



Défis pour le Génie de la Programmation et du Logiciel

GDR CNRS GPL

<http://gdr-gpl.cnrs.fr/>

Une réflexion coordonnée par
Laurence Duchien (LIFL, INRIA, Univ. Lille 1)
et
Yves Ledru (LIG, Univ. Grenoble-1)

Ce document comprend :

- Le document de synthèse publié à l'issue de cette réflexion
- Les défis publiés lors des journées de Pau
 - Software (re)modularization : Fight against the structure erosion and migration preparation
 - Component and Service Farms
 - End-User Modelling
 - Towards Disappearing Languages
 - Vers une réification de l'énergie dans le domaine du logiciel _ L'énergie comme ressource de première classe
- Les 4 nouveaux défis soumis à la journée du 8 juillet
 - Intégration maîtrisée et raisonnée des modèles et méthodes formelles dans un processus industriel de développement logiciel. (*Groupe MFDL*)
 - Nouvelles dimensions du test logiciel (*Groupe MTV2*)
 - Contribution à Ingénierie des Besoins spécifique à l' « Adaptation Diffuse » (*Sylvie Vignes*)
 - Proposition de défis (*groupe LTP*)

Défis pour le Génie de la Programmation et du Logiciel¹

GDR CNRS GPL – <http://gdr-gpl.cnrs.fr/>

Coordonnés par
Laurence Duchien (LIFL, INRIA, Univ. Lille 1) & Yves Ledru (LIG, Univ. Grenoble-1)

Auteurs

- Y. Aït-Ameur (LISI, Poitiers),
- P. Albert (IBM),
- N. Anquetil (LIFL, INRIA, Univ. Lille 1),
- G. Arévalo (LIFIA, UNLP, Argentine),
- Z. Azmeh (LIRMM, Univ. Montpellier 2),
- M. Blay-Fornarino (I3S, Univ. Nice-Sophia-Antipolis),
- P. Castéran (LABRI, U. Bordeaux),
- P. Cointe (LINA, INRIA, Ecole des Mines de Nantes),
- P. Collet (I3S, Univ. Nice-Sophia-Antipolis),
- B. Combemale (IRISA, Univ. Rennes 1),
- C. Consel (LABRI, INRIA, Univ. Bordeaux 1),
- S. Denier (LIFL, INRIA, Univ. Lille 1),
- R. Douence (LINA, INRIA, Ecole des Mines de Nantes),
- C. Dubois (CEDRIC, ENSIIE),
- S. Ducasse (LIFL, INRIA, Univ. Lille 1),
- L. Duchien (LIFL, INRIA, Univ. Lille 1),
- R. Ducournau (LIRMM, Univ. Montpellier 2),
- S. Dupuy-Chessa (LIG, Univ. Grenoble 1),
- A. Front (LIG, Univ. Grenoble 1),
- R. Giroudeau (LIRMM, Univ. Montpellier 2),
- A. Grost (ATOS),
- R. Groz (LIG, Univ. Grenoble),
- A. Gotlieb (INRIA),
- G. Hains (LACL, Univ. Paris-Est-Créteil),
- M. Huchard (LIRMM, Univ. Montpellier 2),
- J.-M. Jézéquel (IRISA, Univ. Rennes 1),
- J.-C. König (LIRMM, Univ. Montpellier 2),
- P. Lahire (I3S, Univ. Nice-Sophia-Antipolis),
- R. Laleau (LACL, Univ. Paris-Est-Créteil),
- J. Laval (LIFL, INRIA, Univ. Lille 1),
- X. Le Pallec (LIFL, Univ. Lille 1),
- A. Lèbre (LINA, INRIA, Ecole des Mines de Nantes),
- T. Ledoux (LINA, INRIA, Ecole des Mines de Nantes),
- L. Ledrich (ALTEN Nord),
- Y. Ledru (LIG, U. Grenoble 1),
- J.-M. Menaud (LINA, INRIA, Ecole des Mines de Nantes),
- T. Nodenot (LIUPPA, Univ. de Pau et des Pays de l'Adour),
- J. Noyé (LINA, INRIA, Ecole des Mines de Nantes),
- A.-M. Pinna-Dery (I3S, Univ. Nice-Sophia-Antipolis),
- D. Pollet (LIFL, INRIA, Univ. Lille 1),
- M.-L. Potet, (Verimag, Univ Grenoble),
- C. Tibermacine (LIRMM, Univ. Montpellier 2),
- R. Rouvoy (LIFL, INRIA, Univ. Lille1),
- S. Rusinek (Psitac),
- L. Seinturier (LIFL, INRIA, Univ. Lille1),
- D. Seriali (LIRMM, Univ. Montpellier 2),
- M. Südholt (LINA, INRIA, Ecole des Mines de Nantes),
- C. Urtado (LGI2P, Ecole des Mines d'Alès),
- S. Vauttier (LGI2P, Ecole des Mines d'Alès),
- S. Vignes (Dpt Informatique et Réseaux, Telecom-ParisTech),
- M. Ziane (LIP6, Univ. P. et M. Curie).

