. C. Polack. Merging models with the epsilon merging language (eml. In *In Proc. ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (Models/UML 2006*, 2006.

[7] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, Amsterdam, The Netherlands, The Netherlands, 1998.

[8] Matias Vara Larsen and Arda Goknil. Railroad Crossing Heterogeneous Model. In *GEMOC workshop 2013 - International Workshop on The Globalization of Modeling Languages*, Miami, Florida, USA, September 2013.

Graphe de Dépendance pour la Recontextualisation de Modèles

Paola Vallejo, Mickaël Kerboeuf and Jean-Philippe Babau

Université de Bretagne Occidentale, Brest, France
vallejoco@univ-brest.fr, kerboeuf@univ-brest.fr, babau@univ-brest.fr

Résumé

Le travail présenté vise à promouvoir la réutilisabilité d'outils dédiés à un domaine spécifique (défini par un métamodèle spécifique). Il cible en particulier les outils de réécriture. La proposition est fondée sur *Modif*, un langage dédié à l'évolution de métamodèles et à la coévolution de modèles. Son processus d'utilisation implique notamment deux migrations de modèles. La première (migration aller), permet de mettre des données dans le contexte d'un outil dédié. La deuxième (migration retour), permet de plonger les données mises à jour par l'outil dans le contexte du modèle initial. La *recontextualisation* complète la migration retour. Celle-ci est rendue possible grâce à la notion de *graphe de dépendance* qui spécifie les relations entre les données produites par l'outil et les données fournies en entrée.

mots clés : réutilisation, recontextualisation, graphe de dépendance

1 Approche

Modif [1] permet de spécifier l'évolution d'un métamodèle par l'application d'un ensemble d'opérateurs. Afin de mieux illustrer notre proposition, nous ne considérons ici que l'opérateur de suppression *Delete* qui s'applique aux classes, aux attributs et aux références. La suppression d'un élément du métamodèle implique la suppression des instances conformes du modèle. Ceci peut amener à des inconsistances et des pertes d'information [3]. Les instances perdues ne peuvent pas être restaurées.

Avec *Modif*, la réutilisation d'un outil se décompose en quatre étapes : la migration aller du modèle, l'utilisation de l'outil, la migration retour et la recontextualisation du modèle. La recontextualisation est décomposée en deux étapes : recontextualisation par clé et recontextualisation par graphe de dépendance. La relation entre la migration aller et l'ensemble migration retour - recontextualisation est une transformation bidirectionnelle qui n'est pas nécessairement bijective [2].

Notre préoccupation est la restauration (recontextualisation) des instances supprimées lors de la migration préalable à l'utilisation d'un outil. La contribution principale est une proposition de recontextualisation de ces instances. La figure 1 illustre les différentes étapes de transformation grâce aux fonctions M (migration aller), T (outil de réécriture), R (migration retour), C_k (recontextualisation par clé) et C_g (recontextualisation par graphe de dépendance). Mo indique que le modèle est dans le contexte initial, Mt indique que le modèle est dans le contexte de l'outil.

$$m.Mo \xrightarrow{M} m_m.Mt \xrightarrow{T} m_t.Mt \xrightarrow{R} m_r.Mo \xrightarrow{C_k} m_k.Mo \xrightarrow{C_g} m_g.Mo$$

FIGURE 1 – Étapes de transformation pour la réutilisation de T sur m

M est produite à partir de la spécification d'évolution du métamodèle et applique la suppression d'instances dans le modèle initial pour produire le modèle migré. T produit un modèle de sortie à partir d'un modèle d'entrée. Les deux modèles sont conformes au même métamodèle. Les transformations exécutées par T doivent être réutilisées sans les modifier [5]. R s'applique à un modèle migré qui a été ensuite modifié par T et produit un nouveau modèle migré inversée conforme au métamodèle du contexte initial. Les actions de l'outil ne sont pas défaites.

C_k reconnecte le contexte initial d'un modèle dans sa version migrée inversée. Chaque instance est identifiée avec une *clé* unique. Si le modèle migré n'a pas été modifié, alors le résultat de la recontextualisation par clé correspond au modèle initial. La recontextualisation par clé ne gère pas le cas où l'outil crée des nouvelles instances. Et si l'outil supprime des instances, le contexte initial ne peut pas être restauré. Ceci nous amène à étendre l'approche en introduisant le graphe de dépendance [4].

C_g restaure le contexte initial d'un modèle migré inversé. Ici, T est considéré comme une *boîte grise* dont les *fonctions de calcul* sont inconnues, mais dont les *dépendances de calcul* sont connues. Ainsi, pour un modèle d'entrée, l'outil T fournit le modèle résultat accompagné d'un *graphe de dépendance*. Il contient la relation entre chaque instance de sortie de T et l'ensemble des instances de l'entrée de T qui ont permis sa création ou modification [6]. L'utilisation d'un graphe de dépendance améliore la recontextualisation. En effet, les liens entre le modèle migré inversé et son contexte initial peuvent être recrées non seulement entre les instances initialement présentes et conservées, mais aussi entre les instances crées par l'outil et les instances qui ont été utilisées pour les calculer.

2 Prototype

Pour expérimenter les idées présentées, nous proposons un prototype simple basé sur Ecore et implementé en Java. Il ne fournit que l'opérateur *Delete*. Les possibilités de la recontextualisation sont illustrées par la réutilisation d'un aplatisseur (Fl) de machines à états hiérarchiques défini sur un métamodèle spécifique (FSM). Il produit un modèle où tous les super-états sont supprimés et potentiellement remplacés par au moins un nouvel état.

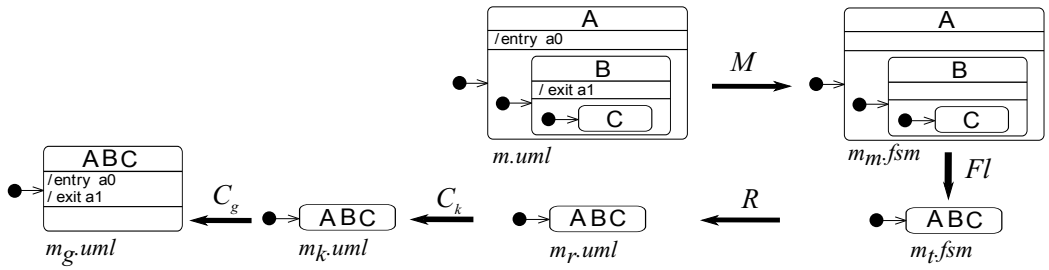


FIGURE 2 – Étapes de transformation pour la réutilisation de Fl sur $m.uml$

La figure 2 présente les modèles produits lors de la réutilisation de Fl. Le modèle initial correspond à un modèle de Statecharts UML ($m.uml$). L'opérateur *Delete* est appliqué pour supprimer les concepts du Statechart qui ne sont pas pris en compte par Fl. Le métamodèle UML est transformé et nous sommes placés dans le contexte de Fl. Le modèle $m.uml$ est alors migré vers $m_m.fsm$, ce dernier est conforme au métamodèle FSM. Le modèle migré ($m_m.fsm$) est aplati par Fl. Le modèle résultant est $m_t.fsm$. Afin que ce modèle puisse être édité, il doit être transformé en un modèle UML valide. La migration retour est appliquée, elle produit

$m_r.uml$, une copie de $m_t.fsm$ mais conforme au métamodèle UML. Maintenant, nous avons besoin de recontextualiser les instances supprimées lors de la migration.

La recontextualisation par clé récupère les instances `entry a0` et `exit a1`, cependant elle n'est pas capable de les restaurer parce que l'état `A` contenant `entry a0` et l'état `B` contenant `exit a1` ne sont plus dans le modèle inversé. La recontextualisation par clé produit le modèle $m_k.uml$. Pour cet exemple, le graphe de dépendance indique que l'état `ABC` (nommé `C` lors de la migration aller) a été modifié à partir des informations des états `A` et `B`. La recontextualisation par graphe de dépendance complète la recontextualisation par clé et restaure les instances `entry a0` et `exit a1`. De cette façon, l'outil Fl peut être utilisé, et le résultat final est un modèle UML valide ($m_g.uml$).

Références

[1] Jean-Philippe Babau and Mickael Kerboeuf. Domain specific language modeling facilities. In *5th MoDELS workshop on Models and Evolution*, New Zealand, 2011.

[2] Krzysztof Czarnecki and al. Bidirectional transformations : A cross-discipline perspective grace meeting notes, state of the art, and outlook, 2009.

[3] Markus Herrmannsdoerfer, Sander Vermolen, and Guido Wachsmuth. An extensive catalog of operators for the coupled evolution of metamodels and models. In *SLE*, 2010.

[4] Mickael Kerboeuf and Jean-Philippe Babau. A dsml for reversible transformations. In *11th OOPSLA Workshop on Domain-Specific Modeling*, United States, 2011.

[5] Sagar Sen and al. Reusable model transformations. *Journal of Software and Systems Modeling (SoSyM)*, 2012.

[6] Paola Vallejo, Mickaël Kerboeuf, and Jean-Philippe Babau. Specification of a legacy tool by means of a dependency graph to improve its reusability. In *7th MoDELS workshop on Models and Evolution*, United States, 2013.

Simulation orientée utilisateur des Systèmes d'Information des Smart Grids

Rachida Seghiri[1,2], Frédéric Boulanger[1], Vincent Godefroy[2] and Claire Lecocq[3]

[1] Supélec, Gif-sur-Yvettes, France
frederic.boulanger@supelec.fr
[2] EDF R&D, Clamart, France
rachida.seghiri@edf.fr, vincent.godefroy@edf.fr
[3] Institut Mines-Télecom, Evry, France
claire.lecocq@telecom-sudparis.eu

1 Introduction

Un Smart Grid est un réseau électrique intelligent permettant d'optimiser la production, la distribution et la consommation de l'électricité grâce à l'introduction des technologies de l'information et de la communication sur le réseau électrique [1]. Les Systèmes d'Information (SI) doivent donc intégrer pleinement les Smart Grids pour les superviser et les piloter. Dans ce contexte, nous proposons de simuler ces SI pour les faire valider par les experts Smart Grid et ainsi analyser la réplicabilité et la scalabilité des solutions mises en oeuvre dans des démonstrateurs en optimisant les coûts associés. Nous identifions principalement trois verrous à lever pour y parvenir. Tout d'abord, les méthodes prônées pour modéliser le SI sont peu adaptées aux profils des utilisateurs non informaticiens, ici les experts Smart Grid. [2] démontre que les langages dédiés sont l'un des principaux moteurs de l'adoption industrielle de l'ingénierie dirigée par les modèles. Des langages de modélisation spécifiques au métier (DSML) doivent être définis pour décrire des modèles exécutables. De plus, un Smart Grid, à la croisée du domaine électrotechnique, des technologies de l'information et des télécoms, est par définition hétérogène. Cette hétérogénéité se retrouve au niveau des modèles issus de différents DSML qu'il faut combiner pour aboutir à une simulation globale du SI. Enfin, l'approche par points de vue, permettant une séparation des préoccupations, est incontournable pour la modélisation de systèmes comme le Smart Grid [3], mais la cohérence entre les différentes vues du système doit être absolument maintenue.

2 Cas d'étude et approche proposée

Notre cas métier est issu des démonstrateurs du projet européen Grid4EU (Fig.1). Ce cas traite la régulation de tension des réseaux présentant une forte densité de sources d'énergie intermittente, appelées Générateurs d'Energie Décentralisés (GED), qui peuvent déséquilibrer le niveau de tension sur le réseau et endommager les équipements électriques. Le SI maintient cet équilibre en envoyant des consignes aux composants du réseau via des instruments de mesure et de commande (rectangles rouges dans Fig 1). Le réseau électrique est constitué du poste source qui transforme la haute tension en moyenne tension, d'interrupteurs, des GED, des consommateurs industriels et enfin des transformateurs alimentant des réseaux basse tension.
Dans des travaux préliminaires, nous avons modélisé et simulé ce cas métier sous Ptolemy [4] en adoptant deux modèles de calculs distincts pour le SI et pour le réseau électrique (respectivement Discret Event et Synchronous DataFlow). Ptolemy permet d'adresser cette première source d'hétérogénéité en adaptant les deux modèles de calcul. Le retour des experts métier a enrichi

notre modèle de simulation. Cependant, les modèles Ptolemy se sont révélés peu adaptés aux profils non informaticiens. Dès lors, le recours aux DSML est pleinement justifié.

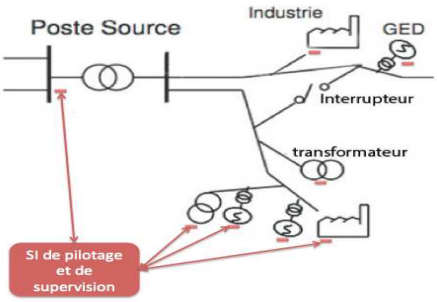


FIGURE 1 – Régulation de la tension

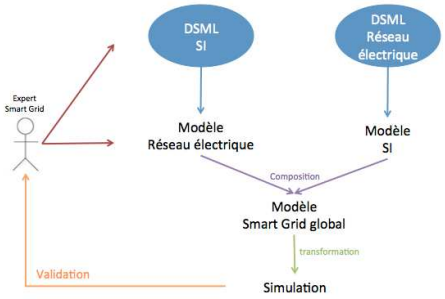


FIGURE 2 – Approche proposée

Le Smart Grid étant le fruit du couplage fort entre les couches SI et réseau électrique, nous proposons d'implémenter des DSML différents pour ces deux couches, avec le concours des experts métier. Les techniques issues de l'ingénierie dirigée par les modèles telle que la composition de modèles peuvent ensuite être utilisées pour assembler ces DSML selon différents scénarios, et ce pour construire le modèle Smart Grid global. Ainsi, nous conférons plus de modularité au modèle final. Par transformation de modèles, nous pouvons ensuite obtenir une simulation du modèle Smart Grid global. Cette simulation permet aux experts de critiquer/valider plus justement les modèles afin de les faire évoluer. Enfin, [3] montre qu'une approche par DSML peut aussi adresser les deux autres verrous que nous avions cités : (1) l'hétérogénéité des modèles et (2) le maintien de la cohérence des points de vue.

3 Conclusion

L'objectif de nos travaux est de définir des méthodes, modèles et outils permettant la simulation des SI des Smart Grids. Nous proposons d'utiliser les langages de modélisation dédiés (DSML) pour construire un modèle global du Smart Grid adapté aux experts métier en mettant à profit les techniques de l'ingénierie dirigée par les modèles. Les DSML peuvent, en outre, contribuer à adresser l'hétérogénéité des modèles et la cohérence des points de vue. Nous devons donc définir ces DSML et souhaitons les éprouver sur le cas métier de la régulation de tension avec le concours des experts Smart Grid.

Références

[1] `http://smartgrids-cre.fr/`.

[2] John Hutchinson, Jon Whittle, Mark Rouncefield, and Steinar Kristoffersen. Empirical assessment of mde in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480. ACM, 2011.

[3] Reinhard von Hanxleden, Edward A. Lee, Christian Motika, and Hauke Fuhrmann. Multi-view modeling and pragmatics in 2020. In *Proceedings of the 17th International Monterey Workshop on Development, Operation and Management of Large-Scale Complex IT System, to appear.*, 2012.

[4] `http://ptolemy.eecs.berkeley.edu/`.

Contribution to Model Verification: Operational Semantics for Systems Engineering Modeling Languages

Blazo Nastov

LGI2P, Ecole des Mines d'Alès, Parc Scientifique G. Besse, 30000 Nimes, France
Blazo.Nastov@mines-ales.fr

System Engineering (SE) [2] is an approach for designing complex systems based on creating, manipulating and analyzing various models. Each model is related to and is specific to a domain (e.g. quality model, requirements model or architecture model). Classically, models are the subject of study of Model Driven Engineering (MDE) [5] and they are nowadays built by using, and conforming to Domain Specific Modeling Languages (DSMLs). Creating DSML for SE objectives primarily consists in defining its abstract and concrete syntaxes. An abstract syntax is given by a meta model, while various concrete syntaxes (only graphical are considered here) define the representation of models instances of metamodels. So proposing the abstract syntax and one concrete syntax of a DSML makes it operational to create models seen then as a graphical representation of a part of a modeled system. Unfortunately, created models may have ambiguous meaning if reviewed by different practitioners. A DSML is then not complete without a description of its semantics, as described in [3], also highlighting four different types of semantics. Denotational, given by a set of mathematical objects which represents the meaning of the model. Operational, describing how a valid model is interpreted as a sequence of computational steps. Translational, translating the model into another language that is well understood and finally pragmatic, providing a tool that execute the model. The main idea of this work is to create and use such DSMLs focusing on the model verification problematic. We aim to improve model quality in terms of construction (the model is correctly build thanks to construction rules) and in terms of relevance for reaching design objectives (the model respects some of the stakeholder's requirements), considering each model separately and in interaction with the other models of the system under study so called System of Interest (SOI).

There are four main ways of verifying a given SOI model, 1) advice of a verification expert, 2) guided modeling, 3) model simulation and 4) formal proof of model properties. SE is a multi-disciplinary approach, so multiple verification experts are requested. Guided modeling is a modeling approach that consists of guiding an expert to design a model, proposing different construction possibilities or patterns in order to avoid some construction errors. In these two cases, the quality of the designed models cannot be guaranteed. "Simulation refers to the application of computational models to the study and prediction of physical events or the behavior of engineered systems" [4]. To be simulated, a model requires the description of its operational semantics. We define an operational semantics as a set of formal rules, describing on the one hand, the conditions, causes and effects of the evolution of each modeling concept and on the other hand, the temporal hypothesis (internal or external time, physic or logic time, synchronism or asynchrony hypothesis of events, etc.) based on which a considered model can be interpreted without ambiguity. Last, formal proof of properties is an approach that consists of using formal methods to check the correctness of a given model. Literature highlights two ways to prove model properties, either through operational semantics, or through translational semantics. In both cases, a property modeling language is used to describe properties which are afterwards proved using a theorem proving or a model checking mechanisms.

Our goal is to provide mechanisms for model simulation and formal proof of properties. We focus on concepts, means and tools allowing to define and to formalize an appropriate operational semantics for a DSML when creating its abstract and concrete syntaxes. Translational semantics however are not considered due to their classical limitations in terms of verification possibilities. There are different ways to formally describe operational semantics; for example, using the first order logic. In this case, a set of activation and deactivation equations are defined and assigned to each DSML concept, describing its behavior. Another way is through state transition system defining the sequence of computational steps, showing how the runtime system process from one state to another as described in [3].

Nowadays, there are multiple approaches and tools for defining operational semantics for a given DSML. Unfortunately, many of them require minimal knowledge in imperative or object-oriented programming and SE experts are not necessarily experts in programming. Indeed, the operational semantics of dedicated DSML is to be described and formalized with minimal efforts from the expert by assisting him and automating the process as much as possible. An approach is proposed in [1] supporting state-based execution (simulation) of models created by DSMLs. The approach is composed of four structural parts related to each other and of a fifth part providing semantics relying on the previous four. Modeling concepts and relationships between them are defined in the Doman Definition MetaModel (DDMM) package. The DDMM does not usually contain execution-related information. Such kind of information is defined in the State Definition MetaModel (SDMM) package. The SDMM contains various sets of states related to DDMM concepts that can evolve during execution. It is placed on the top of the DDMM package. Model execution is represented as successive state changes of DDMM concepts. Such changes are provoked by stimuli. The Event Definition MetaModel (EDMM) package defines different types of stimuli (events) and their relationship with DDMM concepts and SDMM states evolution. The Trace Management MetaModel (TM3) provides monitoring on model execution by scenarios made of stimuli and traces. The last and key part is the package Semantics describing how the running model (SDMM) evolves according to the stimuli defined in the EDMM. It can be either defined as operational semantics using action language or as denotational semantics translating the DDMM into to another language using transformational language. Our initial objectives are to study and to evaluate this approach on DSMLs of the field of SE in order to become able to interpret them and to verify some properties.

This work is developed in collaboration with the LGI2P (Laboratoire de Génie Informatique et d'Ingénierie de Production) from Ecole des mines d'Alès and the LIRMM (Laboratoire d'Informatique, Robotique et Microélectronique de Montpellier) under the direction of V.Chapurlat, F.Pfister (LGI2P) and C.Dony (LIRMM)

References

[1] Benoît Combemale, Xavier Crégut, Marc Pantel, et al. A design pattern to build executable dsmls and associated v&v tools. In *The 19th Asia-Pacific Software Engineering Conference*, 2012.

[2] ISO/IEC. *ISO/IEC 15288 : Systems and software engineering - System life cycle processes*, volume 2008. IEEE, 2008.

[3] Anneke G Kleppe. A language description is more than a metamodel. 2007.

[4] JT Oden, T Belytschko, J Fish, TJR Hughes, C Johnson, D Keyes, A Laub, L Petzold, D Srolovitz, and S Yip. Simulation-based engineering science: Revolutionizing engineering science through simulation–report of the national science foundation blue ribbon panel on simulation-based engineering science, february 2006.

[5] Douglas C Schmidt. Model-driven engineering. *Computer Society-IEEE*, 39(2):25, 2006.

Session 5

Modélisation de la dynamique

Une sémantique multi-paradigme pour simuler des modèles SysML avec SystemC-AMS

Daniel Chaves Café[1,2], Filipe Vinci dos Santos[2], Cécile Hardebolle[1], Christophe Jacquet[1] and Frédéric Boulanger[1]

[1] Département Informatique, Supélec, Gif-sur-Yvette, France.
[2] Chaire Thales/Supélec en Conception analogique avancée, Supélec, Gif-sur-Yvette, France.
{daniel.cafe, filipe.vinci, cecile.hardebolle, christophe.jacquet, frederic.boulanger}@supelec.fr

Abstract

Dans le contexte de la modélisation de systèmes, SysML apparait comme un langage pivot de spécification et de documentation. Ses diagrammes permettent la définition de la structure et du comportement de systèmes. La flexibilité de SysML a pour inconvénient qu'il n'existe pas de méthode standard pour définir leur sémantique. Ce problème est flagrant dans la conception de systèmes hétérogènes, où différentes sémantiques opérationnelles peuvent être utilisées. Cet article présente une manière de donner une sémantique opérationnelle aux éléments de SysML sous la forme de transformations vers le langage SystemC-AMS, permettant ainsi la simulation de modèles SysML.

1 Introduction

Un système hétérogène est constitué de composants de différentes natures, qui sont modélisés selon des formalismes distincts. Par exemple, un accéléromètre MEMS a une partie mécanique, une partie analogique et une interface numérique. Dans le domaine numérique, le formalisme à événements discrets est efficace pour la simulation grâce à l'abstraction des phénomènes analogiques qui font qu'une bascule change d'état. Dans le domaine analogique, la modélisation par réseaux de composants électriques permet de décrire la topologie du réseau pour en déduire les équations différentielles. La simulation des systèmes hétérogènes permet de garantir une fabrication correcte dès le premier essai. Le métier d'architecte de systèmes consiste à intégrer différents paradigmes dans un même modèle, ce qui pose des problèmes d'adaptation et fait appel à des compétences pluridisciplinaires et à la maîtrise des outils de simulation.

Les outils de modélisation hétérogène sont encore en phase d'expérimentation et de maturation. Ptolemy II [3] gère l'hétérogénéité par hiérarchie. Chaque composant est considéré comme une boite noire et la sémantique d'exécution et de communication est définie par une entité appelée *director*. Cette entité définit le modèle de calcul (MoC) de chaque composant. Pour arriver à cet objectif, Ptolemy II définit un moteur d'exécution générique [9] avec trois phases distinctes : l'initialisation, l'itération (pre-fire, fire et post-fire) et la finalisation (wrapup). Chaque *director* compose le comportement des blocs en redéfinissant ces phases d'exécution.

Inspiré de Ptolemy II, ModHel'X [1, 7] a été créé pour modéliser explicitement l'adaptation sémantique en rajoutant au moteur d'exécution générique de Ptolemy des phases d'adaptation sémantique. Cela donne un moyen efficace de définir la sémantique des interactions entre différents modèles de calcul. Dans cet article, nous présentons une méthode pour définir la sémantique opérationnelle (les MoCs) ainsi que l'adaptation sémantique pour des modèles hétérogènes SysML par traduction en SystemC-AMS.

SysML est un langage graphique de spécification de systèmes dont les diagrammes facilitent la communication entre différentes équipes d'un projet pluridisciplinaire. La sémantique

opérationnelle de ces diagrammes n'est toutefois pas précisément définie, ce qui est un obstacle à l'exécution de modèles SysML. L'implémentation de cette sémantique peut être réalisée dans un langage capable de simuler des modèles hétérogènes. Comme une première preuve de concept, nous avons choisi d'utiliser SystemC-AMS [6] (Analog and Mixed-Signal) qui est une bibliothèque C++ contenant un noyau de simulation à événements discrets ainsi qu'un ensemble de blocs de base pour la modélisation des systèmes mixtes. Ce langage supporte les modèles de calcul Dicrete Event (DE), Timed DataFlow (TDF) et Continuous Time (CT).

Notre approche consiste à générer du code SystemC-AMS à partir d'un modèle SysML annoté avec des éléments qui donnent la sémantique d'exécution de chaque bloc SysML et la sémantique d'adaptation entre blocs. Nous utilisons des techniques de l'ingénierie dirigée par des modèles (IDM) pour réaliser les transformations de modèles et la génération de code.

2 L'approche

Notre approche consiste à réaliser deux transformations de modèles. En partant d'un modèle SysML, nous appliquons une transformation "model-to-model" (M2M) écrite en ATL (Atlas Transformation Language)[8] pour générer une représentation intermédiaire du système dans le langage SystemC-AMS. La génération de code se fait juste après, en appliquant une deuxième transformation de type "model-to-text" (M2T) écrite en ACCELEO[10].

