CIEL 2019

8ème Conférence en IngénieriE du Logiciel

École Nationale Supérieure d'Électrotechnique, d'Électronique, d'Informatique, d'Hydraulique et des Télécommunication (ENSEEIHT)
Toulouse
13 et 14 juin 2019

Co-organisée avec les Journées du GDR-GPL

Coprésidence du comité de programme:

Nicolas Anquetil – Université de Lille

Sophie Ebersold – Université de Toulouse

Contents

Avant propos

La conférence CIEL (Conférence en IngénieriE du Logiciel) réunis chaque année la communauté scientifique francophone sur des 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.

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, et surtout servir de tremplin pour la communauté pour s'affirmer au niveau international.

Elle permet à la communauté d'avoir un contact direct avec les auteurs de publications récentes et d'intérêt.

Elle permet enfin aux jeunes chercheurs de faire leur premières armes dans le domaine de la recherche, de confronter leurs idées à un publique élargi et bienveillant et de commencer à construire un réseau de connaissances pour promouvoir la coopération scientifique.

Dans cette vision, les articles présentés répondent à deux critères principaux:

- Travaux de chercheurs confirmés, déjà acceptés par ailleur dans le but d'en assurer une plus large diffusion à l'intérieur de la comunauté francophone.
- Travaux préliminaires, qui ne soient pas entièrement murs pour une publication internationale et que les auteurs souhaitent confronter à l'opinion de leur pairs.

Comité de pilotage

Salah Sadou, Université Bretagne Sud, IRISA-UBS (Président)

Isabelle Borne, Université Bretagne Sud, IRISA-UBS
Jean-Michel Bruel, Université de Toulouse, IRIT
Jean-Rémy Falleri, Bordeaux INP, Bordeaux
Naouel Moha, UQAM, Montréal, Canada
Philippe Merle, CRIStAL, INRIA Lille - Nord Europe, Lille
Marie-Agnès Péraldi, Université Nice Sophia Antipolis, I3S-Inria
Noël Plouzeau, Université de Rennes 1, IRISA, Inria
Pascal Poizat, Université Paris Ouest Nanterre la Défense, LIP6
Romain Rouvoy, CRIStAL, Université de Lille / INRIA,
Chouki Tibermacine, Université de Montpellier,

Christelle Urtado, Ecole des Mines d'Alès, LGI2P
Clémentine Nebut, Université de Montpellier, Montpellier
Tewfik Ziadi, Sorbonne Université,LIP6, Paris

Comité de programme

Nicolas Anquetil (co-président), CRIStAL, Université de Lille
Sophie Ebersold (co-présidente), IRIT, Université de Toulouse

Sebastien Mosser, Université du Québec à Montréal
Antoine Beugnard, IMT Atlantique
Olivier Barais, IRISA/Inria/Université de Rennes1
Marianne Huchard, LIRMM, Université de Montpellier et CNRS
Mireille Blay-Fornarino, I3S, Université Nice Sophia Antipolis
Chouki Tibermacine, Université of Montpellier
Bernard Coulette, IRIT, Université de Toulouse
Marie-Pierre Gervais, LIP6, Université Paris Nanterre
Clément Quinton, CRIStAL, Université de Lille
Lom Messan Hillah, LIP6, Université Paris Nanterre et Université Sorbonne
Anne Etien, CRIStAL, Université de Lille
Gilles Perrouin, PReCISE, Université de Namur
Sophie Dupuy-Chessa, LIG, Université Grenoble Alpes
Dalila Tamzalit, LS2N, Université de Nantes, CNRS UMR 6004

Travaux présentés

Session: Applications mobiles

- Sarra Habchi, Naouel Moha, Romain Rouvoy, “The Rise of Android Code Smells: Who Is to Blame?”
- Lakhdar Meftah, Romain Rouvoy, Isabelle Chrisment, “Testing Nearby Peer-to-Peer Mobile Apps at Large”
- Adel Nassim Henniche, Nabila Bousbia, Naouel Moha , “Forage d’applications mobiles pour la refactorisation des défauts de code”

Session: IDM

- Benoît Verhaeghe, Anne Etien, Stéphane Ducasse, Abderrahmane Seriai, Laurent Deruelle, Mustapha Derras, “Migration de GWT vers Angular 6 en utilisant l’IDM”
- Pierre Lermusiaux, Horatiu Cirstea, Pierre-Etienne Moreau, “Establishing syntactic guarantees for pattern eliminating transformations”
- Elyes Cherfa, Salah Sadou, Chouki Tibermacine, Soraya Kesraoui, Régis Fleurquin, “Bad Smells d’expressivité dans les métamodèles”

Session: Développement de Systèmes

- Nan Messe, Regis Fleurquin, Nicolas Belloir, Vanea Chiprianov, Imane Cherfa, Salah Sadou, “Le Développement de Système de Systèmes Sécurisé Nécessitant un Déploiement Rapide”
- Pierre-Alain Yvars, Laurent Zimmer, “DEPS Studio : Un environnement intégré de modélisation et de résolution de problèmes de conception de systèmes”

Session: Exigences logicielles

- Takwa Kochbati, Shuai Li, Sébastien Gerard, Chokri Mraidha, “Clustering of Software Requirements for Automated Software Architectures”

- Florian Galinier, Ebersold Sophie, Jean-Michel Bruel, “Un langage d’exigences entre langage naturel et langage formel”

Session: Applications mobiles

The Rise of Android Code Smells: Who Is to Blame?

Sarra Habchi, Naouel Moha, and Romain Rouvoy

[1] Inria / University of Lille, Lille, France
sarra.habchi@inria.fr
[2] Université du Québec à Montréal, Montréal, Canada
moha.naouel@uqam.ca
[3] University of Lille / Inria / IUF, Lille, France
romain.rouvoy@inria.fr

Abstract

The rise of mobile apps as new software systems led to the emergence of new development requirements regarding performance. Development practices that do not respect these requirements can seriously hinder app performances and impair user experience, they qualify as code smells. Mobile code smells are generally associated with inexperienced developers who lack knowledge about the framework guidelines. However, this assumption remains unverified and there is no evidence about the role played by developers in the accrual of mobile code smells. In this paper, we therefore study the contributions of developers related to Android code smells. To support this study, we propose Sniffer, an open-source toolkit that mines Git repositories to extract developers contributions as code smell histories. Using Sniffer, we analysed 255k commits from the change history of 324 Android apps. We found that the ownership of code smells is spread across developers regardless of their seniority. There are no distinct groups of code smell *introducers* and *removers*. Developers who introduce and remove code smells are mostly the same.

Keywords: Android, mobile apps, code smells, history mining.

This paper was accepted in the 16th International Conference on Mining Software Repositories. https://hal.inria.fr/hal-02054788

Testing Nearby Peer-to-Peer Mobile Apps at Large

Lakhdar Meftah*, Romain Rouvoy‡ and §Isabelle Chrisment
*Inria / University of Lille, France
‡University of Lille / Inria / Institut Universitaire de France, France
§Telecom Nancy / Inria, France

Abstract

While mobile devices are widely adopted across the population, most of their remote interactions usually go through Internet. However, this indirect interaction model can be assumed as inefficient and sensitive when considering communications with neighboring devices. To leverage such weaknesses, nearby peer-to-peer (P2P) communications are now included in mobile devices to enable device-to-device communications over standard wireless technologies (WiFi, Bluetooth). While this capability sup- ports the design of collaborative whiteboards, local multiplayer gaming, multi-screen gaming, offline file transfers and many other applications, mobile apps using P2P are still suffering from app crashes, battery issues, and bad user reviews and ranking in app stores. We believe that this lack of quality can be partly attributed to the lack of tool support for testing P2P mobile apps at large. In this paper, we therefore introduce a test framework that allows developers to implement reproducible testing of nearby P2P apps. Beyond the identification of P2P-related errors, our approach also helps to tune the discovery protocol settings to optimize the deployment of P2P apps.

This paper was accepted in the 6th IEEE/ACM International Conference on Mobile Software Engineering and Systems

Forage d'applications mobiles pour la refactorisation des défauts de code

Adel Nassim Henniche[1], Nabila Bousbia[1], and Naouel Moha[2]

[1] LMCS-ESI, Ecole nationale Supérieure d'Informatique, Alger, Algérie.
a_henniche@esi.dz, n_bousbia@esi.dz
[2] Université du Québec à Montréal, Montréal, Canada.
moha.naouel@uqam.ca

Résumé

La popularité grandissante des applications mobiles a eu pour conséquence de banaliser les mauvaises pratiques de conception et d'implémentation qu'on appelle défauts de code. Comme première contribution de notre thèse "Forage d'applications mobiles pour la recommandation de refactorisations", nous visons à concevoir un outil de refactorisation automatique des défauts de code Android. Cet outil se base sur les techniques d'apprentissage automatique pour identifier à partir des différentes versions des applications la meilleure refactorisation à appliquer.

Résumé

The ever-growing popularity of mobile applications has resulted in the banalisation of poor design and implementation practices known as code smells. As a first contribution to our thesis "Mining mobile applications to recommend refactorings", we aim to design an automatic Android code smells refactoring tool that will use machine learning techniques to learn from different application versions the best refactoring to apply.

1 Contexte

La popularité grandissante des applications mobiles ces dernières années a fait émerger une forte demande sur le marché de ces logiciels applicatifs. Face à cette demande, les développeurs se retrouvent contraints d'implémenter et de maintenir leurs applications rapidement et sous pression. Malheureusement, une production rapide entraîne généralement une conception plus courte amenant à de mauvais choix de conception et d'implémentation qu'on appelle défauts de code. Ces défauts engendrent des effets négatifs sur la performance [3], la consommation d'énergie [7] et la disponibilité des ressources [12], et doivent donc impérativement être corrigés dans la perspective d'assurer la qualité des applications mobiles.

Cependant, la correction des défauts de code est une tâche complexe qui nécessite l'identification des entités (classe, méthode, etc.) infectées par un défaut et la refactorisation de ces entités. En particulier, il s'agit d'appliquer une succession de modifications de code sur les entités infectées afin d'améliorer leur qualité sans changer leur comportement. On remarque dans la littérature que les travaux de détection automatique des défauts de code dans les applications mobiles commencent à murir, contrairement aux travaux de correction qui restent lacunaires.

Comme premier travail de cette thèse, actuellement en première année, nous voulons développer un outil de refactorisation des défauts de code Android. L'outil se basera sur les techniques d'apprentissage automatique pour connaître, à partir de différentes versions des applications, les pratiques auxquelles les experts ont recours pour corriger les défauts de code. Aussi, nous allons mettre au point un catalogue documentant les pratiques de refactorisation sous Android dans le but d'encourager d'autres travaux dans le domaine.

La suite de l'article se compose de la manière suivante. La section 2 résume les travaux

connexes. La section 3 présente la méthodologie de travail adoptée et la section 4 est une conclusion de notre travail.

2 Travaux connexes

Les premiers travaux relatifs aux défauts de code se sont focalisés sur la description de ces défauts et leurs symptômes. Le catalogue de Reimann [17], qui est une référence concernant les défauts de codes spécifiques aux applications mobiles, a reporté les caractéristiques de 30 défauts de code. La thèse de Hecht [10] décrit 17 défauts dont 13 spécifiques à Android. Ces travaux ont contribué à diverses recherches qui ont porté sur l'analyse des effets des défauts de code sur la qualité logicielle [3, 7, 12], mais également sur leur détection et leur correction automatiques. Nous présentons spécifiquement ces deux derniers axes :

Travaux de détection : certaines études ont utilisé les outils orientés objet pour identifier les défauts de code dans les applications mobiles. Lineras [13] a testé l'outil DECOR [14] pour détecter 18 défauts dans 1 343 applications Android. D'autres recherches basées sur les règles et les normes des métriques de qualité ont abouti à des outils relativement fiables spécifiques aux défauts de code des applications mobiles, comme l'outil PAPRIKA [11]. D'autres travaux [6, 15] ont utilisé les techniques d'apprentissage automatique pour détecter les défauts de code. Par exemple, les travaux de Palomba [15] portent sur les règles d'association pour détecter les défauts en exploitant les informations sur l'historique des changements extraits des commits.

Travaux de correction : La plupart des outils ne considèrent que les défauts qui dépendent de la présence ou non de certains éléments de code comme par exemple l'outil de Reimann [17] qui traite 6 défauts en modifiant le modèle EMF (Eclipse Modeling Framework) des applications. D'autres travaux se sont intéressés à l'optimisation de la performance comme le travail de Dig [5] qui porte sur l'optimisation des tâches en arrière plan. Aussi, de nombreux travaux de correction ont porté sur les défauts impactant la consommation d'énergie [2, 4, 7]. L'outil HOT-PEPPER [2] par exemple corrige automatiquement 3 défauts impactant la batterie. Il se base sur PAPRIKA [11] pour détecter les défauts code puis applique le processus de correction défini manuellement à travers la librairie SPOON [16] qui est une librairie Java à code source ouvert permettant d'analyser et de transformer du code.

Nous constatons que les outils de détection se sont faits une place auprès de la communauté scientifique [8, 2, 9], là où les outils correction des défauts de code manquent de maturité et nécessitent plus d'intérêt. Notons aussi le recours à l'analyse des commits comme sources d'information [15, 9], nous l'utiliserons aussi dans notre travail.

3 Méthodologie de recherche

Dans cette section, nous détaillons notre méthodologie de recherche illustrée dans la figure 1. Nous proposons une approche outillée qui soit capable de reproduire automatiquement les différentes corrections entreprises habituellement par les experts pour corriger les défauts de code. Ainsi, dans la première étape de notre méthodologie, nous extrayons le code source des entités au fil des commits effectués par les développeurs. Pour cela, nous nous appuyons sur une étude qui s'est intéressée à la durée de vie des défauts de code dans les applications mobiles et où nous avons retracé l'historique de plus de 180k instances de défauts [9]. Ensuite, nous analysons les commits et identifions les changements de code qui se sont avérés efficaces pour pallier aux défauts de code. Puis, et grâce aux résultats de l'analyse, nous regroupons les changements recensés sous forme d'opérations de refactorisation dans un catalogue (étape 2) où

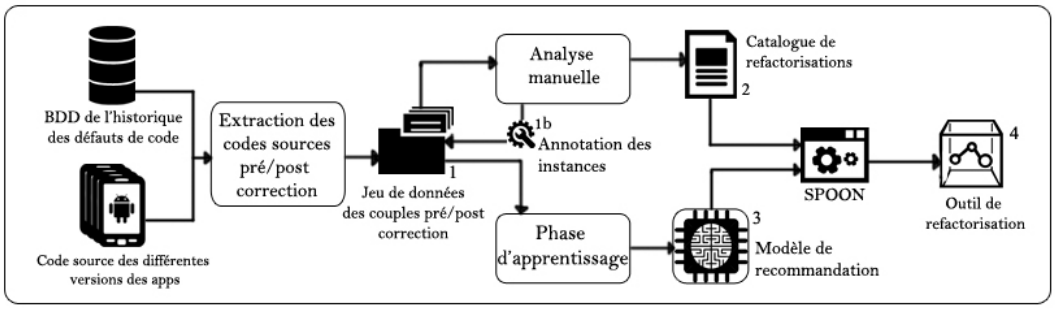


FIGURE 1 – Schéma de la méthodologie de recherche

TABLE 1 – Les informations sur l'historique des défauts de code.

ID Défaut	Nom Défaut	Nom de l'entité infectée	Commit int	Commit sup
0458	code dupliqué	ensure#com.fe...Var	45ca8....54eb	45ca2....45aa

chaque opération aura un nom et une description des étapes à suivre pour l'appliquer. Enfin, nous utilisons les codes extraits et les résultats de l'analyse comme jeu d'apprentissage pour générer notre modèle de recommandation de refactorisations (étape 3). Cette étape permet de recommander, pour une instance infectée par un défaut donné, la refactorisation la plus appropriée parmi les refactorisations possibles pour le défaut de code en cours d'analyse. L'outil de correction intègre le catalogue de refactorisations et le modèle de recommandation (étape 4) en utilisant la plateforme SPOON [16]. Cet outil permet de prédire et d'appliquer automatiquement la méthode de refactorisation adéquate à une nouvelle instance infectée. Les trois processus principaux de notre méthodologie sont détaillés ci-dessous :

Extraction des codes sources : Pour connaître les corrections appliquées par les experts pour corriger les défauts de code Android, nous formons un jeu de données où chaque instance pourrait être schématisée par un couple : le premier élément du couple est le code source d'une entité infectée par un défaut de code et le deuxième élément est le code source de la même entité après la correction du défaut. Pour cela, nous nous appuyons sur l'étude portant sur la durée de vie des défauts de code précédemment citée [9], et qui a répertorié pour chaque instance d'un défaut de code les informations de la table 1 où : **Commit int** référence le commit à partir duquel l'entité a été infectée par ce défaut et **Commit sup** référence le commit à partir duquel le défaut a disparu. Dans le contexte de notre travail, l'information cruciale est le commit qui a conduit à la disparition du défaut (Commit sup). Ce sont les changements apportés par ce commit et qui ont fait disparaître le défaut que nous cherchons à connaître. Pour extraire les parties de code recherchées, nous parcourons la liste des commits du projet auquel appartient l'entité infectée et nous restaurons le code source de cette entité depuis le snapshot du commit de disparition. Cette restauration correspond au code source après correction dans le couple du jeu de données. Ensuite, nous restaurons également le code source de cette entité depuis le snapshot du commit qui précède directement le commit de disparition. Cette restauration correspond au code source avant correction dans le couple.

Analyse manuelle : Une fois le jeu de données prêt, nous analysons manuellement les couples de code (avant correction, après correction). L'objectif de l'analyse est d'identifier les modifications qui ont été appliquées sur le code par les développeurs pour passer du code défaillant au code corrigé. Après que ces modifications de code ont été recensées, nous regroupons sous forme d'opérations les modifications similaires portant sur le même défaut. Chaque opération représente une possibilité de refactorisation pour le défaut en question. Si nous prenons comme exemple un défaut de code commun qui est *le code dupliqué*, les opérations identifiées pourraient être l'extraction de méthode ou l'extraction de classe. Toutes les opérations de refactorisation se voient attribuer un nom et une description du processus qu'il faut suivre pour les appliquer. En regroupant toutes les opérations et leurs descriptions, nous constituons le catalogue de refactorisations. Une fois que nous avons identifié toutes les opérations de refactorisation appliquées, nous annotons chaque couple de code de notre jeu de donnée par le nom de l'opération qui a permis sa correction (1b dans la figure 1). Pour ce faire, chaque entité se voit rajouter la méthode `refactSolution()` à la version avant correction de son code source qui retourne le nom de l'opération de refactorisation appliquée.

Phase d'apprentissage : La façon de corriger un défaut de code peut varier selon le contexte d'occurrence à l'image du code dupliqué qui peut être corrigé par l'extraction de classe ou l'extraction de méthode. Le problème est comment prédire la meilleure opération de refactorisation parmi les opérations possibles ? Ayant annoté chaque entité infectée par l'opération de refactorisation qui l'a efficacement corrigé grâce à l'analyse des commits, nous pouvons considérer ces annotations comme une expertise de spécialistes. Dès lors, le problème peut être vu comme un problème de classification : la fonction de prédiction serait le modèle de recommandation de refactorisations, les exemples annotés seraient les codes sources avant correction des entités infectées et les classes de prédiction sont les différentes valeurs que retournent la méthode `refactSolution()`. Nous nous intéressons à deux algorithmes particulièrement :

1. **Les arbres de décision :** à partir des exemples annotés, nous générons un arbre de décision pour chaque défaut de code en considérant toutes les instances de ce défaut. Les feuilles de l'arbre sont les noms des opérations de refactorisation possibles retournées par la méthode `refactSolution()`. Pour une nouvelle instance infectée par un défaut, nous parcourons l'arbre correspondant à ce dernier à la recherche de la meilleure refactorisation possible.
2. **Les K plus proches voisins :** afin de déterminer la meilleure refactorisation pour une nouvelle instance infectée par un défaut de code donné, nous calculons la distance entre cette instance et les instances infectées par le même défaut de code dans notre jeu de données. Puis, nous cherchons le nom de l'opération de refactorisation la plus fréquente parmi les noms retournés par la méthode `refactSolution()` pour les K plus proches voisins de l'instance étudiée. Une entité pourrait être représenté par un graphe `Neo4J` [1] et la mesure de distance serait une mesure de distance entre graphes.

4 Conclusion

Dans cet article, nous avons positionné notre sujet de thèse par rapport à l'état de l'art et aussi décrit la méthodologie de recherche que nous allons suivre pour concevoir une approche outillée pour la refactorisation des défauts de code Android. La prochaine étape consistera à intégrer et unifier cette approche de correction avec l'étape de détection automatique des défauts de code.

Références

[1] https://neo4j.com/. Accessed : 2019-04-25.

[2] A. Carette, M. Ait Younes, G. Hecht, N. Moha, and R. Rouvoy. Investigating the energy impact of android smells. In Martin Pinzger, Gabriele Bavota, and Andrian Marcus, editors, *SANER*, pages 115–126. IEEE Computer Society, 2017.