¹ Ce document a pour objectif d'être diffusé le plus largement possible à partir du site du GDR GPL. Il sera également envoyé vers les tutelles CNRS et INRIA et les agences de moyens de type ANR.

Résumé des contributions

Jean-Marc Jézéquel constate qu'on n'écrit plus des logiciels en partant de zéro mais en faisant évoluer, en continu, des logiciels existants. Dans ce contexte, les concepts actuels de structuration (classes, paquets,...) s'avèrent insuffisants et il est nécessaire d'introduire de nouveaux concepts de structuration qui supportent mieux les considérations transverses. Nicolas Anquetil et ses collègues constatent que, dans ce contexte d'évolution continue des logiciels, la qualité de la structure des logiciels est déterminante pour faciliter ces migrations. Il s'agit donc d'être capable d'identifier les bonnes structures, et le cas échéant de remodulariser pour adapter la structure et la recréer.

Patrick Albert et ses collègues partent d'un constat largement accepté que l'implication de l'utilisateur dans la construction d'un logiciel est un élément déterminant pour le succès ou l'échec du projet. Leur proposition est de faire participer plus activement l'utilisateur, expert d'un domaine, à l'élaboration des modèles dans une démarche d'Ingénierie Dirigée par les Modèles (IDM). Une autre façon d'impliquer l'utilisateur consiste à lui proposer un langage de description spécifique pour son domaine (DSL). Charles Consel constate que la définition d'un tel langage est une activité complexe et propose d'étudier les manques des propositions actuelles pour ensuite développer des outils de développement dédiés aux DSL.

Dans ce contexte où la complexité des logiciels se traduit par une activité d'adaptation et d'intégration, Gabriela Arévalo et ses collègues attirent notre attention sur le besoin de nouvelles infrastructures pour la recherche de composants et de services. Ces infrastructures devraient aller au delà de la simple recherche pour permettre aux utilisateurs de ces composants et services de partager leurs expériences et connaissances. Lionel Seinturier attire notre attention sur les nombreux défis que posent l'approche service et son prolongement que constitue le "*cloud computing*". Pour le Génie Logiciel, le défi concerne la définition d'un modèle universel de composants et de services qui couvre l'ensemble de leur cycle de vie. Romain Rouvoy anticipe le développement de nouvelles utilisations de l'Internet où chacun aura la possibilité de publier des services virtuels ou physiques. La conception et la publication de ces ressources appellent le développement de nouveaux outils logiciels, adaptés à la diversité de ces objets communicants.

Jean-Marc Menaud et ses collègues constatent que les considérations environnementales nous forcent à considérer les performances des logiciels non seulement en consommation de temps et de mémoire, mais également en termes de consommation énergétique. Il s'agit dès lors de sensibiliser l'utilisateur final et le développeur à ce problème en leur offrant des mécanismes d'introspection/réflexion adaptés. Gaétan Hains va également dans ce sens en proposant de remonter les mécanismes de *monitoring* dans les plates-formes à large échelle du niveau système au niveau de l'intergiciel.

Mikal Ziane refait le constat que l'une des difficultés du développement logiciel vient du fait que ces outils partagent peu de connaissances avec leurs utilisateurs. Dans les années 80, le projet américain *Knowledge-Based Software Engineering* avait fait le même constat, sans parvenir à le résoudre. Depuis, les connaissances et les outils ont significativement évolué et il est probablement temps de tenter à nouveau l'intégration des outils et des connaissances. Finalement, Laurence Duchien attire notre attention sur le besoin de construire des logiciels flexibles qui s'adaptent à leur contexte d'utilisation. Cela revient à prendre en compte l'incertitude dans toutes les phases du développement logiciel. Ceci suppose le développement de modèles de haut niveau pour représenter l'incertitude.