Nous utilisons des contraintes SysML pour indiquer le modèle de calcul utilisé par un bloc donné. Les mots clés "CT Block", "DE Block" et "FSM Block" sont utilisés pour spécifier l'utilisation des MoCs Continuous Time, Discrete Event et Finite State Machine, respectivement. La sémantique de ces MoCs est décrite dans [9].

Nous appliquons ces MoCs au sous-ensemble des diagrammes SysML qui nous semblent leur correspondre. Un automate par exemple, est modélisé par un diagramme d'états-transitions, un modèle à temps continu peut être representé par un diagramme d'interconnections de blocs où chaque bloc répresente une fonction. D'autres solutions peuvent être imaginées, comme l'utilisation des diagrammes paramétriques pour la définition des équations différentielles.

Nous considerons dans ce travail l'adaptation sémantique entre les différents domaines, par exemple l'échantillonnage entre un sous-système à temps continu et un sous-système à temps discret, que nous exprimons dans des commentaires liés aux ports des modules.

3 Cas d'étude

Pour illustrer l'approche, prenons l'exemple d'un véhicule à vitesse contrôlée, présenté en détail dans [2] et résumé ici. Les diagrammes de la figure 1 montrent comment appliquer une sémantique à un bloc SysML en lui ajoutant des annotations textuelles. Nous nous intéressons à une modélisation haut niveau de la dynamique du système en utilisant des équations de la physique classique. Nous déduisons l'accélération de $F = m \times a$, puis la vitesse et le déplacement en intégrant l'accélération. La dynamique du système est modélisée par le formalisme CT. Nous définissons une équation différentielle dans un diagramme IBD en assemblant des fonctions de base, comme l'intégrateur et le gain (voir figure 1 à gauche). Le contrôle est modélisé par un diagramme états-transitions où la rétroaction de la force dépend de l'état du contrôleur (Accelerate, Hold et Brake). Ces deux formalismes sont non seulement exprimés de manières différentes mais ont aussi des sémantiques opérationnelles différentes.

L'adaptation entre différents domaines est décrite par des commentaires liés aux ports du bloc *i_dynamics* dans le diagramme de droite. Une fois l'interface annotée par le mot clé

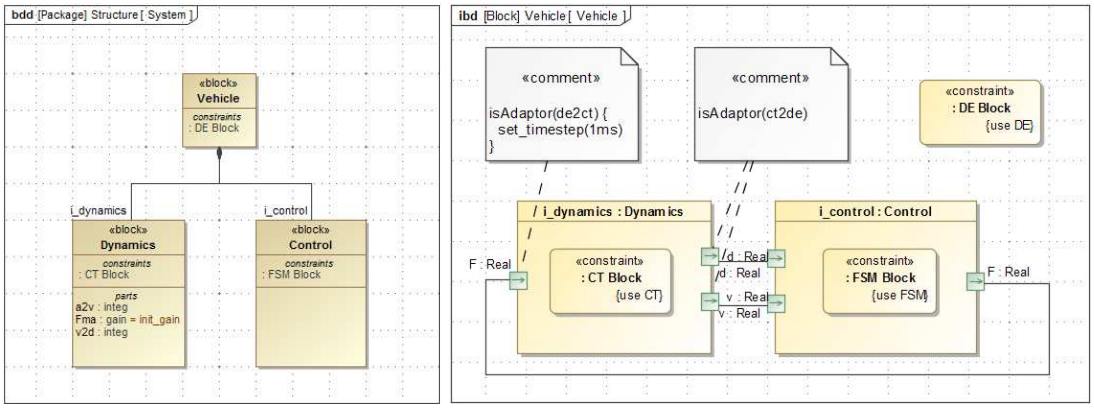


Figure 1: Exemple du véhicule à vitesse contrôlée : modèles SysML annotés

isAdaptor, nous lui appliquons une sémantique d'adaptation spécifique. Dans le cas d'un adaptateur *de2ct* le dernier événement capturé par le port est enregistré et sa valeur répétée à un pas d'échantillonnage fixe, donné par la directive *set_timestep*. En sortie, l'adaptation *ct2de* génère un événement à chaque changement de la valeur produite en sortie.

Cet approche est cependant très dépendante du choix du langage cible (ici SystemC-AMS), de même que les modèles de calcul utilisables. Nous ne pouvons implémenter que les MoCs que SystemC-AMS est capable d'exécuter. Cette limitation n'est pas contraignante car SystemC possède un moteur d'exécution à événements discrets qui supporte une large gamme de modèles de calcul discrets, et la partie AMS supporte les modèles continus. Il est important de souligner que la sémantique des MoCs est prise en compte de deux manières : soit par utilisation directe de la bibliothèque de base de SystemC, soit par implémentation dans la transformation. Dans l'exemple, la sémantique *block dynamics* s'appuie sur l'assemblage de blocs de la bibliothèque LSF (Linear Signal Flow) de SystemC-AMS. Dans le contrôleur, la sémantique de la machine à états est codée par la transformation M2M car SystemC n'a pas de MoC FSM. Cela a une influence sur les performances de simulation comme discuté dans [11].

La même reflexion s'applique aux mécanismes d'adaptation. L'échantillonnage des données et la production d'événements discrets sont réalisés par des adaptateurs spécifiques de la bibliothèque de SystemC-AMS. L'adaptation qui est faite à l'entrée du bloc *dynamics* est traduite en un module SystemC capable de transformer un événement discret de la machine à état en échantillons au pas fixe défini par la commande *set_timestep(1ms)*. L'adaptateur de sortie détecte les changements de valeur pour produire des événements. Une autre adaptation pourrait ne générer que les événements qui correspondent à une transition de l'automate. Cela permettrait une simulation plus performante mais rendrait l'adaptateur dépendant des modules qui lui sont connectés. Les adaptateurs possibles sont également limités par le langage cible. Dans notre exemple, nous n'avons utilisé que les adaptateurs de la bibliothèque SystemC-AMS. La conception d'adaptateurs spécialisées peut être réalisée sous forme de modules dédiés, comme discuté dans [4] avec le concept de *thick adapter*.

Finalement, la simulation de ce système est obtenue par transformation du modèle SysML en code SystemC-AMS qui est ensuite compilé et exécuté. Les résultats de simulation de cet exemple ont été présentés dans [2].

4 Discussions

Cet article définit une première approche pour la modélisation multi-paradigme en SysML ciblant la simulation mixte, que nous avons illustrée sur un modèle à trois formalismes (CT, DE et FSM). Nous avons spécifié la sémantique de chaque bloc SysML avec des annotations textuelles et la sémantique d'adaptation dans les commentaires liés aux ports. Le code exécutable est généré par transformation de modèles en utilisant ATL et ACCELEO.

Cette technique est un premier pas vers un framework générique de génération de code pour d'autres langages, par exemple VHDL-AMS. Notre approche en deux étapes (M2M et M2T) permet d'envisager l'extensibilité à d'autres MoCs par ajout de règles de transformation M2M correspondant au MoC désiré, sans devoir changer le générateur de code (partie M2T). Notre objectif est de découpler au maximum l'aspect sémantique des MoCs et l'aspect génération de code afin de supporter plus facilement de nouveaux MoCs et de nouveaux langages cibles. En ce qui concerne la définition des MoCs et l'adaptation sémantique, nous suivons les travaux en cours sur ce sujet au sein de l'initiative GeMoC [5].

4.1 Réflexions sur l'approche

Depuis la publication de ce travail exploratoire, nous avons amélioré l'intégration de notre approche aux techniques de l'IDM. Nous utilisons désormais des steréotypes pour définir les MoCs au lieu des mots-clés dans les contraintes SysML. Ces contraintes sont désormais réservées à la spécification des équations différentielles dans le formalisme CT. Les annotations des adaptateurs ont aussi évolué vers un langage dédié à la modélisation de leur comportement, ce qui évite de polluer le modèle avec du code spécifique au langage cible.

References

[1] F. Boulanger and C. Hardebolle. Simulation of Multi-Formalism Models with ModHel'X. In *Int. Conference on Software Testing, Verification, and Validation*, pages 318–327. IEEE, 2008.

[2] D. Cafe, F. V. dos Santos, C. Hardebolle, C. Jacquet, and F. Boulanger. Multi-paradigm semantics for simulating SysML models using SystemC-AMS. In *Specification and Design Languages, 2013, Forum on*, pages 82–89. IEEE, 2013.

[3] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity-the ptolemy approach. *Proc. of the IEEE*, 91(1):127–144, 2003.

[4] A. Ferrari, L. Mangeruca, O. Ferrante, and A. Mignogna. Desyreml: a sysml profile for heterogeneous embedded systems. *Embedded Real Time Software and Systems, ERTS*, 2012.

[5] GeMoC. On the globalization of Modeling Languages, May 2011. `http://gemoc.org`.

[6] C. Grimm, M. Barnasconi, A. Vachoux, and K. Einwich. An introduction to modeling embedded analog/mixed-signal systems using SystemC AMS extensions. In *DAC*, 2008.

[7] C. Hardebolle and F. Boulanger. ModHel'X: A component-oriented approach to multi-formalism modeling. In *Models in Software Engineering*, pages 247–258. Springer, 2008.

[8] F. Jouault and I. Kurtev. Transforming models with ATL. *Satellite Events at the MoDELS 2005 Conference*, pages 128–138, 2006.

[9] E. A. Lee. Disciplined heterogeneous modeling. In *Model Driven Engineering Languages and Systems*, pages 273–287. Springer, 2010.

[10] J. Musset, E. Juliot, S. Lacrampe, W. Piers, C. Brun, L. Goubet, Y. Lussaud, and F. Allilaire. Acceleo User Guide, 2006.

[11] H. D. Patel and S. K. Shukla. Systemc kernel extensions for heterogeneous system modeling. *The Netherlands: Kluwer Academic Publishers*, pages 9–71, 2004.

Etendre les Patrons de flot de contrôle dynamique avec des dépendances transactionnelles

Imed Abbassi[1], Mohamed Graiet[2] and Souha Boubaker[2]

[1] ENIT, University of Tunis El Manar, Tunisia
Monastir, Tunisia
abbassi_imed@ymail.com
[2] High School of Computer Science and Mathematics
Monastir, Tunisia
mohamed.graiet@imag.fr, souha.boubaker@gmail.com

Abstract

In this paper, we propose a solution to specify a flexible and reliable services compositions. So, we introduce a new paradigm, called dynamic transactional pattern. This paradigm is a convergence concept between dynamic workflow patterns and advanced transactional model. It can be seen as dynamic coordination and as a structural transaction. Thus, it combines dynamic Control-Flow flexibility and transactional processing reliability. A set of transactional patterns instances are dynamically connected together in order to define a dynamic composite Web service.

keywords: Web Service; Dynamic composition; Dynamic transactional pattern; Dynamic workflow pattern;

Résumé

Dans cet article, nous proposons une solution pour spécifier une composition fiable et flexible. Pour y parvenir, nous introduisons un nouveau concept appelé patron transactionnel dynamique. Ce nouveau concept est une convergence entre les patrons de workflow dynamiques et les modèles transactionnels avancés. Il peut être considéré comme une coordination dynamique et comme une transaction structurelle. Nous intégrons l'expressivité et la puissance des patrons de Workflow dynamique et la fiabilité des modèles transactionnels avancés. Un service composite est défini en connectant un ensemble d'instances de patrons transactionnels dynamiques.

Mots clés : Service Web; composition dynamique; patrons transactionnels dynamiques; patron de workflow dynamique;

1 Introduction

Les services Web peuvent être définis comme des programmes modulaires, indépendants et auto-descriptifs, qui peuvent être découverts, localisés et invoqués dynamiquement à travers Internet en utilisant un ensemble de protocoles standards [17]. Un des concepts les plus intéressants est la possibilité de créer dynamiquement de nouveaux services Web à valeur ajoutée en sélectionnant ceux qui répondent mieux aux exigences des clients puis en les combinant [13].

Assurer la fiabilité et la flexibilité des services composites dans un environnement dynamique, où les composants disponibles peuvent apparaître, disparaître, évoluent selon les besoins des clients, reste l'une des taches les plus difficiles. Nous entendons par flexibilité la capacité de prendre en compte le besoin de changement au niveau de la phase de conception et surtout le besoin de la reconfiguration dynamique qui peut avoir lieu avant/durant l'exécution du processus sans être anticipé durant la phase de conception. Une composition est dite fiable si son exécution amène toujours à un état cohérent.

Les technologies connexes actuelles sont incapables de résoudre ce problème d'une manière efficace. Ces technologies reposent sur deux approches existantes: les modèles transactionnels avancés [7] et le système de workflow [19]. Les modèles transactionnels avancés permettent d'assurer une exécution correcte d'un ensemble d'opérations encapsulées à l'intérieur d'une unité de traitement appelée transaction. Les systèmes de workflow sont focalisés sur la coordination statique (pas de configuration dynamique) et l'aspect organisationnel des processus métiers. Ces deux technologies sont incapables de résoudre le problème de la flexibilité d'une composition de services Web.

Dans cet article, nous proposons une solution permettant d'intégrer la flexibilité lors de l'exécution et d'assurer des compositions fiables de services Web. Pour y parvenir, nous introduisons un nouveau paradigme, appelé patron transactionnel dynamique, en étendant les patrons de workflow dynamique [12] avec des dépendances transactionnelles pour spécifier des compositions dynamiques. Notre approche intègre la flexibilité des patrons de workflow dynamique et la fiabilité des modèles transactionnels avancés.

Dans ce qui suit, nous présentons certains travaux connexes (section 2) et l'exemple fil rouge de l'article (section 3). Dans la section 4, nous présentons le comportement transactionnel des services composés. Dans la section 5, nous proposons des patrons de contrôle flot dynamique qui sont par la suite étendus avec des dépendances transactionnelles. La section 6 montre comment ces patrons peuvent être utilisés pour spécifier des composition fiables et flexibles.

2 Travaux connexes

Le workflow [10] est devenu une technologie clé pour l'automatisation des processus métier, offrant un support pour les aspects organisationnels, l'interface de l'utilisateur, la distribution et l'hétérogénéité [2].

Dans [18], les auteurs proposent une collection de patrons de flot de contrôle statique qui peuvent être classés en six groupes: les patrons de flot de contrôle de base, les patrons de branchement multiple et de synchronisation, les patrons structurels, les patrons d'instanciation multiple, les patrons à base d'état et les patrons d'annulation. Contrairement aux [18] et [15] qui se sont focalisés principalement sur les patrons de workflow statiques, nous nous concentrons dans notre approche sur les patrons de workflow dynamiques [12]. D'autres travaux connexes [4, 16] présentent des collections de patrons de workflow de ressources et d'interaction de services. En outre, [14] propose une série de patrons workflow de données qui visent à capturer les différentes façons dont ces dernières sont représentées et utilisées.

La fiabilité d'exécution des services Web composites est un aspect stimulant qui n'a pas été profondément examiné. Les technologies connexes actuelles sont incapables d'assurer une exécution fiable. Ces technologies peuvent être classées en deux groupes à savoir, les technologies à base de workflow comme WSBPEl [3] et WS-CDL [11] et les technologies à base de transactions comme WS-BusinessActivity [8] et WS-TXM [6]. Les technologies basées sur les transactions assurent seulement la fiabilité d'exécution. Cependant, ils sont trop rigides pour supporter des applications basées sur des processus métier comme les services Web composés dans un environnement dynamique. Les systèmes de workflow se concentrent principalement sur la coordination statique et sur les aspects organisationnels et ils ignorent les problèmes de fiabilité et l'adaptabilité. Par exemple, l'échec ou la non-disponibilité d'un composant.

Différents travaux existants tentent d'assurer la fiabilité d'exécution des services composites. Dans [5], les auteurs proposent une solution pour assurer une composition fiable en combinant les patrons de workflow statique et les modèles transactionnels avancés. La solution ainsi proposée intègre d'une part l'expressivité et la puissance des patrons de workflow statique et d'autre part la fiabilité des modèles transactionnels avancés. Elle utilise un ensemble de règles et une grammaire pour résoudre respectivement le problème de fiabilité et de cohérence du flot de contrôle dans un environnement stable, où les composants disponibles sont stables ou rarement modifiables. Ainsi, la grammaire proposée assure une connexion cohérente d'un ensemble d'instances de patron transactionnel d'un service composite. Elle peut être appliquée de façon hiérarchique si un service composant (voire plusieurs) est lui-même un service composite. Ceci permet ainsi d'utiliser des patrons d'une manière imbriquée à l'intérieur d'une composition. Par rapport à [5], l'avantage principal de notre approche est qu'elle assure non seulement la fiabilité d'exécution, mais aussi la flexibilité des services composites dans un environnement (dynamique), où les attentes des clients et les composants disponibles sont variables. Pour cela, nous introduisons un nouveau concept appelé patron transactionnel dynamique qui est une convergence entre les patrons de workflow dynamiques et les modèles transactionnels avancés.

3 Scénario

Nous choisissons d'illustrer notre approche par un scénario (figure 1) dans lequel un client désire organiser un circuit touristique.

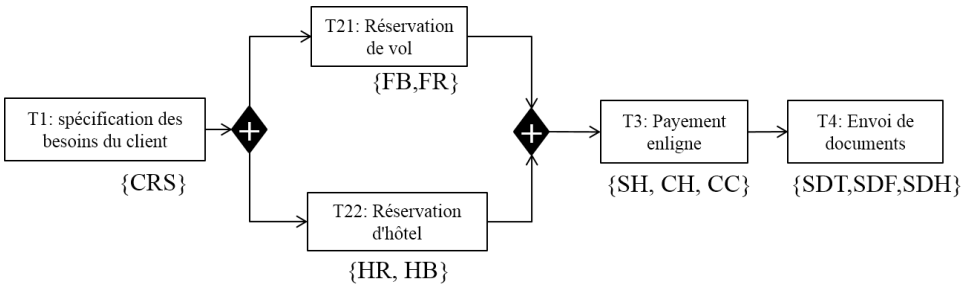


Figure 1: Scénario

Initialement, l'application OTR (Online Touristic-circuit Reservation) est composée de plusieurs tâches à savoir la spécification des besoins des clients (T1), la réservation de vol (T21), la réservation d'hôtel (T22), le paiement en ligne (T3) et enfin l'envoi des documents (T4).

Tout d'abord, le client spécifie ses préférences (destination, hôtel, etc.) en exécutant le service "Customer Requirements Specification" (CRS). Il peut également imposer des contraintes temporelles sur les activités proposées. Après la terminaison de T1, les tâches de réservation de vol et d'hôtel sont exécutées en parallèle. La tâche de réservation de vol peut être réalisée par l'un des services Web suivants:"Flight Booking" (FB) et "Flight Reservation" (FR) (d'autres services Web peuvent être découverts automatiquement). La réservation d'hôtel est exécutée soit par le service "Hotel Reservation" (HR), soit par le service "Hotel Booking" (HB).

L'application permet l'instanciation multiple des tâches de réservation d'hôtel et de vol. Ceci permet aux clients de réserver plusieurs circuits touristiques. Les autres tâches (T1,T3 et T4) sont exécutées une seule fois par requête. Après avoir terminer l'étape de réservation, le client est tenu de payer. La procédure de paiement est réalisée par plusieurs services Web, et ce, selon le mode de paiement, à savoir par carte de crédit "Credit Card" (CC), par chèque "Check" (CH), ou en espèce "caSH" (SH). Finalement, les documents de réservation sont envoyés au client en exécutant l'un des services Web suivants: "Send Document by Fedex" (SDF), "Send Document by DHL" (SDH) ou "Send Document by TNT"(SDT), selon le transporteur.

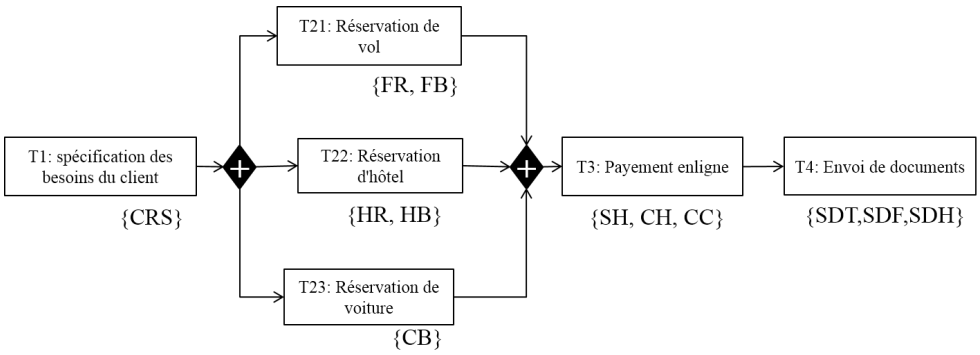


Figure 2: Le modèle du processus du service OTR après la modification

Des améliorations sont apportées au service OTR (figure 2). En effet, une activité réservation de voiture (T23) est ajoutée. Elle est facultatif (les clients ne sont pas obligés d'effectuer cette activité) et permet aux clients de réserver une voiture pour leur voyage. L'activité T23 peut être réalisée par le service "Car Booking" (CB) (d'autres services Web fournissant cette activité peuvent apparaitre ultérieurement). Elle est exécuté après la terminaison de T1 et en parallèle avec les tâches T21 et T22.

Les patrons de workflow standard ne sont pas adaptés pour la modélisation de cet exemple. En effet, ils permettent seulement de modéliser des processus métier dont la structure fixe (ne change pas) et non reconfigurable dynamiquement en cas de dysfonctionnement d'un composant. Par rapport à notre cas, le nombre d'instances des services exécutant les tâches T21 et T22 est déterminé pendant l'exécution. En plus, les clients peuvent choisir seulement les tâches T1, T21, T22, T3 et T4 pour leurs voyages.

4 Comportement transactionnel des services composés dynamiques

Dans cette section, nous proposons notre modèle de services composés dynamiques. Ce dernier intègre l'expressivité, la puissance de modèles de workflow dynamiques et la fiabilité des modèles transactionnels avancés. L'originalité de ce modèle réside dans sa flexibilité, qui est offerte non seulement aux concepteurs, mais aussi aux clients pour spécifier leurs besoins.

4.1 Service Web transactionnel

Un service Web est un composant logiciel mis à disposition sur Internet, indépendant et autonome, qui peut être automatiquement découvert, localisé et invoqué via des protocoles standards. La reutilisabilité est l'une des caractéristiques les plus intéressantes des services Web. Par conséquent, il peut être instancié et invoqué plusieurs fois soit par la même application, ou par deux applications différentes. Ces instances peuvent être exécutées simultanément. Dans notre approche, nous notons par, S_i, la i^{eme} instance du service S. Par exemple, la figure 6.a montre que le service FB est instancié et invoqué deux fois pour répondre à la requête d'un client désirant réserver deux circuit touristique. FB_1 et FB_2 sont les deux instances du service FB créées pendant l'exécution de l'application OTR.

Un service Web transactionnel est un service Web dont le comportement manifeste des propriétés transactionnelles. Les propriétés transactionnelles considérées dans notre approche sont : pivot (p), compensable (c), rejouable(r) [1, 5]. Pour modéliser le comportement transactionnel des services Web, nous utilisons une fonction TP définie comme suit: TP∈SERVICE→{c,p,r,pr,cr}, où SERVICE désigne l'ensemble des services Web transactionnels. Un service Web, s, est dit **rejouable** s'il se termine toujours avec succès après un nombre fini de réactivations (TP(s)=r). Il est dit **compensable** s'il offre des mécanismes de compensation pour annuler sémantiquement son travail (TP(s)=c) tandis qu'un service Web est dit **pivot** si une fois qu'il se termine avec succès, ses effets ne peuvent pas être compensés (TP(s)=p). En plus, il peut combiner un ensemble de propriétés transactionnelles. Il peut être à la fois pivot et rejouable (TP(s)=pr). Il peut être aussi compensable et rejouable (TP(s)=cr). Cependant, un service ne peut pas être compensable et pivot en même temps.

4.2 Service composé transactionnel dynamique

Dans [9], nous définissons un service composé transactionnel dynamique (DTCS) comme une conglomération de services Web transactionnels. Les services Web qui participent dans le processus de composition sont découverts puis combinés à la volée (pendant l'exécution) pour répondre à un objectif défini par un client.
Par exemple, la figure 6 présente deux services composites répondant à deux objectifs différents. Le premier service (figure 6.a) est composé de deux instances du service FB et une instance de chacun des services CRS, HB, CB, SH et SDT. Le deuxième (figure 6.b) est composé d'une seule instance des services CRS, FB, HB, SH et SDT, mais pas d'instance du service CB (le client n'a pas demandé la réservation d'un véhicule pour son circuit touristique).

Le comportement transactionnel d'un DTCS dépend des propriétés transactionnelles de ses services composants. Un DTCS est dit **atomique** si soit tous les services composants se terminent avec succès, soit aucun d'entre eux ne s'est terminé (abandonné, échoué, compensé, ou annulé), et une fois qu'il se termine avec succès (tous les composants se terminent normalement), ses effets ne peuvent plus être annulés. Il est dit **compensable** si tous ses composants

sont compensables, et **rejouable** s'il se termine toujours avec succès après un nombre fini de réactivations.

Les interactions entre les composants d'un DTCS expriment comment les services sont couplés et comment le comportement de plusieurs composants influence sur le comportement d'autres composants. Elles peuvent exprimer différentes sortes de relations (d'activation, de compensation, d'annulation ou d'abandon). Formellement, nous définissons la relation de dépendance (RD) par la fonction suivante: $RD \in SERVICE \leftrightarrow SERVICE$.
Par exemple, le couple (s1,s2) exprime que s1 est en relation de dépendance avec s2 (le comportement de s1 influence le comportement de s2).
Les relation de dépendances doivent respecter les contraintes suivantes:

REQ-1: Une dépendance d'abandon de s1 vers s2 ne peut exister que s'il existe une dépendance d'activation de s1 vers s2 ;

REQ-2: Une dépendance de compensation de s1 vers s2 ne peut être définie que si s2 est compensable, s1 peut échouer et soit il existe une dépendance d'activation de s2 vers s1, soit s1 et s2 s'exécutent en parallèle et ils sont synchronisés ;

REQ-3: Une dépendance d'annulation de s1 vers s2 ne peut être définie que si s1 n'est pas fiable (il peut donc échoué) et si s1 et s2 s'exécutent en parallèle et sont synchronisés ;

REQ-4: Si un service Web est pivot, alors tous les services qui lui succèdent et ceux qui s'exécutent en parallèle sont fiables.