[3] C. Chambers and C. Scaffidi. Smell-driven performance analysis for end-user programmers. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 159–166, Sep. 2013.

[4] L. Cruz and R. Abreu. Using automatic refactoring to improve energy efficiency of android apps. *CoRR*, abs/1803.05889, 2018.

[5] D. Dig. Refactoring for asynchronous execution on mobile devices. *IEEE Software*, 32(6) :52–61, Nov 2015.

[6] F. Fontana, M. Zanoni, A. Marino, and M. Mäntylä. Code smell detection : Towards a machine learning-based approach. In *ICSM*, pages 396–399. IEEE Computer Society, 2013.

[7] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter. Removing energy code smells with reengineering services. In Ursula Goltz, Marcus Magnor, Hans-Jürgen Appelrath, Herbert K. Matthies, Wolf-Tilo Balke, and Lars Wolf, editors, *INFORMATIK 2012*, pages 441–455, Bonn, 2012. Gesellschaft für Informatik e.V.

[8] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha. Code smells in ios apps : How do they compare to android ? In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 110–121, May 2017.

[9] S. Habchi, R. Rouvoy, and N. Moha. On the survival of android code smells in the wild. In *6th IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, May 2019.

[10] G. Hecht. *Détection et analyse de l'impact des défauts de code dans les applications mobiles.* Theses, Université Lille 1 : Sciences et Technologies ; Université du Québec à Montréal, November 2016.

[11] G. Hecht, O. Benomar, R. Rouvoy, N. Moha, and L. Duchien. Tracking the software quality of android applications along their evolution (t). In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *ASE*, pages 236–247. IEEE Computer Society, 2015.

[12] G. Hecht, N. Moha, and R. Rouvoy. An empirical study of the performance impacts of android code smells. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 59–69, May 2016.

[13] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness : A threat to the success of android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 477–487, New York, NY, USA, 2013. ACM.

[14] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. Decor : A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1) :20–36, Jan.-Feb. 2010.

[15] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, Nov 2013.

[16] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. Spoon : A library for implementing analyses and transformations of java source code. *Softw., Pract. Exper.*, 46(9) :1155–1179, 2016.

[17] J. Reimann, M. Brylski, and U. Aßmann. A tool-supported quality smell catalogue for android developers. *Softwaretechnik-Trends*, 34(2), 2014.

Session: IDM

Migration de GWT vers Angular 6 en utilisant l'IDM

Benoît Verhaeghe[1,2,3], Anne Etien[1,3], Stéphane Ducasse[3,1], Abderrahmane Seriai[2],
Laurent Deruelle[2], Mustapha Derras[2]
[1]Univ. Lille, CNRS, Centrale Lille, Inria, UMR 9189 – CRIStAL, 59650 Villeneuve d'Ascq, France
{firstname.lastname}@univ-lille.fr
[2]Berger-Levrault, France
{firstname.lastname}@berger-levrault.com
[3]RMod team, INRIA Lille Nord Europe, Villeneuve d'Ascq, France
{firstname.lastname}@inria.fr

Résumé

Dans le cadre d'une collaboration avec Berger-Levrault, une société d'édition logicielle, nous travaillons à la migration d'une application GWT vers Angular. Nous nous concentrons sur l'aspect GUI de cette migration qui, même si les deux frameworks sont des frameworks d'interface graphique (GUI) pour le web, est rendue difficile parce qu'ils utilisent des langages de programmation différents (Java pour l'un, Typescript — un surensemble de JavaScript — pour l'autre) et différents schémas d'organisation (*e.g.* différents fichiers XML). De plus, la nouvelle application doit pouvoir imiter l'aspect visuel de l'ancienne afin que les utilisateurs de l'application ne soient pas perturbés dans leurs habitudes de travail. Nous proposons une approche en trois étapes qui utilise un méta-modèle pour représenter l'interface graphique. Ce méta-modèle permet à notre approche d'accepter différentes langues sources et cibles. Nous avons évalué cette approche sur une application comprenant 470 classes Java (GWT) représentant 56 pages web. Nous sommes capables de modéliser toutes les pages web de l'application et 93% des widgets qu'elles contiennent, et nous avons migré avec succès (*i.e.* le résultat est visuellement égal à l'original) 26 pages sur 39 (66%). Nous donnons des exemples de pages migrées, avec ou sans succès. Nous présentons également les résultats de quelques expériences de migration sur une application de bureau, non implémentée avec GWT, vers une application web, sans utiliser Angular.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

1 Introduction

Lors de l'évolution d'une application, il est parfois nécessaire de migrer son implémentation vers un langage de programmation ou une interface graphique (GUI) différente [2, 27]. Les frameworks d'interface graphique Web, en particulier, évoluent à un rythme rapide. Par exemple, en 2018, il y avait deux versions majeures de Angular, trois versions majeures de React.js, quatre versions de Vue.js, et trois versions d'Ember.js. Cela oblige les entreprises à mettre à jour régulièrement leurs logiciels afin d'éviter d'être bloquées avec d'anciennes technologies.

Notre travail se fait en collaboration avec Berger-Levrault, une importante entreprise informatique qui vend des applications Web développées avec GWT. Cependant, GWT n'est plus mis à jour avec une seule version majeure depuis 2015. En conséquence, Berger-Levrault a décidé de migrer les interfaces graphiques des ses applications vers Angular 6.

La société développe 8 applications utilisant GWT. Chaque application a plus de 1.5 MLOC [1] représentant plus de 400 pages web. Les applications sont composées de plus de 45 types de widgets et 29 types d'attributs. La société a estimé la migration pour une application à 4 000 jours-homme. Ainsi, migrer automatiquement la partie visuelle d'une application est déjà une étape utile pour la modernisation des applications de l'entreprise. En raison de l'évolution rapide des frameworks d'interface graphique, l'entreprise a également besoin d'une solution réutilisable pour la migration vers le prochain langage de programmation.

Il existe de nombreux articles publiés sur l'aide à la mi-

1. MLOC : Million de lignes de code

gration d'interfaces graphiques (*e.g.* [11, 22, 24]). Aucun d'entre eux ne parle du cas de la migration d'interfaces graphiques d'applications Web.

Nous présentons une approche pour aider les développeurs à migrer l'interface graphique de leurs logiciels basés sur le Web. Cette approche inclut un méta-modèle d'interface graphique, une stratégie pour générer le modèle, et l'exportation du modèle pour créer l'interface graphique cible. Pour valider cette approche, nous avons développé un outil qui migre des applications Java GWT vers Angular. Ensuite, nous avons validé notre approche sur un projet industriel qui sert à présenter tous les widgets et leur utilisation. Il est composé de 470 classes Java et 56 pages web. Notre approche a importé correctement 93% des widgets et 100% des pages. Comme tous les widgets existants ne sont pas réimplémentés dans Angular, nous avons essayé de migrer 39 pages et avons réussi (même apparence visuelle) pour 26 d'entre elles (66%).

Section 2, nous passons en revue la littérature sur la méta-modélisation de l'interface graphique. Nous décrivons le contexte de notre projet Section 3. Section 4, nous décrivons notre approche de migration. Nous présentons notre implémentation Section 5. La Section 6 décrit l'expérience que nous avons faite pour valider notre approche. Section 7, nous présentons nos résultats. Enfin, la Section 8 et la Section 9 discutent des résultats obtenus avec notre outil et des travaux futurs.

2 État de l'art

La Section 2.1 présente les techniques utilisées pour migrer une application. Section 2.2, nous décrivons les méta-modèles d'interface graphiques trouvés dans la littérature.

2.1 Stratégie de migration existantes

Pour définir une stratégie de migration, nous avons identifié des travaux de recherche liés à la migration des applications. Certaines des approches proposées n'effectuent pas une migration complète, mais seulement en partie. Toutes les approches utilisent des techniques de rétroingénierie. De plus, il existe de nombreuses publications traitant de la migration de langage de programmation. Nous ne les considérons cependant pas s'ils ne discutent pas explicitement de la migration d'interfaces graphiques. C'est le cas, par exemple, du travail de Brant *et al.* [2] qui fait état d'une importante migration de Delphi vers C#.

Nous avons identifié trois techniques pour créer une représentation d'interfaces graphiques : statique, dynamique ou hybride.

Statique. La stratégie statique consiste à analyser le code source et à en extraire des informations. Il n'exécute pas le code de l'application analysée.

Cloutier *et al.* [3] analyse directement les fichiers HTML, CSS et JavaScript. L'analyse construit un arbore syntaxique du code source du site web et extrait les widgets des fichiers HTML. Le travail consiste principalement à identifier les liens entre les éléments du programme source (classes JavaScript, balises HTML, etc.). Le travail présenté n'aborde pas la migration complète de l'interface graphique.

Lelli *et al.* [13], Silva *et al.* [25] et Staiger [26] ont utilisé des outils qui analysent le code source des applications de bureau. Les outils recherchent la création des widgets dans le code source, puis ils analysent les méthodes qui ont invoqué ou sont invoquées par les widgets pour identifier les relations entre les widgets et leurs propriétés visuelles.

Sánchez Ramán *et al.* [23] et Garcés *et al.* [8] ont développé des approches pour extraire l'interface graphique des applications Oracle Forms. Avec ce framework, les développeurs définissent les interfaces dans des fichiers externes où la position de chaque widget est spécifiée. Leur approche consiste à créer la hiérarchie des widgets à partir de leur position. Cependant, ces études de cas sont simples et ne comportent que peu de formulaires ou de textes. La mise en page est également simple car tous les éléments sont affichés les uns en dessous des autres.

La stratégie statique permet d'analyser une application sans avoir à l'exécuter ni même à la compiler. Outre le problème classique de montrer tous les faits potentiels plutôt que seulement les faits réels, une autre limitation apparaît par exemple, avec une application client/serveur, quand une partie de l'interface graphique dépend du résultat d'une requête à un serveur.

Dynamique. La stratégie dynamique consiste à analyser les interfaces graphiques d'une application en cours d'exécution. Elle explore les états de l'application en effectuant toutes les actions possibles sur l'interface graphique du logiciel et extrait les widgets et leurs informations.

Memon *et al.* [15], Samir *et al.* [22], Shah and Tilevich [24] et Morgado *et al.* [20] ont développé des outils qui mettent en œuvre une stratégie dynamique. Cependant, les solutions proposées ne sont disponibles que pour les applications de bureau et non pour les applications Web.

L'analyse dynamique permet d'explorer toutes les fenêtres d'une application et de recueillir des informations détaillées à leur sujet. Cependant, l'exécution automatique d'une application pour capturer méthodiquement tous ses écrans peut être une tâche difficile en fonction de la technologie utilisée. De plus, si une requête serveur est utilisée pour construire une des interfaces graphiques, l'analyse dynamique ne détecte pas cette information qui peut être essentielle pour la représentation complète des interfaces.

Hybrid. La stratégie hybride tente de combiner les avantages des analyses statiques et dynamiques.

Gotti and Mbarki [9] a utilisé une stratégie hybride pour analyser des applications Java. Ils créent d'abord un modèle à partir d'une analyse statique du code source. L'analyse statique trouve les widgets et attributs d'une interface graphique et leurs structures. Ensuite, l'analyse dynamique exécute toutes les actions possibles liées à un widget et ana-

lyse si une modification se produit sur l'interface.

Malgré l'utilisation de l'analyse statique et dynamique, la stratégie hybride ne résout pas le problème de requête inhérent aux applications client/serveur. Il a également les mêmes problèmes que l'analyse dynamique de l'exécution automatique d'une application et de la capture de ses écrans.

Fleurey *et al.* [7] et Garcés *et al.* [8] ont travaillé sur la migration complète de logiciels. Ils ont développé un outil qui effectue la migration de façon semi-automatique. Pour ce faire, ils ont utilisé le procédé du fer à cheval (Kazman *et al.* [12]). La migration est divisée en quatre étapes :

1. Génération du modèle de l'application originale.
2. Transformation de ce modèle en modèle pivot. Cela comprend les structures de données, les actions et algorithmes, l'interface utilisateur et la navigation.
3. Transformation du modèle pivot en modèle du langage cible.
4. Génération du code source dans le langage cible.

Aucun des auteurs n'a envisagé la migration des interfaces graphiques du Web vers des interfaces graphiques du Web. De plus, aucun n'avait la contrainte de garder une mise en page similaire sauf Sánchez Ramán *et al.* [23] ; cependant, ils travaillaient sur des applications Oracle Forms qui ont des interfaces très différentes de celles que l'on retrouve dans le Web. Par conséquent, leur travail n'est pas directement applicable à notre étude de cas.

2.2 Représentation de l'interface utilisateur

Dans la section précédente, de nombreuses représentations abstraites d'interfaces graphiques sont utilisées. Nous avons examiné les représentations proposées et les avons comparées. Nous présentons maintenant les deux méta-modèles d'interface graphique définis par l'OMG. Le "Knowledge Discovery Metamodel" (KDM) permet de représenter tout type d'application. L'"Interaction Flow Modeling Language" (IFML) est spécialisé dans les applications avec interface graphique. La Section 2.2.2 présente d'autres représentations décrites dans la littérature et les compare à celles de l'OMG.

2.2.1 Les standards de l'OMG

L'OMG définit la norme KDM pour aider l'évolution des logiciels. La norme définit un méta-modèle pour représenter un logiciel à un haut niveau d'abstraction. Il inclut un module d'interface graphique qui représente les composants et le comportement d'une interface graphique.

La Figure 1 représente les entités principales de la partie de l'interface utilisateur appelée diagramme de classes UIResources. L'entité principale est **UIResource**. Elle peut être affinée comme **UIDisplay** ou **UIField**. **UIDisplay** correspond au support physique sur lequel l'interface sera affichée, *e.g.* un écran d'ordinateur, un rapport imprimé, etc.

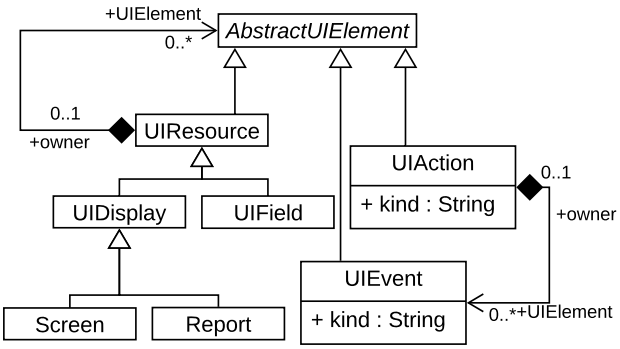


FIGURE 1 – KDM - Diagramme de classes **UIResources**

UIField correspond à un widget d'une interface graphique, *e.g.* un formulaire, un champ texte, un panneau, *etc.* La composition entre **UIResource** et **AbstractUIElement** est utilisée pour définir le DOM (Document Object Model). Chaque **UIResource** peut en contenir un autre pour représenter un widget qui contient un autre widget.

Un **UIResource** peut avoir, par composition, une UIAction pour représenter le comportement de l'interface graphique.

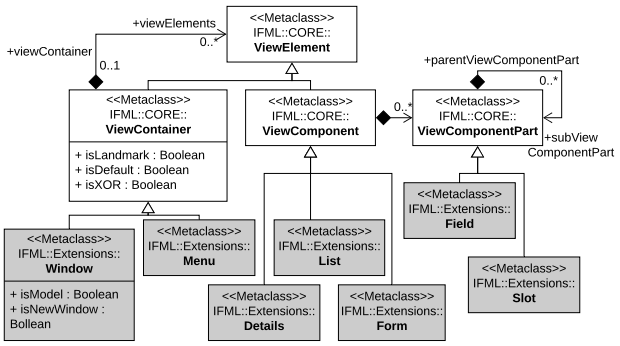


FIGURE 2 – IFML - View Elements

Le but de IFML [1] est de fournir des outils pour décrire la partie visuelle d'une application, avec les composants et les conteneurs, l'état des composants, la logique de l'application et le lien entre les données et l'interface graphique.

La Figure 2 représente le méta-modèle de la partie visuelle d'une application. Les éléments visibles de l'interface graphique sont appelés **ViewElement**. Un **ViewElement** peut être affiné comme un **ViewContainer** ou un **ViewComponent**.

Un **ViewContainer** représente un conteneur d'autres **ViewContainer**s ou **ViewComponent**s. Par exemple, cela peut être une fenêtre, une page HTML, un panneau *etc.*. La composition entre **ViewContainer** et **ViewElement** est utilisée pour définir le DOM.

Un **ViewComponent** correspond à un widget qui affiche son contenu, par exemple un formulaire, un tableau, une galerie d'images, *etc.* Il peut être lié à plusieurs **ViewCom-**

ponentPart. Un **ViewComponentPart** représente un élément d'un **ViewComponent**. Par exemple, un champ de saisie dans un formulaire, un texte qui est affiché à l'intérieur d'un tableau, ou une image d'une galerie.

2.2.2 GUI meta-models

D'autres méta-modèles d'interfaces graphiques ont été proposés dans la littérature. Nous les comparons aux normes de l'OMG.

Tous les méta-modèles utilisent le patron de conception Composite pour représenter le DOM d'une interface graphique et définissent une entité correspondant à **UIResource** pour représenter un élément graphique d'une interface.

Gotti and Mbarki [9] et Sánchez Ramán *et al.* [23] ont proposé un méta-modèle inspiré du modèle KDM. Le méta-modèle est composé des entités principales définies par KDM. Les deux auteurs ont ajouté l'entité Attribute au méta-modèle. Ils définissent également différents types de widgets tels que Button, Label, Panel, *etc.*

Fleurey *et al.* [7] ne décrivaient pas explicitement le méta-modèle de l'interface graphique, mais nous avons extrait des informations de leur modèle de navigation. Ils ont au moins deux éléments dans leur modèle d'interface graphique qui représentent une **Window** et un **GraphicElement**. La **Windows** correspond à l'entité Display du modèle KDM. Et parce que le **GraphicElement** et le **Window** ne sont pas liés, on peut supposer que le **GraphicElement** est une source UIRessource. Le **GraphicElement** a un **Event**.

Morgado *et al.* [20] ont utilisé un méta-modèle d'interface graphique mais ne l'ont pas décrit. Nous savons seulement que l'interface graphique est représentée sous la forme d'un arbre similaire au DOM.

Le méta-modèle UI de Garcés *et al.* [8] diffère beaucoup des précédents. Il y a les attributs, les événements et la page mais la notion de widget est présente comme un champ qui affiche les données d'une table. Ils ont également utilisé une entité **Event** pour représenter l'interaction de l'utilisateur avec l'interface utilisateur. L'entité **Event** correspond aux entités **Action** et **Event** du modèle KDM.

Memon *et al.* [15] ont représenté une interface graphique avec seulement deux entités. Une fenêtre de l'interface graphique qui est composée d'un ensemble de widgets qui peuvent avoir des attributs. Représenter les DOM n'était pas dans le cadre de leur travail. Il n'est pas possible de le représenter avec leur méta-modèle.

Samir *et al.* [22] ont travaillé sur la migration des applications Java-Swing vers des applications Web Ajax. Ils ont créé un méta-modèle pour représenter l'interface graphique de l'application originale. Ce méta-modèle est stocké dans un fichier XUL (langage d'interface graphique basé sur XML) et représente les widgets avec leurs attributs et la mise en page. Ces widgets appartiennent à une **Window** et peut déclencher des événements lorsqu'une action est effectuée sur l'interface graphique. L'action et l'événement correspondent aux entités **Action** et **Event** du modèle KDM. Le format XUL a été abandonné.

Shah and Tilevich [24] ont utilisé une architecture arborescente pour représenter les interfaces graphiques. Cela leur permet de modéliser le DOM. La racine de l'arbre est un **Frame**. Il correspond à l'entité **UIDisplay**. La racine contient les composants avec leurs attributs.

Joorabchi and Mesbah [11] ont représenté une interface graphique avec un ensemble d'éléments graphiques. Ces éléments correspondent à la définition d'un **UIField**. Pour chaque élément de l'interface, l'outil des auteurs détecte de multiples attributs et actions.

Memon [16] utilise un modèle d'interface graphique pour représenter l'état d'une application. Un état est défini à partir des widgets de l'interface graphique et leurs propriétés.

Mesbah *et al.* [17] n'ont pas présenté directement leur méta-modèle pour les interfaces graphiques. Cependant, ils expliquent utiliser une représentation arborescente pour analyser différentes pages Web. Ils ont également utilisé la notion d'événements qui peuvent être déclenchés. Ils ont utilisé différentes instances de leur méta-modèle d'interface graphique pour représenter les pages Web de l'application. Ces instances peuvent être comparées à plusieurs entités UIDisplay.

Tous les auteurs ont utilisé la notion de widget qui représente une entité visuelle de l'interface graphique. La plupart d'entre eux ont une entité attribut qui représente une caractéristique d'un widget. Enfin, les liens de navigation sont représentés par une entité d'action.