Le groupe MFDL (Méthodes Formelles pour le Développement du Logiciel) pose le défi de l'intégration maîtrisée et raisonnée des méthodes formelles dans le développement de systèmes à logiciel prépondérant. Cela amène notamment aux problèmes suivants : établir une chaîne

continue de développement qui va de l'analyse système à la production de code, et garantir la traçabilité sémantique tout au long de cette chaîne. Les difficultés de passage à l'échelle de ces méthodes imposent aussi des choix quant aux parties qui feront l'objet de tels développements et leurs liens avec des parties développées de façon plus classique.

Le groupe MTV2 (Méthodes de Test pour la Validation et la Vérification) identifie trois défis pour la définition de processus et techniques de test mieux étayés, plus fiables en termes de sûreté et sécurité, et capables de passer à l'échelle. Le premier défi concerne l'intégration du test aux nouvelles pratiques de développement que constituent le génie logiciel à base de modèles, les méthodes agiles et le développement « off-shore ». Le deuxième défi répond à la société qui demande des garanties en termes de fiabilité et de sécurité. Le troisième défi est celui de la diversité et de la complexité des environnements où sont plongés les systèmes à tester, ce qui les rend plus difficilement contrôlables et observables, et fait exploser le nombre de configurations et de comportements à tester.

Sylvie Vignes attire notre attention sur le défi que représente « l'adaptation diffuse », c'est-à-dire la capacité d'un système à modifier les comportements individuels de ses artefacts pour faire émerger un comportement global en interaction avec son environnement. De nombreuses applications relèvent de cette problématique : contrôle d'un essaim de micro-robots, auto-organisation d'objets hétérogènes destinés à la domotique, réseaux de capteurs, etc. Un constituant essentiel de ce défi est l'ingénierie des besoins qui doit à la fois exprimer le comportement global et les exigences sur les composants individuels, tout en tenant compte du caractère incertain et des capacités d'adaptation du système.

Le groupe LTP (Langages, Types et Preuves) constate que la diffusion des méthodes de développement basées sur la preuve reste limitée. Il diagnostique que cet état de fait est lié à la difficulté de prise en main des outils de preuve, à leur capacité à passer à l'échelle et à la nécessité de valider les spécifications et hypothèses de départ. Il propose des pistes pour relever le défi de l'adoption des outils de preuve et propose de faciliter la validation des spécifications par l'utilisation de langages dédiés aux domaines concernés.

1. Introduction

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du génie de la programmation et du logiciel à échéance de 5 à 10 ans ?

De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable et les économies d'énergie.

L'appel à défis pour les journées du GDR GPL à Pau en mars 2010, à sa suite, l'envoi du questionnaire de l'Institut INS2I vers les membres du conseil scientifique et les groupes de travail, et finalement la journée à Paris en juillet 2010 ont permis de collecter les questions sur lesquelles les équipes envisagent de travailler dans les années à venir. Les réponses à cet appel sont le fruit de discussions menées dans les groupes de travail et de contributions individuelles. A partir de ce matériel, nous dressons ici un panorama en 6 grandes familles thématiques, à savoir la structuration des logiciels pour leur évolution, la prise en compte de l'utilisateur final, l'accès aux composants et services, la prise en compte de la consommation de ressources dans les outils du logiciel la modélisation de connaissances et de l'incertain et, finalement, la place des méthodes et outils de vérification et de validation.

Sans d'être exhaustif, ce texte regroupe, à notre avis, une grande partie des thèmes de la programmation et du génie logiciel abordés par les équipes françaises. Nous présentons chacune de ces 6 familles, puis nous donnons un ensemble de défis transverses qui sont apparus lors des échanges lors de la journée parisienne en juillet.