Les règles REQ-1, REQ-2 et REQ-3 expriment des contraintes binaires assurant la cohérence des relations de dépendances entre les services Web. Elles seront utilisées par la suite afin de déduire des règles pour générer les dépendances transactionnelles (annulation, abandon et compensation). REQ-4 permet d'assurer la fiabilité d'exécution du service composé. Cette contrainte devrait être intégrée dans le processus de sélection (les services sélectionnés doivent respecter cette contrainte). Nous n'avons pas assez d'espace pour présenter ce processus de sélection dynamique qui permet de garantir des solutions optimales respectant cette contrainte. Dans cet article, nous supposons que les services sélectionnés respectent bien la contrainte REQ-4.

La dépendance d'activation, depAct ($depAct \subset RD$), permet de définir un ordre partiel sur l'activation des services Web. La dépendance de compensation, depCps ($depCps \subset RD$), définie des mécanismes de recouvrement en arrière, tandis que la dépendance d'annulation, depCnl ($depCnl \subset RD$), permet de signaler l'échec d'un service Web à ceux qui s'exécutent en parallèle. La dépendance d'abandon , depAbt ($depAbt \subset RD$), permet de propager l'échec à tous ses successeurs.

Le flot de contrôle définit un ordre partiel sur l'activation des services composants. Intuitivement, il est défini par l'ensemble de ses dépendances d'activation. Dans notre approche, nous définissons un flot de contrôle comme un DTCS dans lequel seules des dépendances d'activation sont définies. Le flot transactionnel définit les mécanismes de recouvrement en cas d'erreur. Intuitivement, il est défini selon les propriétés transactionnelles des composants et de l'ensemble des dépendances transactionnelles. Dans notre approche, nous spécifions le flot transactionnel comme un DTCS dans lequel seules des dépendances transactionnelles sont définies.

5 Patrons Transactionnels dynamiques

Dans cette section, nous introduisons le concept du patron transactionnel dynamique, un nouveau paradigme, que nous proposons pour spécifier des services composés dynamiques fiables

et flexibles.

5.1 Patrons de flot de contrôle dynamique

Les patrons du workflow dynamique [12] sont utilisés principalement pour modéliser des situations où la décision se fait avant/pendant l'exécution des processus. Contrairement aux [18] et [15] qui se concentrent sur les patrons de workflow statique, la structure des patrons de workflow dynamique est définie lors de l'exécution. Dans cet article, nous nous focalisons sur les trois patrons suivants: DAD-split (Dynamic AnD-split), DAD-join (Dynamic AnD-join) and DSQ (Dynamic SeQuence) pour expliquer et illustrer notre approche. Cependant, les concepts ainsi présentés peuvent être appliqués de la même façon à d'autres patrons comme: DOR-split (Dynamic OR-split), DOR-join (Dynamic OR-join) et DNM-join (Dynamic N out Of M Join). Dans ce qui suit, nous décrivons les patrons de workflow dynamique ainsi que leurs sémantiques formelles.

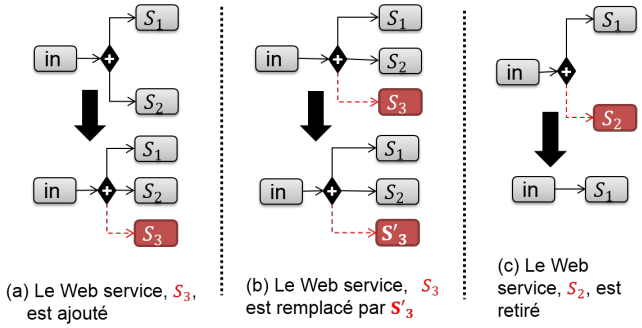


Figure 3: La re-configuration dynamique du patron DAD-split après l'ajout, la suppression ou le remplacement d'un composant

Le patron DAD-split est défini comme un point dans un procédé du workflow dynamique où un flot de contrôle se divise en plusieurs flots de contrôle en parallèle, permettant ainsi une exécution parallèle d'un ensemble de processus. Formellement, nous définissons le patron DAD-split par la fonction suivante: DAD-split $\in \mathbb{P}(CS) \nrightarrow \mathbb{P}(CS \times CS)$.
Comme montré dans la Figure 3, le patron DAD-split peut être reconfigurer dynamiquement. En effet, un service Web peut être soit ajouté (figure 3.a), retiré (figure 3.c), ou remplacé par un autre (figure 3.b). La Figure 3.c montre que le patron DAD-split peut se transformer en un patron séquence.

Le patron DAD-join est défini comme un point dans un procédé du workflow dynamique où plusieurs flots de contrôle en parallèle convergent et se synchronisent en un seul processus d'exécution. Formellement, nous définissons le patron DAD-join par suit:DAD-join $\in \mathbb{P}(CS) \nrightarrow \mathbb{P}(CS \times CS)$.
Comme illustré dans la Figure 4, le patron DAD-join peut être reconfigurer dynamiquement. En effet, un sous-processus simple peut être soit ajouté (figure 4.a), retiré (figure 4.c), ou remplacé par un autre (figure 4.b). La Figure 4.c montre qu'un patron DAD-join peut se transformer en un patron séquence.

Le patron DSQ est défini comme un point dans un procédé du workflow dynamique où un sous-processus déclenche l'exécution d'un autre. Il fournit un flot de contrôle dynamique. En effet, un sous-processus simple peut-être soit remplacé (figure 5.a) soit ajouté. Dans le deuxième

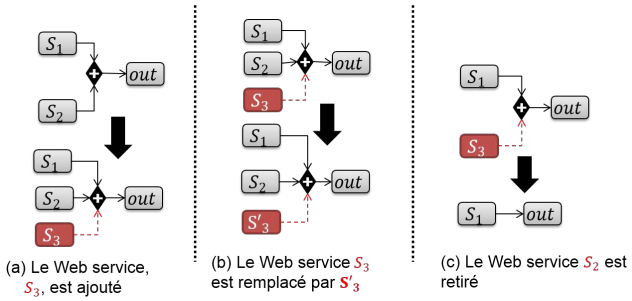


Figure 4: La re-configuration dynamique du patron DAD-join après l'ajout, la suppression ou le remplacement d'un composant

cas, un patron DSQ peut se transformer soit en un patron DAD-split (figure 5.b), soit en un patron DAD-join (figure 5.c). Formellement, nous définissons le patrons DSQ comme suit : $\text{DSQ} \in \mathbb{P}(CS) \nrightarrow \mathbb{P}(CS \times CS)$.

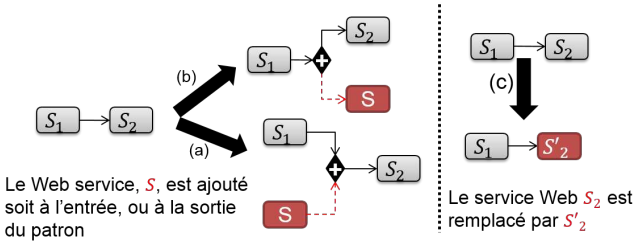


Figure 5: La re-configuration dynamique du patron DSQ après l'ajout, ou le remplacement d'un composant

5.2 Etendre les Patrons de flot de contrôle dynamique avec des dépendances transactionnels

Etant donné des propriétés transactionnelles des services Web et des patrons de flot de contrôle dynamiques, nous avons pu déduire de nouveaux patrons, appelés patrons transactionnels dynamiques, qui seront utilisés pour spécifier à la fois le flot de contrôle et le flot transactionnel.

Un patron transactionnel dynamique peut être considéré comme une coordination dynamique et comme une transaction structurelle. Il combine la flexibilité de flot de contrôle dynamique et de la fiabilité des modèles transactionnels avancés. Dans ce travail, nous présentons les patrons transactionnels dynamiques suivants DTAD-split (Dynamic Transactional AnD split), DTAD-join (Dynamic Transactional AnD join) et DTSQ (Dynamic Transactional SeQuence). Ces patrons sont définis formellement comme une fonction qui retourne une fonction F ($F \in \{act, cps, cnl, abt\} \nrightarrow \mathbb{P}(S \times S)$) déterminant les dépendances d'activation (F(act)), de compensation (F(cps)), d'abandon (F(abt)) et d'annulation (F(cnl)) qui peuvent exister entre les différentes instances de services appartenant à un ensemble S. A partir d'un patron transactionnel dynamique, plusieurs instances peuvent être définies. Une instance est l'application d'un patron transactionnel dynamique à un ensemble de services Web.

Le patron DTAD-split étend DAD-split avec un ensemble de dépendances transactionnelles. Les règles r1.1, r1.2 et r1.3 (voir Table 1) indiquent que DAD-split peut être étendu avec des dépendances d'abandon, d'annulation ou de compensation. Une dépendance d'annulation (ou de compensation) de s1 vers s2 ne peut être définie que si s1 et s2 sont synchronisés. s1 et s2 sont synchronisés si leurs terminaisons déclenchent l'activation d'un service s3 ({(s1,s3), (s2,s3)}⊂ depAct). Par exemple, DTAD-split($\{CRS_1, CB_1, FB_1, FB_2, HB_1\}$) est une instance du patron DTAD-split.

Le patron DTAD-join étend DAD-join avec un ensemble de dépendances transactionnelles. Les règles r2.1, r2.2 et r2.3 (voir Table 1) stipulent que DAD-join peut être étendu par n'importe quelle relation de dépendances (*e.g,* annulation, abandon ou compensation). Par exemple (voir Figure 6.a), DTAD-join($\{CB_1, FB_1, FB_2, HB_1, SH_1\}$) une instance du patron DTAD-join.

Le patron DTSQ étend DSQ avec des dépendances transactionnelles. Comme montré dans la Table 1, les règles, r3.1, r3.2 et r3.3 expriment que le patron DTSQ supporte seulement des relations d'abandon ou de compensation. Par exemple (voir Figure 6.a), DTSQ($\{SH_1, SDT_1\}$) est une instance du patron DTSQ.

6 Composition fiable et flexible utilisant des patrons transactionnels dynamiques

Dans cette section, nous expliquons comment nous utilisons les patrons transactionnels dynamiques pour spécifier des services Web composés fiables et flexibles. En particulier, nous montrons comment un service composite est reconfiguré automatiquement.

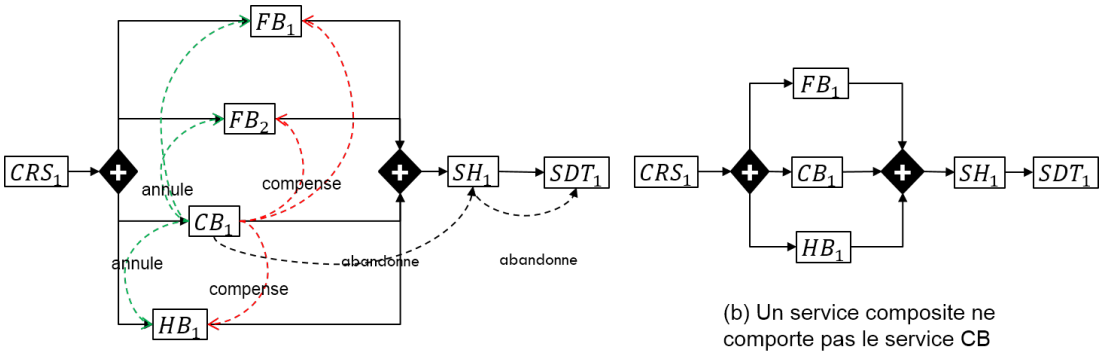


Figure 6: Deux services composés dynamiques différentes correspondant à un même modèle de processus (abstrait) du réservation de circuit touristique

6.1 Composition

Un service composite est défini comme un ensemble d'instances de patrons transactionnels dynamiques correctement reliés entre eux (partage de certains services). Soit CS un DTCS : CS=$\{I_1, I_2, \cdots, I_n\}$, où :

- $I_i = (S_i, F_i)$, $1 \leq$ i $\leq$ n ;

Patrons	Règles de cohérence transactionnels
DTAD-split	**r1.1:** ∀**x,y,S** · **S** ∈ **dom(DTAD-split)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-split(S)(cps)** ⇒ (∃ z · z ∈ S ∧ {(x,z),(y,z)} ⊆ DTAD-split(S)(act)) ∨ (y,x) ∈ DTAD-split(S)(act) **r1.2:** ∀ **x,y,S** · **S**∈ **dom(DTAD-split)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-split(S)(cnl)** ⇒ ∃ z · z∈ S ∧ {(x,z),(y,z)} ⊆ DTAD-split(S)(act) **r1.3:** ∀ **x,y,S** · **S**∈ **dom(DTAD-split)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-split(S)(abt)** ⇒ (x,y) ∈ DTAD-split(S)(act)
DTAD-join	**r2.1:** ∀ **x,y,S** · **S**∈ **dom(DTAD-join)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-join(S)(cps)** ⇒ (∃ z · z∈ S ∧ {(x,z),(y,z)} ⊆ DTAD-join(S)(act)) ∨ (y,x) ∈ DTAD-join(S)(act) **r2.2:** ∀ **x,y,S** · **S**∈ **dom(DTAD-join)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-join(S)(cnl)** ⇒ ∃ z · z∈ S ∧ {(x,z),(y,z)} ⊆ DTAD-join(S)(act) **r2.3:** ∀ **x,y,S** · **S**∈ **dom(DTAD-join)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTAD-join(S)(abt)** ⇒ x↦ y ∈ DTAD-join(S)(act)
DTSQ	**r3.1:** ∀ **x,y,S** · **S**∈ **dom(DTSQ)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTSQ(S)(cps)** ⇒ (y,x) ∈ DTSQ(S)(act) **r3.2:** ∀ **x,y,S** · **S**∈ **dom(DTSQ)** ∧ **{x,y}** ⊆ **S** ∧ **(x,y)** ∈ **DTSQ(S)(abt)** ⇒ (x,y) ∈ DTAD-join(S)(act) **r3.3:** ¬(∃ **x,y** · **{x,y}** ⊆ **dom(DTSQ)** ∧ **(x,y)** ∈ **DTSQ(S)(cnl)**)

Table 1: Les règles de cohérence des patrons transactionnels dynamiques

- $S_i \in \mathbb{P}(SWT)$ and
- $F_i \in \{act, cps, cnl, abt\} \nrightarrow \mathbb{P}(SWT \times SWT)$

L'utilisation des patrons transactionnels dynamiques permet la création des services composés flexibles (reconfigurables automatiquement). Par exemple, si un composant n'est pas disponible pendant l'exécution, alors il est possible de remplacer ce composant par un autre réalisant le même processus métier afin de valider au maximum possible la requête du client. Le processus de substitution provoque des modifications non seulement structurelles, mais aussi comportementales. La reconfiguration d'un service composite consiste à redéfinir son comportement transactionnel (flot de contrôle et flot transactionnel).

Nous proposons un algorithme (voir Algorithme 1) permettant de reconfigurer dynamiquement un service composite en remplaçant un composant qui n'est plus fonctionnel par un autre qui fourni la même fonctionnalité. Cet algorithme a comme entrée : un ensemble d'instances de patrons transactionnels CS = $\{I_1, \cdots, I_n\}$ et deux services Web : s_1 (le service Web à remplacer), s_2 (le remplaçant approprié de s_1). Nous utilisons une fonction "patternType()" pour déterminer le type de chaque instance de patron, qui est soit DTAD-split, DTAD-join, ou DTSQ.

Revenant à notre exemple fil rouge, la Figure 6.a montre un exemple d'un service composite CS=$\{I_1, I_2, I_3\}$, créé dynamiquement en utilisant les trois instances suivantes I_1, I_2 and I_3 :

- I_1 = (S_1,F_1), I_2=(S_2,F_2) et I_3=(S_3,F_3);
- S_1 = {CRS_1, FB_1, CB_1, FB_2}, S_2={FB_1, CB_1, FB_2, SH_1} et S_3={SH_1, SDT_1};
- $F1$ = {(act, SD11), (cnl, SD12), (cps, SD13)}, $F2$ = {(act, SD21), (cnl, SD22), (cps, SD23), (abt, SD24)} et $F3$ = {(act, SD31), (abt, SD32)};
- SD11 = {(CRS_1, FB_1), (CRS_1, FB_2), (CRS_1, CB_1), (CRS_1, HB_1)}, SD12 = {(CB_1, FB_1), (CB_1, FB_2), (CB_1, HB_1)} et SD13 = {(CB_1, FB_1), (CB_1, FB_2), (CB_1, HB_1)};
- SD21 = {(FB_1, SH_1), (FB_2, SH_1), (CB_1, SH_1), (HB_1, SH_1)}, SD22 = SD12, SD23 = SD13 et SD24={(CB_1, SH_1)};
- SD31 = {(SH_1, SDT_1)} et SD32 = {(SH_1, SDT_1)}.

Nous considérons maintenant le cas où le service SDT n'est plus disponible (il est retiré par son propriétaire par exemple). Après une étape de recherche exhaustive, le service SDH est sélectionné. Le remplacement de SDT par SDH peut conduire à des modification dans certaines instances de patrons. L'exécution de l'algorithme 1 indique que seule l'instance I_3 = (S,F) est modifiée comme suit :

- S = $\{SH_1, SDH_1\}$;
- F = {(act,SD1), (abt,SD2)} ;
- SD1 = $\{(SH_1, SDH_1)\}$ et SD2 = $\{(SH_1, SDH_1)\}$.

Algorithme 1 : La reconfiguration dynamique de services composés en cas de remplacement d'un composant par un autre service équivalent

```
Données : CS={I_1, ··· , I_n}, s1 (un service non-disponible) and s2 (un service alternative pour s1)
Résultat : CS'={I_1, ··· , I_n} avec s1 est remplacé par s2
début
    SF ⟵ {act,abt,cnl,cps} ;
    pour chaque I=(S,F) in CS faire
        si (s1 ∈ S) alors
            PT ⟵ patternType(I,CS) ;
            S1 ⟵ S\{s1} ∪ {s2} ;
            F1 ⟵ {} ;
            pour chaque sf in SF faire
                SD ⟵ {} ;
                pour chaque x ≠ s1 in S faire
                    si ( (x,s1) ∈ PT(S)(sf)) alors
                        SD ⟵ SD∪ {(x,s2)} ;
                    fin
                    si ((s1,x) ∈PT(S)(sf)) alors
                        SD ⟵ SD∪ {(s2,x)} ;
                    fin
                    pour chaque y ≠ x in S faire
                        si ((x,y) ∈ PT(S)(sf)) alors
                            SD ⟵ SD∪ {(x,y)} ;
                        fin
                    fin
                fin
                F1 ⟵ F1∪ { (sf,SD) } ;
            fin
            I1 ⟵ { (S1,F1) } ;
            CS ⟵ CS\{I} ;
            CS ⟵ CS∪{I1} ;
        fin
    fin
    return CS ;
fin
```

La connexion d'un ensemble d'instances de patrons transactionnels dynamiques peut conduire à une incohérence dans le flot de contrôle et dans le flot transactionnel. En effet, le problème de la cohérence du flot de contrôle peut apparaître lorsque les instances sont disjointes, ou incohérentes (avec des sémantiques incompatibles). De même, l'incohérence transactionnelle peut apparaître quand un service Web échoue provoquant l'abandon de tout le DTCS. Un DTCS est dit fiable lorsque le flot de contrôle et le flot transactionnel sont cohérents. Nous présentons dans ce qui suit comment assurer la cohérence du flot de contrôle et du flot transactionnel.

6.2 Cohérence du flot de contrôle et du flot transactionnel

La première étape pour assurer la fiabilité d'un service composite est la définition d'une structure de contrôle cohérente (flot de contrôle). Pour cela, nous proposons un automate fini déterministe

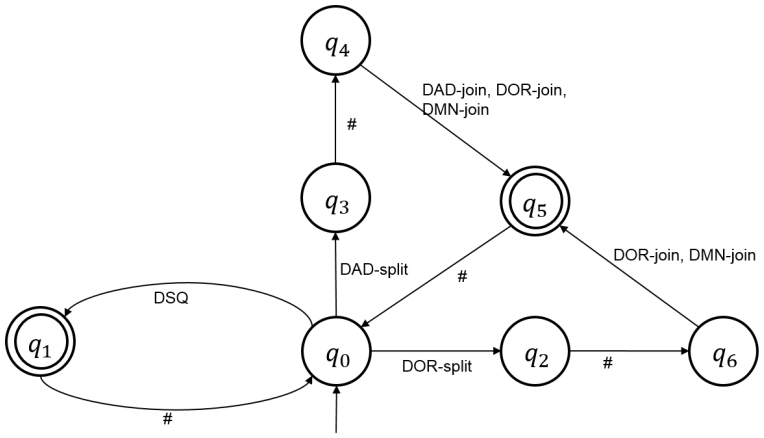


Figure 7: Un automate fini définissant le langage L(A) des flot de contrôle cohérent

défini par un quintuplet A = ($\sum$,Q,q_0, $\{q_1, q_5\}$, σ), où:

- $\sum$ ={DAD-split, DAD-join, DSQ, DOR-split, DOR-join, DMN-join, #} est un ensemble fini de symboles (l'alphabet)
- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$ est un ensemble fini d'états
- q_0 est l'état initiale
- $\{q_1, q_5\}$ sont les états finaux
- σ est une fonction partiel de (Q $\times$ $\sum$) dans Q, appelée fonction de transition.

L'automate A admet une représentation graphique qui correspond à un graphe orienté (voir Figure 7). Dans cette représentation, certains des nœuds (états) sont distingués et marqués comme initial (q_0) ou finaux (q_1 et q_5) et dans lequel les arcs (transitions) sont étiquetés par des symboles de $\sum$. Le symbole # est utilisé comme un séparateur. Pratiquement, cet automate peut être facilement implémenté et par la suite utilisé pour spécifier un flot de contrôle cohérent. Un flot de contrôle est cohérent s'il correspond à un mot appartenant à L(A). Etant donné un mot ω de L(A), ω doit commencer par l'un des alphabets suivants: DSQ, DAD-split, ou DOR-split. Le symbole DSQ est suivi par #DSQ, #DAD-split ou par #DOR-split. Le symbole DAD-split doit être suivi soit par #DAD-join, #DOR-join, ou #DNM-join, tandis que le symbole DOR-split doit être suivie par #DOR-join, ou #DNM-join. Par exemple, le mot ω = DAD-split#DAD-join#DSQ correspondant au flot de contrôle du service de réservation de circuit touristique (voir Figure 1) appartient au langage reconnu par A (L(A)). En outre, un composant (service Web) d'un DTCS donné peut être lui-même un service composite où son flot de contrôle est cohérent. Ceci permet d'utiliser des patrons de contrôle de flot dynamique d'une manière imbriquée à l'intérieur d'une composition.

La deuxième étape consiste à définir un flot transactionnel cohérent. Un flot transactionnel est cohérent s'il respecte les règles suivistes : (R1) si le processus de substitution échoue, alors les activités des services qui s'exécutent en parallèle sont annulées, et (R2) les activités réalisées sont compensées selon les dépendances de compensation définies. Pour cela, nous proposons un algorithme permettant de générer un ensemble de dépendances transactionnelles

(voir Algorithme 2). Les entrées de l'algorithme sont un ensemble d'instances de service SWT et une structure de contrôle CF (CF ∈ SWT ↔ SWT). L'algorithme utilise les règles REQ-1, REQ-2 et REQ-3 pour déduire respectivement les dependances d'abandon, de compensation et d'annulation.

Algorithme 2 : Générer les dépendances transactionnelles

Données : SWT: un ensemble de services Web transactionnels et CF: un ensemble de dépendances d'activation.
Résultat :

- depAbt: est l'ensemble de dépendances d'abandon,
- depCnl: est l'ensemble de dépendances d'annulation et
- depCps: est l'ensemble de dépendances de compensation.

début
  pour chaque s_i *in SWT* **faire**
    pour chaque $s_j \neq s_i$ *in SWT* **faire**
      si *(*$(s_i, s_j) \in CF \wedge$ *TP(*s_i*)* $\in$ {p,c}*)* **alors**
        depAbt ⟵ depAbt ∪ $\{(s_i, s_j)\}$;
      fin
      si *(TP(*s_i*)* $\in$ {p,c} $\wedge$ *synchronized(*s_i,s_j,CF*))* **alors**
        depCnl ⟵ depCnl ∪ $\{(s_i, s_j)\}$;
      fin
      si *(TP(*s_i*)* $\in$ {p,c} $\wedge$ *TP(*s_i*)* $\notin$ {c,cr} $\wedge$ *(synchronized(*s_i ,s_j,CF*)* $\vee (s_j, s_i) \in CF$*))* **alors**
        depCps ⟵ depCps ∪ $\{(s_i, s_j)\}$;
      fin
    fin
  fin
fin

Pour plus d'explication, nous considérons le cas de l'exemple fil rouge de l'article (Voir Figure 2):

- SWT=$\{CRS_1, CB_1, FB_1, FB_2, HB_1, SH_1\}$,
- CF = $\{(CRS_1,FB_1), (CRS_1,FB_2), (CRS_1,CB_1), (CRS_1,HB_1), (FB_1,SH_1), (FB_2,SH_1), (CB_1,SH_1), (HB_1,SH_1), (SH_1,SDT_1)\}$.