3 Contexte du projet de migration

Le but de notre travail est de migrer les interfaces graphiques d'un framework d'interface graphique à un autre. Il s'agit d'un projet industriel de migration d'applications web de GWT vers Angular. L'objectif est de produire une interface graphique exécutable dans le framework cible. Nous présentons maintenant les conditions du projet. Dans la Section 3.1 nous énumérons quelques contraintes que nous devons respecter. Dans la Section 3.2 nous décrivons les principales différences entre les applications GWT et Angular. Dans la Section 3.3 nous présentons une catégorisation du code source du front-end.

3.1 Contraintes

D'après les travaux précédents de Moore *et al.* [18] et Sánchez Ramán *et al.* [23], nous identifions les contraintes suivantes pour notre étude de cas :

— *Depuis GWT vers Angular*. Dans le cadre de la collaboration avec Berger-Levrault, notre approche de migration doit fonctionner avec Java GWT comme langue source et TypeScript Angular comme langue cible.

TABLE 1 – Comparaison de l'organisation des applications GWT et Angular

	GWT	Angular
Définition d'une page web	Une classe Java	Un fichier TypeScript et un fichier HTML
Aspect visuel principal de l'application	Un fichier CSS	Un fichier CSS
Aspect visuel spécifique d'une page web	Inclus dans le fichier Java	Un fichier CSS optionnel
Nombre de fichiers de configuration	Un fichier	Deux fichiers

— *Adaptabilité de l'approche*. Notre approche doit être aussi adaptable que possible pour des contextes différents. Par exemple, elle peut être utilisée avec différentes langues source et cible. Cette contrainte inclut les contraintes *Source and target independence* et de *Modularity*.

— *Préservation du visuel*. La migration doit générer l'aspect visuel de l'application cible le plus près possible de l'original. Cette contrainte inclut le *Layout-preserving migration* ce qui est en opposition avec le *GUI Quality improvement*.

— *Conservation de la qualité du code*. En tant que contrainte *Code Quality improvement* allégé, notre approche devrait produire un code qui semble familier aux développeurs de l'application source. Dans la mesure du possible, le code cible devrait conserver la même structure, les mêmes identificateurs et les mêmes commentaires. Cependant, nous verrons dans la prochaine section qu'il existe de fortes différences dans l'organisation des applications GWT et Angular.

— *Automatique*. Une solution automatique rend l'approche plus accessible. Il serait plus facile d'utiliser une approche automatique sur un grand système [18]. Cette contrainte correspond à la contrainte *Automation* de la littérature.

3.2 Comparison de GWT et Angular

Dans ce projet, la langue source et la langue cible imposent deux schémas d'organisation différents. Leurs différences sont syntaxiques et sémantiques.

GWT est un framework qui permet aux développeurs d'écrire une application web en Java. Le code GUI est compilé en code HTML, CSS et JavaScript. Angular est un framework d'application Web qui permet aux développeurs d'écrire une application web avec le langage TypeScript. Il est utilisé pour créer des Single-Page Applications [2]

La Table 1 résume les différences entre les applications GWT et Angular concernant : la définition des pages web, leur style et les fichiers de configuration. Avant d'expliquer ces trois différences, nous notons une similitude majeure : les applications GWT et Angular ont toutes deux un fichier CSS principal pour définir l'aspect visuel général de l'application.

2. Les Single-Page Applications (SPA) sont des applications Web qui chargent une seule page HTML et mettent à jour dynamiquement cette page lorsque l'utilisateur interagit avec l'application.

— **Définition de page web.** Dans le framework GWT, un seul fichier Java est nécessaire pour définir une page web (un extrait est proposé Figure 5, page 7). Le fichier Java (GWT) comprend les principaux composants graphiques (widgets) de la page Web, leurs positions et leurs organisations hiérarchiques. Dans le cas d'un widget actionnable (comme un bouton), l'action est implémentée dans le même fichier. Dans Angular, il existe une hiérarchie de fichiers pour chaque page Web. Chaque page web est considérée comme un sous-projet indépendant des autres. Un sous-projet contient deux fichiers : un fichier HTML, contenant les widgets de la page Web et leurs organisations ; et un fichier TypeScript, contenant le code à exécuter lorsqu'une action est exécutée.

— **Aspect visual** L'aspect visuel d'une page Web comprend la couleur ou les dimensions spécifiques des éléments affichés. Dans le cas de GWT, l'aspect visuel spécifique est défini dans le fichier Java du fichier de définition de la page web. Dans Angular, il existe un fichier CSS distinct optionnel.

```
1 <application name="CORE-Incubator">
2   <module name="KITCHENSINK">
3     <phase codePhase="KITCHENSINK_HOME"
4       className="fr.bl.client.kitchensink.
      PhaseHomeKitchenSink"
5       title="Home"/>
6   </module>
7 </application>
8
```

FIGURE 3 – Exemple d'un fichier de configuration GWT en XML

— **Fichiers de configuration** Pour les fichiers de configuration, GWT utilise un fichier XML qui définit les liens entre un fichier Java, une page Web et l'URL de la page Web. Figure 3 présente un extrait du fichier XML d'une application de Berger-Levrault. La balise **application** est la balise racine du fichier. Elle définit le nom de l'application GWT. La balise **phase** (ligne 3) définit une page Web de l'application GWT : Le titre de la page Web est "Home" ; elle est définie par la classe Java `PhaseHomeHomeKitchenSink` (dans le paquetage `fr.bl.client.kitchensink`); et l'URL pour accéder à la page Web est

`mserver.com/KITCHENSINK_HOME`. Pour Angular, il existe deux fichiers de configuration : *module* qui définit les composants de l'application, *e.g.* pages web, services distants et composant graphiques, et *routing* qui définit pour chaque page web son URL associée.

3.3 Structure d'application front-end

Comme proposé par Hayakawa *et al.* [10], nous avons divisé le projet de migration en plusieurs sous-problèmes. Pour ce faire, nous définissons trois catégories de code source : le code visuel ; le code comportemental ; et le code métier.

— **Code visuel** Le code visuel décrit l'aspect visuel de l'interface graphique. Il contient les éléments de l'interface. Il définit les caractéristiques inhérentes aux composants, telles que la possibilité d'être cliqué ou leur couleur et leur taille. Il décrit également la position des composants par rapport aux autres.

— **Code comportemental** Le code comportemental définit le flux d'action/navigation qui est exécuté lorsqu'un utilisateur interagit avec l'interface graphique. Le code comportemental contient les structures de contrôle (boucle et alternative).

— **Code métier** Le code métier est spécifique à une application. Il comprend les règles de l'application, les adresses des serveurs distants et les données spécifiques à l'application.

En raison de la taille et de la diversité du code source, la migration d'une de ces catégories de code est déjà un problème important.

4 Approche pour la migration

Cette section présente l'approche pour la migration que nous avons conçue. Dans la Section 4.1, nous décrivons le processus de migration que nous avons conçu. La Section 4.2 présente notre méta-modèle GUI.

4.1 Processus de migration

À partir de l'état de l'art, des contraintes et de la décomposition des interfaces utilisateurs, nous avons conçu une approche pour la migration.

Le processus, représenté Figure 4, est divisé en trois étapes :

1. *Extraction du modèle du code source.* Nous construisons un modèle représentant le code source de l'application originale. Dans notre étude de cas, le programme source est écrit en Java GWT. L'extraction produit un modèle FAMIX [5] de l'application utilisant un méta-modèle capturant les concepts Java. Nous devons également analyser le fichier de configuration XML décrit dans la Section 3.2.

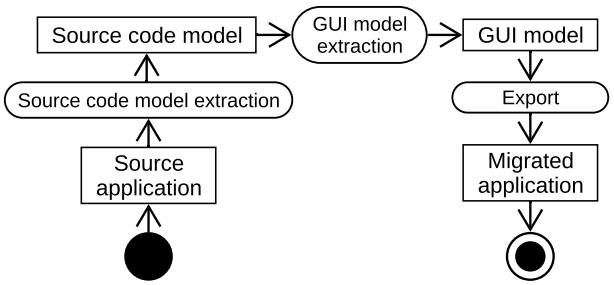


FIGURE 4 – Notre processus de migration

2. *Extraction du modèle GUI.* Nous analysons le modèle de code source pour détecter les éléments du *code visuel* décrivant l'interface graphique et nous construisons un modèle d'interface graphique à partir de ces éléments. Le méta-modèle de l'interface graphique est décrit Section 4.2.
3. *Export.* Nous recréons l'interface graphique dans la langue cible. Cette étape exporte les fichiers de l'interface utilisateur et les fichiers de configuration de l'application.

Notez qu'actuellement nous ne traitons ni le *code métier* ni le *code comportemental* de l'application. C'est sur ce point que porteront les travaux futurs.

4.2 Méta-modèle GUI

Afin de représenter les interfaces graphiques d'applications de bureau ou Web, nous avons conçu un méta-modèle GUI à partir de ceux présentés dans la Section 2.2.2. Dans la suite de cette section, nous présentons les entités du méta-modèle.

Notre méta-modèle est une adaptation du méta-modèle KDM (voir Figure 1). Comme beaucoup d'autres, nous séparons les éléments graphiques correspondant au DOM des actions et événements. Dans notre méta-modèle, les éléments graphiques sont appelés **Widget**. Ils peuvent être raffinés comme **Leaf** ou **Container**.

Dans notre contexte, l'interface graphique sera toujours affichée sur un écran. Nous ne représentons donc pas tout le type d'UIDisplay et définissons une entité **Page**. La **Page** représente le conteneur principal d'une interface graphique. Cela peut être une fenêtre d'une application de bureau ou une page web. La **Page** est un type de **Container**.

Comme proposé par de nombreux autres auteurs, nous avons ajouté l'entité **Attribute** dans notre méta-modèle d'interface graphique. Un **Attribute** représente les informations d'un widget et peut changer son aspect visuel ou son comportement. Quelques attributs communs sont la hauteur et la largeur pour définir précisément la taille d'un widget. Il y a aussi des attributs qui contiennent des données. Par exemple, un widget représentant un bouton peut avoir un attribut *text* qui contient le texte du bouton. Un attribut peut

changer le comportement d'un widget, c'est le cas de l'attribut *enable*. Un bouton avec l'attribut *enable* réglé sur *false* représente un bouton sur lequel on ne peut pas cliquer. Enfin, les widgets peuvent avoir des attributs qui ont un impact sur l'aspect visuel de l'application. Ce type d'attribut permet de définir une mise en page à respecter par les widgets contenus dans le widget principal et peut éventuellement modifier les dimensions de ce dernier pour respecter une disposition particulière.

5 Implémentation

Pour tester notre approche, nous avons implémenté un outil de migration. Il est implémenté en Pharo [3] et le méta-modèle est représenté à l'aide de la plateforme Moose [4].

5.1 Étude de cas

Les applications chez Berger-Levrault (notre partenaire industriel) sont basées sur le framework BLCore. Ce framework se compose de 763 classes dans 169 paquetages. Il est développé par la société comme extension de GWT et définit les widgets que les développeurs doivent utiliser, l'aspect visuel par défaut des applications, et les classes Java pour connecter le front-end de l'application au back-end. Il encourage également certaines conventions de développement.

Afin d'adapter notre approche aux applications de Berger-Levrault, nous ajoutons une nouvelle entité (**Business Page**) au méta-modèle GUI présenté Section 4.2. C'est une sorte de **Container**. Une convention est que chaque **Page** a une ou plusieurs **Business Pages** représenté comme onglets dans la **Page**. Les **widgets** (boutons, champs texte, tables, ...) sont inclus dans les **Business Pages**, jamais directement dans la **Page**.

5.2 Importation

En partie à cause de la complexité de la mise en place d'un outil pour capturer automatiquement tous les écrans de ces grandes applications web, nous comptons sur l'analyse statique pour créer notre modèle. Les résultats obtenus jusqu'à présent semblent indiquer qu'elle sera suffisante.

Tel que présenté Section 4.1, la création du modèle d'interface graphique est divisée en deux étapes : l'extraction du modèle de code source et l'extraction du modèle GUI. Pour le méta-modèle de code source, nous utilisons le méta-modèle Java de Moose [5, 21] qui vient avec un extracteur de modèle Java [5]. La Figure 5 présente un extrait du code source d'une application de Berger-Levrault. Il montre la méthode buildPageUi(Object object) qui construit l'interface graphique de la "business page" SPMetier1 (une "page métier simple").

```
1  class SPMetier1 extends AbstractSimplePageMetier
       {
2    @Override
3    public void buildPageUi(Object object) {
4      BLLinkLabel lblPg = new BLLinkLabel("Next");
5      lblPg.addClickHandler(new ClickHandler(){
6        public void onClick(ClickEvent event) {
7          SPMetier1.this.fireOnSuccess("param");
8        }
9      });
10     lblPg.setEnabled(false);
11     vpMain.add(new Label("<Business content>"));
12     vpMain.add(lblPg);
13     super.setBuild(true);
14   }
15 }
```

FIGURE 5 – Création de l'interface graphique en GWT

Pour la deuxième étape de l'extraction, notre outil crée le modèle GUI à partir du modèle de code source et une analyse du fichier de configuration XML. Les entités que nous voulons extraire sont d'abord les **Pages**. Nous analysons le fichier de configuration XML dans lequel est définie l'information sur les pages (voir Section 3.2). Il fournit pour chaque **Page** (appelé *phase* dans le fichier XML, Figure 3) son nom et le nom de la classe Java qui le définit. Ensuite, l'outil recherche les **Widgets**.

Tout d'abord, l'outil détermine les widgets disponibles. Pour ce faire, il collecte toutes les sous-classes Java de la classe GWT Widget. Pour les **Business Pages**, l'outil recherche les classes qui implémentent l'interface IPageMetier. Ensuite, l'outil regarde où les constructeurs des **Widgets** sont appelés et crée les liens entre les **Widgets** et leurs **Widget** parents. Dans la Figure 5, il y a deux appels aux constructeurs d'un**Widget** : ligne 4, le constructeur de BLLinkLabel est appelé, et ligne 11, le constructeur de Label. La variable vpMain correspond au panneau principal de la **Business Page**. Les lignes 11 et 12 correspondent à l'ajout d'un widget dans le panneau principal grâce à la méthode add().

Enfin, pour détecter les attributs et les actions qui appartiennent à un widget, l'outil détecte dans quelle variable Java le widget a été affecté. Ensuite, il recherche les méthodes invoquées sur cette variable. Si un widget invoque la méthode *"addClickHandler"*, il crée un événement. S'il invoque une méthode *"setX"*, il crée un attribut. Ces heuristiques ont été trouvées dans la littérature [22, 25]. Dans la Figure 5, le BLLinkLabel, dont la variable est lblPg, est lié à un événement et à un attribut. Les lignes 5 à 9 correspondent à la création d'un événement avec le code exécutable. La ligne 10 correspond à l'ajout de l'attribut enabled. avec la valeur false.

3. Pharo est un langage de programmation orienté objet inspiré de Smalltalk. http://pharo.org/

4. Moose est une plateforme d'analyse de logiciels et de données. http://www.moosetechnology.org/

5. verveineJ https://github.com/moosetechnology/verveineJ

5.3 Exportation

Une fois le modèle GUI généré, il est possible d'exporter l'application. Pour générer le code de l'application cible, l'outil inclut un exportateur. L'exportateur crée les dossiers de l'application cible et les fichiers de configuration. Puis, il visite les pages. Pour chaque **Page**, l'exportateur crée un sous-projet Angular sous la forme d'un répertoire contenant plusieurs fichiers de configuration et une page web vide par défaut. Ensuite, pour chaque business page de la **Page** visité, l'exportateur génère un fichier HTML et un fichier TypeScript. Pour le fichier HTML, l'exportateur construit le DOM grâce au patron de conception Composite utilisé dans le méta-modèle GUI (voir Section 4.2). Chaque widget fournit ses attributs et actions à l'exportateur.

6 Validation

Dans cette section, nous décrivons l'application industrielle sur laquelle nous avons utilisé notre outil pour valider notre approche. La Section 6.1 présente l'application industrielle. La Section 6.2 présente les métriques que nous avons utilisées pour évaluer notre approche.

6.1 Application industrielle

Nous avons expérimenté notre approche sur l'application *kitchensink* de Berger-Levrault. Ce logiciel, dédié aux développeurs, a pour but de regrouper en une seule et même application l'ensemble des composants disponible pour construire une interface graphique. Cette application est plus petite qu'une application de production mais utilise le framework BLCore. Le framework de l'entreprise nous garantit que le fonctionnement de l'application *kitchensink* est exactement le même que les applications industrielles. L'application contient 470 classes Java et représente 56 pages web. Bien que ce soit l'application de démonstration pour les développeurs, l'application *kitchensink* contient des irrégularités dans le code.

Notez que l'application *kitchensink*, comme les autres applications industrielles de la société, n'a pas de test. Il n'est donc pas possible d'utiliser des tests pour valider la migration.

6.2 Métriques de validation

La validation se fait en trois étapes : premièrement, nous vérifions les contraintes définies dans la Section 3.1 ; deuxièmement, nous validons que toutes les entités d'intérêt de l'interface graphique sont extraites et correctement extraites ; troisièmement, nous validons que nous pouvons réexporter ces entités dans Angular et que le résultat est correct.

Pour la première validation, nous identifions et comptons manuellement les entités de l'application *kitchensink* et comparons les résultats obtenus avec l'outil. Notre analyse porte sur la migration de trois entités : **Pages**, **Business Pages** et **Widgets**

- **Pages.** À partir du fichier de configuration XML de l'application, nous comptons manuellement 56 pages. Ce fichier de configuration fournit également le nom de chaque page.
- **Business Pages.** Comme expliqué précédemment, les business pages correspondent à un concept propre à Berger-Levrault. Ils sont définis dans le framework BLCore comme une classe Java qui implémente l'interface IPageMetier. Grâce à cette heuristique, nous comptons manuellement 76 instances **Business Page** dans l'application originale.
- **Widgets.** Dans l'état de l'art, nous n'avons pas trouvé de moyen automatique d'évaluer la détection de widgets. La vérification de tous les widgets de l'application serait longue et sujette aux erreurs car il y en a des milliers. Comme solution de repli, nous prenons un échantillon des pages de l'application *kitchensink* et comptons les widgets dans le DOM de ces pages. Nous considérons un échantillon de 6 **Pages** qui représente un peu plus de 10% des **Pages** de l'application. Ces **Pages** sont de différentes tailles et contiennent différents types de widgets. Au total, nous avons trouvé 238 **Widgets** dans ces 6 **Pages**. Pour avoir une idée plus précise de la représentativité de notre échantillon, nous comptons aussi le nombre de création de **Widgets** (*i.e.* `new AWidgetClass()`) dans le code. Il existe 2 081 créations de ce type. Cela peut ne pas représenter le nombre exact de widgets dans l'application entière, mais c'est une bonne estimation. Nous notons que le nombre de **Widgets** dans notre échantillon (un peu plus de 10 % des pages) représente également un peu plus de 10 % de notre estimation du nombre total de widgets.
  Pour l'évaluation, nous vérifions également que les **Widgets** sont correctement placés dans le DOM de l'interface graphique (*i.e.* ils appartiennent au bon **Container** dans le modèle GUI).

Dans nos résultats, nous ne considérons que le rappel de l'outil car la précision est toujours de 100 % (il n'y a pas de faux positif). C'est un signe que le framework BLCore fournit des heuristiques claires (sinon complètes) pour identifier les entités.

Pour la deuxième validation, nous vérifions que les entités sont correctement exportées. Dans l'application Angular, chaque **Page** correspond à un sous-projet et est représentée par un dossier. Le nom du dossier doit correspondre au nom de la **Page**. Les **Business Pages** sont représentées par un sous-dossier dans le projet de la **Page**. Les noms doivent également correspondre à ce niveau.

Nous vérifions également visuellement que le **Page** exporté "ressemble" à l'original. Il s'agit d'une évaluation

subjective, et nous cherchons des options pour l'automatiser à l'avenir.

7 Résultats

Cette section présente les résultats de la validation de la migration de l'application *kitchensink* de Berger-Levrault. La Section 7.2 résume les résultats de l'extraction. Dans la Section 7.1, nous confrontons le résultat exporté avec les contraintes définies Section 3.1.

7.1 Satisfaction des contraintes

Nous avons défini les contraintes suivantes Section 3.1 : *Depuis GWT vers Angular*, *Adaptabilité de l'approche*, *Préservation du visuel*, *Conservation de la qualité du code*, and *Automatique*.

Notre outil peut utiliser du code Java en entrée et générer du code Angular. Le code exporté est compilable et exécutable. L'application cible peut être affichée. Nous pouvons donc confirmer que notre outil remplit la contrainte depuis GWT vers Angular.

Notre outil est utilisable sur d'autres technologies cibles source. Nos heuristiques ont été conçues pour être faciles à adapter, un utilisateur de notre outil peut ainsi ajouter un nouveau type de widget pour les phases d'importation ou d'exportation. Nous décrivons brièvement une petite expérience dans ce sens Section 8.5. Ces possibilités répondent à la contrainte d'adaptabilité.