2. Structuration des logiciels pour leur évolution

Concepts pour préparer un logiciel à évoluer – Jean-Marc Jézequel

Les systèmes informatiques deviennent de plus en plus complexes. Cette complexité augmente de façon exponentielle avec un facteur dix tous les dix ans, ceci depuis plus de quarante ans. En dehors de toute considération académique ou industrielle, le fait que le développement d'un logiciel commence par une spécification bien définie et que le code soit écrit *from scratch* n'est plus vrai. Dans le monde réel de la programmation, le changement et l'évolution en continu sont la norme. En lieu et place de produire un logiciel pour un problème donné, l'objectif réel maintenant en génie logiciel est de produire des familles de logiciels évolutifs, soit dans l'une ou l'autre dimension, ou les deux. Ces deux dimensions sont le temps (versions successives) et l'espace (variantes d'un produit). En fonction du contexte, la prise en considération du délai de mise sur le marché, du coût et de la qualité rentrera en jeu dans la technique choisie pour la mise en œuvre de l'évolution de ce logiciel.

Un besoin croissant de concepts se fait sentir pour la construction de programmes à partir de préoccupations développées indépendamment les unes des autres et pouvant être combinées de façon flexible, ceci en vue de faciliter la gestion d'évolution. Ces concepts doivent venir en complément des éléments structurants disponibles dans les langages actuels : modules, classes,

fonctions, paquets, etc. Quelles que soient les structures choisies pour définir un système complexe, certaines préoccupations ne se composent pas si facilement, avec des méthodes de conception traditionnelle. Ces préoccupations peuvent correspondre à des moyens de synchronisation, de gestion de mémoire, des politiques de cache, du monitoring, etc. Elles sont connues comme des préoccupations transverses, parce que leur prise en compte doit être mis en place dans des petites portions de code réparties partout dans un programme correctement structuré, ce qui rend leur évolution coûteuse, pénible et risquée.

Remodularisation des logiciels : lutter contre l'érosion de la structure et préparer la migration [Anquetil et al. 2010]

Les systèmes orientés objet sont des modèles du monde réel qui manipulent une représentation de ses entités au travers de modèles de processus. Le monde réel n'est pas statique, de nouvelles lois sont créées, des concurrents offrent de nouvelles fonctionnalités, les utilisateurs ont de nouveaux besoins, des contraintes physiques peuvent s'ajouter. Un logiciel doit alors s'adapter continuellement, au risque, dans le cas contraire, de devenir rapidement obsolète. Dans le même temps, les applications sont devenues complexes et très grandes. Au-delà de la maintenance, une bonne structure permet d'avoir des systèmes logiciels de bonne qualité pour la migration vers des paradigmes modernes tels que les web services ou les composants et le problème de l'extraction de l'architecture est très proche du problème de remodularisation. Les lois de l'évolution du logiciel définies par Lehman et Belady indiquent que les changements sont continus, c'est-à-dire qu'un logiciel qui résout un problème doit être continuellement adapté, sinon, il devient progressivement moins satisfaisant, et que plus un programme évolue, plus sa complexité augmente, alors que moins de travail est fait pour le maintenir ou le réduire.

Différentes solutions ont été étudiées pour construire des abstractions modulaires, pour proposer des approches de remodularisation, pour définir des techniques de modules. Aucune n'est pour le moment satisfaisante. Aussi plusieurs challenges se présentent. Le premier concerne une meilleure maîtrise de l'érosion et la préparation de la migration vers des technologies avancées. Le second devrait permettre la définition de bonnes abstractions et de leurs relations entre-elles, de la proposition d'algorithmes de remodularisation complémentaires en termes d'analyses, de complexité et d'approximation, mais également ces algorithmes devront passer à l'échelle.

Il est donc urgent de conduire des études complètes dans ce contexte de remodularisation, aussi bien verticalement par l'étude de tous les aspects du problème de modularisation (modélisation du logiciel, définition de métriques de qualité de modularisation, présentation des résultats) et horizontalement en considérant les différentes approches de modularisation. La solution sera sans doute multiple, prenant en compte différentes compétences dans différents domaines de recherche.

3. Prise en compte de l'utilisateur final

End-user modelling [Albert et al. 2010]

D'après le *Standish group*, l'amélioration du taux de réussite des projets informatiques dépend, entre autres, de l'implication des utilisateurs finaux. L'ingénierie logicielle a intégré depuis quelque temps dans ses approches, le fait d'associer plus étroitement l'utilisateur futur dans les phases de conception. Les démarches agiles, officialisées en 2001, ont popularisé ce principe. En parallèle, la production logicielle et la taille des applications informatiques ont accentué la nécessité de réutiliser les éléments logiciels. Une réponse est d'accorder plus d'importance à la modélisation des logiciels et d'y faire participer l'utilisateur final. Cette modélisation doit être le vecteur pour une discussion et une compréhension entre utilisateurs et concepteurs.