Les propretés transactionnelles des services Web sont requises pour générer les dépendances transactionnelles. Elles sont définies comme suit : TP(CRS) = {r}, TP(FB) = {c,r}, TP(CB) = {c}, TP(HB) = {c,r} , TP(SH) = {p,r} et TP(SDT) = {r}.
L'exécution de l'algorithme2 produit les résultats suivants:

- depAbt = { $(CB_1, SH_1), (SH_1, SDT_1)$ }
- depCnl = { $(CB_1, FB_1), (CB_1, FB_2), (CB_1, FB_1)$ }
- depCps = { $(CB_1, FB_1), (CB_1, FB_2), (CB_1, FB_1)$ }

Il est clair que ces dépendances de compensation, d'annulation et d'abandon respectent bien les règles R1 et R2. Les dépendances ainsi générées corresponds au services composé présenté dans la figure 6.a.

7 Conclusion et perspectives

Dans cet article, nous proposons une approche à la base d'un nouveau concept, appelé patron transactionnel dynamique, pour la composition des services web. Cette composition profite ainsi

de la flexibilité des patrons de workflow dynamique et de la fiabilité des modèles transactionnels. Les patrons transactionnels dynamiques s'étendent des patrons de workflow dynamiques avec des dépendances transactionnelles, permettant ainsi de réduire leur manque transactionnel. Nous montrons comment nous les utilisons pour définir des services Web composites dynamiques et comment nous assurons leur fiabilité.

Quant aux perspectives de ce travail, nous proposons de développer une stratégie efficace pour la sélection des services Web respectant essentiellement la contrainte REQ-4 et répondant mieux aux besoins des clients. Cette stratégie est basée sur les heuristiques. Intuitivement, elle permet de réduire l'espace de recherche des solutions possibles tout en garantissant en même temps des solutions optimales. La solution optimale (une composition de services Web) doit respecter la propriété d'atomicité.

References

[1] Divyakant Agrawal and Amr El Abbadi. Transaction management in database systems. In *Database Transaction Models for Advanced Applications*, pages 1–31. Morgan Kaufmann, 1992.

[2] Gustavo Alonso, Divyakant Agrawal, and Amr El Abbadi. Process synchronization in workflow management systems. In *Parallel and Distributed Processing, 1996. Eighth IEEE Symposium on*, pages 581–588, 1996.

[3] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte, et al. Business process execution language for web services, 2003.

[4] Alistair Barros, Marlon Dumas, and Arthur HM Ter Hofstede. Service interaction patterns. In *Business Process Management*, pages 302–318. Springer, 2005.

[5] Sami Bhiri, Claude Godart, and Olivier Perrin. Transactional patterns for reliable web services compositions. In *Proceedings of the 6th international conference on Web engineering*, ICWE '06, pages 137–144, New York, NY, USA, 2006. ACM.

[6] D Bunting, M Chapman, O Hurley, M Little, J Mischkinsky, E Newcomer, J Webber, and K Swenson. Web services transaction management (ws-txm) version 1.0. *Arjuna, Fujitsu, IONA, Oracle, and Sun*, 2003.

[7] Ahmed K Elmagarmid. Transaction models for advanced database applications. 1991.

[8] Robert W Freund, Hitachi Tom Freund, Frank Leymann, Ian Robinson, and Tony Storey. Web services business activity framework (ws-businessactivity). 2005.

[9] Mohamed Graiet, Imed Abbassi, Lazhar Hamel, Mohamed Tahar Bhiri, Mourad Kmimech, and Walid Gaaloul. Event-b based approach for verifying dynamic composite service transactional behavior. In *Proceedings of the 2013 IEEE 20th International Conference on Web Services*, ICWS '13, pages 251–259, Washington, DC, USA, 2013. IEEE Computer Society.

[10] Stefan Jablonski and Christoph Bussler. Workflow management: modeling concepts, architecture and implementation. 1996.

[11] Nickolas Kavantzas, David Burdett, Gregory Ritzinger, Tony Fletcher, Yves Lafon, and Charlton Barreto. Web services choreography description language version 1.0. *W3C candidate recommendation*, 9, 2005.

[12] Vitus S. W. Lam. Dynamic workflow patterns. In *Enterprise Information Systems and Web Technologies*, pages 160–166, 2008.

[13] F. Mustafa and T. L. McCluskey. Dynamic web service composition. In *Proc. Int. Conf. Computer Engineering and Technology ICCET '09*, volume 2, pages 463–467, 2009.

[14] Nick Russell, Arthur HM Ter Hofstede, David Edmond, and Wil MP van der Aalst. Workflow data patterns: Identification, representation and tool support. In *Conceptual Modeling–ER 2005*, pages 353–368. Springer, 2005.

[15] Nick Russell, Arthur HM Ter Hofstede, and Nataliya Mulyar. Workflow controlflow patterns: A revised view. 2006.

[16] Nick Russell, Wil MP van der Aalst, Arthur HM ter Hofstede, and David Edmond. Workflow resource patterns: Identification, representation and tool support. In *Advanced Information Systems Engineering*, pages 216–232. Springer, 2005.

[17] Doug Tidwell, James Snell, and Pavel Kulchenko. *Programming Web Services with SOAP*. O'Reilly, 2001.

[18] Wil MP van Der Aalst, Arthur HM Ter Hofstede, Bartek Kiepuszewski, and Alistair P Barros. Workflow patterns. *Distributed and parallel databases*, 14(1):5–51, 2003.

[19] A. H. M. ter Hofstede W. M. P. van der Aalst, A. P. Barros and B. Kiepuszewski. Advanced workflow patterns. In *O. Etzion and Peter Scheuermann, editors, 5th IFCIS Int. Conf. on Cooperative Information Systems (CoopIS'00)*, volume number 1901 in LNCS, pages 18–29, 2000.

Session commune

Posters et démos AFADL / CAL / CIEL / GDR GPL

ModHel'X, un outil expérimental pour la modélisation multi-paradigmes

Christophe Jacquet, Cécile Hardebolle and Frédéric Boulanger

SUPELEC
3 rue Joliot-Curie
91192 Gif-sur-Yvette Cedex
France
Prenom.Nom@supelec.fr

Résumé

La modélisation multi-paradigmes vise à permettre la modélisation d'un système en utilisant pour chacune de ses sous-parties le formalisme (ou paradigme) de modélisation le plus adapté. Nous nous intéressons plus spécifiquement à la modélisation du *comportement* de systèmes : dans ce cadre, il est nécessaire de pouvoir combiner la sémantique des différents formalismes utilisés dans un modèle de sorte à déterminer le comportement global du système modélisé.

Nous avons conçu un outil appelé ModHel'X qui permet a) de décrire les formalismes de modélisation utilisés grâce au concept de modèle de calcul, b) de décrire l'adaptation sémantique à la frontière entre deux formalismes, c) de construire des modèles faisant appel à plusieurs de ces formalismes et d) de simuler le comportement de tels modèles.

1 Introduction

La modélisation multi-paradigmes [4] vise à permettre la modélisation d'un système en utilisant pour chacune de ses sous-parties le formalisme (ou paradigme) de modélisation le plus adapté. Nous nous intéressons spécifiquement à la modélisation du comportement des systèmes. Dans ce cadre, les formalismes de modélisation utilisés sont par exemple les automates, les flots de données, les systèmes d'équations différentielles en temps continu, etc.

Ce type de modélisation prend une importance croissante dans la conception des systèmes embarqués dans le cadre de l'ingénierie dirigée par les modèles. Actuellement, la plupart des formalismes sont soit décrits de manière informelle en langage naturel, soit décrits par une implémentation de référence qui est généralement l'outil qui supporte le modèle. Pour permettre la spécification de systèmes de façon indépendante des outils utilisés, nous cherchons à décrire les formalismes de façon précise, mais à un niveau d'abstraction plus élevé que celui des langages de programmation.

Nous développons depuis plusieurs années ModHel'X, un outil expérimental pour la modélisation mutli-paradigmes. ModHel'X s'inspire de Ptolemy II [2]. Notamment, il emprunte à ce dernier la description des formalismes de modélisation en tant que *modèles de calculs* (abrégés MoC pour *Model of Computation*).

ModHel'X propose un méta-modèle permettant de décrire la structure de modèles de systèmes, et un moteur d'exécution générique qui s'appuie sur un certain nombre d'opérations primitives pour décrire la sémantique des modèles de calcul, ainsi que les interactions entre modèles de calculs différents. Ceci permet de donner une sémantique d'exécution précise aux modèles multi-paradgimes. ModHel'X a été décrit en détails dans [1].

Cette démonstration s'intéresse aux principaux aspects de ModHel'X : la description de modèles grâce à un méta-modèle générique, indépendant du MoC (section 2), la description

des modèles de calcul eux-mêmes (section 3), la description de l'adaptation sémantique entre MoC (section 4). La section 5 résume la contribution, indique comment obtenir ModHel'X pour expérimenter soi-même, et donne quelques perspectives.

2 Description de modèles : méta-modèle générique

Afin de permettre la construction de modèles multi-paradigmes, ModHel'X introduit un méta-modèle générique, indépendant du modèle de calcul. Tout modèle, quelle que soit sa sémantique, est donc décrit en utilisant un petit nombre de constructions syntaxiques :

Bloc C'est l'unité élémentaire de comportement. Il s'agit d'une boîte noire dont l'interface est spécifiée.

Patte (pin) C'est le mécanisme d'échange d'informations entre blocs. L'ensemble des pattes d'un bloc forme son interface.

Relations Elles connectent entre elles des pattes de différents blocs.

Un modèle est composé d'une structure (assemblage syntaxique de blocs dont les pattes sont connectées par des relations) et d'un MoC qui spécifie sa sémantique. Une même structure peut donc être interprétée différemment, selon le MoC qui lui est associé (voir figure 1).

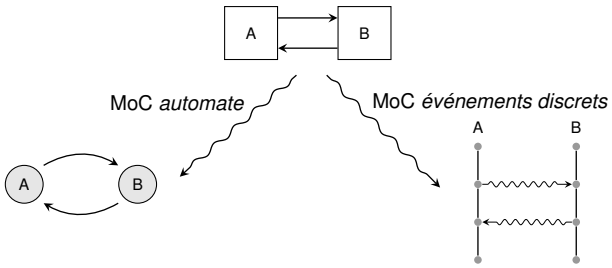


FIGURE 1 – Différents modèles de calcul donnent une interprétation différente de la même structure : états et transitions dans le cas d'un automate, processus s'échangeant des messages dans le cas d'un modèle à événements discrets.

L'hétérogénéité dans les modèles multi-paradigmes est obtenue en structurant hiérarchiquement un modèle : des blocs spéciaux appelés *blocs d'interface* permettent d'embarquer un modèle obéissant à un certain MoC dans un modèle obéissant à un autre MoC (voir figure 2).

3 Description de modèles de calculs

ModHel'X repose sur une approche boîte noire : pour déterminer le comportement global d'un modèle, le moteur d'exécution doit observer les comportements des blocs qui le composent, puis combiner ces observations selon les règles du MoC.

ModHel'X détermine donc une succession d'observations instantanées appelées *snapshots*. Chaque snapshot est calculé grâce à trois étapes :

Schedule Le MoC détermine quel est le prochain bloc à observer.

Update Ce bloc est observé.

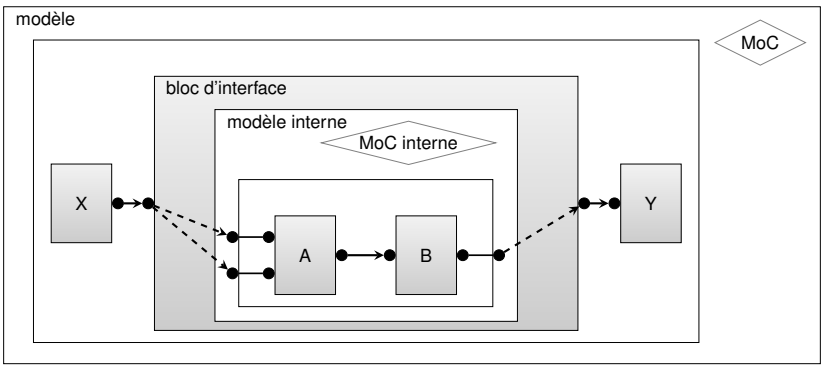


FIGURE 2 – Utilisation de la hiérarchie pour construire un modèle multi-paradigmes.

Propagate Le MoC propage les données dans le modèle.

Chacune de ces étapes permet d'observer une partie du modèle. Cet enchaînement est itéré jusqu'à la détermination complète de l'observation du modèle, c'est-à-dire jusqu'à trouver un point fixe.

L'ensemble de ces trois étapes représente la sémantique abstraite de ModHel'X. Le comportement de chaque bloc vient concrétiser l'opération *update*, et de même chaque MoC concrétise *schedule* et *propagate*. Ainsi, créer un MoC revient à créer des opérations *schedule* et *propagate* pour expliciter sa sémantique. Actuellement, ces opérations sont codées en Java, mais nous envisageons à l'avenir de les décrire dans un langage dédié.

4 Description de l'adaptation sémantique entre MoC

L'adaptation sémantique recouvre trois aspects :

L'adaptation des données Dans le cas général, les données doivent être transformées lorsqu'on passe d'un modèle de calcul à un autre. Par exemple, un automate manipule des événements purs. S'il est intégré dans un modèle de type *événements discrets* (DE), il faut déterminer quelles données faire circuler dans DE suite à l'émission d'événements par l'automate.

L'adaptation du contrôle Les instants d'observation dépendent du modèle de calcul. Par exemple, un automate n'est observé que lorsqu'il réagit, tandis qu'un modèle échantillonné de type *flot de données synchrone* (SDF) doit être observé à chaque instant d'échantillonnage.

L'adaptation du temps Différents modèles peuvent avoir des notions de temps différentes, qu'il est nécessaire de connecter entre elles. Par exemple, différents sous-modèles peuvent avoir des échelles de temps différentes.

Les blocs d'interface sont chargés de mettre en œuvre les trois facettes de l'adaptation. Nous avons créé un langage appelé TESL (Tagged Events Specification Language), qui permet de spécifier déclarativement une partie importante de l'adaptation du temps et du contrôle. Pour le moment, une partie de l'adaptation, notamment l'adaptation des données, est réalisée par du code Java appelé par l'algorithme d'exécution générique de ModHel'X. Cependant, des travaux sont en cours pour remplacer ce code Java par l'utilisation d'un langage dédié à l'adaptation [3].

5 Conclusion et perspectives

Nous avons présenté ModHel'X, un outil expérimental de modélisation multi-paradigmes. Nous avons vu comment sont décrits les modèles, les formalismes de modélisation (MoC), et les adaptateurs entre MoC. En utilisant ces trois éléments, le moteur générique d'exécution de ModHel'X peut simuler des modèles.

ModHel'X est inspiré de Ptolemy II, auquel il ajoute la modélisation explicite et modulaire de l'adaptation sémantique entre modèles de calcul. Les *blocs d'interface* n'ont pas d'équivalents en Ptolemy II, ce qui oblige soit à utiliser une sémantique d'adaptation implicite, soit à modifier les modèles pour y intégrer l'adaptation, ce qui limite fortement la réutilisabilité des modèles.

ModHel'X est actuellement implémenté dans l'écosystème Java. Il est diffusé sous forme de logiciel libre. La page web `http://wwwdi.supelec.fr/software/ModHelX/` permet de télécharger l'outil en lui-même ainsi que différentes démonstrations. Elle donne également accès aux publications sur le sujet.

Pour le moment, nos efforts se sont tournés essentiellement vers l'expression d'une sémantique opérationnelle pour les MoC et les modèles combinant différents MoC. À l'avenir, il serait intéressant de donner aux MoC et aux modèles une sémantique qui soit analysable. Cela permettrait par exemple de prouver des propriétés portant sur un MoC, ou sur un bloc, et d'en déduire des propriétés portant sur un modèle dans son ensemble.

Références

[1] Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Dominique Marcadet. Semantic Adaptation for Models of Computation. In *Proceedings of ACSD 2011 (Application of Concurrency to System Design)*, pages 153–162. IEEE Computer Society, June 2011.

[2] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity – the Ptolemy approach. *Proceedings of the IEEE, Special Issue on Modeling and Design of Embedded Software*, 91(1) :127–144, January 2003.

[3] Bart Meyers, Joachim Denil, Frédéric Boulanger, Cécile Hardebolle, Christophe Jacquet, and Hans Vangheluwe. A DSL for Explicit Semantic Adaptation. In *Proceedings of MPM 2013 (Multi-Paradigm Modeling workshop at Models 2013)*, number 1112 in CEUR Workshop Proceedings, pages 47–56. CEUR, December 2013.

[4] P. J. Mosterman and H. Vangheluwe. Computer Automated Multi-Paradigm Modeling : An Introduction. *SIMULATION*, 80(9) :433–450, 2004.

Évaluation de la substituabilité comportementale de composants UML

Thomas Lambolais[1], Anne-Lise Courbis[1] and Thanh-Liêm Phan[2]

[1] École des mines d'Alès, Parc scientifique G. Besse, 30035 Nîmes Cedex, France
thomas.lambolais@mines-ales.fr, anne-lise.courbis@mines-ales.fr
[2] School of Information and Communication Technology, Hanoi University of Science and Technology, 601 B1, Hanoi University of Science and Technology, Bach Khoa, Hanoi, Vietnam
phanliem@soict.hust.edu.vn

Résumé

Nous présentons le problème de la substituabilité de composants dans une architecture UML. Le composant de substitution doit assurer que la nouvelle architecture préserve les propriétés comportementales offertes par l'ancienne architecture, propriétés pouvant se traduire par des concepts de vivacité et de sûreté. Nous montrons quels sont les problèmes liés à la substitution de composants à travers un exemple et une démonstration de l'outil IDCM (*Incremental Development of Compliant Models*) que nous avons développé.

1 Introduction

La construction et l'évolution de modèles architecturaux s'appuie sur l'utilisation et la réutilisation de modèles de composants. Dès lors, la question de la substitution de composants est cruciale dans les processus de construction d'architectures. La notion de substitution est centrale dans les langages orientés-objet, où la relation de spécialisation a été présentée comme permettant la substitution. De même, dans le langage de spécification formel B, la relation de raffinement est une relation de substitution.

Néanmoins, telle qu'elle est définie, la relation de spécialisation des langages orientés-objet n'est pas suffisante pour effectivement garantir la substituabilité. D'une part, elle n'est définie que sur les interfaces des composants, sans prendre en compte les séquences d'interaction entre le composant à substituer et son environnement. D'autre part, elle permet l'ajout de nouvelles opérations, rendant impossible la préservation des propriétés de sûreté.

En langage B, cette lacune est corrigée : le raffinement préserve les propriétés de sûreté, mais ce sont alors les propriétés de vivacité qui ne sont pas toutes préservées.

Afin d'offrir les garanties de qualité escomptées aux modèles, la relation de substitution que nous avons retenue et outillée préserve les propriétés de sûreté et de vivacité. Nous montrons quels sont les problèmes liés à la substitution de composants sur un exemple et une démonstration de l'outil IDCM (*Incremental Development of Compliant Models*) [4] que nous avons développé.

2 Illustration de la problématique

Afin d'illustrer les difficultés liées à la substitution de composants, nous représentons plusieurs variantes de machines d'états UML pour un composant de connexion à un service via un réseau bluetooth (figure 1). Le composant ne fournit que deux opérations connect et disconnet. Il cherche à statisfaire la demande de connexion bluetooth connect par l'activité privée connectionRequest. Cette activité se termine après un temps indéterminé. Dans une première

approche, le réseau est fiable et la tentative de connexion aboutira de manière sûre, après une durée inconnue. Ce sont les machines M_1 et M_2 de la figure 1. Dans une seconde approche plus réaliste (cas des machines M_3, M_4 et M_5) le réseau n'est pas fiable et la tentative de connexion connectionRequest peut échouer. On peut voir M_1 comme une spécification, ou abstraction, de M_2, et M_3 comme une spécification de M_4 : la machine M_2 doit pouvoir remplacer la machine M_1 ; la machine M_4 doit pouvoir remplacer la machine M_3. En revanche, M_5 est conçue de façon erronnée et ne doit pouvoir remplacer ni M_3 ni M_4.

Cherchons à vérifier ces hypothèses grâce à des relations formelles entre machines. Nous donnons une sémantique des machines d'états UML sur les systèmes de transitions étiquetées (LTS : *Labelled Transition Systems*) et une sémantique des structures composites sur une algèbre de processus communicants. Les algèbres de processus telles que CCS (*Calculus of Communicating Systems*) [5] et CSP (*Communicating Sequential Processes*) [2] nous apportent des résultats théoriques centraux, où des relations d'équivalence, de préordre, et de congruence ont été définies. Rappelons qu'une relation d'équivalence est une relation de congruence si elle est préservée dans tout contexte. C'est précisément ce que l'on attend d'une relation de substitution, au moins pour les contextes architecturaux de composition parallèle et d'encapsulation. Les relations de raffinement de CSP sont trop restrictives dans la mesure où elles considèrent toutes les divergences (activités qui ne sont plus sous contrôle de l'environnement) comme critiques. Ainsi, la machine M_2 de la figure 1 ne pourrait pas remplacer la machine M_1 au sens de la relation d'équivalence $=_{\mathsf{FDR}}$ associée au préordre FDR (*Failure Divergence Refinement*) de CSP [6], car elle offre un risque de divergence dans l'activité interne connectionRequest. Nous préférons adopter le point de vue de Milner dans CCS qui ne considère pas cela comme une divergence critique : M_1 est observationnellement congruent à M_2 ($M_1 =_{obs} M_2$). Cependant, les relations de bisimulation de Milner, même pour la congruence observationnelle, ne reflètent pas toujours fidèlement le côté observationnel et distinguent M_3 et M_4. Les seules opérations visibles étant connect et disconnect, si l'on fait abstraction du temps d'exécution, ces deux machines ne peuvent pas être distinguées. Non seulement elles possèdent les mêmes traces d'exécution (à savoir $\{\mathsf{connect}^*, (\mathsf{connect}^*.\mathsf{connect}.\mathsf{disconnect})^*\}$), mais de plus, après toute trace, elles peuvent refuser les mêmes actions : elles sont équivalentes au sens de la conformité ($M_3 =_{\mathrm{conf}} M_4$).

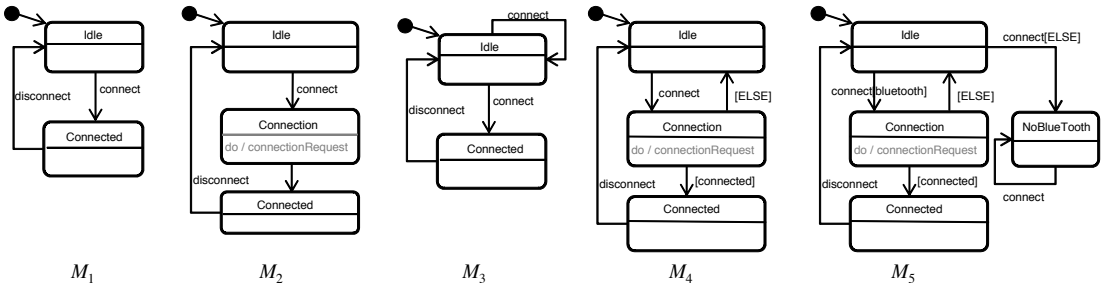
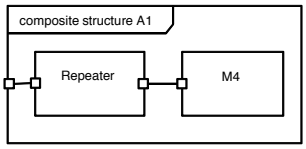


FIGURE 1 – L'équivalence de conformité traduit la proximité de M_1 avec M_2 et M_3 avec M_4 : $M_1 \neq_{\mathsf{FDR}} M_2$, $M_1 =_{obs} M_2$, $M_1 =_{\mathrm{conf}} M_2$ $M_3 \neq_{\mathsf{FDR}} M_4$, $M_3 \neq_{obs} M_4$, $M_3 =_{\mathrm{conf}} M_4$. Cependant, elle ne détecte pas les différences entre M_4 et M_5 : $M_4 =_{\mathrm{conf}} M_5$.

L'équivalence de conformité traduit le fait que M_4 est une implantation correcte de M_3. Elle garantit que les deux machines possèdent les mêmes propriétés de vivacité et de sûreté. Malheureusement, cette relation n'est pas congruente dans le cas général, comme en témoignent les machines M_4 et M_5 sur lesquelles nous nous appuierons pour la démonstration de l'outil IDCM. À la différence de M_4, M_5 est une machine qui ne renouvelle pas les tentatives de

connexion connectionRequest si le bluetooth n'a pas été activé. Elle peut ainsi tomber dans un état où les demandes de connexion connect n'aboutiront pas. Par l'équivalence de conformité, il est toutefois impossible de voir la différence entre M_4 et M_5, car déjà dans M_4, les tentatives de connexion connectionRequest peuvent échouer de façon répétée.



On place à présent M_4 dans une architecture A_1, en communication avec un composant répétant autant de fois que nécessaire l'opération connect et déclarant cette opération comme privée. L'architecture A_2 s'obtient à partir de A_1 en substituant M_4 par M_5 : on constate alors que $A_1 \neq_{\text{conf}} A_2$. A_1 devra accepter disconnect après un temps indéterminé, alors que A_2 peut le refuser éternellement.

Cet exemple montre que l'équivalence de conformité est une relation trop faible pour caractériser la substituabilité. Inversement, l'équivalence observationnelle est trop forte.

3 Analyse de la substituabilité des modèles par IDCM

Trouver une relation plus forte que $=_{\text{conf}}$ qui soit une congruence est resté un problème ouvert pendant de nombreuses années. Le préordre should [1] et l'équivalence associée sont une réponse à ce problème. Cependant, aucun algorithme n'était proposé jusque là. Rensink et Vogler [7] ont montré la décidabilité de cette relation, ce qui nous a permis d'implanter l'algorithme de calcul de substitution au sein d'IDCM.