Les contraintes *Conservation de la qualité du code* et *Préservation du visuel* sont discutées dans la Section 7.3, dans les résultats de la troisième validation.

Enfin, les résultats décrits ici ont été obtenus automatiquement en applicant notre outil sur l'application de Berger-Levrault. Ceci valide la dernière contrainte.

7.2 Résultats de l'extraction

La Table 2 résume les résultats de l'extraction.

TABLE 2 – Résultats de l'extraction

	Pages	**Business Pages**	**Widgets** (sample)
Nombre	56	76	238
Correctement importé	100%	100%	89%

L'outil a extrait 56 **Pages** de l'interface graphique originale. Cela correspond au nombre de pages définies dans le fichier de configuration de l'application *kitchensink*.

L'outil a extrait 76 **Business Pages**. Cette valeur correspond exactement au nombre de business pages de l'application d'origine. De plus, l'outil assigne correctement chaque **Business Page** à sa **Page**.

Nous avons obtenu 100 % des **Widgets** sur l'échantillon évalué qui ont été correctement détectés. Cependant, 27 des 238 **Widgets** de notre échantillon (11%) n'ont pas été correctement affectés à leur conteneur. Tous ces problèmes viennent d'une seule et unique **Page** (contenant 75 **Widgets** au total).

7.3 Résultats à l'exportation

Nous avons vérifié manuellement le nom des 56 pages exportées. Elles conservent toutes leur nom d'origine.

(a) GWT original (b) Angular migration

FIGURE 6 – Comparaison du visuelle de la migration d'une **Page**

Figure 6 présente les différences visuelles entre la version originale (GWT), à gauche, et celle migrée (Angular 6), à droite. On voit qu'il y a peu de différences. Dans la version exportée, la couleur de l'en-tête est un peu plus claire, et les lignes sont un peu plus éloignées.

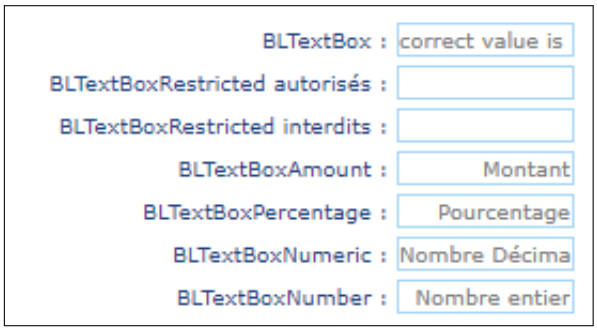
Figure 7 présente les différences visuelles pour la **Page** *Input box*. De nouveau sur le côté gauche il y a la **Page** original et sur le côté droit la même **Page** après la migration. Comme les deux images sont grandes, nous les avons rognées pour afficher cette zone d'intérêt. Même si les deux images semblent complètement différentes, tous les widgets sont présents dans la version migrée. Les différences visuelles sont dues à un problème dans la gestion de la mise en page. La contrainte visuelle est ainsi partiellement satisfaite. Ce point est discuté Section 8.1.

8 Discussion

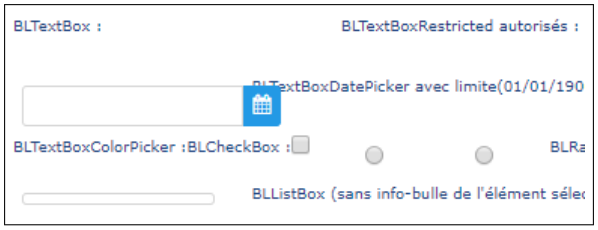
La Section 8.1 et la Section 8.2 présentent deux parties de l'interface utilisateur sur lesquelles nous n'avons pas travaillé. La Section 8.3 discute de l'impact du choix de *kitchensink* comme étude de cas. La Section 8.4 met en évidence les difficultés de validation à grande échelle de notre outil. Enfin, la Section 8.5 traite de l'impact du framework BLCore.

8.1 Gestion de la mise en page

Comme indiqué dans la Section 7.3, l'exportation peut donner des résultats incorrects à cause de problèmes de mise en page (Figure 7). Cela est dû à la représentation de la mise en page dans notre méta-modèle d'interface graphique. Actuellement, les modèles sont représentés dans notre méta-modèle d'interface graphique sous forme d'attribut sur un **Widget Conrtainer** définissant si les enfants de ce widget sont placés l'un à côté de l'autre ou l'un en dessous de l'autre (*i.e.* Mise en page verticale ou horizontale). Cependant, il existe beaucoup d'autres mises en page [14]. Par exemple, le framework BLCore fournit le widget



(a) GWT original



(b) Angular migration

FIGURE 7 – Comparaison du visuelle de la migration d'une **Page** : Tous les **Widgets** sont migrés mais avec un mauvais layout

`BLGrid`, un **Widget** héritant de la classe GWT `Grid` et implémentant une mise en page de grille. Actuellement, les mises en page complexes ne sont pas prises en compte dans notre méta-modèle d'interface graphique.

Une solution est proposée par Sánchez Ramán *et al.* [23]. Ils ont conçu un méta-modèle de mise en page. L'idée consiste à lier des widgets à une mise en page et à combiner les mises en page pour créer une représentation précise. Les auteurs ont défini un sous-ensemble de mise en page possible à connecter aux widgets.

De plus, avec de telles mises en page, la position des **Widgets** enfants peut être calculée au moment de l'exécution. Par exemple, dans une grille, les enfants peuvent être positionnés en fonction des valeurs de certaines variables `ligne` et `colonne`. Trouver ces valeurs est une tâche compliquée avec une analyse statique. C'est un cas où une approche hybride pourrait être nécessaire.

8.2 Gestion du code comportemental et métier

Actuellement, seule la partie visuelle de l'interface graphique est migrée. Pour prendre en compte l'ensemble de l'application, les migrations du *code comportemental* et du *code métier* (voir Section 3.3) sont nécessaires. Le *code comportemental* représente les interactions de l'utilisateur (clic, glisser-déposer, survoler, ...) et les structures de contrôle (*i.e.* loop et alternative). Dans le cas d'une application client/serveur, les requêtes au serveur font partie du *code comportemental*, alors que la requête en elle-même et les données appartiennent au *code métier*.

8.3 Application demo

Bien que les résultats soient encourageants, nous avons évalué notre outil sur l'application *kitchensink*. L'application *kitchensink* est un bon terrain d'entraînement pour notre outil car elle contient toute les types de widgets que les développeurs ont à leur disposition et la façon de les utiliser. Cependant, elle peut s'écarter des applications de production car elle doit contenir moins d'irrégularités ou d'astuces de codage que ces dernières.

8.4 Outils de validation

La validation automatique de la migration est actuellement un problème non résolu. Il est possible de vérifier manuellement le résultat de la migration pour quelques pages mais il est préférable de le faire automatiquement pour des centaines de pages (plus de 400 sur les applications de Berger-Levrault).

Nous n'avons trouvé, dans la littérature, que peu d'approches envisageant une validation visuelle automatique. Dans deux articles [11, 23], les auteurs comptent simplement le nombre de widgets dans les applications sources et les applications cibles. Mais nous avons vu dans Figure 7 que cela ne garantit pas la similarité visuelle. Un autre article [19] propose de comparer les captures d'écran de l'application originel et de l'application exportée pixel par pixel. Cependant, nous avons vu dans Figure 6 que des écrans aux différences à peine distinguables peuvent avoir des différences au niveau des pixels.

8.5 Impacte de BLCore

Comme expliqué dans la Section 5.1, les applications de Berger-Levrault sont basées sur le framework BLCore. En spécialisant GWT, BLCore fournit des widgets spécifiques et une API dédiée. Cela peut ou non avoir un impact sur notre approche. Pour évaluer ce possible impact, et aussi pour valider la généralité de notre approche, nous avons réalisé deux petites expériences considérant (i) Spec (un framework d'interface utilisateur de bureau dans Pharo[6]) comme framework source et, (ii) Seaside (un framework Web dans Pharo [4] – également décrit à seaside.st) comme framework cible. Ces expériences prennent donc en compte différents langages de programmation (Pharo au lieu de Java (GWT) et TypeScript), différents frameworks d'interface graphique et différentes applications web et de bureau. Nous avons expérimenté la migration de l'interface graphique de l'application de démonstration de Spec vers Angular, et la migration de l'application *kitchensink* de Berger-Levrault vers Seaside.

Certaines conclusions sont :

— Il était plus difficile d'importer du code Spec que GWT à cause d'une plus grande variabilité dans la définition de l'interface graphique. Nous concluons que le framework BLCore a facilité notre travail sur l'importation en normalisant la façon de construire les pages.
— Pour Seaside, il était facile de migrer des widgets simples (*e.g.* Label, Button, Panel), mais le framework BLCore définit également des widgets complexes sans équivalent direct dans Seaside. Une bibliothèque similaire à BLCore devrait être définie dans Seaside pour faciliter la migration.
— La capacité de migration de notre méta-modèle d'interface graphique et l'extraction en deux étapes (d'abord, l'extraction du code source, puis l'extraction du modèle d'interface graphique, voir Figure 4) est validée par le fait que nous avons pu migrer une application de bureau Pharo avec peu de travail supplémentaire.

9 Conclusion et travaux futurs

Nous avons créé un outil avec des résultats prometteurs sur la représentation de l'interface graphique pour migrer des applications GWT vers Angular. Dans ce qui suit, nous concluons la présentation de ce travail et proposons quelques pistes de recherche que nous voulons explorer.

9.1 Conclusion

Dans cet article, nous avons exposé un travail préliminaire sur le problème de la préservation visuelle et le respect de l'architecture cible lors de la migration de l'interface graphique d'une application. Nous avons proposé une approche basée sur un méta-modèle d'interface graphique et un processus de migration en trois étapes. Nous avons implémenté ce processus dans un outil pour effectuer la migration des applications GWT vers Angular 6. Ensuite, nous avons validé notre outil avec une expérience sur une application de démonstration (*kitchensink*). Nous avons pu extraire correctement toutes les pages de l'application et 89 % des widgets. L'application migré a le même visuel que l'application originelle tant que des widgets complexes (*e.g.* GridLayout) ne sont pas utilisés. Notre prochain défi sera de faire face aux problèmes de mise en page.

Notre solution nous permet également de respecter les conventions de nommage utilisées dans l'application source ainsi que la structure du code dans la mesure où les différences entre les frameworks GUI le permettent.

9.2 Travaux futurs

Afin d'améliorer la migration d'une interface graphique, nous allons améliorer notre méta-modèle et notre outil pour supporter la gestion des mises en page et du code comportemental et métier.

Nous n'avons pas trouvé d'approche ou de métriques pour évaluer automatiquement la validité des écrans migrés. Il est donc important de trouver une nouvelle façon d'évaluer si les écrans migrés préservent l'aspect visuel des écrans originaux.

Avoir un bon méta-modèle d'interface graphique ouvre également la porte à un constructeur d'interface graphique générique qui pourrait la créer dans plusieurs frameworks d'interface graphique.

Références

[1] Marco Brambilla and Piero Fraternali. *Interaction flow modeling language : Model-driven UI engineering of web and mobile apps with IFML*. Morgan Kaufmann, 2014.

[2] John Brant, Don Roberts, Bill Plendl, and Jeff Prince. Extreme maintenance : Transforming Delphi into C#. In *ICSM'10*, 2010.

[3] Jonathan Cloutier, Segla Kpodjedo, and Ghizlane El Boussaidi. WAVI : A reverse engineering tool for web applications. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–3. IEEE, 2016.

[4] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010.

[5] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0 : an Interexchange Format and Source Code Model Family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[6] Johan Fabry and Stéphane Ducasse. *The Spec UI Framework*. Square Bracket Associates, 2017.

[7] Franck Fleurey, Erwan Breton, Benoit Baudry, Alain Nicolas, and Jean-Marc Jezéquel. Model-Driven Engineering for Software Migration in a Large Industrial Context. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Model Driven Engineering Languages and Systems*, volume 4735, pages 482–497, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[8] Kelly Garcés, Rubby Casallas, Camilo Álvarez, Edgar Sandoval, Alejandro Salamanca, Fredy Viera, Fabián Melo, and Juan Manuel Soto. White-box modernization of legacy applications : The oracle forms case study. *Computer Standards & Interfaces*, pages 110–122, October 2017.

[9] Zineb Gotti and Samir Mbarki. Java swing modernization approach - complete abstract representation based on static and dynamic analysis :. In *Proceedings of the 11th International Joint Conference on Software Technologies*, pages 210–219. SCITEPRESS - Science and Technology Publications, 2016.

[10] Tomokazu Hayakawa, Shinya Hasegawa, Shota Yoshika, and Teruo Hikita. Maintaining web applications by translating among different ria technologies. *GSTF Journal on Computing*, page 7, 2012.

[11] Mona Erfani Joorabchi and Ali Mesbah. Reverse engineering iOS mobile applications. In *2012 19th Working Conference on Reverse Engineering*, pages 177–186. IEEE, 2012.

[12] R. Kazman, S.G. Woods, and S.J. Carriére. Requirements for integrating software architecture and reengineering models : Corum ii. In *Proceedings of WCRE '98*, pages 154–163. IEEE Computer Society, 1998.

[13] Valéria Lelli, Arnaud Blouin, Benoit Baudry, Fabien Coulon, and Olivier Beaudoux. Automatic detection of GUI design smells : The case of blob listener. *EICS '16 Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, page 12, 2016.

[14] Simon Lok and Steven Feiner. A survey of automated layout techniques for information presentations. *Proceedings of SmartGraphics*, 2001 :61–68, 2001.

[15] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI ripping : reverse engineering of graphical user interfaces for testing. In *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, pages 260–269. IEEE, 2003.

[16] Atif M. Memon. An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability*, 17(3) : 137–157, 2007.

[17] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *ACM Transactions on the Web*, 6(1) :1–30, 2012.

[18] Moore, Rugaber, and Seaver. Knowledge-based user interface migration. In *Proceedings 1994 International Conference on Software Maintenance*, pages 72–79. IEEE Comput. Soc. Press, 1994.

[19] Kevin Moran, Cody Watson, John Hoskins, George Purnell, and Denys Poshyvanyk. Detecting and Summarizing GUI Changes in Evolving Mobile Apps. *arXiv :1807.09440 [cs]*, July 2018.

[20] I Coimbra Morgado, Ana Paiva, and J Pascoal Faria. Reverse engineering of graphical user interfaces. In *ICSEA 2011 : The Sixth International Conference on Software Engineering Advances*, 2011.

[21] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose : an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press.

[22] Hani Samir, Amr Kamel, and Eleni Stroulia. Swing2script : Migration of Java-Swing applications to Ajax Web applications. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, 2007.

[23] Óscar Sánchez Ramán, Jesús Sánchez Cuadrado, and Jesús García Molina. Model-driven reverse engineering of legacy graphical user interfaces. *Automated Software Engineering*, 21(2) :147–186, 2014.

[24] Eeshan Shah and Eli Tilevich. Reverse-engineering user interfaces to facilitate porting to and across mobile devices and platforms. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE ! 2011, AOOPES'11, NEAT'11, \& VMIL'11*, pages 255–260. ACM, 2011.

[25] João Carlos Silva, Carlos C. Silva, Rui D. Goncalo, João Saraiva, and José Creissac Campos. The GUISurfer tool : towards a language independent approach to reverse engineering GUI code. In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, pages 181–186. ACM Press, 2010.

[26] Stefan Staiger. Reverse engineering of graphical user interfaces using static analyses. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 189–198. IEEE, 2007.

[27] Christian Zirkelbach, Alexander Krause, and Wilhelm Hasselbring. On the modernization of explorviz towards a microservice architecture. In *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. CEUR Workshop Proceedings, 2018.

Pattern eliminating transformations

Pierre Lermusiaux, Horatiu Cirstea, and Pierre-Etienne Moreau

Université de Lorraine – LORIA
name.surname@loria.fr

Abstract

Program transformation is a common practice in computer science, and its many applications can have a range of different objectives. For example, a program written in an original high level language could be either translated into machine code for execution purposes, or towards a language suitable for formal verification. Languages often have a lot of different constructions, and thus, transformations often focus on eliminating some of these constructions, or at least processing some specific sequence of constructions. Rewriting is a widely established formalism to describe the mechanism and the logic behind such transformations. It relies mainly on the principle of rewrite rules to describe the different operations performed. Generally type-preserving, these rewrite rules can ensure that the transformation result has a given type and thus give syntactic guarantees on the constructions it consists of. However, we sometimes want the transformation to provide more guarantees on the shape of its result by specifying that some patterns of constructions does not appear. For this purpose, we propose in this paper an approach based on annotating transformation function symbols with (anti-)pattern in order for such transformation to guarantee stronger properties on the shape of its result. With the generic principles governing term algebra and rewriting, we believe this approach to be an accurate formalism to any language providing pattern-matching primitives.

1 Introduction

Rewriting is a widely established formalism for a number of applications in both computer science and mathematics and has been used, in particular, to describe program semantics [13] and transformations [11, 4]. In general, compilation consists in several phases, also called passes, eventually obtaining a program in a different target language. These phases use some corresponding intermediate languages which generally contain less and less constructions of the original language.

Consider, for example, a very simple language allowing to express some form of λ-expressions:

$$\begin{array}{rcccc} Expr & = & Var(String) & | & Apply(Expr, Expr) \\ & | & Lambda(String, Expr) & | & Let(String, Expr, Expr) \end{array}$$

A first step in the compilation of such expressions is to eliminate the *Let* constructor using the rewrite rule $Let(name, e_1, e_2) \twoheadrightarrow Apply(Lambda(name, e_2), e_1)$ to obtain pure λ-expressions.

The goal is to provide a formalism allowing to describe such transformations and to guarantee that some language constructs are eliminated by these transformations.

Existing works addressed this problem from different perspectives. In particular, a number of approaches rely on the use of automata to implement transformation. Such approach is particularly relevent as the input and output of the transformation can be viewed as a regular language [1] or a tree [3]. While this approach focus on the decidability and complexity of some specific cases it does not handle the general problems we target here. Functionnal approaches to transformation [12] focus on the simplicity and expressiveness of the formalism more than on verifying the elimination of constructs. This verification can be performed by using fine grained typing systems which combine overloading, subtyping and polymorphism through the use of variants [7] but is somewhat limited w.r.t. the type of constructs which can be eliminated.

We use rewriting to define such transformations and annotate the function symbols in order to check that the associated transformation verifies the desired elimination properties.

First, in the next section, we will introduce all the notions and notations used in the article. We will then, in the following section, give a formal definition to the property we want pattern eliminating transformation to guarantee: pattern-freeness. The 2 subsequent sections will be focused respectively on introducing a method to reliably verify such properties and on studying how they can be guaranteed by the considered transformations, using and extending notions of pattern semantics introduced in the notions of the next section.

2 Pattern matching and term rewriting systems

We define in this section the basic notions and notations used in this paper; more details can be found in [2, 14].

A *many-sorted signature* $\Sigma = (\mathcal{S}, \mathcal{F})$, consists of a set of sorts $\mathcal{S}$ and a set of symbols $\mathcal{F}$. We distinguish constructors symbols from functions symbols by partitioning the alphabet $\mathcal{F}$ into $\mathcal{D}$, the set of *defined symbols*, and $\mathcal{C}$ the set of *constructors*. A symbol f with *domain* $\mathcal{D}om(f) = s_1 \times \ldots \times s_n \in \mathcal{S}^*$ and *co-domain* $\mathcal{C}o\mathcal{D}om(f) = s \in \mathcal{S}$ is written $f : s_1 \times \ldots \times s_n \mapsto s$. Variables are also sorted and we write $x : s$ or x^s to indicate that variable x has sort s. The set $\mathcal{X}_s$ denotes a set of variables of sort s and $\mathcal{X} = \bigcup_{s \in \mathcal{S}} \mathcal{X}_s$ is the set of sorted variables.

The set of terms of sort $s \in \mathcal{S}$, denoted $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$ is the smallest set containing $\mathcal{X}_s$ and such that $f(t_1, \ldots, t_n)$ is in $\mathcal{T}_s(\mathcal{F}, \mathcal{X})$ whenever $f : s_1 \times \ldots \times s_n \mapsto s$ and $t_i \in \mathcal{T}_{s_i}(\mathcal{F}, \mathcal{X})$ for $i \in [1, n]$. We write $t : s$ when the term t is of sort s, *i.e.* when $t \in \mathcal{T}_s(\mathcal{F}, \mathcal{X})$. The set of *sorted terms* is defined as $\mathcal{T}(\mathcal{F}, \mathcal{X}) = \bigcup_{s \in \mathcal{S}} \mathcal{T}_s(\mathcal{F}, \mathcal{X})$. The set of variables occurring in $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is denoted by $\mathcal{V}ar(t)$. If $\mathcal{V}ar(t)$ is empty, t is called a *ground* term. $\mathcal{T}_s(\mathcal{F})$ denotes the set of all ground first-order terms of sort s and $\mathcal{T}(\mathcal{F})$ denotes the set of all ground first-order terms, while members of $\mathcal{T}(\mathcal{C})$ are called *values*. A *linear* term is a term where every variable occurs at most once, and linear terms in $\mathcal{T}(\mathcal{C}, \mathcal{X})$ are called *patterns*.