L'objectif de ce défi est de proposer aux experts d'un domaine les moyens de construire leur propre système informatique en utilisant des techniques de modélisation avancée telles que

l'ingénierie dirigée par les modèles. Il s'agit ici de proposer une approche de la conception, non pas dirigée par la syntaxe d'un langage, mais par les concepts, les usages et les contraintes du métier. Les enjeux sont une productivité plus grande et une meilleure adéquation des produits aux problèmes. Ces enjeux ne seront atteints que si les artefacts de modélisation donnés sont adéquats et si les produits résultants sont fiables et opérationnels.

Les verrous identifiés sont des difficultés en termes de représentations à donner aux multiples profils des utilisateurs à mettre en adéquation avec leurs usages, au niveau de l'interprétation de ces notations par le système pour assurer l'aide, la validation, la collaboration et l'exécution des systèmes produits, et finalement l'impact sur les démarches usuelles de développement.

Les jalons de ce défi sont la circonscription de la complexité par des études de cas, la définition des artefacts de modélisation cognitivement adaptés aux différents rôles impliquées, la corrélation entre modélisation métier et mise en œuvre au niveau des plates-formes.

L'évaporation des langages [Consel 2010]

L'approche par DSL (*Domain Specific Language*) est utilisée depuis longtemps avec succès dans deux domaines historiques que sont la téléphonie et le développement d'applications Web. De l'ingénierie du logiciel aux langages de programmation, il y a un sentiment partagé que qu'il reste encore beaucoup de travail pour que l'approche DSL soit un succès.

A la différence de langages de programmation généralistes qui ciblent des programmeurs entraînés, un DSL tourne autour d'un domaine : il vient d'un domaine et cible les membres de ce domaine. Ainsi, un DSL réussi doit être une sorte de langage qui se fait oublier. En quelque sorte, la programmation étend sa portée aux utilisateurs finaux, ce qui inclut les utilisateurs pour qui écrire des programmes vient en support de leur métier premier. Les exemples bien connus sont Excel ou encore MatLab.

Créer un langage qui sait se faire oublier repose conjointement sur une analyse de domaine et sur la conception de langage. Les expériences pratiques montrent que ces deux phases sont consommatrices de temps et à hauts risques. Comment ces deux phases doivent-elle être outillées ? Quelles améliorations pouvons nous en attendre ?

Un DSL réussi est surtout un DSL utilisé. Pour atteindre ce but, le concepteur doit avoir besoin de réduire, de simplifier, et de personnaliser un langage. Pour faire cela, le développement d'un DSL contraste avec la recherche dans les langages de programmation pour lesquels la généricité, l'expressivité et la puissance doivent caractériser tout nouveau langage. Une conséquence de ceci est que les experts de langage de programmation ne sont pas nécessairement les bonnes personnes pour développer un DSL. Est-ce que cela veut dire que pour un domaine donné, ses membres devraient développer leur propre DSL ? ou qu'il devrait exister une nouvelle communauté d'ingénieurs langage qui font le lien entre les experts en langage de programmation et les membres d'un domaine ? La communauté de recherche dans les langages de programmation s'est toujours très largement intéressée à la conception et à la réalisation des langages. Très peu d'effort ont été portés sur la compréhension de la façon dont les humains utilisent les langages. Cette piste de recherche est une clé pour étendre le domaine de la programmation au delà des chercheurs en informatique. Comment devons-nous concevoir les langages pour qu'ils soient facilement utilisables ? Comment mesurons-nous le bénéfice de la productivité avec l'utilisation d'un langage ? Comment mesurons-nous la qualité des logiciels venant de programmes issus de DSL ?

Finalement un DSL est souvent une version simplifiée d'un langage de programmation généraliste : les constructions syntaxiques sont personnalisées, la sémantique est simple, les propriétés vérifiables par construction également. Ces différences clés peuvent provenir du manque d'outils pour le développement de DSL. Il y a aussi beaucoup d'outils de manipulation de programmes (générateurs de *parsers*, éditeurs, IDE) qui peuvent facilement être

personnalisés pour de nouveaux langages, qu'ils soient graphiques ou textuels. De plus, pour une large classe de DSL, la compilation consiste à produire un code au-dessus d'un cadre de programmation spécifique à un domaine, et à permettre l'utilisation d'outils de transformation de haut niveau. Finalement, les propriétés peuvent souvent être vérifiées par des outils de vérification générique. Alors, que manque-t-il pour développer des DSL ? A-t-on besoin d'un environnement intégré pour le développement de DSL, orchestrant une librairie d'outils ? Devrait-il exister un nouveau type de compilateur et de générateur de vérification en relation avec les propriétés des DSL ?