Nous avons développé la boîte à outils IDCM [4, 8] afin de réaliser des vérifications entre des modèles construits dans un cadre incrémental. Les principales relations vérifiées sont la conformité [3], l'extension, le raffinement et la substitution [8]. Les principales étapes menant à la vérification sont les suivantes : la transformation des modèles en LTS, la construction de graphes d'acceptance ou d'arbres de refus selon la relation, et enfin la vérification de la relation conduisant à un verdict qui, en cas d'échec, met en évidence une trace et des ensembles de refus en expliquant la cause. Illustrons sur l'exemple des machines M_4 et M_5 de la figure 1 les étapes de transformation et de vérification de la substituabilité.

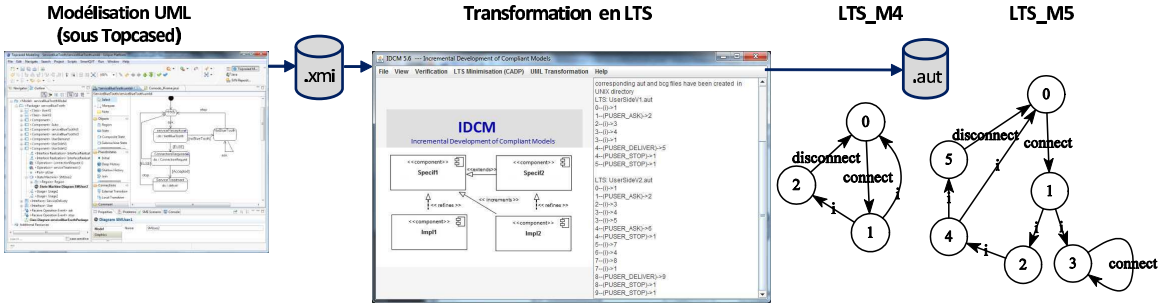


FIGURE 2 – Processus de transformation de machines d'états en LTS sous IDCM

IDCM possède un module (cf. figure 2) d'analyse et de transformation des machines d'états UML (.xmi) en LTS, au format Aldebaran de CADP .aut. Pour chaque composant, le transformateur identifie les opérations privées, celles des interfaces fournies et requises ainsi que ses ports de communication. Pour répondre aux critères d'encapsulation et de réutilisabilité des composants, toute communication avec le composant doit être faite au travers de ses ports. La transformation analyse la machine d'états du composant en faisant abstraction des données et des opérations internes afin de ne mettre en évidence que les interactions du composant avec son environnement. La partie droite de la figure 2 montre les LTS générés pour l'analyse des

machines M_4 et M_5. La transformation est réalisée en Java-DOM et repose sur un ensemble de règles spécifiques à des configurations d'états/transitions en fonction de la présence de triggers, gardes et effets.

Le calcul de substituabilité de M_4 par M_5 sous IDCM détecte (cf. figure 3) que M_5 ne peut être substitué à M_4. L'explication donnée dans la partie droite de l'interface met en évidence que M_4 doit accepter l'automate correspondant à la trace infinie (connect.connect*.disconnect) alors que M_5 peut refuser cet automate. Cette relation de substitution proposée par Brinksma,

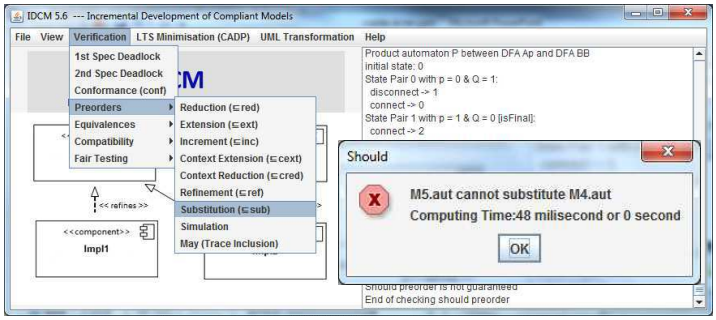


FIGURE 3 – Échec de la substituabilité de M_4 par M_5

Rensink et Vogler correspond effectivement à la plus faible congruence, plus forte que $=_{\text{conf}}$. Cependant, l'algorithme est de complexité exponentielle. Pour réduire les temps de calcul, il est donc judicieux de ne l'utiliser que dans les cas où la relation $=_{obs}$ n'est pas satisfaite (cette dernière est de complexité linéaire), ainsi que de l'appliquer sur des systèmes de transitions minimisés au sens de $=_{obs}$.

Références

[1] Ed Brinksma, Arend Rensink, and Walter Vogler. Fair testing. In Insup Lee and Scott A. Smolka, editors, *Concurrency Theory*, volume 962 of *Lecture Notes in Computer Science*, pages 313–327, Berlin, Heidelberg, 1995. Springer-Verlag.

[2] Charles Antony Richard Hoare. *Communicating sequential processes*. Prentice Hall International., 2004.

[3] Hong-Viet Luong, Thomas Lambolais, and Anne-Lise Courbis. Implementation of the Conformance Relation for Incremental Development of Behavioural Models. In Krzysztof Czarnecki, editor, *Proceedings of 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2008)*, volume 5301/2009 of *LNCS*, pages 356–370. Springer-Verlag, 2008.

[4] Hong-Viet Luong, Liem Phan, Thomas Lambolais, and Anne-Lise Courbis. IDCM : un outil d'analyse de composants et d'architectures dédié à la construction incrémentale. In *AFADL*, pages 50–53, 2012.

[5] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., 1989.

[6] Oxford University Computing Laboratory. Failures-Divergence Refinement (FDR2 User Manual), October 2010.

[7] Arend Rensink and Walter Vogler. Fair testing. *Information and Computation*, 205(2) :125–198, 2007.

[8] Thanh-Liêm Phan. *Développement incrémental de spécifications d'architectures en UML intégrant des procédures de vérification*. PhD thesis, université Montpellier II, December 2013.

BUT4Reuse Feature identifier: Identifying reusable features on software variants

Jabier Martinez[1,2], Tewfik Ziadi[1], Jacques Klein[2] and Yves le Traon[2]

[1] LIP6, Université Pierre et Marie Curie, Paris, France
(jabier.martinez|tewfik.ziadi)@lip6.fr
[2] SnT, University of Luxembourg, Luxembourg, Luxembourg
(jacques.klein|yves.letraon)@uni.lu

Abstract

Software-intensive companies apply software reuse to increase their productivity. However different levels of software reuse exist ranging from ad-hoc reuse such as copy-paste, to systematic software reuse as promoted by Software Product Line Engineering (SPLE). In SPLE, the commonality and variability in the functionalities of a set of product variants is captured by a feature model. Therefore, the feature model captures the reusable features of the scoped products. Companies with existing similar products could increase the productivity and maintainability by achieving this higher levels of reuse. Bottom-Up Technologies for Reuse (BUT4Reuse) tackle the challenging issue of SPL adoption given a set of existing variants. In this paper we present one of BUT4Reuse main functionalities: Identifying features from software variants. The tool integrates and provides new feature identification techniques that could be applied and extended to any kind of software artifacts.

keywords: Software Reuse, Software Product Line Engineering, Feature identification

1 Introduction

Software Product Line Engineering (SPLE) [7] is a software development paradigm designed to handle software products variants that share a common set of features. While objects and components enhance the reuse of software, SPLE performs a step further by allowing the reusability and management of software features. The benefits of SPLE include the reduction of both the maintenance effort and the time to market [3]. The products of a Software Product Line (SPL) are usually represented by a variability model called Feature Model (FM) [1] which allows building tailored products by combining the features [4].

SPLE can be implemented as a top-down approach. In that case, features and variability are first specified at the design stage and then software products are built. The top-down process is useful when SPLE is adopted from the beginning. However, current practices are different. As reported by *Berger et al.* [2], most of the industrial practitioners first implement several software products and then try to manage them. These product variants are created using ad-hoc techniques, e.g., copy-paste-modify. This is not the most efficient practice leading to a complex management and a low product quality. Thus, migrating such products to an SPL is the challenge faced by extractive approaches in SPLE [5].

In this paper we present one of the main functionalities of Bottom-Up Technologies for Reuse (BUT4Reuse): The Feature identifier[1]. This functionality allows the extraction of reusable features from similar product variants. The objectives of this development are 1) to transfer state-of-the-art feature identification techniques to the companies in a user-friendly way to help

[1]The source-code of BUT4Reuse Feature identifier could be found at https://bitbucket.org/jabi/but4reuse

them adopting SPLE in a semi-automatic guided process, and 2) develop a community around this open-source tool for researchers and practitioners dealing with mining existing assets for SPL adoption.

2 Bottom-Up Technologies for Reuse: Feature identifier

One of the major requirements of BUT4Reuse is to be a generic framework that can support any kind of software artifacts. Software artifacts includes for example source code, documentation files, requirements, models or configuration files.

For the feature identifier functionality of BUT4Reuse, we considered two claims:

- Each software artifact variant can be divided in a set of atomic pieces.
- The equivalence relationship between the atomic pieces can be formally defined.

Given these claims, considering any kind of artifacts it is sufficient to: 1) define the atomic pieces related to these artifacts, 2) define the equivalence relationship. To illustrate this with a simple example we will consider plain text configuration files: The atomic pieces could be the configuration statements on each line and the equivalence relationship between two atomic pieces could be defined as a string equivalence. As we have proven more complex artifacts could be managed with this approach such as source code [9] or models conforming to any metamodel [6].

Based on the representation of the input artifact variants as a set of atomic pieces, we implemented within BUT4Reuse an algorithm [8] that formally identifies commonality and variability between the artifact variants as a set of features. A given set of product variants could share atomic pieces among them, and this algorithm is able to identify the features that corresponds to these intersections as illustrated in figure 1.

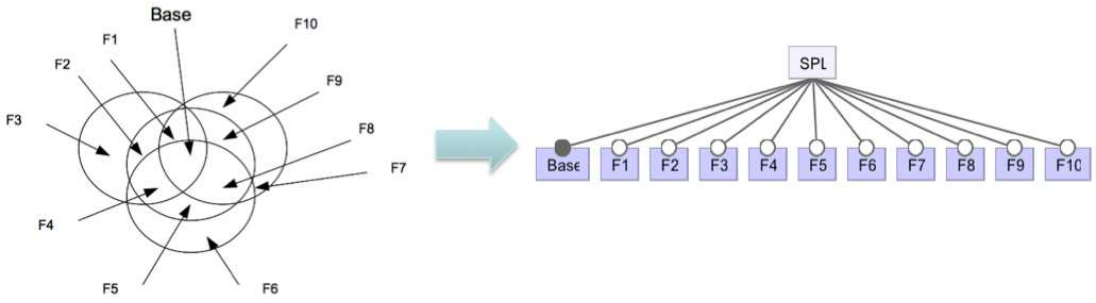


Figure 1: Input: Software variants. Output: Identified features

BUT4Reuse is a built-on Eclipse tool that provides facilities to simplify the Feature identification process, in particular to manage the location of the software variants, and to visualize from different perspectives the automatic feature identification results. BUT4Reuse contains extension points to easily include support to new types of artifacts. However, the current version contains ready-to-use feature identification adapters for source code, models conforming to any metamodel, plain text files, and files/folders structures.

References

[1] David Benavides, Sergio Segura, and Antonio Ruiz Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.*, 35(6):615–636, 2010.

[2] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. A survey of variability modeling in industrial practice. In *VaMoS*, page 7, 2013.

[3] Hyunsook Do, Siavash Mirarab, Ladan Tahvildari, and Gregg Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *FSE*, pages 71–82, 2008.

[4] K.C. Kang, Jaejoon Lee, and P. Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58 – 65, jul/aug 2002.

[5] Charles W. Krueger. Easing the transition to software mass customization. In *PFE*, volume 2290, pages 282–293, 2001.

[6] Jabier Martinez, Tewfik Ziadi, Jacques Klein, and Yves le Traon. Identifying and visualising commonality and variability in model variants. In *Proceedings of the 2014 European Conference on Models Foundations and Applications (ECMFA 2014)*, July 2014.

[7] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. 2005.

[8] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature identification from the source code of product variants. In *CSMR*, pages 417–422, 2012.

[9] Tewfik Ziadi, Christopher Henard, Mike Papadakis, Mikal Ziane, and Yves le Traon. Towards a language-independant approach for reverse-engineering of software product lines. In *Proceedings of the 29th Symposium On Applied Computing (SAC 2014)*, March 2014.

Recherche de sous-modèles

Gilles Vanwormhoudt[1,2], Bernard Carré[1], Olivier Caron[1]
and Christophe Tombelle[1,2]

[1] LIFL, UMR CNRS 8022
University of Lille
France - 59655 Villeneuve d'Ascq cedex
Gilles.Vanwormhoudt@lifl.fr
[2] Institut Mines-TELECOM

Résumé

La notion de sous-modèle intervient dans de nombreuses opérations et activités de l'IDM. Dans le cadre de nos travaux, nous avons établi une formulation ensembliste qui a permis d'isoler plusieurs concepts de sous-modèle avec leurs propriétés. Nous présentons ici une application des résultats pour la recherche de sous-modèles dans des entrepôts. Cette démonstration est réalisée dans l'environnement Eclipse à l'aide d'un ensemble de plugins offrant des fonctionnalités génériques pour manipuler des sous-modèles dans le monde EMF.

1 Introduction

Dans l'IDM, les opérations de manipulation, de comparaison, d'extraction de parties de modèles sont au coeur de nombreuses activités de modélisation : construction incrémentale, composition [9], compréhension de modèles par décomposition [7], versionnement [1], ... Cependant, malgré son importance pour structurer l'espace de modélisation, la notion de sous-modèle a été relativement peu étudiée et soulève de nombreuses questions fondamentales et récurrentes comme celle de savoir si un modèle est inclus dans un autre ou de déterminer si un modèle est la composition d'autres.

Afin de répondre à ces questions, nous avons établi dans [5] une formulation ensembliste qui permet de définir formellement et de façon uniforme des modèles, des sous-modèles et n'importe quelle partie de modèle mais aussi de caractériser précisément leur relation d'inclusion. À partir de cette formulation, nous avons isolé plusieurs concepts de sous-modèles (inclus, fermé, covariant et invariant) qui ont permis de dégager des propriétés intéressantes comme la fermeture limite d'un sous-modèle par rapport à un autre ou la transitivité entre sous-modèles invariants. Comme indiqué dans [5], ces notions de sous-modèle et leurs propriétés ont de nombreuses applications pour la composition et la transformation de modèles mais aussi leur édition collaborative.

Dans ce travail, nous présentons une application des sous-modèles pour des entrepôts de modèles. De tels entrepôts visent à faciliter la mémorisation des efforts de conception et leur réutilisation sous la forme de bibliothèques de modèles ou encore de modèles sur étagères [3, 2, 6, 11, 8]. Grâce à ces entrepôts, il est possible de construire de nouveaux modèles à partir de modèles existants. Cet objectif réclame des capacités d'enregistrement, d'organisation et de recherche d'artefacts allant du simple ensemble non structuré d'éléments de modèle jusqu'à des modèles à part entière en passant par toute partie bien ou mal formée.

Après une présentation rapide de nos résultats sur les sous-modèles, nous précisons l'objet de la démonstration en lien avec ceux-ci.

2 Sous-modèles et leurs propriétés

Dans les travaux existants, la notion de sous-modèle a principalement été utilisée pour construire et extraire des parties d'un modèle complexe [7, 10, 4]. Dans ces utilisations, l'objectif est d'obtenir des sous-modèles bien formés par rapport au métamodèle ou du moins une partie de celui-ci. Dans nos travaux, nous sommes partis d'une définition de modèle assouplie afin de prendre en compte une famille plus large de (sous)-modèles. Cette définition consiste en un ensemble d'éléments muni de contraintes de dépendance entre ces éléments. La figure 1 montre des exemples de sous-modèles manipulables grâce à cette définition, tous issus du même modèle *HeatingSystem* (*m4* en bas de la figure). Elle est suffisamment générale pour représenter des modèles complets, des parties de modèles qui peuvent être bien formées (par exemple *m2'*, *HeatingSubCircuit*) ou non (dégénérées, par exemple *m1*, *m3'*) ainsi que de simples ensembles d'éléments de modèles (*m0*).

Plusieurs relations de sous-modèles ont été établies afin de déterminer le niveau d'inclusion d'un modèle dans un autre. Au niveau le plus simple, nous avons une relation de sous-modèle qui se base sur l'inclusion d'ensemble d'éléments de modèle, sans prendre en compte leur structure. Une seconde relation nommée "closed" permet de déterminer la fermeture d'un modèle dans un autre, i.e de déterminer si les éléments d'un modèle inclus dans un autre n'ont pas d'autres dépendances dans ce dernier. Deux autres relations de sous-modèles prennent en compte leur structure. Il s'agit des relations de sous-modèle "covariant" et "invariant" selon l'inclusion ou l'égalité de leur contraintes relatives. La figure 1 synthétise les relations qui existent entre les modèles pris en exemple.

Plusieurs propriétés ont été démontrées. On citera en particulier : la propriété de "clôture limite" qui permet de caractériser la localité et les qualités modulaires de sous-modèles fermés ; la propriété de "transitivité" de la relation d'invariance qui caractérise la stabilité de contenu et de structure entre sous-modèles.

Les résultats précédents fournissent des fondements pour analyser les différentes opérations qui manipulent des sous-modèles dans l'IDM. Par exemple, nous les avons mis à profit pour l'étude systématique de l'opérateur d'extraction d'un modèle à partir d'un autre en sélectionnant un ensemble d'éléments. Cette étude a permis de qualifier les relations d'inclusion entre modèles d'entrée et modèles résultats et donc de caractériser précisément cet opérateur [5].

3 Application à la recherche de sous-modèles

À partir des fondements précédents, nous avons développé un outillage Eclipse offrant des fonctionnalités génériques pour manipuler des sous-modèles et leurs différentes relations dans le monde EMF, quel que soit le métamodèle. Cet outillage est composé de plusieurs plugins :

- un plugin noyau qui est un moteur de calcul des relations de sous-modèles entre modèles EMF (bien formés ou non) mais aussi entre un modèle et un ensemble d'éléments. Une particularité intéressante de ce moteur vient de son extensibilité par des plugins adaptateurs. En effet, il peut s'appliquer à tout métamodèle moyennant la représentation de ses modèles dans un format pivot répondant aux définitions précédentes.
- un plugin pour rechercher des modèles liés par des relations de sous-modèles dans un entrepôt de modèles CDO (Eclipse Connected Data Objects). Ce moteur réalise les deux phases nécessaires à une telle recherche : l'indexation des modèles avant leur stockage dans l'entrepôt et la récupération des modèles stockés qui satisfont une relation de sous-modèle avec un modèle requête.

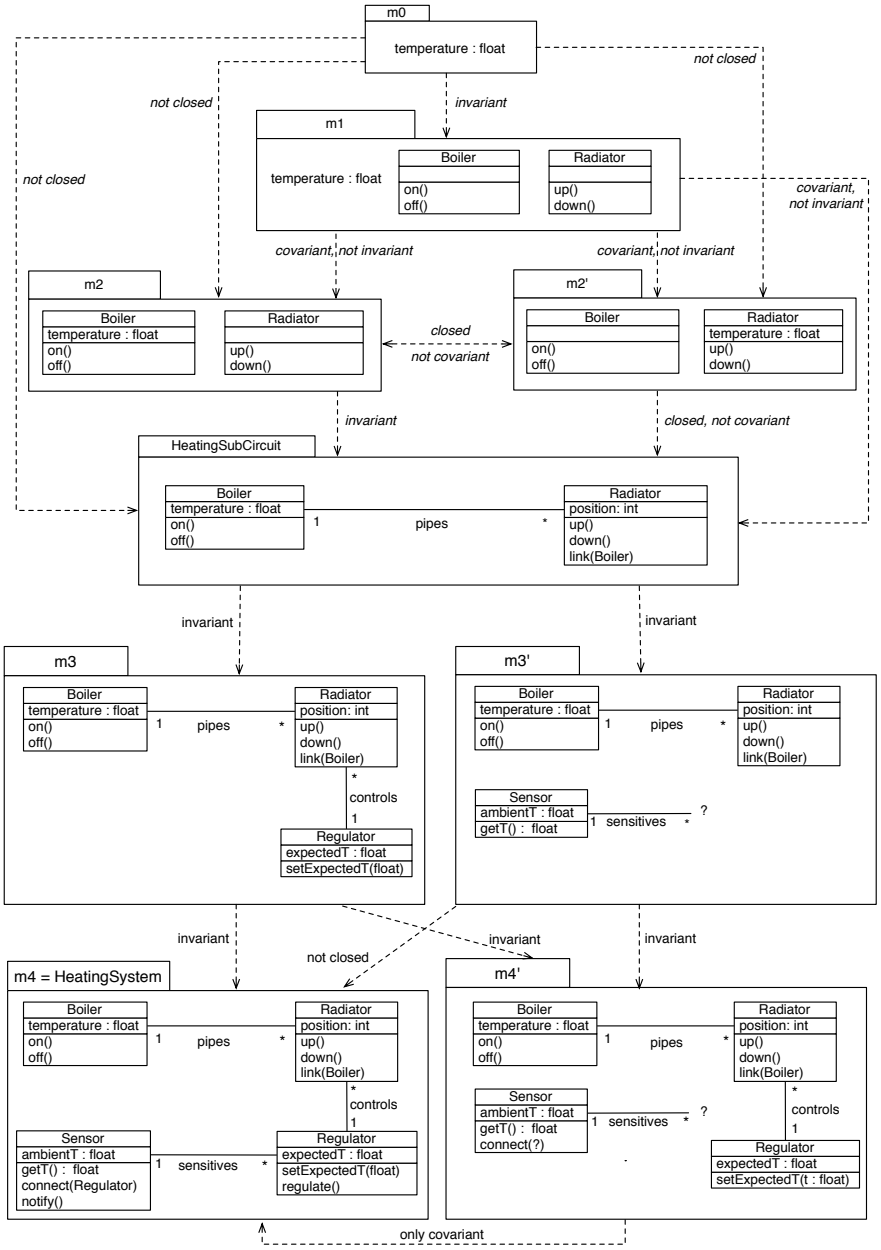


FIGURE 1 – Relations de sous-modèles

– un plugin permettant de trier un ensemble de modèles en relation de sous-modèles avec un modèle donné. Ce plugin peut servir à sélectionner les modèles les plus pertinents parmi les modèles appartenant au résultat d'une recherche.
– un plugin pour visualiser graphiquement des réseaux de modèles en relation de sous-modèles et naviguer dans ces réseaux pour, par exemple, détecter des parties communes

et des différences entre modèles [1].
Ces plugins sont utiles dans de nombreuses activités et environnements de modélisation qui impliquent des sous-modèles. On peut citer notamment leur utilisation pour la navigation dans des entrepôts de modèles mais aussi pour leur organisation (hiérarchisation, classification) de façon automatique. D'autres utilisations intéressantes en cours de développement sont la recherche ou la recommandation de modèles ou de morceaux enregistrés dans l'entrepôt pour compléter un modèle en cours de construction.

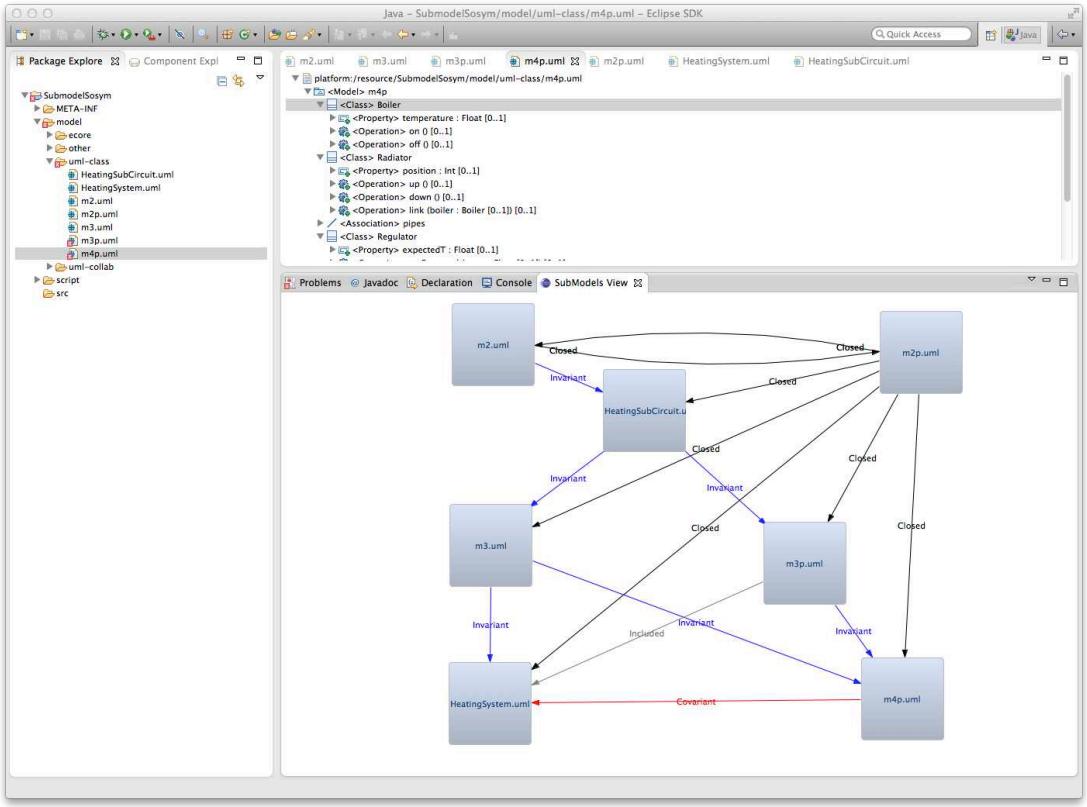


FIGURE 2 – Copie d'écran de l'outil en démonstration

Dans le cadre de la démonstration, nous montrerons une utilisation combinée des plugins précédents pour réaliser différents types de recherche dans un entrepôt de modèles CDO, la requête étant constituée à partir d'un ensemble d'ingrédients sélectionnés dans l'éditeur standard d'EMF. La figure 2 montre une copie d'écran de l'outil permettant ces recherches. A l'aide de cet outil, nous montrerons en particulier :

- comment retrouver les modèles qui incluent l'ensemble des éléments de la requête.
- comment retrouver les modèles dont les éléments de la requête forment un sous-modèle.
- inversement, comment retrouver les parties entreposées (contributions) d'un modèle en cours d'édition.