A *position* of a term t is a finite sequence of positive integers describing the path from the root of t to the root of the sub-term at that position. The empty sequence representing the root position is denoted by ε. $t_{|\omega}$ denotes the sub-term of t at position ω and $t[s]_\omega$ denotes the term t with the sub-term at position ω replaced by s. We note $\mathcal{P}os(t)$ the set of positions of t.

We call *substitution* any mapping from $\mathcal{X}$ to $\mathcal{T}(\mathcal{F}, \mathcal{X})$ which is the identity except over a finite set of variables called its domain; any substitution extends as expected to an endomorphism of $\mathcal{T}(\mathcal{F}, \mathcal{X})$. Given a sort s, a value $v : s$ and a constructor pattern p, we say that p *matches* v (denoted $p \lessdot v$) if there exists a substitution σ such that $v = \sigma(p)$.

A *constructor rewrite rule* is of the form $\varphi(l_1, \ldots, l_n) \twoheadrightarrow r \in \mathcal{T}_s(\mathcal{F}, \mathcal{X}) \times \mathcal{T}_s(\mathcal{F}, \mathcal{X})$ with $s \in \mathcal{S}$, $\varphi \in \mathcal{D}$, $l_1, \ldots, l_n \in \mathcal{T}(\mathcal{C}, \mathcal{X})$ and such that $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(l)$. A *constructor term rewriting system* (CTRS) is a set of rewrite rules $\mathcal{R}$ inducing a *rewriting relation* over $\mathcal{T}(\mathcal{F})$, denoted by $\longrightarrow_{\mathcal{R}}$ and such that $t \longrightarrow_{\mathcal{R}} t'$ iff there exist $l \twoheadrightarrow r \in \mathcal{R}$, $\omega \in \mathcal{P}os(t)$, and a substitution σ such that $t_{|\omega} = \sigma(l)$ and $t' = t[\sigma(r)]_\omega$.

Starting from the observation that a pattern can be interpreted as the set of its instances, the notion of *ground semantics* was introduced in [6] as the set of all ground constructor instances of a pattern $p \in \mathcal{T}_s(\mathcal{C}, \mathcal{X})$: $[\![p]\!] = \{\sigma(p) \mid \sigma(p) \in \mathcal{T}_s(\mathcal{C})\}$.

Proposition 1. *Given a pattern p and a value v, $v \in [\![p]\!]$ iff $v = \sigma(p)$.*

Note that the ground semantics of a variable x^s is the set of all possible ground patterns: $[\![x^s]\!] = \mathcal{T}_s(\mathcal{C})$, and since patterns are linear we can use a recursive definition for the non variable patterns: $[\![c(p_1, \ldots, p_n)]\!] = \{c(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in [\![p_1]\!] \times \ldots \times [\![p_n]\!]\}, \forall c \in \mathcal{C}$. Moreover

$[\![x^s]\!] = \{c(t_1, \ldots, t_n) \mid c \in \mathcal{C} \text{ s.t. } c : s_1 \times \ldots \times s_n \mapsto s \wedge \forall i, t_i \in [\![x^{s_i}]\!]\} = \bigcup_{c \in \mathcal{C}_s} [\![c(x_1, \ldots, x_n)]\!]$.

We consider also a special pattern $\bot$ with $[\![\bot]\!] = \emptyset$. Given two patterns p, q we can compute [6] their complement $p \setminus q$, *i.e.* a set $p_1, \ldots, p_n$ of patterns s.t. $[\![p \setminus q]\!] = [\![p_1, \ldots, p_n]\!]$.

3 Pattern free terms

The sort provides some information on the shape of the terms of the respective sort and in particular allows one to check if a given symbol may be present or not in these terms. In fact, the precise language of the values of a given sort is implicitly given by the signature. Sorts are less informative concerning the shape of the values obtained when reducing terms containing defined symbols since they strongly depend on the CTRS defining these symbols.

We want to ensure that the normal form of a term, if it exists, does not contain a constructor and more generally that no subterm of this normal form matches a given pattern. For this we annotate all defined symbols with the patterns that are supposed to be absent from the normal form and we check that the CTRS defining the corresponding functions are consistent with these annotations. We will see later on how this consistence can be verified and we focus first on the notion of pattern-free term and the corresponding ground semantics.

We consider that every defined symbol $f \in \mathcal{D}$ is annotated with a pattern $p \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \cup \{\bot\}$ and we use this notation to define *pattern-free* terms. Intuitively, a term of the form $f^{-p}(t_1, \ldots, t_n)$ should ultimately be reduced to a value containing no subterms matched by p.

Definition 3.1 (Pattern-free)**.** *Given a pattern $p \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \cup \{\bot\}$,*

- *a value $v \in \mathcal{T}(\mathcal{C})$ is p-free iff $\forall \omega \in \mathcal{P}os(v), p \not\lessdot v_{|\omega}$;*
- *a ground term $t \in \mathcal{T}(\mathcal{F})$ is p-free iff $\forall \omega \in \mathcal{P}os(t)$ such that $t_{|\omega} = f^{-q}(t_1, \ldots, t_n)$ with $f \in \mathcal{D}, q \in \mathcal{T}(\mathcal{C}, \mathcal{X}) \cup \{\bot\}$, $\mathcal{C}o\mathcal{D}om(f) = s$, we have $\forall v \in \mathcal{T}_s(\mathcal{C})$ q-free, $t[v]_\omega$ is p-free;*
- *a linear term $t \in \mathcal{T}(\mathcal{F}, \mathcal{X})$ is p-free iff $\forall \sigma$ such that $\sigma(t) \in \mathcal{T}(\mathcal{F})$, $\sigma(t)$ is p-free.*

A value is p-free if and only if p matches no subterm of the value. A ground term is p-free if and only if replacing (all) the subterms headed by a defined symbol f^{-q} by any appropriate q-free value results in a p-free term. For general terms verifying the pattern-freeness comes to verifying the property for all ground instances of the term. While the pattern-freeness of a value can be checked by exploring all its subterms this is not possible for a general term since we potentially have to check the pattern-freeness of an infinite number of values. We present in the next section an approach for solving this problem.

4 Extensions of ground semantics

The previously introduced ground semantics can be used to compare the shape of the root of a constructor pattern p_1 to another constructor pattern p_2. Indeed, by definition, if $[\![p_1]\!] \cap [\![p_2]\!] = \emptyset$, then $\forall \sigma, \sigma(p_1) \notin [\![p_2]\!]$ so $p_2 \not\lessdot \sigma(p_1)$.

Example 4.1. *Consider the signature Σ with $\mathcal{S} = \{s_1, s_2\}$ and $\mathcal{F} = \mathcal{C} = \{c_1 : s_2, s_1 \mapsto s_1, c_2 : s_2 \mapsto s_1, c_3 : [] \mapsto s_1, c_4 : s_2 \mapsto s_2, c_5 : s_1 \mapsto s_2\}$. Using the transformation method proposed in [6] the complement $c4(c3()) \setminus c4(x)$ is reduced to $\bot$ indicating that $[\![c4(c3()) \setminus c4(x)]\!] = [\![c4(c3())]\!] \setminus [\![c4(x)]\!] = \emptyset$ and thus that $c4(c3())$ is redundant w.r.t. $c4(x)$. Moreover $c4(c3()) \setminus (c4(c3()) \setminus c4(x))$ is reduced to $c4(c3()) \setminus \bot$ and then to $c4(c3())$ indicating now that $[\![c4(c3())]\!] \cap [\![c4(x)]\!] \neq \emptyset$ and thus that $c4(c3())$ is not $c4(x)$ free.*

Similarly, $c4(c3()) \setminus (c4(c3()) \setminus c5(x))$ and $c3() \setminus (c3() \setminus c5(x))$ are both reduced to $\bot$ and consequently we can deduce that $c4(c3())$ is $c5(x)$-free.

The pattern-freeness properties in the above example could have been checked by trying to match all the subterms and we can see this method as a starting point for the generalisation to general terms. We first introduce an extended ground semantics:

Definition 4.1 (Extended ground semantics). *Given $s \in \mathcal{S}$ and a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$*

- $[\![x^s_{-p}]\!] = \{v \mid v \in \mathcal{T}_s(\mathcal{C}) \wedge v \text{ p-free}\}$;
- $[\![f^{-p}(t_1,\ldots,t_n)]\!] = [\![x^s_{-p}]\!]$ *with* $\mathcal{CoDom}(f) = s$;
- $[\![c(t_1,\ldots,t_n)]\!] = \{c(v_1,\ldots,v_n) \mid v_i \in [\![t_i]\!]\}$ *with* $c : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{C}, t_i \in \mathcal{T}_{s_i}(\mathcal{F},\mathcal{X})$.

The ground semantics of a term rooted by a defined symbol represents an over-approximation of all the possible values obtained by reducing the term with respect to a TRS preserving the pattern-freeness, this by taking into account the annotation of the respective defined symbol. When restricting to patterns we retrieve the original definition of ground semantics.

The extended ground semantics of a p-free variable of sort s can be also defined as:

$$[\![x^s_{-p}]\!] \quad = \quad \bigcup_{c \in \mathcal{C}_s} [\![c(x^{s_1}_{-p},\ldots,x^{s_i}_{-p})]\!] \setminus [\![p]\!]$$

and when $p = \bot$ we retrieve the corresponding definition for the classical ground semantics. This observation allows us to easily adapt the method introduced in [6] for computing constructor pattern complements to general annotated terms.

We can now establish pattern-free properties using this extended ground semantics and the *reachable sorts w.r.t.* a given sort s: $\lfloor s \rfloor = \{s' \mid \exists\, t \in \mathcal{T}_s(\mathcal{C}), \omega \in \mathcal{Pos}(t) \text{ s.t. } t_{|\omega} : s'\}$

Proposition 2 (Pattern-free vs Extended Ground Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}_s(\mathcal{F},\mathcal{X})$:*

- *if $t = x^s$ and $\forall s' \in \lfloor s \rfloor, [\![x^{s'}_{\bot}]\!] \cap [\![p]\!] = \emptyset$ then t is p-free;*
- *if $t = f^{-q}(t_1,\ldots,t_n)$ and $\forall s' \in \lfloor s \rfloor, [\![x^{s'}_{-q}]\!] \cap [\![p]\!] = \emptyset$ then t is p-free;*
- *If $t = c(t_1,\ldots,t_n)$ with $c \in \mathcal{C}$, t is p-free iff $[\![t]\!] \cap [\![p]\!] = \emptyset$ and $\forall i \in [1,n]$, t_i is p-free.*

The extended semantics of annotated terms together with the corresponding transformation of complement patterns (corresponding to semantics intersections) into sets of constructor patterns over-approximating their semantics allows us to systematically check if a term is pattern-free. Unfortunately equivalent extended semantics do not guarantee the preservation of pattern-freeness: having $[\![u]\!] = [\![v]\!]$ and u p-free does not necessarily mean that v is p-free.

We introduce the notion of *deep semantics* of a term $t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, denoted $\{\!|t|\!\}$, which can be seen as an over-approximation of the set of all subterms of all values of its ground semantics, *i.e.* $\{u_{|\omega} \mid u \in [\![t]\!], \omega \in \mathcal{Pos}(u)\} \subseteq \{\!|t|\!\}$:

Definition 4.2. *Given the sorts $s_1,\ldots,s_n,s \in \mathcal{S}$ and a pattern $p \in \mathcal{T}(\mathcal{C},\mathcal{X}) \cup \{\bot\}$*

- $\{\!|x^s_{-p}|\!\} = \bigcup_{s' \in \lfloor s \rfloor} [\![x^{s'}_{-p}]\!]$;
- $\{\!|f^{-p}(t_1,\ldots,t_n)|\!\} = \bigcup_{s' \in \lfloor s \rfloor} [\![x^{s'}_{-p}]\!]$ *with* $f : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{D}$;
- $\{\!|c(t_1,\ldots,t_n)|\!\} = [\![c(t_1,\ldots,t_n)]\!] \cup \left(\bigcup_{i=1}^n \{\!|t_i|\!\}\right)$ *with* $c : s_1 \times \ldots \times s_n \mapsto s \in \mathcal{C}$.

The deep semantics of a variable must not only take into account its own sort, but also the sorts of all the subterms of its instances. Similarly, for terms headed by a constructor we consider the semantics of terms and of all its subterms.

We can now identify terms that have such a shape that a given pattern p does not appear in any of their ground instances. In other words, we have an alternative method for establishing pattern-free properties for linear terms in $\mathcal{T}(\mathcal{F},\mathcal{X})$:

Proposition 3 (Pattern-free vs Deep Semantics). *Let $p \in \mathcal{T}(\mathcal{C},\mathcal{X}), t \in \mathcal{T}(\mathcal{F},\mathcal{X})$, if $\{\!|t|\!\} \cap [\![p]\!] = \emptyset$ then t is p-free.*

Example 4.2. *Let's consider the signature from Example 4.1: We add defined symbols* $\mathcal{D} = \{f : s_1 \mapsto s_1, g : s_2 \mapsto s_2\}$ *such that* f *is supposed to eliminate the* $p_1 = c_1(c_4(x), y)$ *and* g *eliminates* $p_2 = c_4(x)$*. We therefore have the following constructor TRS:*

$$\begin{aligned} f(c_1(c_4(x), y)) &\twoheadrightarrow c_1(g(x), f(y)) \\ f(c_1(c_5(x), y)) &\twoheadrightarrow c_1(c_5(f(x)), f(y)) \\ f(c_2(c_4(x))) &\twoheadrightarrow c_2(c_4(g(x))) \\ f(c_2(c_5(x))) &\twoheadrightarrow c_2(c_5(f(x))) \\ f(c_3()) &\twoheadrightarrow c_3() \end{aligned} \qquad \begin{aligned} g(c_4(x)) &\twoheadrightarrow c_5(c_1(g(x), c_3())) \\ g(c_5(c_1(x, y))) &\twoheadrightarrow c_5(c_1(g(x), g(c_5(y)))) \\ g(c_5(c_2(x))) &\twoheadrightarrow c_5(c_2(g(x))) \\ g(c_5(c_3())) &\twoheadrightarrow c_5(c_3()) \end{aligned}$$

Let's consider the term $c_1(g^{-p_2}(x), f^{-p_1}(y))$*.* $f^{-p_1}(y)$ *and* $g^{-p_2}(x)$ *have respectively the same semantics has* $y^{s_1}_{-p_1}$ *and* $x^{s_2}_{-p_2}$*. Therefore the deep semantics of the whole term is the union of the ground semantics of* $c_1(x^{s_2}_{-p_2}, y^{s_1}_{-p_1})$*,* $y^{s_1}_{-p_1}$*,* $x^{s_2}_{-p_1}$*,* $y^{s_1}_{-p_2}$ *and* $x^{s_2}_{-p_2}$*. Similarily as in Example 4.1, we can thus compare* $c_1(x^{s_2}_{-p_2}, y^{s_1}_{-p_1})$ *to* p_1 *and prove that it is* p_1*-free.*

5 Semantics preservation for CTRS

Pattern-freeness properties rely on the symbol annotations and assume thus a specific shape for the normal forms of reducible terms. This assumption should be checked by verifying that the CTRSs defining the annotated symbols are consistent with these annotations, *i.e.* verifying that the semantics is preserved by reduction.

Definition 5.1 (Semantics preservation)**.** *A constructor rewrite rule* $\varphi^{-p}(l_1, \ldots, l_n) \twoheadrightarrow r$ *is* semantics preserving *w.r.t. the extended semantics, resp. deep semantics, iff* $[\![r]\!] \subseteq [\![l]\!]$*, resp.* $\{\!\{r\}\!\} \subseteq \{\!\{l\}\!\}$*. A CTRS is semantics preserving iff all its rewrites rules are.*

Since the semantics of the left-hand side of a rewrite rule is the set of p-free values then such a rule is semantics preserving iff its right-hand side is p-free. It is easy to check that:

Proposition 4 (Semantics preservation)**.** *Given a semantics preserving CTRS* $\mathcal{R}$ *we have that then,* $\forall t, u \in \mathcal{T}(\mathcal{F})$*:* $t \longrightarrow_{\mathcal{R}} u \implies [\![u]\!] \subseteq [\![t]\!]$

Example 5.1. *Let's consider the case presented in Example 4.2:*

We want each of the rules of the constructor TRS to be pattern-free preserving, so that for all $v_1 \in \mathcal{T}_{s_1}(\mathcal{C})$*,* $f(v_1)$ *is and stays* $c_1(c_4(x), y)$*-free through every reduction step, and for all* $v_2 \in \mathcal{T}_{s_2}(\mathcal{C})$*,* $g(v_2)$ *is and stays* $c_4(x)$*-free through every reduction step.*

In terms of pattern-free preservation we have:

Proposition 5 (Pattern-free preservation)**.** *Given a semantics preserving CTRS* $\mathcal{R}$ *we have that* $\forall t, u \in \mathcal{T}(\mathcal{F}), p \in \mathcal{T}(\mathcal{C}, \mathcal{X})$*:* t *p-free* $\wedge\, t \longrightarrow_{\mathcal{R}} u \implies u$ *p-free*

6 Conclusion and perspectives

We have proposed a method to statically analyse constructor term rewrite systems and verify the absence of patterns from the corresponding normal forms. We can thus guarantee not only that some constructors are not present in the normal forms but we can also be more specific and verify that more complex constructs cannot be retrieved in the result of the reduction.

We suppose the existence of normal forms but the formalism does not rely on the termination of the analysed rewriting systems; if the property is not verified a final value is not obtained but the intermediate terms in the infinite reduction verify nevertheless the pattern-freeness properties *w.r.t.* the specified annotations. Moreover, different termination techniques and

tools [8, 10] on termination analysis can be used to analyse the termination of the rewriting systems we addressed in this paper.

We believe this formalism opens a lot of opportunities for further developments. In particular, this method could be extended in the context of automatic rewriting rule generation techniques, such as the one introduced in [5], in order to implement transformation approaches of passes such as in [9]. Indeed, the formalism considered here relies on the same pattern matching primitives as these techniques.

References

[1] Rajeev Alur and Pavol Černý. Streaming transducers for algorithmic verification of single-pass list-processing programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 599–610. ACM, 2011.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[3] Patrick Bahr and Laurence E. Day. Programming macro tree transducers. In *Proceedings of the 9th ACM SIGPLAN Workshop on Generic Programming*, WGP '13, pages 61–72. ACM, 2013.

[4] Françoise Bellegarde. Program transformation and rewriting. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, pages 226–239, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[5] Horatiu Cirstea, Sergueï Lenglet, and Pierre-Etienne Moreau. A faithful encoding of programmable strategies into term rewriting systems. In *RTA 2015*, volume 36 of *LIPIcs*, pages 74–88, 2015.

[6] Horatiu Cirstea and Pierre-Etienne Moreau. Generic encodings of constructor rewriting systems. arxiv:1905.06233, arXiv, 2019.

[7] Jacques Garrigue. Programming with polymorphic variants. In *In ACM Workshop on ML*, 1998.

[8] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM Trans. Program. Lang. Syst.*, 33(2):7, 2011.

[9] Andrew W. Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 343–350. ACM, 2013.

[10] Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle functions via termination of rewriting. In *ITP 2011*, pages 152–167, 2011.

[11] David Lacey and Oege de Moor. Imperative program transformation by rewriting. In *Proceedings of the 10th International Conference on Compiler Construction*, CC '01, pages 52–68, London, UK, UK, 2001. Springer-Verlag.

[12] François Pottier. Visitors unchained. *Proceedings of the ACM on Programming Languages*, 1(ICFP):28:1–28:28, 2017.

[13] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[14] Terese. *Term Rewriting Systems.* Cambridge University Press, 2003. M. Bezem, J. W. Klop and R. de Vrijer, eds.

Bad Smells d'Expressivité dans les Métamodèles

Elyes Cherfa[1,2], Salah Sadou[1], Soraya Kesraoui[2], Chouki Tibermacine[3], and Régis Fleurquin[1]

[1] Université Bretagne Sud, Vannes, France
prénom.nom@univ-ubs.fr
[2] Segula technologies, Lorient, France
soraya.kesraoui@segula.fr
[3] LIRMM, Université de Montpelier, Montpelier, France
chouki.tibermacine@lirmm.fr

Abstract

1 Introduction

Dans l'ingénierie dirigée par des modèles (IDM), les métamodèles représentent la syntaxe abstraite des langages de modélisation spécifiques au domaine (DSML) [11]. La structure du métamodèle capture les concepts de base du domaine, ainsi que les relations qu'il existe entre les concepts. Considérant la difficulté ou l'impossibilité d'exprimer certaines informations à travers des diagrammes, des contraintes textuelles, autrement appelées Well-Formedness Rules (WFR) exprimées généralement avec le langage OCL sont spécifiées dans le métamodèle pour restreindre la portée de certains concepts. En conséquent, la puissance de l'ingénierie dirigée par des modèle est pleinement exploitée seulement si les métamodèles sont suffisamment précis pour capturer pleinement les parties syntaxiques et sémantiques du domaine représenté.