4. Accès aux composants et services

Fermes de composants et de services [Arévalo et al. 2010]

Les composants logiciels et les web services sont des blocs de code qui sont utilisés dans la composition de logiciels modernes. Ils fournissent tous les deux des fonctionnalités qui demandent à être enregistrées dans des bases de données de façon à être accessibles et réutilisées dans des processus de construction de logiciels. Assembler des composants logiciels pour construire de nouveaux logiciels ou pour réparer ou faire évoluer des logiciels existants demande à ce que l'on puisse sélectionner le composant ou le service qui fournit une partie de la fonctionnalité applicative voulue et de le connecter facilement (avec un minimum d'adaptation) aux autres composants sélectionnés. De nombreux composants existent déjà. Par exemple, le moteur de recherche Seekda de web services contient plus de 28 000 références et le consortium OW2 regroupe plus de 40 projets open source dans le domaine des intergiciels. Cependant les modèles d'accès à ces références ne sont pas pour le moment satisfaisants. Il est alors nécessaire de concevoir des bureaux d'enregistrement (*registries*) pour collecter les composants et assister les utilisateurs dans leur recherche, leur sélection, leur adaptation et leur connexion d'un composant à d'autres. Nous imaginons que ces tâches pourraient être automatisées le plus possible et intégrées dans des environnements ouverts et dans des contextes dynamiques (périphériques embarqués administrés à distance, intelligence ambiante, applications ouvertes et extensibles, informatique mobile, etc.).

Le défi majeur consiste à proposer une architecture pour un bureau d'enregistrement des composants et des services, disponible en ligne, qui ne donnera pas uniquement un accès aux composants adéquats, mais proposera également un support tout au long du cycle de vie aux développeurs et aux applications (si possible en mode automatique). Ce bureau d'enregistrement sera une plate-forme de partage de connaissances et d'expériences sur les composants en direction des développeurs. Celle-ci leur permettra aussi bien de tester leurs développements efficacement que de les exécuter dans un environnement fiable. Une extension de ce moteur de recherche est naturellement une ferme de composants où les composants pourront être soit localisés physiquement (modèle de dépôt de composants) ou uniquement référencés dans un répertoire de composants organisé de manière adéquat. Cette ferme de composants pourra offrir des vues multiples sur les composants et proposer des mécanismes de recherche efficaces.

Software as a Service (SaaS) et Cloud Computing – Lionel Seinturier

L'approche service et l'informatique dans les nuages sont deux tendances actuelles fortes du développement logiciel. Au delà de l'attente suscitée à tort ou à raison, il s'agit d'une remise en cause profonde de la façon dont les logiciels sont développés et hébergés et du modèle économique qui sous-tend leur utilisation. L'approche SaaS peut être définie simplement comme un logiciel déployé comme un service hébergé et accédé via Internet. Les gains attendus d'une telle approche résident dans l'accès universel et ubiquitaire au logiciel, les gains en terme de maintenance et de mise à jour des infrastructures matérielles et système la tolérance aux pannes et le passage à l'échelle.