1. Vidéo disponible sur `http://www.lifl.fr/cocoa/pmwiki.php?n=Main.SubmodelEngineForEMF`

Nous montrons également les capacités offertes pour trier les modèles qui résultent de telles requêtes et leur présentation en réseaux.

Références

[1] M. Alanen and I. Porres. Difference and Union of Models. In *Proceedings of 6th International Conference on the Unified Modeling Language, Modeling Languages and Applications (UML'03)*, volume 2863 of *LNCS*, pages 2–17. Springer, 2003.

[2] M. Barbero and J. Bézivin. Structured Libraries of Models. In *Proceedings of 1st International Workshop on Towers of Models (TOWERS'07)*, 2007.

[3] J. Bézivin, F. Jouault, P. Rosenthal, and P. Valduriez. Modeling in the Large and Modeling in the Small. In *Model Driven Architecture*, volume 3599 of *LNCS*. Springer, 2005.

[4] A. Blouin, B. Combemale, B. Baudry, and O. Beaudoux. Modeling Model Slicers. In *Proceedings of 14th International Conference on Model Driven Engineering Languages and Systems (MoDELS'11)*, volume 6981 of *LNCS*, pages 62–76. Springer, 2011.

[5] B. Carré, G. Vanwormhoudt, and O. Caron. From subsets of model elements to submodels, a characterization of submodels and their properties. *Software and Systems Modeling*, page 29, 4 2013.

[6] R. B. France, J. M. Bieman, and B. H. C. Cheng. Repository for Model Driven Development (ReMoDD). In *Proceeding of MoDELS'06 Workshops*, volume 4364 of *LNCS*, pages 311–317. Springer, 2006.

[7] P. Kelsen, Q. Ma, and C. Glodt. Models within Models : Taming Model Complexity Using the Sub-model Lattice. In *Proceedings of 14th International Conference on Fundamental Approaches to Software Engineering, FASE'11*, volume 6603 of *LNCS*, pages 171–185. Springer, 2011.

[8] D. Lucrédio, R. Pontin de Mattos Fortes, and J. Whittle. Moogle : a metamodel-based model search engine. *Software and System Modeling*, 11(2) :183–208, 2012.

[9] T. Reiter, E. Kapsammer, W. Retschitzegger, and W. Schwinger. Model Integration Through Mega Operations. In *Proceedings of the International Workshop on Model-driven Web Engineering (MDWE'05)*, 2005.

[10] M. Siikarla, J. Peltonen, and J. Koskinen. Towards unambiguous model fragments. *Nordic Journal of Computing*, 13 :180–195, 2006.

[11] Harald Störrle. VMQL : A visual language for ad-hoc model querying. In *Journal of Visual Languages and Computing*, 2010.

Session 6

Modélisation de propriétés non fonctionnelles

ORQA : modélisation de l'énergie et de la qualité de service*

Borjan Tchakaloff[1,2], Sébastien Saudrais[1], and Jean-Philippe Babau[2]

[1] CERIE, ESTACA
Laval, France
firstname.lastname@estaca.fr
[2] Univ. Bretagne Occidentale, UMR 6285, Lab-STICC
Brest, France
firstname.lastname@univ-brest.fr

Résumé

Les véhicules électriques n'embarquent qu'une faible capacité énergétique, il est nécessaire de gérer efficacement leurs organes pour optimiser l'autonomie du véhicule. La gestion d'un véhicule est assurée par les systèmes embarqués, modélisés selon le standard Autosar. Ce standard couvre la plupart des besoins du monde automobile mais il lui manque des modèles de consommation et de Qualité de Service orientée utilisateur. Ce papier présente Orqa, un canevas logiciel pour modéliser et gérer les organes d'un véhicule électrique à travers leurs consommations énergétiques et leurs Qualités de Service. Les architectes choisissent et paramètrent à la conception les organes du véhicule par leurs puissances requises et, si approprié, leurs niveaux de qualité. L'implémentation générée est ensuite embarquée dans les modèles Autosar existants. Le système du véhicule est ainsi capable d'évaluer à l'exécution la consommation d'un trajet et de proposer au conducteur une stratégie de conduite. Les organes optionnels sont gérés tout au long du trajet selon les préférences du conducteur. L'utilisation d'Orqa est illustrée par un cas d'utilisation classique : un trajet journalier de retour du travail.

1 Introduction

Le véhicule électrique est arrivé à maturité industrielle. Malgré plusieurs modèles disponibles, leur autonomie reste faible (100 kilomètres environ) et ne permet qu'une utilisation journalière. Actuellement, la plupart des véhicules électriques ne fournissent pas de gestion avancée de l'énergie embarquée : la vitesse du véhicule est limitée lorsque le niveau de batterie est bas sans considérations pour les souhaits du conducteur ou sa destination. La gestion ne garantit pas l'atteinte de l'objectif du trajet. Il aurait été nécessaire d'anticiper la réduction de consommation afin d'atteindre la destination.

Pour effectuer une gestion efficace et adaptée, un contrôle complet de la configuration de tous les organes consommateurs est requis tout en respectant les préférences du conducteur. Le contrôle est possible grâce aux logiciels embarqués sur les calculateurs du véhicule et doit être intégré en respectant les contraintes du standard Autosar (AUTomotive Open System ARchitecture).

La méthodologie Autosar permet le développement d'applications automobiles mais n'offre pas la possibilité d'intégrer des propriétés extra-fonctionnelles comme la consommation. L'estimation de la consommation du véhicule en fonction des conditions d'utilisations (le type de route, la vitesse, les organes en fonctionnement) permet d'optimiser la stratégie de conduite. L'objectif est de trouver une solution réalisable pour atteindre la destination du conducteur.

*Papier original accepté à QoSA 2013 [1].

Dans ce papier, nous proposons Orqa, un canevas logiciel pour modéliser la consommation énergétique des organes d'un véhicule et de la Qualité de Service pour l'utilisateur. Afin d'assurer au conducteur l'atteinte de sa destination, la consommation énergétique doit être prévue pour toutes les routes disponibles et la meilleure doit être proposée au conducteur.

2 Approche

Le processus d'Orqa est réalisé en deux étapes. La première étape est la conception des modèles énergétiques et de Qualité de Service (QdS). Le concepteur définit les exigences de puissance et (s'il y a lieu) les niveaux de QdS des organes du véhicule en utilisant des modèles spécifiques. Afin d'aider le concepteur, le canevas logiciel offre une bibliothèque de modèles prédéfinis pour le moteur et pour chaque organe tel que les lampes, la climatisation, le système multimédia et un agglomérat des organes peu consommateurs.

La seconde étape est effectuée à l'exécution et concerne l'utilisation des organes. Elle se déroule comme suit :

1. L'utilisateur choisit un objectif et une politique de consommation qui seront utilisés plus tard pour calculer et choisir une stratégie. Dans l'exemple du trajet journalier, l'utilisateur choisit sa destination et sélectionne une politique de consommation. Le point de départ est défini par la position actuelle du véhicule.
2. Les stratégies sont issues ou générées à partir du système. Deux sous-étapes existent dans l'exemple. Plusieurs routes peuvent exister entre les points de départ et d'arrivée. Un GPS classique offre trois types de routes : la plus rapide, la plus courte et un compromis entre les deux. Les types de routes sont récupérés un à un (première sous-étape). Les coefficients de vitesse (réduisant les vitesses maximales à des vitesses inférieures) sont appliqués à l'ensemble des routes disponibles, créant une matrice des routes à évaluer (seconde sous-étape).
3. Les stratégies sont évaluées et l'énergie nécessaire est calculée pour chacune d'elle. La consommation des organes et des fonctions obligatoires ainsi que la durée du trajet sont calculées pour chaque trajet en tenant compte d'un coefficient de vitesse.
4. La meilleure stratégie est sélectionnée. Les différentes stratégies sont d'abord filtrées en fonction de la consommation maximale possible, en tenant compte de l'énergie restante et d'une marge de sécurité. Ensuite, des scores sont assignés aux stratégies restantes selon les objectifs de l'utilisateur et la stratégie ayant le score le plus élevé est sélectionnée. Enfin, l'utilisateur est informé de la stratégie choisie.
5. Un gestionnaire énergétique embarqué (un composant spécial communicant avec les interfaces des organes) est utilisé pour contrôler les organes. Pour respecter les contraintes du système (comme celles d'Autosar), l'architecture est statique et ne peut être modifiée à l'exécution. Le gestionnaire modifie donc le comportement des organes en utilisant des composants intermédiaires *(brokers)* attachés aux composants utilisant les interfaces existantes.

Le processus est basé sur un ensemble de modèles spécifiques pour l'énergie et la qualité.

3 Architecture

L'architecture d'Orqa est présentée dans la figure 1. Le modèle décrit en Autosar est enrichi avec deux types de composants : un gestionnaire énergétique embarqué et un ensemble

de *brokers* d'organes. Le gestionnaire énergétique est l'élément principal de ORQA. Il requiert des données provenant des capteurs ou calculées à partir d'autres composants. Toutes les données requises sont déjà existantes dans le système.

Le gestionnaire énergétique est composé des composants *Rater* et *Controller*. Le composant *Rater* est chargé de sélectionner la meilleure stratégie par rapport à l'objectif de l'utilisateur. Le composant *Controller* exécute la stratégie en gérant les organes consommateurs à l'aide des *brokers*.

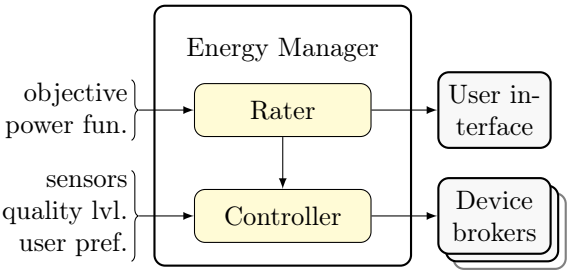


FIGURE 1 – Le composant gestionnaire énergétique.

Après le choix de la destination par l'utilisateur, le composant *Rater* évalue les routes possibles en fonction de leurs durées et de leurs énergies consommées. La figure 2 illustre l'évaluation d'un trajet ayant trois routes possibles : une mixte urbaine/extra-urbaine, une à majorité extra-urbaine et une à majorité urbaine. La ligne en pointillés représente le niveau d'énergie restant dans la batterie. Pour chaque route, quatre coefficients de vitesse sont appliqués (100%, 90%, 80%, 60%).

Nous pouvons remarquer que la route 2 consomme trop d'énergie pour la destination désirée dans tous les cas.

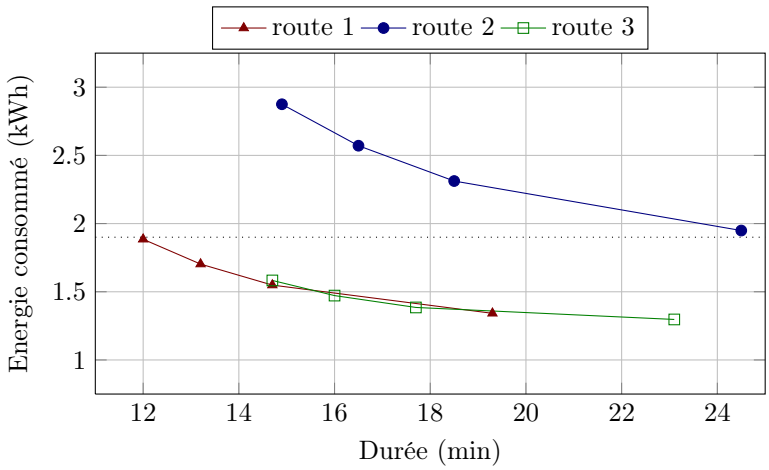


FIGURE 2 – Résultats d'évaluation pour les routes disponibles.

Chaque route est ensuite notée selon les critères de l'utilisateur, qui dans ce cas souhaite favoriser l'économie d'énergie. La figure 3 montre les scores obtenus pour chacune des routes en prenant en compte les différents coefficients de vitesse. Plusieurs stratégies ont des scores

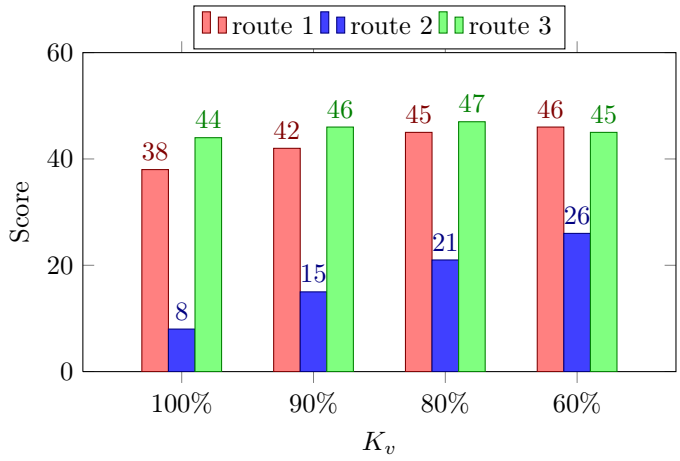


FIGURE 3 – Scores des routes disponibles.

comparables, la stratégie choisie est la route 3 avec un coefficient de 80%. En fonction de l'énergie restante, les organes optionnels sont sélectionnés afin de maximiser la Qualité de Service.

Le composant *Controller* est chargé d'appliquer la stratégie en s'assurant que les organes ne consomment pas plus que l'énergie qui leur a été alloué. Il vérifie régulièrement l'énergie restante. Si elle respecte la quantité estimée, le composant *Controller* garde les composants optionnels à leur niveau de Qualité de Service. Dans le cas contraire, les niveaux sont abaissés afin de garantir l'arrivée à destination.

La stratégie choisie permet d'atteindre l'objectif en environ 18 minutes et d'obtenir une Qualité de Service utilisateur de 80% (les organes optionnels sont accessibles et la climatisation est réglée en automatique). Nous pouvons remarquer que pour les trois routes, le conducteur n'aurait pas atteint sa destination en utilisant les organes optionnels à leur niveau maximal de Qualité de Service.

4 Conclusion

Dans ce papier, nous proposons un canevas logiciel permettant d'assurer l'arrivée à destination d'un véhicule électrique. Le canevas logiciel est exécuté par un gestionnaire énergétique embarqué agissant sur le système global. Le gestionnaire utilise un modèle pré-calculé de la consommation de chaque organe du véhicule, un ensemble de préférences utilisateur et les différents niveaux de Qualité de Service de chaque organe optionnel. À partir de ces modèles, le composant *Rater* cherche les routes possibles et calcule pour chacune la durée et la consommation pour une conduite nominale et dans le cas de vitesses réduites.

Références

[1] Borjan Tchakaloff, Sébastien Saudrais, and Jean-Philippe Babau. ORQA : Modeling Energy and Quality of Service within AUTOSAR Models. In *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures - QoSA'13*, page 3, Vancouver, British Columbia, Canada, 2013. ACM Press.

Mise à jour dynamique des applications Java Card
Une approche pour une mise à jour sûre du tas

Razika LOUNAS[1], Mohamed MEZGHICHE[1] et Jean-Louis LANET[2]

[1] Laboratoire LIMOSE, Département d'informatique, Faculté des Sciences
Université M'Hamed Bougara, Boumerdès, Avenue de l'Indépendance, 35000 Algérie
lounas_razika@umbb.dz, mohamed-mezghiche@umbb.dz
[2] Université de Limoges, 123 Avenue Albert Thomas, 87700 Limoges, France
jean-louis.lanet@unilim.fr

Abstract

La mise à jour dynamique des programmes consiste à modifier ceux-ci sans arrêter leur exécution. Lors de la mise à jour dynamique des applications Java Card, trois parties principales sont considérées : le code octal, la pile des blocs d'activation des méthodes et le tas des instances utilisées par l'applications. La mise à jour des instances dans le tas passe par l'application de Fonctions de Transfert d'Etat (FTE). Les différentes relations existant entre les objets ainsi qu'entre les objets et les méthodes en exécution rendent la mise à jour dynamique du tas délicate : en effet, une mise à jour hasardeuse peut introduire des erreurs dans l'application. Dans ce travail, nous proposons une approche sûre pour la mise à jour dynamique du tas basée sur une modélisation formelle de celui-ci et sur un algorithme d'ordonnancement des classes à mettre à jour et des FTEs.
Mots clés : Java Card, Mise à jour dynamique des programmes, Modèle formel, Manipulation du tas, Fonctions de Transfert d'Etat

Abstract

Dynamic Software Updating (DSU) consists in updating running programs on the fly without any downtime. When dynamically updating Java Card applications, three main parts are considered: the code, the stack of the frames of methods and the heap of instances manipulated by the application. A way to update instances in the heap is the application of State Transfer Functions (STF). The relations between objects and between objects and running methods make the update of the heap tedious, and a hazardous update may lead to errors in the application. In this paper, we present a safe approach for dynamically updating the heap. The approach is based on a formal model of the heap and an algorithm for updated classes and state transfer functions ordering.
Keywords: Java Card, Dynamic software updating, Formal model, Heap manipulation, State Transfert Funcions.

1 Introduction et motivations

Dans le but d'ajouter des fonctionnalités ou de corriger des erreurs dans des applications, celles-ci sont appelées à être modifiées. La mise à jour dynamique des programmes consiste à modifier ceux-ci sans arrêter leur exécution. Ce type de mise à jour est important dans les applications critiques, où le fait d'arrêter le système pour effectuer la mise à jour puis le redémarrer peut avoir de néfastes conséquences. Il est encore plus important dans le cadre des applications devant résister à des attaques durant une durée d'exécution longue. L'exemple du passeport électronique est éloquent : il doit résister pendant 10 ans à des attaques imprévisibles. De ce

fait, si un algorithme cryptographique utilisé par le passeport venait à être attaqué, le système doit être mis à jour pour charger un nouvel algorithme sans être arrêté.

Les systèmes utilisés dans les domaines critiques doivent passer par des procédures de certification pour évaluer leurs niveaux de sûreté , comme par exemple: le standard Common Criteria [1]. Un niveau élevé de sûreté est apporté aux systèmes par l'utilisation des méthodes formelles pour spécifier et / ou vérifier ceux-ci. Il est nécessaire donc d'utiliser une démarche formelle pour modéliser et analyser les systèmes utilisant la mise à jour dynamique des programmes. Notre travail s'intéresse à la mise à jour dynamique des applications Java Card. Dans ce cadre, EmbedDSU [8] est un système de mise à jour dynamique pour Java Card. Ce système effectue la mise à jour selon trois niveaux : le code octal, le tas et la pile des blocs d'activation des méthodes. Nous avons effectué une étude formelle de la partie code code octal dans [6]. Cet article est consacré à la mise à jour des instances dans le tas de la machine virtuelle. La mise à jour des instances passe par l'application des fonctions de transfert d'état (FTEs) pour l'initialisation des champs. L'application des FTEs s'avère critique de par les différentes relations entre les classes et entre les instances et des parties du code en exécution ainsi que les contraintes relatives à Java Card comme la notion du contexte peuvent conduire à des erreurs dans l'applications si la mise à jour n'est pas envisagée par une approche sûre. Ce travail présente une modélisation formelle du tas, ensuite il propose une approche de la mise à jour des instances en tenant compte des contraintes existantes et reposant sur le modèle et sur un algorithme d'ordonnancement de la mise à jour des classes et des FTEs. Ce papier est organisé de la façon suivante: le système EmbedDSU est présenté en section 2. La section 3 présente les FTEs. Nous parlons des problèmes relatifs à la mise à jour du tas en section 4. La section 5 présente notre approche. Les travaux connexes sont présentés en section 6. Nous concluons en section 7.

2 Le système EmbedDSU

EmbedDSU [8] est un système de mise à jour dynamique pour les applications Java Card basé sur une modification de la machine virtuelle. Il est divisé en deux parties : la partie off-card pour la préparation de la mise à jour et la partie on-card pour l'application de la mise à jour : **La partie externe.** Un module appelé DIFF generator détermine les changements syntaxiques entre deux versions d'une classe (instructions, méthodes ou champs ajoutés, supprimés ou modifiés). Ces changements sont sauvegardés dans un fichier DIFF qui est transféré ensuite sur la carte pour effectuer les mises à jour. **La partie interne.** Plusieurs modules sont implémentés au niveau de la machine virtuelle. On retrouve principalement : 1) Le module d'interprétation du fichier de DIFF permet d'interpréter le fichier de DIFF et d'initialiser les structures de données nécessaires à la mise à jour. 2) Un module d'introspection qui fournit des fonctions de recherche à travers les structures de la machine virtuelle comme la table des classes, le tas, la pile des blocs d'activation et la table des références dans le but de fournir aux autres modules les informations nécessaires à la mise à jour. 3) Un module de mise à jour permet d'effectuer les mises à jour dans le tas, le corps des méthodes, les références, les méta données et les structures de données affectées par la mise à jour. 4) Un module de détection de point sûr qui permet de détecter le moment opportun pour réaliser la mise à jour en préservant la cohérence du système. Le processus de mise à jour se déroule sur trois niveaux: le processus met à jour d'abord le code octal des classes modifiées et les méta données qui lui sont associées: la table des champs, le constant pool, la table des méthodes... Au niveau du tas, le processus met à jour les instances actives de la classe existant dans le tas. Le processus met à jour ensuite chaque bloc d'activation dans la pile des threads pour pointer vers les nouvelles instances.

3 Les fonctions de transfert d'état

La modification des instances dans le tas passe par la définition des fonctions de transfert pour les classes dont les objets seront mis à jour. La FTE d'une classe est appliquée sur chaque instance de la classe durant la mise à jour. La FTE retourne une instance initialisée, le nouvel objet prend ainsi l'identité de celui sur lequel la FTE est appliqué [4] [3]. Les FTEs sont écrites par le programmeur. Selon le corps des FTEs, celles-ci sont divisées en trois catégories : 1)Les FTEs à valeurs par défaut : il s'agit de mettre à jour les instances avec des valeurs par défaut. 2) Les FTEs à calcul simple : mettre à jour l'instance après un calcul qui n'utilise pas un autre objet à mettre à jour.3) Les FTEs avec accès aux autres instances : ce type contient des références vers des objets concernés par la mise à jour.

4 Problèmes soulevés par la mise à jour du tas

1)Problème avec les dépendances entre objets et les FTEs. L'ajustement apporté à l'état de l'application durant l'application de la mise à jour dynamique, est spécifié dans les FTEs. Deux problèmes peuvent survenir avec ces dépendances. Le premier provient des dépendances entre les classes. En effet, si une classe fait référence à une autre classe et si les deux sont concernées par la mise à jour alors le résultat peut dépendre de l'ordre de l'exécution de la mise à jour et de l'ordre de l'exécution des FTEs relatives aux deux classes. Le deuxième problème est posé par les FTEs qui peuvent utiliser dans leurs traitements des instances de classes concernées par la mise à jour. L'algorithme que nous proposons en section 5.2 permet de résoudre ces deux problèmes.
2)Problème avec les méthodes en exécution. Une méthode en cours d'exécution peut utiliser un objet plusieurs fois. Si par exemple, une méthode non concernée par la mise à jour utilise une instance à mettre à jour deux fois, dans le cas où la mise à jour se produit entre la première et la deuxième utilisation, la méthode utilisera deux versions différentes d'une même instance ce qui produira des erreurs dans le programme.
3) Problème avec le contexte Java Card. Dans Java Card, un contexte est associé à chaque applet en exécution. La machine virtuelle utilise le contexte pour renforcer la sécurité dans les opérations inter applets [2]. Chaque objet appartient à un contexte qui est le contexte du package l'ayant créé (owning context). Ce mécanisme permet d'assurer la sécurité de l'objet en effectuant des vérifications pour les accès aux objets du tas.

5 Une approche pour la mise à jour dynamique du tas

5.1 Le modèle formel

Les objets Java Card sont alloués dans le tas. Nous modélisons le tas comme un graphe orienté $G = (V; E)$ ou Vest un ensemble de noeuds (instances). Un objet est défini par le nom de la classe dont il est instance, une référence et par l'identifiant de son contexte (context ID). E est l'ensemble d'arcs. Il représente les relations existantes entre les objets. Pour chaque arc $e = (s; t; l)$: s est le noeud source de e, t est le noeud cible de e et $l = \{héritage, dépendence\}$ est un ensemble d'étiquettes sur les arcs pour spécifier le type de relation existant entre les objets. On dit qu'une classe A dépend d'une classe B si la classe A utilise un champ de la classe B. Nous ajoutons à notre modèle un noeud spécial appelé noeud des blocs d'activation. Il représente les méthodes en cours d'exécution dans l'application. Un arc étiqueté "utilisé-par" existe entre chaque méthode et chacune des instances qu'elle utilise.