Malheureusement, la majorité des métamodèles présents dans les répertoires (comme l'OMG [8]) incluent la description de la partie structurelle seulement. Les contraintes OCL sont rarement incluses, et parfois, spécifiées grossièrement et donc n'empêchent pas toutes les ambiguïtés sémantiques. Ceci est dû au fait que l'élicitation des contraintes OCL est réalisée manuellement, ce qui prend un temps conséquent et qui est sujet aux erreurs. Simplifier la procédure de spécification des contraintes OCL a été le centre d'attention de plusieurs travaux de recherche. Plusieurs approches ont été explorées, en particulier l'utilisation de la structure du métamodèle et un ensemble d'exemples de modèles pour inférer les contraintes OCL ([7, 5]). D'autres approches ont ciblées directement le langage OCL dans le but d'identifier les patterns de contraintes qui permettent de spécifier la majorité des contraintes OCL ([12, 3, 2, 4, 10, 6]).

Nous avons pu constater que certaines contraintes OCL sont récurrentes quel que soit le domaine représenté par le métamodèle. Nous pensons que la présence de certaines structures implique la spécification de ces contraintes. Par conséquent, ce travail a pour but de faire une enquête sur le rôle de ces structures sur l'existence de certaines contraintes. Notre objectif est de caractériser les structures qu'on complète avec des contraintes OCL. Pour cela, nous avons étudié des métamodèles ayant déjà des contraintes OCL. Nous analysons manuellement chaque contrainte avec la structure qu'elle cible pour identifier les structures qui causent des inconsistances, et qui ont conduit à la définition de contraintes.

Il est important de souligner que selon le domaine, la présence de ces structures n'implique pas automatiquement des inconsistances, leurs occurrences doivent être inspectés par le concepteur du métamodèle pour vérifier si ça cause réellement des inconsistances, d'où le nom "bad smells d'expressivité de Métamodèles" (BSEM). Nous estimons que UML est le métamodèle

le plus approprié sur lequel nous pouvons nous appuyer dans notre étude pour identifier les BSEM. Afin de valider les résultats, nous analysons d'autres métamodèles issus de diverses sources (SysML, CWM, ER2RE, RBAC, ...) dans le but de confirmer si les BSEM identifiés sont purement liés à UML ou bien génériques et souvent corrigés avec des contraintes OCL.

Dans ce qui suit, nous présentons la liste de BSE dans la section 2. Avant de conclure en section 4, nous présentons les résultats de l'analyse des autres métamodèles dans la section 3

2 Caractérisation des BSEM

Nous sommes convaincus que le meilleur moyen de caractériser des BSEM est d'analyser des métamodèles déjà existants ayant des contraintes OCL. En analysant dans quelle partie du métamodèle la contrainte a été ajoutée, et pourquoi a-t-elle été spécifiée, nous serons en mesure de conclure si la structure représente un BSEM ou pas.

2.1 Le métamodèle utilisé

Étant donné qu'il faut s'appuyer sur des métamodèles déjà existant pour caractériser les BSEM, leur choix est très important pour assurer la qualité et la complétude des BSEM, surtout l'ensemble des contraintes OCL qui doit être le plus complet possible. Nous sommes convaincus que UML est le métamodèle le plus approprié à cette étude. D'une part, UML 2.5 [9] regroupe plus de 400 contraintes, ce qui est une bonne base d'étude. D'autre part, étant l'un des plus grands métamodèles dans la littérature, nous estimons qu'il est plus probable que ce dernier englobe diverses formations structurelles.

2.2 BSEMs caractérisés

Dans ce qui suit, nous allons présenter les BSEM retrouvés. Par manque de place, seuls trois des neuf BSEM sont présentés. Nous rappelons qu'un fragment de métamodèle est soupçonné d'être un BSEM si l'expert est amené à poser une question à ce sujet. Ainsi, pour chaque BSEM identifié, nous donnons la question formulée, ainsi qu'un exemple d'UML.

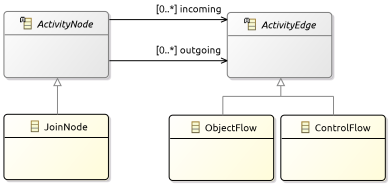


Figure 1: Restriction de la valeur d'attribut

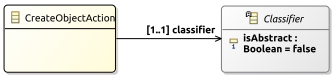


Figure 2: Restriction de la multiplicité d'une association héritée

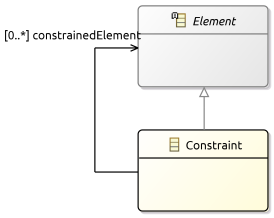


Figure 3: Cycle non spécifié

2.2.1 L'héritage d'associations

La généralisation permet de regrouper des éléments communs (association, attribut, opération) depuis différentes métaclasses, et de les définir dans une seule classe. En conséquent, les classes

ayant besoin de ces propriétés héritent de la nouvelle classe sans avoir à redéfinir localement les propriétés qu'elle contient. La généralisation permet de simplifier la structure du métamodèle et d'éviter les redondances. Cependant, la fusion des propriétés de différentes classes en une seule classe mène souvent à une perte de précision. Ceci est dû au fait que lors de la fusion d'une propriété ayant été dans plusieurs classes avec des multiplicités différentes, la plus grande multiplicité est choisie afin d'inclure toutes les autres multiplicités, ce qui rend la propriété incohérente par rapport au domaine dans certains cas.

La question relative à ce BSEM est la suivante : est-ce que les sous-métaclasses héritent une association avec la même multiplicité spécifiée dans la super-métaclasse ?

Comme illustré dans la Figure 1 prise du diagramme d'activité d'UML, les sous-métaclasses de ActivityNode héritent l'association *incoming* avec la multiplicité [0..*]. Cependant, un nœud d'union (ForkNode) ne peut avoir plus d'une entrée (*incoming*). La contrainte OCL suivante restreint la multiplicité de *incoming* dans la classe ForkNode à 1.

context ForkNode **inv** :
incoming $->$ size() $= 1$

2.2.2 Restriction de la valeur d'un attribut

Dans MOF, il n'y a aucun moyen de spécifier les valeurs précises qu'un attribut peut prendre. En conséquence, un attribut peut prendre n'importe quelle valeur du domaine par défaut (i.e. un entier peut prendre une valeur dans $\mathbb{N}$). Cependant, certaines valeurs sont incorrectes en se basant sur la sémantique du domaine. Le seul moyen de restreindre le domaine d'un attribut est d'écrire des contraintes pour spécifier le champ de valeurs correct, et ainsi exclure toute valeur insignifiante dans le domaine. Il est important de souligner que la restriction peut s'appliquer sur un sous ensemble d'instances et n'est pas appliquée automatiquement sur toutes les instances de l'attribut. La question qui se pose pour ce BSEM est : est-ce que le domaine de définition par défaut de l'attribut correspond à celui dicté par le domaine ?

Comme le montre la Figure 2, un *Classifier* contient un attribut *isAbstract* qui indique si un classifieur est abstrait ou pas. Selon la sémantique d'UML, les classifieurs qui sont associés à *CreateObjectAction* doivent toujours être abstraits. Ainsi, la contrainte suivante est spécifiée.

context CreateObjectAction **inv** :
not classifier.isAbstract

2.2.3 Les cycles

Dans le contexte de métamodèles, un cycle est une succession d'associations et d'opérations qui relie la classe à elle-même. Certains cycles plus subtils peuvent être présents, qui relient une classe à sa superclasse par le biais de navigations. La présence de cycles permet non seulement les auto-associations, mais aussi les configurations en diamant [13]. Les concepteurs de modèles doivent être conscients de cela et doivent analyser si ces associations sont valides vis-à-vis du domaine applicatif représenté par le métamodèle.

La question qui se pose en cas de cycle est la suivante : est-ce qu'une instance de cette métaclasse peut être associée à elle-même ?

Par exemple, dans la Figure 3, une contrainte est appliquée à un élément (Element). En conséquence, étant donné qu'une contrainte est un élément et hérite de la métaclasse *Element*, il est possible d'appliquer une contrainte à elle-même, ce qui ne peut pas être fait dans UML. Dans ce cas, il est nécessaire de spécifier une contrainte OCL qui restreint cette association pour la classe *Constraint*. [9].

context Constraint **inv** :
not constrainedElement − > includes(self)

3 Investigation sur l'impact des BSEM

Après avoir caractérisé un ensemble de BSEM à partir du métamodèle UML, nous devons nous assurer que les BSEM caractérisés dans UML sont également présents dans d'autres métamodèles et, surtout, raffinés avec des contraintes OCL pour compléter leur sémantique. Pour ce faire, nous évaluons l'importance des BSEM du point de vue quantitatif, et ce en analysant les contraintes OCL d'autres métamodèles et en calculant les occurrences des contraintes complétant les BSEM.

Nous avons choisi un ensemble de métamodèles ayant des contraintes OCL, à savoir SysML, ODM, CWM, DD, SAD3, CPFSTool, ER2RE, RBAC, SAM. Ces derniers sont présents dans la bibliothèque d'artefacts [1].

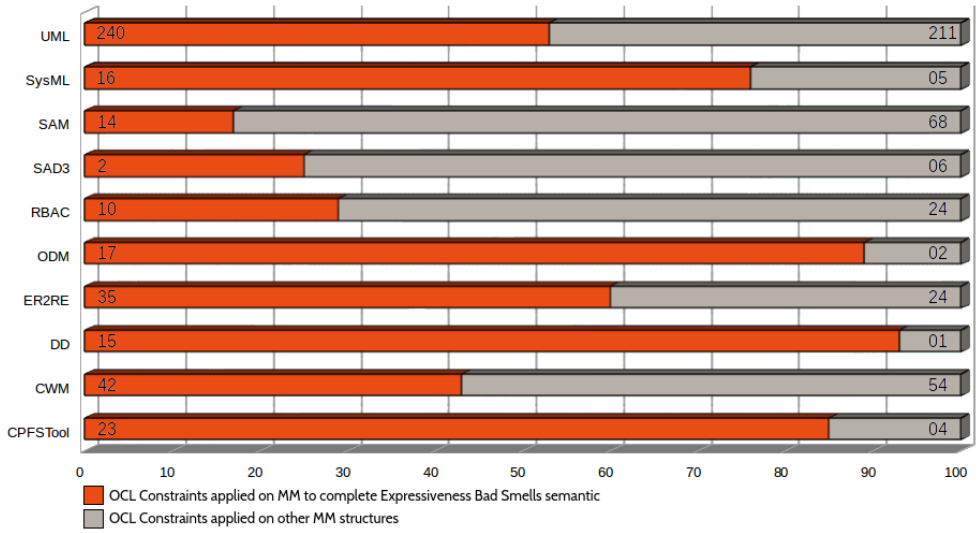


Figure 4: Le taux de contraintes spécifiées pour les BSEM

La Figure 4 représente le taux de contraintes qui complètent les BSEM. Nous pouvons constater que le taux de contraintes liées au BSEM varie d'un métamodèle à l'autre. Par exemple, dans le métamodèle DD, les contraintes liées à BSEM dépassent 90 % de l'ensemble des contraintes spécifiées pour ce métamodèle, alors que sur le métamodèle SAM, elles ne représentent que 17 % du nombre total de contraintes OCL spécifiées. En moyenne, le taux de contraintes liées à MEBS est d'environ 52 % du nombre total des contraintes spécifiées dans les métamodèles étudiés. Ces résultats prouvent que l'impact de la présence de ces BSEM est réel sur la sémantique du métamodèle, et qu'il est nécessaire d'analyser les occurrences de ces BSEM dans les métamodèles lors de la phase de spécification des contraintes OCL afin de compléter leur sémantique avec des contraintes OCL.

4 Conclusion

Dans ce papier, des bad smells d'expressivité dans les métamodèles ont été caractérisés. Ces derniers sont des structures à qui on applique souvent des contraintes OCL parce qu'ils ne sont pas suffisamment expressifs pour capturer pleinement et avec précision la sémantique du domaine. En premier lieu, une analyse a été effectuée sur le métamodèle UML et ces contraintes OCL dans le but d'identifier les structures manquant d'expressivité, ceci étant fait par des experts qui ont étudié chaque contrainte avec la structure qu'elle cible dans le métamodèle. Par conséquent, un ensemble de BSEM a été proposé. Ensuite, afin de s'assurer que ces BSEM ne sont pas propres à UML et peuvent être retrouvés sur d'autres métamodèles, d'autres métamodèles ayant des contraintes OCL ont été collectés et une analyse visant à quantifier les contraintes OCL qui complètent chaque BSEM a été effectuée. Nous estimons que ce travail permet d'aider les concepteurs de métamodèles lors de la création ou le refactoring de métamodèle à détecter les structures qui manquent d'expressivité et qui peuvent engendrer des inconsistances dans les artefacts générés si elles ne sont pas accompagnées de contraintes OCL qui spécifient précisément leur sémantique. Comme travaux futurs, nous souhaitons créer un outil qui permet de rechercher automatiquement les occurrences de chaque BSEM dans les métamodèles, et qui propose des contraintes OCL en fonction du BSEM recherché pour compléter la sémantique.

References

[1] The repository for model-driven development. http://remodd.org/.

[2] Juan Cadavid, Benoit Combemale, and Benoit Baudry. *Ten years of Meta-Object Facility: an analysis of metamodeling practices*. PhD thesis, INRIA, 2012.

[3] Dan Chiorean, Vladiela Petraşcu, and Ileana Ober. Testing-oriented improvements of ocl specification patterns. In *2010 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, volume 2, pages 1–6. IEEE, 2010.

[4] Dan Chiorean, Vladiela Petrascu, and Ileana Ober. Mde-driven ocl specification patterns. *Journal of Control Engineering and Applied Informatics*, 14(1):83–92, 2012.

[5] Duc-Hanh Dang and Jordi Cabot. Automating inference of ocl business rules from user scenarios. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 156–163. IEEE, 2013.

[6] Hammad et AL. Iocl: An interactive tool for specifying, validating and evaluating ocl constraints. *Science of Computer Programming*, 2017.

[7] Martin Faunes, Juan Cadavid, Benoit Baudry, Houari Sahraoui, and Benoit Combemale. Automatically searching for metamodel well-formedness rules in examples and counter-examples. In *MODELS*, pages 187–202. Springer, 2013.

[8] Object Management Group. https://www.omg.org/.

[9] Object Management Group. Uml 2.5. https://www.omg.org/spec/UML/2.5/, 2015.

[10] Ali Hamie. Constraint specifications using patterns in ocl. *International Journal on Computer Science and Information Systems*, 8(1), 2013.

[11] Anneke Kleppe. *Software language engineering: creating domain-specific languages using metamodels*. Pearson Education, 2008.

[12] Michael Wahler, Jana Koehler, and Achim D Brucker. Model-driven constraint engineering. *Electronic Communications of the EASST*, 5, 2007.

[13] Michael S Wahler. *Using patterns to develop consistent design constraints*. PhD thesis, ETH Zurich, 2008.

Session: Développement de Systèmes

Le Développement de Systèmes de Systèmes Sécurisé Nécessitant un Déploiement Rapide

Nan Messe, Nicolas Belloir, Vanea Chiprianov, Imane Cherfa, Régis Fleurquin, and Salah Sadou

Université Bretagne Sud - IRISA, France
firstname.lastname@irisa.fr

1 Introduction

Le domaine des systèmes de systèmes (SoS) traite de la mise en commun de systèmes plus ou moins complexes, indépendants managérialement et opérationnellement, évolutifs, de manière à offrir un comportement qui ne peut être obtenu par les systèmes constituants de manière indépendante [10]. Dans ce travail, nous traitons d'un type particulier de SoS : les SoS nécessitant un déploiement rapide (SoSnRD) comme une opération de sauvetage après un tremblement de terre ou une opération militaire. Ce type de système doit être mis en place le plus rapidement possible (en quelques heures ou quelques jours) pour répondre à une situation d'urgence. Ces SoS usent de plus en plus d'objets connectés (IoT), comme des véhicules autonomes sans pilote (UAV) ou des capteurs médicaux intelligents. Ils manipulent également davantage d'informations souvent sensibles. Pour construire ces SoSnRD, on ne peut user des méthodes traditionnelles de développement de SoS, comme NAF [13], COMPASS [14] ou DANSE [9]. Car elles supposent généralement : i) avoir du temps (plusieurs mois ou années) et ii) avoir un nombre important d'acteurs pouvant être mobilisés. A l'inverse le développement d'un SoSnRD se fait sur un temps court et souvent sous la responsabilité d'une seule personne : un expert de domaine métier. Cet expert a une expérience significative dans la mise en place de SoS dans son domaine d'intervention. Il est ainsi capable d'imaginer très rapidement une solution. Il adapte pour cela une solution générique et éprouvée au contexte particulier de la mission qui lui est soumise. Ce type de développement entre dans le champs de l'ingénierie des missions et offre une vue plutôt opérationnelle.

Comme les SoSnRD peuvent être déployés dans des environnements hostiles, ils sont exposés aux cyber-attaques. Pour préserver l'information sensible, les biens et les personnes il est donc essentiel de prendre en compte l'aspect sécurité [11] et cela dès la conception. L'expert du domaine métier ne peut s'appuyer sur les méthodes de sécurisation existantes telles que Microsoft SDL [8], OWASP [15], Secure i* [6] ou SysML-Sec [4], car elles ne sont pas adaptées aux SoS. Les méthodes dédiées aux SoS, comme SoSSec [7] exigent également beaucoup de temps et nécessitent de solides compétences en sécurité. Il n'est pas réaliste de s'attendre à ce que cet expert, qui a déjà fait l'effort de maintenir continuellement son domaine d'expertise à jour, devienne en outre un expert en sécurité. Nous proposons dans cet article une vision prospective sur le développement de SoSnRD intégrant la préoccupation de la sécurité. Pour cela, nous définirons une assistance logicielle basée sur les modèles qui permettra à un expert du domaine métier de documenter et d'intégrer la préoccupation sécurité lors du processus de conception de l'architecture du SoSnRD. Le challenge consiste à établir un pont entre le modèle de domaine de l'expert et celui de la sécurité. Nous définissons un modèle intermédiaire basé sur le concept 'bien' (*asset* en anglais). Nous montrerons que les 'biens' peuvent aider à combler le fossé entre les compétences de l'expert du domaine métier et la préoccupations de la sécurité.

2 Approche

2.1 Développement rapide de SoS

Pour relever le défi du développement rapide, nous proposons d'utiliser deux principes du génie logiciel : i) promouvoir la réutilisation autant que possible, ii) automatiser au maximum les tâches de développement.

Pour faciliter la réutilisation, nous proposons de mêler deux techniques : les langages de modélisation dédiés (DSML) et la généricité. Le DSML devra permettre de décrire aussi bien des architectures abstraites que concrètes. Il intégrera dans sa syntaxe les concepts liés aux systèmes constituants du domaine métier et à leurs différents types de relations. Les experts devront pouvoir capitaliser leurs connaissances sous forme de modèles d'instances réutilisables. Le DSML doit donc offrir des mécanismes de points de variation tels que ceux que l'on trouve dans la littérature de lignes de produits logiciels ou dans la littérature de l'expression de patrons architecturaux. Comme il n'est pas réaliste de définir à partir de zéro autant de DSML (et d'outils) que de domaines métiers, nous proposons d'utiliser l'ingénierie dirigée par les modèles. Nous définirons un langage de modélisation de référence que nous spécialiserons en autant de 'profils' que de domaines métiers. Nous travaillons actuellement sur ce langage. Il est basé sur les concepts de 'rôle' (un système constituant abstrait ou concret comme l'humain, le logiciel ou le matériel) offrant une certaine 'capacité' (services) à d'autres rôles lorsqu'ils sont liés par une relation de 'collaboration' [5].

Pour automatiser le développement, nous prônons l'usage d'outils logiciels, chacun dédié à un expert. Ces outils seront dérivés de l'outil construit pour supporter le langage de référence. Cela consiste à : i) adapter le langage de référence au domaine métier de l'expert (en utilisant des stéréotypes), ii) choisir une syntaxe concrète adaptée au domaine métier (associer une représentation graphique à un stéréotype). L'outil obtenu permettra à l'expert d'éditer, stocker, rechercher, réutiliser, généraliser (créer un patron architectural ou une architecture de référence), adapter ou spécialiser (appliquer un patron ou configurer une architecture de référence) des modèles conformes à son DSML. Une bibliothèque de rôles sera gérée par l'expert. L'expert pioche dans cette bibliothèque pour concevoir ses modèles. Il y a donc deux bases de connaissances maintenues par l'outil qui facilitent la réutilisation : la première stocke les modèles d'architecture (générique ou non) tandis que la seconde stocke une hiérarchie de rôles.

La résolution de ce premier défi n'est pas l'objet de cet article, nous ne donnerons pas ici plus de détails sur ce travail en cours. Dans ce qui suit, nous ferons l'hypothèse que l'expert dispose d'un DSML adapté à son domaine métier.

2.2 SoSnRD sécurisé

Pour relever ce second défi, nous proposons de définir une assistance supportée par un outil permettant à l'expert de domaine métier d'évaluer et de gérer les exigences de sécurité lors de la conception de l'architecture. Cette assistance suppose que les experts ont peu de compétences en sécurité. Ils doivent cependant au moins pouvoir : i) distinguer les éléments qu'ils veulent protéger sur le modèle d'architecture et ii) être conscients des propriétés de la sécurité : confidentialité, intégrité, disponibilité, authenticité, non-répudiation, fiabilité et responsabilité [2]. Sur la base de ces informations et d'une base de connaissances en sécurité, notre outil analyse les risques et fournit des conseils pour diminuer les risques. Cette assistance se décline en trois niveaux : 1) lever des alertes (liste des vulnérabilités détectées et des risques associés), 2) proposer des modifications locales de l'architecture pour limiter les risques (configuration ou

remplacement de certains systèmes constituants) ou 3) modifications globales (application de modèles de sécurité). L'expert peut demander à tout moment le lancement de l'assistance. Cependant, le niveau de détail et la pertinence des conseils seront meilleurs si le modèle est concret et utilisera donc de véritables systèmes constituants, dont les vulnérabilités sont répertoriées dans la base de connaissances de la sécurité. Par conséquent, le processus sera itératif, ce qui permettra à l'expert d'intégrer les conseils de l'assistance.

2.3 Vision globale de l'assistance orientée sécurité

Une méthode traditionnelle d'analyse des risques de sécurité commence par l'énumération de tous les biens d'une organisation, puis par l'identification des menaces, puis par l'identification des vulnérabilités exploitables par ces menaces et enfin se termine par l'estimation des risques. Plusieurs définitions du terme 'bien' (asset en anglais) existent dans la littérature [12], [3]. ISO/CEI définit qu'un bien est tout ce qui a de la valeur pour une organisation [1]. Dans notre vision, un bien est : tout ce qu'un expert de domaine métier ou un expert en sécurité veut protéger contre des attaques potentielles. Les experts du domaine métier et les experts en sécurité ont des perspectives ou des visions différentes sur les biens.

Nous montrons notre vision pour combler le fossé entre l'expert de domaine métier et les préoccupations en sécurité dans la Fig. 1. L'axe horizontal illustre trois mondes : domaine métier, bien et sécurité. Le monde du bien est le lien entre le monde du domaine métier et le monde de la sécurité. L'axe vertical montre trois niveaux d'abstraction différents : l'exigence, la conception et l'implémentation. La partie bleue indique la vision de l'expert du domaine métier tandis que la partie verte indique les préoccupations en sécurité. Dans notre vision, nous faisons la promotion de trois types de biens : métiers, pivots et de support. Les biens pivots sont introduits pour faciliter la transition entre les biens métiers et les biens de support.

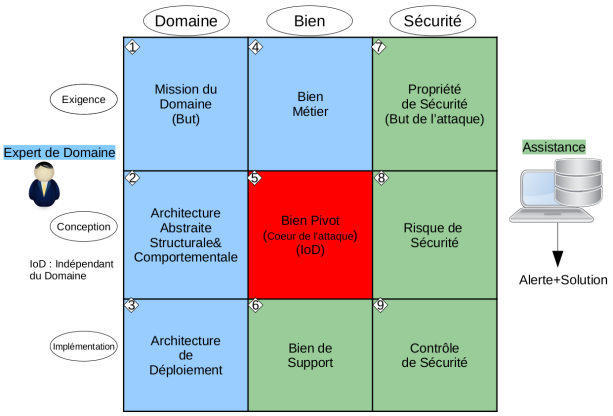


FIGURE 1 – La Vision Globale

Les experts de domaine métier sont prêts à protéger les propriétés financières, les données confidentielles ou la réputation de leur organisation. Nous les appelons les 'bien métier' (*business asset* en anglais) parce qu'ils ont une valeur pour l'expert de domaine métier. Le bien métier, que l'expert du domaine métier veut protéger, est taggé par l'expert de domaine métier sur un modèle d'architecture pour exprimer les exigences de la sécurité. Le bien métier est l'objectif d'une attaque. N'étant pas des experts en sécurité, les experts de domaine métier ne savent pas comment exactement concevoir les architectures pour protéger ces biens. Les biens métiers

sont le point de départ de toute politique de sécurité. Ils délimitent également la limite de compétence de l'expert du domaine métier. Par exemple, un expert du domaine veut protéger un plan militaire. Il le définit en tant que bien métier et précise qu'il veut assurer la *confidentialité* de cette *donnée*.

Le bien pivot (*pivot asset* en anglais) est le cœur d'une attaque, il est l'intention d'une attaque (contraire à l'action ou mécanisme) pour atteindre l'objectif final. C'est un élément essentiel, sensible et intangible supporté par le bien de support. Il représente l'axe que les attaquants ciblent. Ces axes sont indépendants des domaines et des plateformes. On évalue son niveau de sensibilité ou de criticité, mais pas sa vulnérabilité, car il ne présente aucune vulnérabilité. Prenons par exemple notre plan militaire. Il est stocké dans une base de données. Vu les propriétés *confidentialité* et *donnée* spécifiées sur le bien métier, le bien pivot est de type *information*. Il existe d'autres types de bien pivot : *ressource*, *identité*, *fonction*, *communication*, *accès*, etc. Ils appartiennent à trois familles : *données*, *processus*, et *biens reliés à l'humain* .

Un expert en sécurité considère la protection des biens métiers comme une exigence de sécurité. Leur connaissance des mécanismes d'attaque leur permet d'identifier d'autres éléments architecturaux qui jouent un rôle clé dans l'obtention du niveau de sécurité souhaité. Nous appelons ces éléments les 'biens de support' (*supporting asset* en anglais), tels que la base de données, le serveur, etc. Le bien de support fournit un support pour le bien pivot. Il est le point d'entrée ou la source d'une action d'attaque. C'est un élément (instance) tangible. En exploitant les vulnérabilités qui se retrouvent dans ce bien de support, l'attaquant peut compromettre le bien pivot. On identifie les vulnérabilités mais pas la sensibilité ou la criticité. Dans l'exemple ci-dessus, la base de données contenant le plan est le bien de support qui supporte le bien pivot 'information'. Une attaque réussie n'est accessible qu'en attaquant la combinaison du bien pivot et du bien de support. L'expert en sécurité s'appuie sur des connaissances appelées 'les chaînes de causalités' : ce sont des chemins de propagation typiques des attaques dans le modèle d'architecture, décrites par exemple à l'aide d'arbres d'attaques. Par exemple, les données sur la santé d'un patient doivent être confidentielles. Un attaquant peut avoir plus d'une façon d'obtenir cette information. Chaque chemin implique une chaîne causale de biens. Un scénario d'attaque possible est le suivant : les attaquants se connectent à distance au réseau via un logiciel malveillant. Le logiciel malveillant se propagera alors (parfois même directement à partir d'un point d'accès physique), jusqu'au poste de travail où une base de données contenant des informations personnelles est installée. La chaîne de causalité correspondante est la suivante : information ← base de données ← mécanismes de protection ← poste de travail ← réseau ← point d'accès physique (objectif final : les données du patient).

L'assistance est basée sur une base de connaissances comprenant : i) une table de correspondance pour identifier les biens pivots (elle fait le lien entre les propriétés de sécurité (confidentialité ...) et ce sur quoi elles s'appliquent (donnée ...), ii) une table de correspondance associant les biens pivots aux catégories de risque, iii) une base de données décrivant pour chaque système constituant sa liste de vulnérabilités connues (si c'est un système concret) ou (si abstrait) les catégories de vulnérabilité, iv) des chaînes causales décrivant des trajectoires possibles pour les attaques. Cette base de connaissances est créée et maintenue par un expert en sécurité indépendamment des activités de l'expert du domaine métier.

3 Conclusion

Dans ce papier, nous avons introduit un type particulier de systèmes de systèmes : les systèmes de systèmes nécessitant un déploiement rapide qui sont parfois déployés dans des environnements où la sécurité est une préoccupation importante. Nous avons montré que leurs

développements nécessitent des méthodes et des outils spécifiques. Ces développements sont menés par des experts de domaine métier qui, dans la plupart des cas, ont peu de compétences en sécurité. Ainsi, nous avons présenté le principe d'une assistance capable de dériver automatiquement des alertes et des contrôles de sécurité à partir d'une liste de biens métiers spécifiés par des experts de domaine métier. Nous avons identifié les actions que l'expert de domaine métier doit mener et celles exécutées par le futur outil d'assistant de sécurité. Nous prévoyons maintenant d'utiliser des techniques d'ingénierie dirigées par les modèles, telles que les transformations de modèles, pour développer une première version de cet outil d'aide. Nous nous concentrerons d'abord sur la fourniture d'une assistance de niveau 1 (lever des alertes) et 2 (proposer de changements locaux). A l'avenir, le troisième niveau de l'aide (changements globaux proposés) devra être abordé.

Remerciement

Ce travail est financé par la Direction Générale de l'Armement et le Pôle d'Excellence Cyber.

Références

[1] ISO/IEC 13335-1 :2004. Information technology – security techniques – management of information and communications technology security – part 1 : Concepts and models for information and communications technology security management, 2004.

[2] ISO/IEC 27000 :2018. Information technology - security techniques - information security management systems - overview and vocabulary, 2018.

[3] ANSSI. Classification method and key measures, 2014.

[4] L. Apvrille and Y. Roudier. *Model-Driven Engineering and Software Development*, chapter Designing Safe and Secure Embedded and Cyber-Physical Systems with SysML-Sec. Springer, 2016.

[5] Imane Cherfa, Salah Sadou, Nicolas Belloir, Régis Fleurquin, and Djamal Bennouar. Involving the application domain expert in the construction of systems of systems. In *13th Annual Conference on System of Systems Engineering, SoSE, Paris, France, June 19-22, 2018*, pages 335–342, 2018.

[6] G. Elahi and E. Yu. A goal oriented approach for modeling and analyzing security trade-offs. In *Proc. of the 26th Int. Conf. on Conceptual Modeling*, pages 375–390, 2007.

[7] J. E. Hachem, Z. Y. Pang, V. Chiprianov, A. Babar, and P. Aniorte. Model driven software security architecture of systems-of-systems. In *23rd Asia-Pacific Software Engineering Conf*, 2016.

[8] M. Howard and S. Lipner. *The Security Development Lifecycle*. Microsoft Press, Redmond, WA, USA, 2006.

[9] B. Josko. Designing for adaptability and evolution in system of systems engineering, 2015.

[10] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4) :267–284, 1998.

[11] P. H. Nguyen, S. Ali, and T. Yue. Model-based security engineering for cyber-physical systems : A systematic mapping study. *Information and Software Technology*, 83 :116 – 135, 2017.

[12] NIST. Specification for asset identification 1.1. Technical Report NISTIR 7693, June 2011.

[13] North Atlantic Treaty Organization. Nato architecture framework, version 4, 2018.

[14] COMPASS Project. Accident response use case engineering analysis report using current methods & tools. Technical Report D41.1, COMPASS Project, www.compass-research.eu/Project/Deliverables/D411.pdf, 2013.

[15] The OWASP Foundation. OWASP Secure Software Development Lifecycle Project, 2017.

DEPS Studio :

Un environnement intégré de modélisation et de résolution de problèmes de conception de systèmes

Pierre-Alain Yvars[1], Laurent Zimmer[2]

[1] SupMéca, QUARTZ, Saint-Ouen, France
pierre-alain.yvars@supmeca.fr
[2] Dassault Aviation, Saint-Cloud, France
laurent.zimmer@dassault-aviation.com

Résumé

Nous présentons DEPS Studio un environnement intégré de modélisation et de résolution conçu pour spécifier et résoudre des problèmes de conception de systèmes. Cet environnement permet de décrire, mettre au point et résoudre des problèmes modélisés avec le langage DEPS. Celui-ci combine des traits de représentation des connaissances par objets et des traits de spécification de problème de la programmation par contraintes. Les problèmes de conception adressés par DEPS peuvent-être à dominante technique, matérielle, logicielle (systèmes embarqués) ou bien mixtes (systèmes cyber physiques). Après quelques rappels sur le langage de modélisation, nous présentons les différents éléments qui composent l'environnement intégré. Nous détaillons particulièrement l'organisation de la chaîne de compilation qui va de l'édition du modèle jusqu'à la génération du modèle de calcul ainsi que les principales caractéristiques du solveur de programmation par contraintes qui a été développé pour être intégré à cet environnement. Nous terminons par un point sur les développements en cours qui seront intégrés dans la prochaine version.

Mots-clefs : spécification, conception de système, synthèse de système, représentation des connaissances, programmation par contraintes

Abstract

We present DEPS Studio an integrated modeling environment developed for system design problem solving. This environment enables to describe, debug and solve problems which have been modeled with the DEPS language. The design problems addressed by DEPS can be engineering design, hardware or software design or a mix (Cyber Physical System design). DEPS (Design Problem Specification) combines the modeling capabilities of object oriented knowledge representation and the problem solving capabilities of constraint programming. After a survey of the main distinctive features of the language, we put the emphasis on the compiling process from the model edition to the computational model generation. Then we describe the main characteristics of the constraint solver we developed. Finally, we give some information about the current work that will be integrated in the next version of the tool.

Keywords: specification, system design, system synthesis, knowledge modeling, constraint programming

1 Introduction

Acronyme de « Design Problem Specification », le langage DEPS a été initialement développé pour modéliser des problèmes de conception de produits et plus récemment des problèmes de conception de systèmes [1]. L'outil DEPS Studio exploite donc ce langage pour spécifier et résoudre des problèmes de dimensionnement, configuration, déploiement, allocation, vérification ou synthèse de système. Les problèmes de conception adressés par DEPS peuvent-être à dominante technique, matérielle, logicielle (systèmes embarqués) ou bien mixtes (systèmes cyber physiques).
Le point commun de tous ces problèmes est que leur résolution revient à compléter une représentation sous-définie (au sens de sa structure) du système étudié pour obtenir le ou les seuls systèmes qui satisfont l'ensemble des propriétés qui les caractérisent. Ces dernières pouvant provenir soit d'exigences du cahier des charges soit de contraintes physiques ou technologiques.
Ce que nous résumerons par notre slogan :

Résoudre un problème de conception = Compléter un modèle sous-défini

Nous avons affaire à une problématique que nous qualifierons de synthèse par rapport à la problématique plus classique d'analyse. Dans la problématique d'analyse on exploite un modèle de définition (donc bien défini ou complet) du produit ou du système avec des outils d'évaluation qu'on appelle communément des simulateurs. Dans la problématique de synthèse on cherche à obtenir un modèle de définition avec un outil de synthèse.
Ainsi, DEPS Studio est un outil de synthèse en conception de produits ou de systèmes.

2 Le langage DEPS

2.1 Paradigme

DEPS [1, 2] est un langage de modélisation dédié ou DSML (Domain Specific Modeling Language) C'est une combinaison entre un langage de modélisation et un langage de programmation par contraintes. Aux premiers ont été empruntés les traits de structuration et d'abstraction qui permettent de représenter les composants et l'architecture (éventuellement partielle) du système étudié. Aux seconds ont été empruntés les concepts logico-mathématiques nécessaires à la résolution des problèmes de l'ingénieur. Cette combinaison de paradigmes déclaratifs donne un langage lui-même déclaratif qui permet de représenter à la fois l'organisation des systèmes étudiés ainsi que les propriétés qui les régissent [3].

2.2 Langage orienté objet

Le langage DEPS est un langage objet orienté classes. En DEPS une classe est appelée un modèle et une instance de classe un élément. Les classes peuvent être spécialisées par un mécanisme d'héritage.
Un modèle est constitué de constantes, de variables, d'éléments et de propriétés. Les propriétés sont des relations entre constantes et variables. Un modèle peut comporter des arguments qui sont soit des constantes soit des éléments. Passer des constantes en argument d'un modèle permet de le paramétrer. Passer des éléments en argument d'un modèle permet de définir une relation d'agrégation entre ceux-ci. Cela permet aussi de définir des propriétés entre leurs variables. Dans ce dernier cas un modèle peut représenter une relation générale entre des éléments. A noter que dans DEPS le polymorphisme des modèles tient compte du nombre et du type des arguments.

2.3 Programmation par contraintes et ontologie

DEPS emprunte les aspects déclaratifs du modèle de représentation des problèmes de satisfaction de contraintes (CSP) [4], le modèle <V, D, C> (Variables, Domaines, Contraintes) mais les adapte pour représenter spécifiquement les problèmes de l'ingénieur.
En DEPS, on distingue donc les constantes et les variables. Les deux ont un domaine de définition mais les premières ont une valeur fixée dans leur domaine tandis que les secondes ont un domaine de définition non réduit à une valeur qui caractérise le caractère sous-défini du modèle auquel elles appartiennent.
Les domaines sont soit discrets à valeurs entières ou réelles soit continus à valeurs réelles.
Les constantes et les variables sont elles aussi à valeurs entières ou réelles mais dès lors qu'elles représentent des grandeurs physiques ou technologiques elles peuvent être typées par leurs grandeurs que l'on appelle des quantités.
Une quantité comporte :
- un type de quantité de base ; par exemple une longueur,
- une borne min (respect. max) qui représente la valeur minimale (respect. maximale) autorisée,
- une dimension qui représente la dimension au sens de l'analyse dimensionnelle de la quantité,
- une unité de la quantité ; par exemple le mètre m pour une longueur.

Les quantités, les dimensions et les unités constituent une ontologie du domaine de l'ingénieur.
Les propriétés d'un modèle sont l'équivalent des contraintes dans un CSP. Dans les sciences de l'ingénieur elles sont souvent définies en intension par des équations ou des inéquations algébriques qui relient des constantes et des variables. C'est le cas dans la version actuelle du langage mais toutes les autres formes de définition qu'on retrouve en programmation par contraintes regroupées sous l'étiquette de « contraintes globales » seront permises ultérieurement.

3 DEPS Studio

3.1 L'Environnement

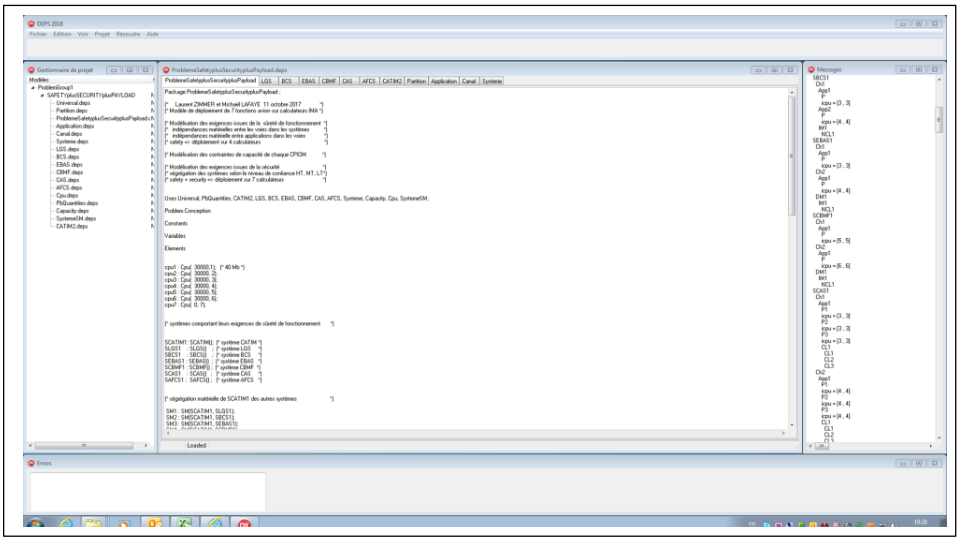


Figure 1 : L'environnement DEPS Studio

L'environnement de modélisation et de résolution intégré associé au langage DEPS comprend des fonctions d'édition de modèle et de gestion de projet, un compilateur et un solveur. L'expérience montre que la spécification d'un problème de conception de système n'est jamais bonne du premier coup et que beaucoup d'erreurs de modélisation ne sont détectables que par le calcul. Nous avons donc décidé de développer et d'intégrer notre propre solveur dans l'environnement de développement pour que la résolution contribue efficacement à la mise au point du modèle du problème. On se place ainsi dans une approche de développement rapide de modèles (analogue à une approche RAD) qui contrairement à une approche par transformation de modèle, diminue le temps d'exécution de la boucle de mise au point du modèle. Elle permet aussi de remonter les erreurs au bon niveau d'abstraction qui est celui de la modélisation.