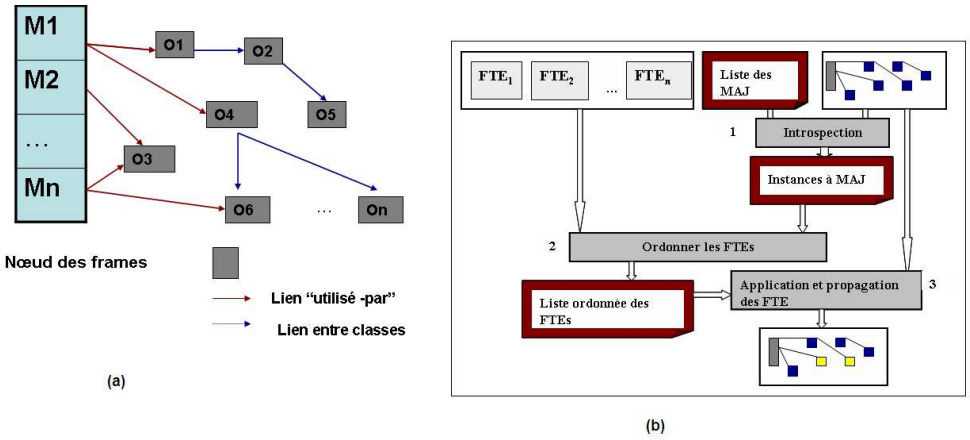


Figure 1: Modèle du tas et schéma de l'approche

5.2 Les étapes de notre approche

Étant donné: G, un graphe représentant un modèle du tas, P, un correctif contenant une description des mises à jour à effectuer (classes modifiées, champs ajoutés...). F, un ensemble de FTEs. Une FTE ft est un triplet (I, C, Oc) ou I est le nom de la classe des instances à mettre à jour, C est le corps de ft et Oc est l'ensemble des objets utilisés par tf, une mise à jour dynamique du tas MDTest une transformation d'une modélisation du tas en entrée (graphe G) vers un modèle G', de telle façon que les conditions suivantes soient respectées:1) Cohérence dans les versions des instances: une méthode en cours d'exécution lors du processus de la mise à jour n'utilise pas deux versions différentes d'une instance.2) Préservation du contexte: le mécanisme de mise à jour concerne seulement le contexte actifs. 3) Cohérence des dépendances entre les instances. L'application des FTEs n'introduit pas des erreurs sur les relations entre les instances. La figure 1(b) représente les différentes étapes de notre approche.

Étape 1: ***Introspection du tas.*** Cette étape prend des informations à partir du patch sur les classes à mettre à jour puis parcourt le graphe représentant le tas pour trouver les instances concernées par la mise à jour. La liste des instances à mettre à jour est retournée par cette étape. Ce résultat sera utilisé lors de la mise à jour.

Étape 2: ***Ordonnancement des classes et des FTEs.*** L'ordre de la mise à jour des classes et de l'exécution des FTEs sur les instances dans le tas est important. Nous proposons un algorithme pour calculer ces ordres de façon automatique en se basant sur les informations contenues dans le modèle du tas et les informations véhiculées par les FTEs elles mêmes. Le graphe en entrée est préalablement construit par une étude des liens de dépendance entre les classes. Dans un premier temps (lignes de 0 à 9 de l'algorithme 1), on visite les sommets du graphe représentant les liens entre les classes par un parcours en largeur. Pour cela, nous utilisons la file $file\_parcours$ qui contiendra les noeuds lors du parcours. Un noeud est retiré de la file puis inséré dans la liste C_ord représentant l'ordre de la mise à jour des classes. Ensuite il est examiné pour vérifier s'il correspond au nom d'une classe correspondant à une des FTEs (FTC (f1)) contenue dans la liste des FTEs en entrée. Dans ce cas, la FTE est ajoutée à la liste ordonnée des FTE F_ord (initialement vide []). Le symbole symbole :: représente la construction de liste. L'algorithme enfile ensuite les voisins du noeud en cours pour les examiner. Cette partie finit quand la file est vide. On obtient une liste de l'ordre de la mise à jour des classes

Algorithme 1: Ord_FTE

entrée: Une liste des FTEs FTE_list
Un graphe representant les classes et les liens entre elles $G =< V, E >$
Un sommet S

sortie: Une liste ordonnée des FTEs,une liste ordonnée des classes
et une liste de contraintes sur l'ordre des FTEs

variables: file_parcours: file pour parcourir le graphe en largeur
F_ord: liste ordonnée des FTEs, C_ord:liste ordonnée des classes
list_contr:liste des contraintes sur l'ordre des FTEs

0:$F\_ord \leftarrow []$;$C\_ord \leftarrow []$;
1:$enfiler(S, file\_parcours)$
2: **tant que** $\neg(vide(file\_parcours))$
3:$x \leftarrow defiler(file\_parcours)$;
4:$C\_ord \leftarrow x :: C\_ord$;
5: **pour chaque** $FTEf1 \in FTE\_list$
6: **si** $FTC(f1) = x$
7: $F\_ord \leftarrow f1 :: F\_ord$; finpour ;
8:**pour chaque** $y \in voisins(x)$
9: $enfiler(y, file\_parcours)$; finpour ; finTantque;
10:$list\_contr \leftarrow []$
11:**pour chaque** $FTEf1 \in F\_ord$
12: **pour chaque** $FTEf2 \in F\_ord - \{f1\}$
13: **si** $FTC(f1) \in FTE\_usef2$
14: $list\_contr \leftarrow (f1; f2) :: list\_contr$; finpour;finpour; fin.

Figure 2: Algorithme d'ordonnancement

et une liste de l'ordre des FTEs lui correspondant. Dans la deuxième partie de l'algorithme, on examine les liens existants entre les classes à mettre à jour et celles utilisées dans les corps des FTEs (FTE_use). Á partir de ces liens, on construit une liste de contraintes sur F_ord. Cette liste ($list\_contr$) est prise en compte ensuite par un mécanisme de programmation avec résolution des contraintes pour valider l'ordre vis-à-vis des relations entre les classes et traiter les références récursives si elles existent dans ($list\_contr$)

Étape 3: ***Synchronisation et propagation.*** Étant donné les listes ordonnées des classes à mettre à jour et des FTEs, du tas et de la liste des instances à mettre à jour obtenue par introspection, cette étape consiste à effectuer la mise à jour selon l'ordre calculé en tenant en compte le contexte actuel et des méthodes en cours d'exécution utilisant les instances. L'application d'une FTE sur une instance x dépend de l'ensemble des méthodes utilisant x et du contexte de x. La synchronisation est relative au fait de mettre à jour l'instance après vérification de l'absence d'incohérences vis-à-vis des méthodes en cours d'exécution. Ceci est basé sur les liens entre les les instances et le noeud des blocs d'activation de notre modèle. La propagation concerne la mise à jour des classes dependantes de la classe mise à jour avec les nouvelles informations (nouvelles références).

6 Travaux connexes

Plusieurs travaux se sont intéressés à la mise à jour de l'état de programmes en cours d'exécutions. Dans [5], l'auteur définit des conditions de sûreté pour appliquer la mise à jour en se basant sur le critère d'atteignabilité de l'état pour des programmes SmallTalk. Quatre

modèles pour la mise à jour des instances sont définis dans [9]: 1) Le modèle à barrière pour version qui consiste à ne porter la mise à jour que si toutes les instances de l'ancienne version ne sont plus utilisées. 2) Le modèle du partitionnement passif où tout les objets créés avant la mise à jour restent inchangés et ceux crées après la mise à jour correspondent à la nouvelle classe. 3) La mise à jour globale qui consiste à modifier toutes les instances de la classe. 4) Le partitionnement actif qui permet à l'utilisateur de sélectionner quels objets modifier et lesquels laisser selon l'ancienne version. Dans [7], la mise à jour des instances dans le tas est basée sur le concept de compatibilité. Les objets sont classifiés selon leur compatibilité avec l'ancienne ou la nouvelle version de l'application ainsi que leurs utilisation par les unités de codes (boucles, méthodes...). Cette étude est basée sur la définition de quatre comportements pour les activités que l'utilisateur spécifie dans un patch. Dans notre travail, le patch est généré automatiquement. Notre approche permet de faire un mise à jour automatique et sûre des instances,et d'ordonner automatiquement les FTEs et les classes à mettre à jour.

7 Conclusion

La mise à jour dynamique est une caractéristique clé dans les applications critiques. Notre travail s'inscrit dans le cadre de la certification de la mise à jour dynamique pour applications Java Card. Nous avons présenté une approche pour la mise à jour sûre du tas basée sur l'ordonnencement automatique des fonctions de transfert d'état. Actuellement, nous étendons le système EmbedDSU pour intégrer l'approche proposée. Le système EmbedDSU se base initialement sur les fonctions de transferts à valeurs par défaut, l'ordre de leurs exécutions n'étant pas pris en compte. La mise à jour des instances est basée sur une approche globale, l'approche que nous proposons permet d'éliminer les problèmes de cohérence des versions des instances vis-à-vis des méthodes qui les utilisent. Pour nos travaux futurs nous envisageons d'étudier formellement la mise à jour de la pile des blocs d'activation des méthodes et effectuer des démonstrations sur des propriétés de mise à jour dynamique dans le cadre Java Card.

References

[1] Common criteria: http://www.commmoncriteria.org/.

[2] Java card. the java card 3.0 specification: http://java.sun.com/javacard/.

[3] R.A. Bazzi, K. Makris, P. Nayeri, and J. Shen. Dynamic software updates: The state mapping problem. In *Proceedings of the 2Nd International Workshop on Hot Topics in Software Upgrades*, HotSWUp '09, pages 7:1–7:2, New York, NY, USA, 2009. ACM.

[4] C. Boyapati, B. Liskov L., Shrira, C.H. Moh, and S. Richman. Lazy modular upgrades in persistent object stores. *SIGPLAN Not.*, 38(11):403–417, October 2003.

[5] D. Gupta. *Online software version change*. PhD thesis, Indian Institute of Technology, India, 1994.

[6] R. Lounas, M. Mezghiche, and J.L. Lanet. Towards a general framework for formal reasoning about java bytecode transformation. In A. Bouhoula, T. Ida, and F. Kamareddine, editors, *SCSS*, volume 122 of *EPTCS*, pages 63–73, 2013.

[7] Y. Murarka. *Online Update of Concurrent Object Oriented Programs*. PhD thesis, Indian Institute of Technology, India, 2010.

[8] A.C. Noubissi, J. Iguchi-Cartigny, and J.L. Lanet. Hot updates for java based smart cards. In *ICDE Workshops*, pages 168–173, 2011.

[9] S.Malabarba, R. Pandey, J. Gragg, E. Barr, and J.F. Barnes. Runtime support for type-safe dynamic java classes. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 337–361, London, UK, UK, 2000. Springer-Verlag.

Challenges in Security Engineering of Systems-of-Systems

Vanea Chiprianov[1], Laurent Gallon[1], Manuel Munier[1], Philippe Aniorte[1] and Vincent Lalanne[1]

LIUPPA, Univ Pau & Pays Adour, France
name.surname@univ-pau.fr

Abstract

Systems of systems (SoS) are large-scale systems composed of complex systems with difficult to predict emergent properties. One of the most significant challenges in the engineering of such systems is how to model and analyze their Non-Functional Properties (NFP), such as security. In this review paper we identify, describe, analyze and categorize challenges to security engineering for SoS. This catalog of challenges offers a road-map of major directions for future research activities, and a set of requirements against which present and future solutions of security for SoS can be evaluated.

1 Introduction

Strategic attacks on a nation's infrastructure represent a great risk of disruption and loss of life and property. As the National Security Advisor, Condoleezza Rice, noted on 22 March 2001: 'US businesses, which own and operate more than 90% of the nation's banks, electric power plants, transportation systems, telecommunications networks, and other critical systems, must be as prepared as the government for the possibility of a debilitating attack in cyberspace.' Compounding the vulnerability of such systems is their interdependencies, with the result that impacts of attacks on one system can cascade into other systems [35].

As critical infrastructures are getting more and more dependent on Information Communication Technologies (ICT), the protection of these systems necessitates providing solutions that consider the vulnerabilities and security issues found in computers and digital communication technologies. However, the ICT systems that support these critical infrastructures are ubiquitous environments of composed heterogeneous components, and diverse technologies. These systems exhibit a variety of security problems and expose critical infrastructures to cyber attacks. Theses security challenges spread computer networks, through different ICT areas such as: cellular networks, operating systems, software, etc.

1.1 Engineering of System-of-Systems

Critical infrastructures are considered a type of a larger class of systems, Systems-of-Systems (SoS). SoS are large-scale concurrent and distributed systems, comprised of complex systems [22]. Several definitions of SoS have been advanced, some of them are historically reviewed in [18] for example. SoS are complex systems themselves, and thus are distributed and characterized by interdependence, independence, cooperation, competition, and adaptation [10].

Examples of SoS comprise critical infrastructures like: electric grid interconnected with other sectors [45], the urban transportation sector interconnected with the wireless network [3], but also home devices integrated into a larger home monitoring system, interoperability of clouds [55], maritime security [44], embedded time-triggered safety-critical SoS [48], federated health information systems [9], communities of banks [4], self-organizing crowd-sourced incident

reporting [42]. For example, a systematic review of SoS architecture [29] identifies examples of SoS in different categories of application domains: 58 SoS in defense and national security, 20 in Earth observation systems, 8 in Space systems, 6 in Modeling and simulation, 5 in Sensor Networking, 4 in Health-care and electric power grid, 3 in Business information system, 3 in Transportation systems.

Characteristics that have been proposed to distinguish between complex but monolithic systems and SoS are [36]:

- *Operational Independence of the Elements*: If the SoS is disassembled into its component systems the component systems must be able to usefully operate independently. The SoS is composed of systems which are independent and useful in their own right.
- *Managerial Independence of the Elements*: The component systems not only *can* operate independently, they *do* operate independently. They are separately acquired and integrated but maintain a continuing operational existence independent of the SoS.
- *Evolutionary Development*: The SoS does not appear fully formed. Its development and existence is evolutionary with functions and purposes added, removed, and modified with experience.
- *Emergent Behavior*: The SoS performs functions and carries out purposes that do not reside in any component system. These behaviors are emergent properties of the entire SoS and cannot be localized to any component system. The principal purposes of the SoS are fulfilled by these behaviors.
- *Geographic Distribution*: The geographic extent of the component systems is large. Large is a nebulous and relative concept as communication capabilities increase, but at a minimum it means that the components can readily exchange only information and not substantial quantities of mass or energy.

Other sets of characteristics of SoS, partially overlapping, have been identified, e.g. [5]:

- *Autonomy*: The reason a system exists is to be free to pursue its purpose; this applies to both the whole SoS and constituent systems.
- *Belonging*: The component systems can choose to belong to the SoS based on their needs and enhance the value of the system's purpose.
- *Connectivity*: There has to be the means provided for the systems to communicate with each other for the exchange of information.
- *Diversity*: The SoS should be diverse and exhibit a variety of functions as a system compared to the limited functionality of the constituent systems.
- *Emergence*: The formation of new behaviors due to development or evolutionary processes.

Taking into account these characteristics specific to SoS needs specific engineering approaches. Most researchers agree that the SoS engineering approaches need to be different from the traditional systems engineering methodologies to account for the lack of holistic system analysis, design, verification, validation, test, and evaluation [22], [8]. There is consensus among researchers [5], [37] and practitioners [2] that these characteristics necessitate treating a SoS as something different from a large, complex system. Therefore, SoS is treated as a distinct field by many researchers and practitioners, with its own conferences (e.g. IEEE International Conference on System of Systems Engineering, first one in 2006) and journals (e.g. International Journal of System of Systems Engineering).

As part of the SoS engineering initiative, an SoS life-cycle is essential. However, as underlined also by the *Evolutionary Development* characteristic for example, the SoS life-cycle depends on the degree of dependence between the SoS and its constituting systems [33]. If there are strong dependencies between the development of the SoS and the development of the

participating systems, they require synchronization between the construction and deployment of the composing systems and that of the SoS. For example, such SoS are regional health information organizations and intelligent transportation systems. On the other hand, when missing elements are not critical, or when alternative solutions exist, SoS development can proceed without waiting. For example, data mash-ups - web applications that compose services - are such SoS. However, for these SoS is even more difficult to ensure their NFPs, exactly because their dependencies from their composing systems are looser.

1.2 Security Engineering of Systems-of-Systems

Security engineering within SoS and SoS security life-cycle are influenced by SoS engineering and the SoS life-cycle. They need to take into account the characteristics specific to SoS, and how they impact security of SoS. At a general, abstract level, these impacts include [52]:

- *Operational Independence*: In an SoS, the component systems may be operated separately, under different policies, using different implementations and, in some cases, for multiple simultaneous purposes (i.e. including functions outside of the SoS purpose under consideration). This can lead to potential incompatibilities and conflict between the security of each system, including different security requirements, protocols, procedures, technologies and culture. Additionally, some systems may be more vulnerable to attack than others, and compromise of such systems may lead to compromise of the entire SoS. Operational independence adds a level of complexity to SoS that is not present in single systems.
- *Managerial Independence*: Component systems may be managed by completely different organizations, each with their own agendas. In the cyber security context, activities of one system may produce difficulties for the security of another system. What rights should one system have to specify the security of another system for SoS activities and independent activities? How can systems protect themselves within the SoS from other component systems and from SoS emerging activities? Does greater fulfillment require a component system to allow other component systems to access it?
- *Evolutionary Development*: An SoS typically evolves over time, and this can introduce security problems that the SoS or its components do not address, or are not aware of. Therefore, the security mitigations in place for an evolving SoS will be difficult to completely specify at design time, and will need to evolve as the SoS evolves.
- *Emergent Behavior*: SoS are typically characterized by emerging or non-localized behaviors and functions that occur after the SoS has been deployed. These could clearly introduce security issues for the SoS or for its component systems, and therefore the security of the SoS will again need to evolve as the SoS evolves. In addition, responsibility for such behaviors could be complex and shared, leading to difficulties in deciding who should respond and where responses are needed.
- *Geographic Distribution*: An SoS is often geographically dispersed, which may cause difficulties in trying to secure the SoS as a whole if national regulations differ. These may restrict what can be done at different locations, and how the component systems may work together to respond to a changing security situation.

And for the other set of characteristics [18]:

- *Autonomy*: Ensuring security of component systems.
- *Belonging*: Restricting/allowing component systems access to SoS.
- *Connectivity*: Protecting from unauthorized integration.
- *Diversity*: Restricting/allowing diverse behavior.
- *Emergence*: Preventing/managing policies "bad" emergent behavior.

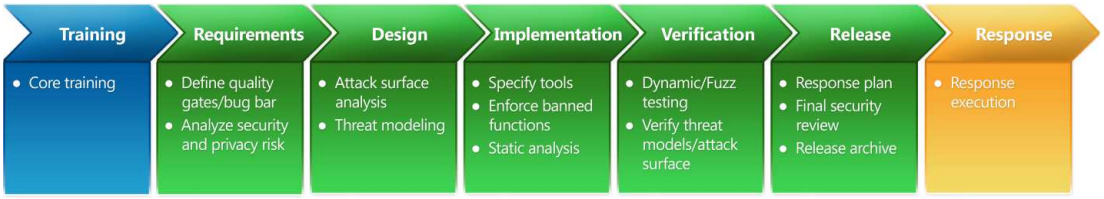


Figure 1: Microsoft Security Development Life-cycle, [20]

The specificities of security engineering for SoS, identified by analyzing the impact of SoS-specific characteristics, run of course much deeper. They touch all phases of the SoS life-cycle and all phases of the security engineering process. A generic security process is for example that proposed by Microsoft, Fig. 1. It comprises the phases of: training, requirements, design, implementation, verification, release, and response. A process for SoS security engineering has been proposed e.g. by [7]. This consists in the following activities, which can be largely concurrent and iterative: information gathering, flow analysis, security test and evaluation and end-to-end testing, target architecture and transition planning, security of SoS modeling, SoS security policy developing and risk management. In this paper, we further identify challenges to security engineering within SoS and organize them according to the security process.

Identifying challenges to security engineering within SoS is the first step in engineering security within SoS. As highlighted by [37], a very desirable research direction would be an integrated description and analysis method that can express and guarantee user level security, reliability, and timeliness properties of systems built by integrating large application layer parts - SoS. Moreover, systems engineering of defense systems and critical infrastructure must incorporate consideration of threats and vulnerabilities to malicious subversion into the engineering requirements, architecture, and design processes; the importance and the challenges of applying System Security Engineering beyond individual systems to SoS has been recognized [11]. Additionally, secure cyberspace has been recognized as one of the major challenges for 21st century engineering [53], [24].

2 Challenges in Security Engineering of Systems-of-Systems

Starting from the challenges related to characteristics specific to SoS, we further identify, describe and analyze challenges to security engineering of SoS. We organize them according to the activity of the security process in which they have the most impact. Of course, most challenges impact several activities, but for clarity purposes, we present them in the activity in which we consider they have the most impact.

2.1 Challenges impacting all Activities of the Security Process

Long life of SoS Most composing systems of a SoS are used in a context with other systems that have not necessarily gone through the same security engineering processes. Even if all the systems in a given composition had gone through the same security engineering process individually, composing them to achieve a new capability has the potential for new vulnerabilities and threats, and thus new risks to the end-to-end SoS. And the number of new systems organizations (e.g. Department of Defense) fields is small compared to the number of legacy ones. How to approach constraints associated with these legacy systems [7]? Consequently, will

most SoS be composed of systems with uneven levels of system protection [11]?

2.2 Requirements Challenges

Identifying SoS security requirements Because requirements are taken on by the constituent systems to meet the SoS objectives, identifying the security requirements for the overarching SoS provides a framework for assessing the adequacy of the system security engineering actions on the part of the constituent systems for security for the SoS and its mission [11]. How to identify these overarching security requirements?

Security requirements modeling SoS security engineering involves a tension between near-term risk mitigation and long-term evolution to a more secure SoS architecture. In the near term, risks can usually be mitigated effectively by controls at policy domain boundaries and at interfaces between individual systems. In the long term, uniform enforcement mechanisms within and between policy domains not only mitigate risks more effectively but also improve interoperability and maintainability. How can security be integrated into requirements modeling [7]? How can a balance between near-term and long-term security requirements be achieved?

Ownership Who should have the ultimate ownership responsibility for the SoS? Who will be responsible for dealing with issues arising from the SoS, for example if the system was used for malicious purposes, who would be legally culpable? Who will be responsible for testing and proving the system is running as expected and fulfilling its security requirements [27]?

Risk management This is concerned with management and control for the assessment, updating and mitigating of risks [27]. Security-related risks would be part of SoS risk identification and mitigation. They include new security risks resulting from new SoS capabilities composed from interacting constituent systems, as well as any residual security risk of constituent systems [11]. How to identify and mitigate risks associated with end-to-end flow of information and control, without, if possible, focusing on risks internal to individual systems [7]? While there are standards for risk management of standalone systems [21], there are not for SoS. To what extent do they apply to SoS; should such standards be extended to SoS?

Security of interoperability Several important aspects of enterprise interoperability have been the focus of European research programmes and initiatives such as European Interoperability Framework (EIF), INTEROP-Vlab and ATHENA-IP. How should these interoperability approaches consider organizational and human factors, such as personal responsibilities (policies and best practice for system security) from the earliest stage of the analysis? How should they address information protection, trust and security [41]?

Holistic security For many years the focus has been mainly on IT security (e.g. cryptographic analysis) and usually the implementation of security tools has been done by IT departments and computer experts. During the early 90s key aspects started to change and the first draft of an information security management standard BS 7799 [1] was produced. It focused on security related to synergies of people, processes, and information as well as IT systems. Since then, early security management standards have been transformed into international standards published by ISO/IEC [21]. These standards are being used by enterprises and organizations, and there are currently several initiatives holistically capturing security issues related to people, organization and technology. However application of standards does not guarantee solving

the broad spectrum of systems security problems. Information security comprises: 1) Physical software systems security based on applying computer cryptography and safety or software criticality implementation; 2) Human / personnel security based on the procedure, regulations, methodologies that make an organization / enterprise / system safe; 3) Cyber / Networking level that is mainly concerned with controlling cyber attacks and vulnerabilities and reducing their effects [41]. How can such holistic standards be extended to encompass SoS? How can they be applied and enforced in the context of SoS?

Requirements as source of variability It may be difficult to identify a specific SoS configuration that will meet the objectives and quantify the requirements allocated for each specific composing system. Requirements are often the largest source of variability and unknown quantities at the start of traditional system engineering projects. From a SoS perspective, increased difficulty is added with multiple systems and independently managed requirements teams whose efforts need to be coordinated in order to achieve the SoS objectives. How to adequately identify and allocate requirements to constituent systems for their respective teams to manage [16]?

Security metrics for SoS What could be security-specific metrics and measures for an SoS [11]? Is it possible to define a set of metrics which can be evaluated on the entire SoS, or are some security assessments limited to subparts of the SoS? Is it possible to define probability-theoretic metrics that can be associated with prediction models? How the mix of deterministic and uncertain phenomena, that come into play when addressing the behavior of a SoS faced with malicious attacks, can be represented [47]?

Balancing costs and gains of information sharing Information sharing participation carries with it costs which need to be balanced by direct expected gain or to be subsidized in order to have a critical number of composing systems to agree to share information and to discourage free riding. Agreeing to share information entails some cost to the participating system: these include costs of acquiring and maintaining equipment, training staff to use it, and integrating it into existing business practices. Although these expenses may be small in relation to existing operating costs, non-monetary costs might have some influence on the decision to participate. The benefits of information sharing include a decreased probability that a particular attack will be successful, and an increased rate of detection and recovery should an attack succeed. A large part of this benefit is naturally seen by the composing systems; however there may be important positive externalities as well. If an attack or even an (in)voluntary input error on a particular system is successful, it may create problems for other systems in the SoS. If the attack introduces propagating malware, for example, it might be spread to other systems through business or social communications. Operational disruptions in one system might impose costs on other systems by preventing clearing and settlement of inter-system transactions or client transactions. There are also possible reputation costs to the SoS as a whole arising from a successful attack on a single system. Such externalities create an incentive for each system in the SoS to see other systems join in an information sharing arrangement. Conversely some of the benefits created by a joining system are experienced by other systems in the SoS, whether or not they themselves participate [4].

2.3 Design Challenges

Bridging the gap between requirements and design The use of standards and architectural frameworks does not always guarantee achievement of desired levels of interoperability

security. How to breach the gap between frameworks and implementation [41]? How to assure a level of system and information availability consistent with stated requirements [15]?