3.2 Edition de modèles et gestion de projet

Un problème à résoudre est organisé en projet. Un projet est constitué de plusieurs packages. Chaque package est sauvegardé dans un fichier. Les packages contiennent des modèles, des éléments d'autres modèles et à la déclaration des constantes, des variables d'autres modèles.
Un des packages doit contenir un modèle particulier sans argument déclaré comme problème. Ce modèle représente le problème à résoudre exprimé sous forme de constantes, de variables, de relations et d'éléments.
L'environnement dispose :

- d'un éditeur multi-modèle pour charger, modifier et sauvegarder des packages,
- d'un gestionnaire de projet pour charger, modifier et sauvegarder le projet de modélisation d'un problème constitué de l'ensemble de ses packages.

3.3 Le compilateur

Le compilateur que nous avons développé transforme directement la modèle « source » DEPS d'un problème de conception en un réseau de contraintes « objet » associé au modèle <V, D, C>. C'est donc un compilateur « natif » qui n'est pas une surcouche d'un langage de programmation par contraintes. La compilation est anticipée ; tout le réseau est donc généré avant la résolution.
Le typage statique du langage DEPS est exploité par le compilateur pour détecter les erreurs de type des éléments et les erreurs de grandeur des constantes et des variables avant la résolution.
La compilation se fait en deux passes :

- la première passe de compilation contrôle les packages utilisés par le projet, analyse lexicalement et syntaxiquement leur contenu et crée la hiérarchie des modèles du projet ;
- La deuxième passe crée l'ensemble des éléments qui définissent le problème en partant de la création de l'unique élément instance du modèle déclaré comme problème.

Des erreurs sont traitées et remontées à l'utilisateur à toute les étapes de la compilation : contrôle des packages, analyse lexicale, analyse syntaxique, création de la hiérarchie des modèles, création des éléments sous-définis (avec des variables dont le domaine n'est pas réduit à une valeur).
Si l'étape de compilation réussit alors l'étape de résolution aura la tâche d'affecter des valeurs aux variables satisfaisant l'ensemble des propriétés/contraintes du problème.

3.4 Le solveur

Résoudre un problème de conception nécessite de disposer de capacités de résolution permettant de prendre en compte :

- Des problèmes sous contraints
- Des équations et inéquations algébriques non linéaires sur des domaines mixtes
- D'autres types de relations telles que des tables de valeurs, des relations logiques …

Pour ce faire, nous avons développé un solveur à base de contraintes dédié au calcul sur des modèles DEPS structurés. Les méthodes de calcul que nous utilisons sont tirées des travaux sur la résolution des CSP. La structure des modèles DEPS est conservée tout au long de la chaine de compilation jusque dans les modèles de calcul.
Le solveur implémente une méthode de propagation de type HC4 révisé [5] sur des équations et inéquations. Initialement prévue pour des domaines continus, nous avons étendu la méthode à quatre types de domaines : les intervalles ouverts de réels, les intervalles d'entiers, les ensembles énumérés de valeurs flottantes et les ensembles énumérés de valeurs entières signées. Les contractions sont réalisées directement sur les domaines typés sans repasser dans les intervalles de réels. L'algorithme de recherche de solution est une méthode de branch and prune. Les stratégies round-robin et first-fail sont disponibles.
Dans le cas d'un problème sur-contraint, un échec peut apparaitre dès la première propagation ou bien au final après avoir exploré les parties restantes de l'arbre de recherche. Dans ce cas, l'échec s'interprète comme la preuve qu'il n'y a pas de solution au problème posé et non pas comme une défaillance de l'algorithme de résolution.
L'architecture orientée-objet du solveur a été pensée de manière à pouvoir être étendue à d'autres méthodes de propagation et/ou de résolution (box-consistance, méthodes locales, …).

4 Conclusion et perspectives

Nous avons présenté dans ce papier DEPS Studio l'environnement de modélisation et de résolution de problèmes de conception de systèmes exprimés en DEPS. Dans sa version actuelle, cet environnement intègre un nouveau compilateur natif produisant directement un modèle de résolution et un solveur dédié dont les caractéristiques répondent aux problèmes rencontrés en ingénierie des systèmes. L'association qui en résulte est actuellement en cours d'évaluation sur des problèmes de synthèse d'architecture système.
Les développements en cours portent sur la prise en charge d'évolutions importantes du langage DEPS avec notamment l'introduction de collecteurs d'éléments ainsi que la définition de propriétés définies en extension. Ces développements seront disponibles dans une prochaine version de DEPS Studio.
L'association DEPSLink [2] supporte le développement et fait la promotion du langage DEPS.

Bibliographie

[1] P.A. Yvars, L. Zimmer, *"DEPS Un langage pour la spécification de problèmes de conception de Systèmes"*, proc of the 10th International Conference on Modeling, Optimization & SIMulation (MOSIM 2014), France.
[2] www.depslink.com
[3] L.Zimmer, M. Lafaye, P.A. Yvars, "Modélisation d'exigences pour la synthèse d'architecture avionique : Application à la sûreté de fonctionnement", 16eme journées AFADL, Approche Formelle dans l'Assistance au Développement de Logiciel, Montpellier, 2017.
[4] E. Tsang, Foundations of Constraint Satisfaction.London and San Diego:Academic Press, 1993.
[5] Benhamou F., Goualard F., Granvilliers L., Puget J.F., Revising Hull and Box consistency,16th International Conference on Logic Programming, 1993.

Session: Exigences logicielles

Clustering of Software Requirements for Automated Software Architectures

Takwa Kochbati[1,2], Shuai Li[1], Sébastien Gerard[1], and Chokri Mraidha[1]
[1] CEA List
[2] Université Paris-Sud, Université Paris-Saclay, F-91405 Orsay, France.
takwa.kochbati@cea.fr, shuai.li@cea.fr, sebastien.gerard@cea.fr, chokri.mraidha@cea.fr

Abstract

Requirements analysis is the first phase of software engineering cycle and it is essential for the success of the software development process. Software requirement specifications are often expressed in natural language, which is comprehensible by stakeholders. One of the goals of requirements analysis is to organize them into hierarchical clusters. These clusters constitute a mean to identify main packages of a software architecture. Thus, automating requirements clustering would be a first step towards a tooled assistance for software architectures design. Automating clustering of requirements written in natural language is not straightforward, due to the richness of natural languages and requires the use of Natural Language Processing techniques that have several limitations.

In this context, one of the objectives of this doctoral research, presented in this paper, is to develop an automated approach for software requirements clustering in order to help the developer in the design phase by automating the transition from an unstructured model of software requirements into a UML model denoting a preliminary software design architecture.

Keywords—software requirements, clustering, machine learning, natural language processing, software design architecture, UML.

1 Introduction

Requirements analysis is critical to the success of a software development project [1]. Usually, natural language is preferred to denote the software requirements including different constraints and properties of the target software [2]. When it comes to the clustering of requirements, the tasks to manually extract semantic clusters from the natural language requirements can be tedious and error-prone. This is

particularly because engineers and other stakeholders use different terminologies and sentence structures to describe the same kind of requirements [3].

Moreover, building a software architecture is an important step for transitioning from informal requirements expressed in natural language to precise and analyzable specifications through architecture models [4]. Furthermore, this can be a tedious and cumbersome task especially for large systems. Therefore, this raises the need of an automated approach for architecture artifacts generation from natural language based requirements.

Several approaches already exist to assist engineers with the task of automating the transitioning from requirements to architecture models generation [5], [6], [7]. Most of these approaches are commonly based on Natural Language processing (NLP) techniques to extract architecture artifacts expressed in UML.

However, these approaches address many limitations in term of accuracy. Some are due to the complexity of the case study system [5]. Others are because some NLP based rules used in these approaches present weaknesses in term of semantic relations extraction between artifacts [6]. In fact, most of the proposed rules for the extraction of architecture artifacts lack of semantic support. Therefore, increasing relevance in these cases requires a human in-the-loop strategy that is a difficult task.

Motivated by addressing the above limitations, we propose an approach based on semantic clustering of requirements as an initial step towards automatic generation of software architecture model.

The global approach is based on two main contributions: 1) enabling automatic clustering of unstructured description of requirements in order to get a first refinement of the system into subsystems; 2) the mapping of requirements clusters into architectural models artifacts. For this classification task, we plan to use supervised learning combined with NLP techniques in order to automate the transition between the requirements analysis phase and the architecture design phase.

In this paper, we initially focus on the very first step of our approach, which is the clustering of software requirements. In fact, requirements need to be grouped into semantic clusters in order to get a general view of the system and its subsystems (i.e., the clusters). We believe that this step will enable to improve the accuracy issue in the case of large systems by refining it into subsystems. This step will hence help developers to design an initial architecture of the system [8]. Eventually, each subsystem will cover a set of semantically related requirements and it will be implemented by a specialized developer team.

However, in this context, there is a lack of tools or methodologies for requirements clustering with the goal of automating architecture artifacts generation. Some works have studied non-functional software requirements classification [9], [10], others have addressed a solution to the problem of software requirements clustering using machine learning techniques [10], [11], [12], [13], [14], [15]. Nevertheless, their research direction is not to automate architecture artifacts generation based on the semantic similarity between requirements.

In [13], [14], [15], the proposed requirements clustering approaches are commonly based on the use of NLP techniques for text preprocessing, then, traditional Vector Space Model (VSM) [15] and Latent Semantic Analysis (LSA) [14] were proposed for text representation as well as term frequency-inverse term frequency (tf-idf) for terms weighting [15] and cosine similarity for terms similarity measurement [14]. Furthermore, the most used algorithms for the clustering were K-means [14], Hierarchical Agglomerative Clustering (HAC) [13], [14] and K-medoids [15].

However, all of these approaches require extensive manual intervention. Some are due to the textual structure of the input requirements data, which varies from a case study to another; others are due to the performance of the used techniques especially when using traditional Distributional Semantic Models (DSMs) [16] for the clustering such as LSA.

In this paper, we initially aim to propose an approach to overcome these limitations, thus, achieving better accuracy in the context of requirements clustering via applying neural word embedding models [17] followed by a clustering algorithm for requirements grouping.

2 Requirements clustering as an initial step towards automated software architectures

Based on these insights, in this PhD thesis, the first goal of this research project is to propose a software requirements clustering approach, from which requirements clusters can then be derived and exploited for software architecture generation.

For this first goal, the proposed approach is based on three parts as shown in the Fig 1:

1) Preprocessing: in this step, we aim to normalize each statement of the requirements by using NLP techniques.

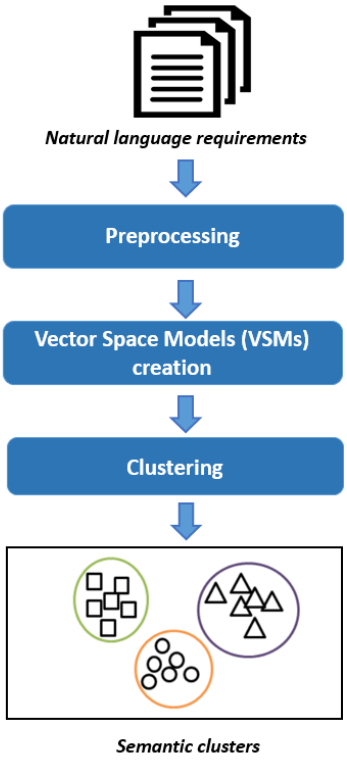


Fig 1: Requirements Clustering Approach

2) Creation of Vector Space Models (VSMs): The input to the clustering algorithm should be represented in VSM. Thus, this step consists in converting the textual data into numerical values.

Different models have been proposed in literature to obtain a VSM. They give nothing but a numerical vector representation of the textual data. This vector works as feature vector of a document and determines how many features would be there. Hence, we propose to use neural word embedding models as prediction models to represent the requirements statements. In fact, neural word embedding models work well in understanding the contextual meaning of text compared to most of the proposed approaches that use traditional DSMs when representing textual requirements [17].

In addition, DSMs can be seen as count models as they "count" co-occurrences among words by operating on co-occurrence matrices. They are more used in classification. Neural word embedding models, in contrast, can be viewed as predictive models, as they try to predict surrounding words by training a neural network against neighbourhood word of a randomly sampled word and vice versa. They are used to compute similar words and they allow words with similar meaning to be understood by machine learning algorithms. In the context of requirements clustering, neural word embedding models, as prediction models, can give best requirements representation, as they have shown to outperform common traditional count models [17].

3) Clustering: the resulted vectors that represent requirements will be then fed into a clustering algorithm. In this step, we will experiment K-means algorithm to cluster the requirements. In fact, K-means is one of the simplest unsupervised learning algorithms that solve the clustering problems. Furthermore, it has shown better results compared to other common document grouping techniques according to a comparison made in [18].

Thus, for our initial work, we plan to make two contributions. (1) We plan to make a systematic literature review to compare requirements clustering approaches and measure how well several existing techniques perform for requirements clustering in the context of transitioning between the requirements analysis phase and the design phase. (2) We propose to apply neural word embedding models for requirements representation in the clustering step to handle the limitations in term of accuracy and performance, and then, contributing in automating the decomposition of the system into subsystems (i.e. clusters). This will improve the automation of the system decomposition. In fact, each subsystem will be represented by a cluster, which contains a group of semantically related requirements. Consequently, this will reduce the manual effort for developers when decomposing the system.

3 Case study and evaluation metrics

We will start by conducting our approach on a case study called Traffic Jam chauffer (TJC). The TJC is conducted as part of the French innovation project SVA (numerical Simulation for the Autonomous Vehicle Safety) project [19] at IRT-SystemX. The TJC requirements document contains a hundred functional requirements expressed in French that we will translate into English.

To evaluate the accuracy of the resulted clusters, we will use a simple set of accuracy metrics, *precision* and *recall* [20], then we will compute the *F-measure* [20] to combine *precision* and *recall* into one metric. Below, we outline the procedure for calculating *precision*, *recall* and *F-measure* for clusters.
Let $I_1, \ldots, I_t$ denote the set of ideal clusters, and let $G_1, \ldots, G_u$ denote the set of generated clusters.

For every pair (I_i,G_j) $1 \leq i \leq t$; $1 \leq j \leq u$, let n_{ij} be the number of common data points between I_i and G_j:

– *Precision* $(I_i,G_j) = \frac{n_{ij}}{|G_j|}$

– *Recall* $(I_i,G_j) = \frac{n_{ij}}{|I_i|}$

– *F-measure* $(I_i,G_j) = \frac{2\times\text{Precision}(I_i,G_j)\times\text{Recall}(I_i,G_j)}{\text{Precision}(I_i,G_j)+\text{Recall}(I_i,G_j)}$

4 Conclusion

In this paper, we propose an approach for requirements clustering as a first step towards automated software architecture. In fact, clustering the requirements allows to get a first refinement of the system into subsystems. Thus, it will make it easier to automate the architecture models generation task especially for large systems, as well as moving from informal requirements expressed in natural language to analyzable domain architecture.

Up to now, we are working on performing a systematic literature review in order to carefully classify the existing clustering techniques, and to identify their limitations. We intend to overcome these limitations by first improving the clustering results and thus, getting clusters that represent the system and its subsystems with better cohesiveness. To this end, we plan to use NLP and neural word embedding models, followed by the K-means clustering algorithm. Then, we aim to improve the accuracy of the used NLP based approaches in the context of automating the architectural model artifacts extraction from each cluster. Thus, we will focus on adding relevant semantic rules to improve the architectural model artifacts extraction process.

Acknowledgment

This work has been partially funded by the ECSEL Joint Undertaking (JU) Arrowhead Tools project under grant agreement No 826452.

References

[1] J. Verner, K. Cox, S. Bleistein, et N. Cerpa, « Requirements Engineering and Software Project Success: an industrial survey in Australia and the U.S », *Australas. J. Inf. Syst.*, vol. 13, nº 1, nov. 2005.

[2] A. Gangopadhyay, « Conceptual modeling from natural language functional specifications », p. 12, 2001.

[3] Z. S. H. Abad, A. Shymka, S. Pant, A. Currie, et G. Ruhe, « What are Practitioners Asking about Requirements Engineering? An Exploratory Analysis of Social Q&A Sites », in *2016 IEEE 24th International Requirements Engineering Conference Workshops (REW)*, Beijing, China, 2016, p. 334‑343.

[4] T. Yue, L. C. Briand, et Y. Labiche, « A systematic review of transformation approaches between user requirements and analysis models », *Requir. Eng.*, vol. 16, nº 2, p. 75–99, juin 2011.

[5] A. Al-Hroob, A. T. Imam, et R. Al-Heisa, « The use of artificial neural networks for extracting actions and actors from requirements document », *Inf. Softw. Technol.*, vol. 101, p. 1‑15, sept. 2018.

[6] Chetan Arora, Mehrdad Sabetzadeh, Lionel Briand, et F. Zimmer, « Extracting domain models from natural-language requirements: approach and industrial evaluation », in *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems - MODELS '16*, Saint-malo, France, 2016, p. 250‑260.

[7] S. K.Shinde, V. Bhojane, et P. Mahajan, « NLP based Object Oriented Analysis and Design from Requirement Specification », *Int. J. Comput. Appl.*, vol. 47, nº 21, p. 30‑34, juin 2012.

[8] A. Alebrahim, « Framework for Identifying Meta-Requirements », in *Bridging the Gap between Requirements Engineering and Software Architecture: A Problem-Oriented and Quality-Driven Method*, Wiesbaden: Springer Fachmedien Wiesbaden, 2017, p. 51–109.

[9] I. Hussain, L. Kosseim, et O. Ormandjieva, « Using Linguistic Knowledge to Classify Non-functional Requirements in SRS documents », in *Natural Language and Information Systems*, vol. 5039, E. Kapetanios, V. Sugumaran, et M. Spiliopoulou, Éd. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, p. 287‑298.

[10] R. Deocadez, R. Harrison, et D. Rodriguez, « Automatically Classifying Requirements from App Stores: A Preliminary Study », in *2017 IEEE 25th International Requirements Engineering Conference Workshops (REW)*, Lisbon, Portugal, 2017, p. 367‑371.

[11] D. Gupta, V. Kr. Goyal, et H. Mittal, « Analysis of Clustering Techniques for Software Quality Prediction », in *2012 Second International Conference on Advanced Computing & Communication Technologies*, Rohtak,Haryana, India, 2012, p. 6‑9.

[12] Y. Li, S. Schulze, et G. Saake, « Extracting features from requirements: Achieving accuracy and automation with neural networks », in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Campobasso, 2018, p. 477‑481.

[13] H. Eyal Salman, M. Hammad, A.-D. Seriai, et A. Al-Sbou, « Semantic Clustering of Functional Requirements Using Agglomerative Hierarchical Clustering », *Information*, vol. 9, nº 9, p. 222, sept. 2018.

[14] N. H. Bakar, Z. M. Kasirun, et H. A. Jalab, « Towards Requirements Reuse: Identifying Similar Requirements with Latent Semantic Analysis and Clustering Algorithms », vol. 5, nº 1, p. 6,2015.

[15] R. Barbosa, D. Januario, A. E. Silva, R. Moraes, et P. Martins, « An Approach to Clustering and Sequencing of Textual Requirements », in *2015 IEEE International Conference on Dependable Systems and Networks Workshops*, Rio de Janeiro, Brazil, 2015, p. 39‑44.

[16] C. Fabre et A. Lenci, « Distributional Semantics Today Introduction to the special issue », p. 15.

[17] M. Baroni, G. Dinu, et G. Kruszewski, « Don't count, predict! A systematic comparison of context-counting vs. context-predicting semantic vectors », in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Baltimore, Maryland, 2014, p. 238‑247.

[18] D. Sonagara et S. Badheka, « Comparison of Basic Clustering Algorithms », p. 4, 2014.

[19] https://www.irt-systemx.fr/project/sva/

[20] C. Manning, P. Raghavan, et H. Schuetze, « Introduction to Information Retrieval », p. 581, 2009.

Un langage d'exigences entre langage naturel et langage formel

Florian Galinier, Sophie Ebersold et Jean-Michel Bruel

Abstract

Les exigences sont les briques de base de tout développement de système. Même dans le cadre de petits projets, on peut faire appel à plusieurs intervenants ayant des connaissances et des expertises différentes et, par conséquent, le langage naturel (NL) est souvent le langage commun utilisé pour exprimer les besoins et faciliter la communication entre ces intervenants. Pour faciliter l'analyse des besoins et sa vérification, la formalisation des exigences peut être utilisée. Néanmoins, bien qu'elles soient plus faciles à analyser et à vérifier, ces exigences formelles sont moins abordables pour les parties prenantes et exigent une certaine expertise dans les langages formels utilisés. Pour combler cette lacune, nous proposons dans ce papier un DSL (Domain Specific Language) proche du langage naturel, appelé RSML, dédié à l'expression des exigences. Tous les éléments du métamodèle RSML sont sémantiquement définis afin que les artefacts nécessaires puissent être utilisés pour la vérification et la validation des exigences du système. De plus, RSML est abordable pour les utilisateurs non spécialistes et, grâce aux transformations des modèles RSML en d'autres langages d'exigences, peut être utilisé en combinaison avec des approches, langages ou outils classiques (tels queKAOS, SysML, ou même Word ou Excel). mots clés : ingénierie des exigences, ingénierie système par modèle, langage spécifique au domaine, transformation de modèle, développement sans couture, vérification et validation