Designing security Security is often taken into account only at the end of the development life-cycle of a system. Consequently, any a posteriori modification is very expensive. That is why, security must be taken into account as soon as possible and designed-in rather than relying on hardening of systems post implementation [30]. Moreover, SoS must be designed for robustness under planned malicious assault [35]. But how can security be integrated into the SoS architecture [7]? How to represent an exchange policy specification so as to verify some properties like: completeness, consistency, applicability and minimality [12]?

SoS security modeling and (continuous) analysis The protection of SoS requires a better understanding of how control needs to be designed in a top-down design approach in order to cover all the aspects of the composing systems. High interconnectivity, complexity, and dependency in SoS are spread through multiple levels and evolve over time with unpredictable behavior. Therefore, it is hard to understand, design, and manage composing systems [38]. How to represent the SoS in a form that lends itself to detailed analysis, especially when full details of the component systems may not be readily available? SoS analysis can drive changes in the composing systems to exploit opportunities or correct problems that were not originally anticipated. A key architectural tool in this respect may be the use of predictive modeling and simulation to compare architectural alternatives. In any process for improving a SoS, alternative architectures would need to be carefully considered and modeled to ensure the SoS is not compromised or undesirable emergent behaviors result. Some of the elements that comprise a SoS may be outside the control of other elements in the SoS. This means the SoS may not be responsive to a single analysis. Should, therefore, SoS analysis be incremental and the SoS should be available for testing almost on a continuous basis [25]?

Interdependency analysis It is concerned with examining the possible cumulative effects (escalation or cascading) of a single security incident on multiple systems. Such cascading failures could result in a complete blackout. Hence, it is important to identify cascading vulnerabilities before they happen [38]. How to identify threats that may appear insignificant when examining only first-order dependencies between composing systems of a SoS, but may have potentially significant impact if one adopts a more macroscopic view and assesses multi-order dependencies? How to asses the hidden interdependencies [31]? How to represent the interdependencies existing among a group of collaborating systems? How such an approach can be integrated in a risk assessment methodology in order to obtain a SoS risk assessment framework [17]? How to understand dependencies of a constituent system, on systems that are external to the formal definition of the SoS, but that nonetheless have security-relevant impacts to SoS capabilities [11]?

Security agreements How to manage security protections and security risk acceptance relationships among multiple systems? How about also supporting SoS evolution in terms of authorization to connect a constituent system to some other constituent system; the handling of risks propagated from one constituent system to some other constituent system of the SoS? How about the distribution and inheritance of security measures; assessment processes; authorization and acceptance processes; the roles and responsibilities for operation and sustainment of protections necessitated by SoS capabilities [11]?

New architectural processes Governance plays a significant role; designing SoS needs to be addressed different from the traditional process of stove-piped systems [15]. Which would be the best suited process for architecting SoS? Should it contain iterative elements, should it be agile, or model-based, etc? How does the type of dependencies between the development of SoS and the development of its constituent systems influence the design process of the SoS?

Defensive design SoS must be designed defensively, and this holds for composing systems as well. Defensive design is nothing new in areas involving physical processes. Aircraft engines and canopies are designed to withstand bird impacts, and structures are designed for the one-hundred year storm and magnitude 8 earthquakes. In ICT there seems to be no equivalent stressing design criterion, and such systems are easily overwhelmed by nothing more sophisticated than just driving up the rates at which system resources are accessed [35].

Design for evolution It is not sensible to assume that present security controls will provide adequate protection of a future SoS. Should there be a transition from system design principles based on establishing complex defensive measures aimed at keeping threats at bay, to postures that maintain operations regardless of the state of the SoS, including compromised states [14]? Should the focus move from avoiding threats from adversary action, over which we have limited control, to the larger, more inclusive goal of controlling SoS vulnerabilities [54]?

Security assurance cases How to model security assurance cases of SoS? While a system may be deemed secure by itself in a particular configuration, the introduction of other systems into that environment increases the complexity of the assurance model and must be considered and evaluated as part of a larger system [30].

Geographical aspects How should data be conveyed by geographical proximity? Like in wide area networks? Data that are related could be stored in different repositories belonging to several distinct systems, each with different security assurances [9].

Scalability of security A larger number of users can interact with the SoS than with any of its composing systems. This means a possible increased number and/or scale of attacks [9]. Therefore, the security mechanisms for the security of SoS should be scaled up consequently.

Multiplicity of security mechanisms There are different security mechanisms at different levels. Defensive capabilities include for example physical security measures, personnel security measures, configuration control, intrusion detection, virus and mal-ware control, monitoring, auditing, disaster recovery, continuity of operations planning [15], cryptography, secure communications protocols, and key management methods that are time tested, reviewed by experts, and computationally sound [14]. How to use together effectively and efficiently all these mechanisms [9]?

2.4 Implementation Challenges

Authentication The confirmation of a stated identity is an essential security mechanism in standalone systems, as well as in SoS. To achieve system interoperability, authentication mechanisms have to be agreed upon among systems to facilitate accessing resources from each system. How and when can this agreement be reached? Two kinds of authentication mechanisms are commonly presented: HTTP Authentication and Public Key Infrastructure [55].

Without authentication mechanisms to limit access, there is limited protection for the integrity of the information being transmitted [38].

What mechanism would allow various identity systems to inter-operate, so as all identity providers, relying parties (identity consumers) and subjects (users) work together using existing systems. Component systems may be developed by anyone; no single party has control. This mechanism should ensure: 1) consent: location systems must only reveal a user's location information with the user's consent; 2) minimal disclosure: location systems should reveal only the location information necessary; 3) granularity: location providers should specify all the levels of granularity of location information they are able to provide, and location consumers should specify all the levels of location information they are able to consume and switch between providers when one is shut down or temporarily unavailable [13].

Time constraints One of the main prerequisites for security of real-time SoS is that devices properly mutually authenticate themselves to prevent insertion of malicious devices or messages in case of a man-in-the middle attack. The main challenge in the design and implementation of device authentication mechanisms is to retain the temporal properties of a real-time system, i.e., the designer has to take care that introducing an authentication scheme in the real-time communication does not spoil the original real-time properties of the time-triggered system. Any additional and unpredictable delay in the communication path is critical for the communication and consequently for the access control and traffic separation based on the time-triggered protocol [48]. Of course, the authentication case can be generalized to other security mechanisms that may introduce delays in time-constrained SoS.

Authorization It is concerned with the management and control of the authorization schemes used and the ability to grant SoS authentication to interested parties [27]. Systems need to allocate the resources or rights according to a user's credentials after the user has proven to be what they stated. In a SoS, users with different backgrounds and requirements should be granted accesses to different resources of each composing system. Therefore, a proper authorization mechanism is necessary for the composing systems to cooperate together and provide the best user experience possible for the SoS users [55]. How would delegation of rights be handled? Who would be responsible for it?

Accounting / Auditing In conjunction to security, accounting is necessary for the record of events and operations, and the saving of log information about them, for SoS and fault analysis, for responsibility delegation and transfer, and even digital forensics. Interoperability among systems can be seriously affected if no such information is available. However, there is no well-defined best-practice guideline (not necessarily standard) on accounting agreed upon and adopted [55]. Where will this information be tracked and stored and who will be responsible for the generation and maintenance of logs [27]? How could this be reconciled with privacy concerns for example?

Non-Repudiation It is particularly important in a SoS where systems are legally bound by certain contracts. Verifying that one of the systems has indeed performed a certain action becomes necessary. How can an evidence of the origin of any change to certain pieces of data be obtained in the context of an SoS [9]? Who should collect these data, who can be trusted?

Encryption Challenges arise when ensuring the security between SoS endpoints through communication encryption. Without encryption to protect data as it flows through these insecure

connections, there is limited protection for the integrity of the information being transmitted [38]. Encryption mechanisms should be agreed upon in order for SoS users from different end-points to access the resources of a SoS. Encryption mechanisms like SSL, TSL, VPN are some of the widely adopted protocols [55]. Cryptographic keys must be securely exchanged, then held and protected on either end of a communications link. This is challenging for a utility with numerous composing systems [14].

Cryptographic key management Are the current trust models for cryptographic systems appropriate for SoS? The hierarchical trust model on which Public Key Infrastructure systems depend on is only as strong as the keys and trust points near the top of the pyramid (i.e. the keys used to issue certificates). As the SoS increases in size, the potential impact of a root compromise event also increases, particularly as the SoS crosses organizational boundaries. In the case of a Certificate Authority compromise, all systems that have the compromised certificate in their certificate stores are susceptible to compromise [14].

On the other hand, various trust management systems and associated trust models are being introduced, customized according to their target applications. The heterogeneity of trust models may prevent exploiting the trust knowledge acquired in one context in another context although this would be beneficial for the overall SoS. How to achieve interoperability between heterogeneous trust management systems [46]?

Security classification of data SoS may have multiple security domains [26]. In each composing system, data may have a specific security classification. It is possible that when combining two different sources at different classification levels, the new synthesized SoS product results with yet an additional classification - this necessitates a cross-domain solution [49]. How to provide the ability to securely and dynamically share information across security domains while simultaneously guaranteeing the security and privacy required to that information [15]? How to define multiple security policy domains and ensure separation between them? At the boundary between domains on different systems, information is often handed off from one set of enforcement mechanisms to another. Inconsistencies between policies and enforcement mechanisms frequently create vulnerabilities at policy boundaries which must be addressed by SoS security engineering [7].

Composing control policies How to express security and information assurance control into a uniformly, verifiable form so that they can be easily composed to form functional security requirements [19] [51]? In a top-down approach, how to flow down security policies to lower levels in the program? This enables specific interpretation of policy to different levels: work-packages, operational focus areas and individual projects, i.e. the top-level SoS goals need to be instantiated in and tailored to the high-level objectives for each system [30].

Context-based policies Composing systems might not be comfortable disclosing sensitive data to other entities except under certain conditions including transient conditions at the time of access. This shared data should be accessed exclusively by authorized parties, which may vary depending on the context (e.g. in emergency situations, or based on the location of the requester) [50]. In many cases it is the context-based policy that drives the data sharing while the number or recipients or their identities may not be known in advance. Interestingly, it is not just the data that is sensitive but also the policies for sharing the data. Therefore, there may be a need for policy-based data encryption techniques that support: 1) multiple recipients, 2) data and policy secrecy and 3) context-based policy enforcement [6].

Meta-data What kind of data should meta-data contain? What kind of meta-data should be legally-conformant to collect and employ? What kind of meta-data would technically be available? Should meta-data tags include data classification to provide controlled access, ensure security, and protect privacy? Should meta-data be crypto-bound to the original data to ensure source and authenticity of contents [15]?

Heterogeneity and multiplicity of platforms How to detect cross-protocol, cross-implementation and cross-infrastructure vulnerabilities? These vulnerabilities may be created for example when bridging two types of networks, e.g. VoIP and PSTN. How to correlate information across systems to identify such vulnerabilities and attacks [28]?

Hardware-enabled security Devices and systems that can place trust in a hardware mechanism to ensure operational integrity, force attacks to physically compromise a device in order to successfully perpetrate an attack. This provides a significant mechanism that devices can use to not only detect compromise, but also manage it and recover from it [14]. This would certainly benefit the security of the SoS as well.

2.5 Verification Challenges

Verifying the implementation satisfies the requirements When multiple, interacting components and services are involved, verifying that the SoS satisfies chosen security controls increases in complexity over standalone systems. This complexity is because the controls must be examined in terms of their different applications to the overall SoS, the independent composing systems, and their information exchange [19].

2.6 Release/Response Challenges

Configuration It is related to managing and altering security configuration settings associated with the SoS. Who will be responsible for investigating any configuration issues and performing changes [27]?

Monitoring It deals with monitoring for faults and issues within the SoS and ultimately who will be responsible for addressing any issues that may occur [27]?

Operational environment It deals with the control and assessment of the operational environment and the creation and enforcing of policies to control environmental security related to the SoS [27].

Runtime re-engineering In some cases, the SoS is only created at runtime, and the exact composition may not be known in advance. In such cases, it is difficult to fully plan and design the security of an SoS as part of the pre-deployment design. However, security currently takes time to establish, and there are many interrelated security issues that could create delay or loss of critical information. For some applications, runtime delays will have a big impact. Balance is therefore required in order to ensure security doesn't have a negative impact on operational effectiveness [43]. Moreover, in some cases, evolution of SoS may impact the conformance to requirements. At runtime, it is important to ensure that the current required level of security is achieved. If not, then re-engineering is required to resolve the situation. As part of the re-engineering, it is important to monitor and assess the SoS security state in order to determine

the nature of any security inadequacies, choose an appropriate course of action to resolve the deficiencies and implement it [52].

3 Related Work

A survey [27] examines selected approaches for the provision of security within SoS. The survey identifies some challenges to SoS security, like ownership, auditing, configuration, monitoring, authorization, risk management, operational environment, but focuses mainly on the running and operation of the SoS, which corresponds mainly to the Release/Response activities in the security process. We take into account all activities of the security process and identify a greater number of challenges related to each of them.

A systematic review of SoS architecture research [29] identifies 14 studies that discuss security of SoS. However, it does not identify the challenges to SoS security, and it does not analyze in further detail these studies.

The paper [11] identifies a number of challenges, issues to security of defense SoS. However, it is focused on defense SoS, while we take into account all types of SoS.

The paper [26] presents a framework for secure SoS composition, with a substantial related work. However, it focuses on solutions, not on challenges.

4 Conclusions and Future Work

In this review paper we have provided a catalog of challenges that have been identified in the literature regarding the subject of security engineering for Systems-of-Systems (SoS). Organized according to the security process activities, they represent an easy to consult, clear road-map of major directions for future research. Future research can position their research questions according to the challenges identified here. Moreover, these challenges can serve as a set of requirements against which existing and future solutions to security engineering of SoS can be evaluated.

Concerning our own work in the field of security engineering for SoS, we are tackling the **Security requirements modeling** and the **SoS security modeling and analysis** challenges with Model Driven Engineering (MDE) approaches by proposing Domain Specific Modeling Languages (DSML) both at the Computer Independent Model and at the Platform Independent Model levels, based on our works for systems [39]. Model Driven Security has more than a decade of existance, with major approaches reviewed e.g. in [34]. Using MDE also offers the advantage of naturally tackling the **Bridging the gap between requirements and design** challenge with model transformations between the DSML for requirements and the one for design.

The security aspects modeled in the DSMLs are in the case of our work mainly related to **Authorization**. In this direction, we are also investigating how to express and compose context-based access control policies: **Composing control policies** and **Context-based policies**.

A complementary research direction we are pursuing deals with **Risk management**, looking at how new threats and vulnerabilities, and thus new risks can be identified and further refined and accounted for - **Accounting / Auditing**. This is based on our previous work about information security risk management for information systems interconnected through services [32]. Moreover, a survey of the aspects related to data risks in business processes composed through the cloud was presented in [23]. The data, and **Meta-data** needed and legally permitted for accounting is another aspect we investigate [40].

Acknowledgments

The authors of this paper particularly thank the authors of the systematic review [29] for providing the list of the studies they have found to deal with security of SoS.

References

[1] BS 7799:Part 1:1995 information security management code of practice for information security management systems. Technical report, BSI British Standards, 1995.

[2] Systems engineering guide for systems of systems, version 1.0., 2008.

[3] C. Barrett, R. Beckman, K. Channakeshava, Fei Huang, V.S.A. Kumar, A. Marathe, M.V. Marathe, and Guanhong Pei. Cascading failures in multiple infrastructures: From transportation to communication network. In *Critical Infrastructure, 5th Intl Conf on*, pages 1–8, 2010.

[4] Walter Beyeler, Robert Glass, and Giorgia Lodi. Modeling and risk analysis of information sharing in the financial infrastructure. In Roberto Baldoni and Gregory Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 41–52. Springer Berlin Heidelberg, 2012.

[5] J. Boardman and B. Sauser. System of systems - the meaning of OF. In *System of Systems Engineering, 2006 IEEE/SMC International Conference on*, pages 6 pp.–, April 2006.

[6] Rakesh Bobba, Himanshu Khurana, Musab AlTurki, and Farhana Ashraf. Pbes: a policy based encryption system with application to data sharing in the power grid. In *4th International Symposium on Information, Computer, and Communications Security*, ASIACCS, pages 262–275, 2009.

[7] D.J. Bodeau. System-of-systems security engineering. In *Computer Security Applications Conference, 1994. Proceedings., 10th Annual*, pages 228–235, Dec 1994.

[8] Roland T. Brooks and Andrew P. Sage. System of systems integration and test. *Information, Knowledge, Systems Management*, 5:261–280, 2006.

[9] Mario Ciampi, Giuseppe Pietro, Christian Esposito, Mario Sicuranza, Paolo Mori, Abraham Gebrehiwot, and Paolo Donzelli. On Securing Communications among Federated Health Information Systems. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *LNCS*, pages 235–246. Springer, 2012.

[10] Cihan H. Dagli and Nil Kilicay-Ergin. *System of Systems Architecting*, pages 77–100. John Wiley & Sons, 2008.

[11] J. Dahmann, G. Rebovich, M. McEvilley, and G. Turner. Security engineering in a system of systems environment. In *Systems Conference (SysCon), 2013 IEEE Intl*, pages 364–369, 2013.

[12] Remi Delmas and Thomas Polacsek. Formal methods for exchange policy specification. In Camille Salinesi, MoiraC. Norrie, and scar Pastor, editors, *Advanced Information Systems Engineering*, volume 7908 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2013.

[13] Nick Doty. The case for a location metasystem. In *2nd International Workshop on Location and the Web*, LOCWEB, 2009.

[14] Michael Duren, Hal Aldridge, Robert K. Abercrombie, and Frederick T. Sheldon. Designing and operating through compromise: Architectural analysis of ckms for the advanced metering infrastructure. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13, pages 48:1–48:3, New York, NY, USA, 2013. ACM.

[15] D.L. Farroha and B.S. Farroha. Agile development for system of systems: Cyber security integration into information repositories architecture. In *IEEE Systems Conf*, pages 182 –188, 2011.

[16] David Flanigan and Peggy Brouse. Evaluating the allocation of border security system of systems requirements. *Procedia Computer Science*, 16(0):631 – 638, 2013. Conf on Systems Eng Research.

[17] I.N. Fovino and M. Masera. Emergent disservices in interdependent systems and system-of-systems. In *IEEE Intl Conf on Systems, Man and Cybernetics*, volume 1, pages 590–595, 2006.

[18] A. Gorod, R. Gove, B. Sauser, and J. Boardman. System of systems management: A network management approach. In *System of Systems Engineering, 2007. SoSE '07. IEEE International Conference on*, pages 1–5, April 2007.

[19] J. Hosey and R. Gamble. Extracting security control requirements. In *6th Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW, 2010.

[20] Michael Howard and Steve Lipner. *The security development lifecycle*. O'Reilly Media, 2009.

[21] ISO/IEC. ISO/IEC 27005:2011: Information security risk management. Technical report, International Organization for Standardization (ISO), Geneva, Switzerland, 2011.

[22] M. Jamshidi. System of Systems - Innovations for 21st Century. In *Industrial and Information Systems, 2008. ICIIS 2008. IEEE Region 10 and the Third intl Conf on*, pages 6–7, Dec 2008.

[23] Elena Jaramillo, Manuel Munier, and Philippe Aniorte. Information security in business intelligence based on cloud: A survey of key issues and the premises of a proposal. In *WOSIS*, 2013.

[24] Roy S. Kalawsky. The Next Generation of Grand Challenges for Systems Engineering Research. *Procedia Computer Science*, 16(0):834 – 843, 2013. Conf. on Systems Engineering Research.

[25] Roy S. Kalawsky, D. Joannou, Y. Tian, and A. Fayoumi. Using architecture patterns to architect and analyze systems of systems. *Procedia Computer Science*, 16(0):283 – 292, 2013. 2013 Conference on Systems Engineering Research.

[26] M. Kennedy, D. Llewellyn-Jones, Q. Shi, and M. Merabti. A framework for providing a secure system of systems composition. In *The 12th Annual Conference on the Convergence of Telecommunications, Networking & Broadcasting (PGNet 2011)*, 2011.

[27] Michael Kennedy, David Llewellyn-Jones, Qi Shi, and Madjid Merabti. System-of-systems security: A survey. In *The 11th Annual Conference on the Convergence of Telecommunications, Networking & Broadcasting (PGNet 2010)*, 2010.

[28] A.D. Keromytis. A comprehensive survey of voice over ip security research. *Communications Surveys Tutorials, IEEE*, 14(2):514–537, Second 2012.

[29] John Klein and Hans van Vliet. A systematic review of system-of-systems architecture research. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 13–22, New York, NY, USA, 2013. ACM.

[30] R. Koelle and M. Hawley. Sesar security 2020: How to embed and assure security in system-of-systems engineering? In *Integrated Communications, Navigation and Surveillance Conference (ICNS), 2012*, pages E8–1–E8–11, April 2012.

[31] Panayiotis Kotzanikolaou, Marianthi Theoharidou, and Dimitris Gritzalis. Interdependencies between critical infrastructures: Analyzing the risk of cascading effects. In Sandro Bologna, Bernhard Hmmerli, Dimitris Gritzalis, and Stephen Wolthusen, editors, *Critical Information Infrastructure Security*, volume 6983 of *Lecture Notes in Computer Science*, pages 104–115. Springer, 2013.

[32] Vincent Lalanne, Manuel Munier, and Alban Gabillon. Information security risk management in a world of services. In *PASSAT*, 2013.

[33] G. Lewis, E. Morris, P. Place, S. Simanta, D. Smith, and L. Wrage. Engineering systems of systems. In *Systems Conference, 2008 2nd Annual IEEE*, pages 1–6, April 2008.

[34] Levi Lucio, Qin Zhang, Phu Hong Nguyen, Moussa Amrani, Jacques Klein, Hans Vangheluwe, and Yves Le Traon. Advances in model-driven security. *Advances in Computers*, 93:103–152, 2014.

[35] S.J. Lukasik. Vulnerabilities and failures of complex systems. *Int. J. Eng. Educ.*, 19(1):206–212, 2003.

[36] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998.

[37] M.W. Maier. Research challenges for systems-of-systems. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, volume 4, pages 3149–3154, Oct 2005.

[38] M. Merabti, M. Kennedy, and W. Hurst. Critical infrastructure protection: A 21st century challenge. In *Communications and Information Technology (ICCIT), 2011 International Conference*

on, pages 1–6, March 2011.

[39] D. Munante, L. Gallon, and P. Aniorte. An approach based on model-driven engineering to define security policies using orbac. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 324–332, Sept 2013.

[40] Manuel Munier, Vincent Lalanne, Pierre-Yves Ardoy, and Magali Ricarde. Legal issues about metadata: Data privacy vs information security. In *DPM*, 2013.

[41] E.I. Neaga and M.J. de C Henshaw. Modeling the linkage between systems interoperability and security engineering. In *5th Intl Conference on System of Systems Engineering*, SoSE, June 2010.

[42] Craig Nichols and Rick Dove. Architectural patterns for self-organizing systems-of-systems. *Insight*, 4:42–45, 2011.

[43] Charles E. Phillips, Jr., T.C. Ting, and Steven A. Demurjian. Information sharing and security in dynamic coalitions. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 87–96, New York, NY, USA, 2002. ACM.

[44] Nicola Ricci, Adam M. Ross, and Donna H. Rhodes. A generalized options-based approach to mitigate perturbations in a maritime security system-of-systems. *Procedia Computer Science*, 16(0):718 – 727, 2013. 2013 Conference on Systems Engineering Research.

[45] S.M. Rinaldi, J.P. Peerenboom, and T.K. Kelly. Identifying, understanding, and analyzing critical infrastructure interdependencies. *Control Systems, IEEE*, 21(6):11–25, Dec 2001.

[46] Rachid Saadi, MohammadAshiqur Rahaman, Valrie Issarny, and Alessandra Toninelli. Composing trust models towards interoperable trust management. In Ian Wakeman, Ehud Gudes, Christian-Damsgaard Jensen, and Jason Crampton, editors, *Trust Management V*, volume 358 of *IFIP Advances in Information and Communication Technology*, pages 51–66. 2011.

[47] Luca Simoncini. Dependable and historic computing. chapter Socio-technical Complex Systems of Systems: Can We Justifiably Trust Their Resilience?, pages 486–497. Berlin, Heidelberg, 2011.

[48] Florian Skopik, Albert Treytl, Arjan Geven, Bernd Hirschler, Thomas Bleier, Andreas Eckel, Christian El-Salloum, and Armin Wasicek. Towards secure time-triggered systems. In *Proc of the 2012 Intl Conf on Computer Safety, Reliability, and Security*, pages 365–372. Springer, 2012.

[49] M.A. Solano. Sose architecture principles for net-centric multi-int fusion systems. In *6th International Conference on System of Systems Engineering*, SoSE, pages 61 –66, June 2011.

[50] Daniel Trivellato, Nicola Zannone, and Sandro Etalle. Poster: protecting information in systems of systems. In *18th ACM Conf on Computer and Communications Security*, pages 865–868, 2011.

[51] Daniel Trivellato, Nicola Zannone, Maurice Glaundrup, Jacek Skowronek, and Sandro Etalle. A semantic security framework for systems of systems. *Int. J. Cooperative Inf. Syst.*, 22(1), 2013.

[52] A. Waller and R. Craddock. Managing runtime re-engineering of a system-of-systems for cyber security. In *System of Systems Engineering (SoSE), 2011 6th Intl Conf on*, pages 13–18, 2011.

[53] W. A. Wulf. Great achievements and grand challenges. Technical report, National Academy of Engineering, 2000.

[54] William Young and Nancy G. Leveson. An integrated approach to safety and security based on systems theory. *Commun. ACM*, 57(2):31–35, February 2014.

[55] Zhizhong Zhang, Chuan Wu, and David W.L. Cheung. A survey on cloud interoperability: Taxonomies, standards, and practice. *SIGMETRICS Perform. Eval. Rev.*, 40(4):13–22, April 2013.