Les défis impliqués par l'approche service et l'informatique dans les nuages sont nombreux tant en terme de développement logiciel qu'en termes d'usage, confidentialité et propriété des données, mode économique de facturation. De part la nature distribuée des traitements et des données qui sont mises en jeu, ces défis concernent plusieurs communautés dont celle du génie logiciel (GDR GPL), mais aussi celles des systèmes distribués (GDR ASR) et des systèmes d'information (GDR I3). En terme de génie logiciel, un défi concerne la définition d'un modèle de services et composants universel et standardisé qui supporte le cycle de développement complet du SaaS, de la conception, à l'implémentation, au packaging, au déploiement, à l'exécution, au monitoring, à la reconfiguration. Ce modèle devra supporter un degré élevé d'hétérogénéité en terme de langages de description des services et de langages de programmation. Les descriptions de services devront pouvoir être enrichies avec des informations de niveau sémantique et non fonctionnelles liées par exemple à leur consommation énergétique. Ce modèle de services devra pouvoir être mis en œuvre aussi bien pour les applications SaaS que pour le développement des plates-formes de Cloud Computing supportant ces applications. En terme de plate-forme, il s'agira de faire en sorte de proposer différents niveaux de virtualisation des ressources logicielles et matérielles afin d'assurer une cohabitation harmonieuse, fiable et sûre des différents services hébergés. L'architecture de ces plates-formes devra être scalable afin de pouvoir envisager des usages de style Home Cloud pour l'hébergement de services dans des domaines tels que l'immotique et la domotique, sur des équipements tels que des set-top box, les compteurs intelligents ou les boîtiers de raccordement Internet. En conclusion, ce défi consiste à faire émerger un modèle viable et fédérateur pour l'industrie du logiciel.

Internet des ressources : Connecter le yocto physique au yotta virtuel – Romain

Rouvoy

Internet ne cesse de se développer afin d'offrir toujours plus de fidélité vis-à-vis du monde physique. Si l'émergence du Web 2.0 a offert de nouveaux outils collaboratifs pour diffuser un contenu plus dynamique et faciliter l'expression des internautes au travers des applications Internet riches (blog, forums, wiki), le monde physique demeure partiellement connecté au monde virtuel. La prochaine évolution de l'Internet se devra donc d'intégrer plus facilement toutes les dimensions du monde physique pour offrir aux internautes de nouvelles expériences. Dans cette perspective, on observe une mouvance croissante autour du concept de *ressource logicielle* car il semble offrir une abstraction adéquate pour connecter les objets du quotidien, du plus petit au plus grand. À l'avenir, l'internaute pourra ainsi naturellement rechercher et publier des services virtuels (co-voiturage) ou physiques (station météo) en les exposant sous la forme de ressources logicielles sur Internet. Du point de vue des usages, la prise en compte de la sécurité (authentification, cryptage) et du respect de la vie privée (diffusion aux membres du réseau social, diffusion sous anonymat) représente une dimension importante du succès de cet Internet démocratisé. D'un point de vue technique, cette intégration est une tâche complexe qui nécessite de prendre en compte la diversité matérielle et logicielle des objets communicants (protocoles de communication, de découverte, d'interaction). Dès lors, il apparaît critique de reconsidérer les méthodes et les outils d'ingénierie logicielle permettant de décrire, concevoir, configurer, et composer ces ressources logicielles afin d'offrir aux utilisateurs des interfaces flexibles et intuitives pour contrôler la diffusion et l'accès du publique à leurs objets. Ainsi, au travers de cette mise en relation des ressources distribuées sur l'Internet, il sera possible à terme de déployer des infrastructures collaboratives à très large échelle permettant aux internautes d'interagir à distance via les objets de leur quotidien.

5. Modèles de développement pour environnements contraints

De la réification de l'énergie dans le domaine du logiciel [Menaud et al. 2010]

En quelques années, le problème de la gestion de l'énergie est devenu un enjeu de société. En informatique, les principaux travaux se sont concentrés sur des mécanismes permettant de

maîtriser l'énergie au niveau du matériel. Le renforcement du rôle de l'informatique dans notre société (développement des centres de données, prolifération des objets numériques du quotidien) conduit à traiter ces problèmes aussi au niveau du logiciel. Nous nous posons la question de la réification de l'énergie comme fut posée en son temps celle de la réification de la mémoire (l'espace) et de l'interpréteur (la machine d'exécution).

Le défi est ici de sensibiliser dans un premier temps l'utilisateur final au problème de la consommation énergétique en lui offrant des mécanismes d'introspection visualisant la ressource énergie à l'image de ce qui se fait aujourd'hui dans le domaine automobile (consommation d'essence instantanée). Il s'agira ensuite de proposer des mécanismes d'intercession aux développeurs aptes à contrôler cette consommation énergétique. Ces mécanismes réflexifs devront concerner l'ensemble du cycle de vie du logiciel.

Programmation pour le calcul haute performance - Gaétan Hains

L'adaptation des modèles et outils de programmation parallèle aux architectures hétérogènes GPU-multicoeur-multiprocesseur-cloud et l'intégration de notions de très haut niveau pour la gestion des ressources de calcul: cpu/temps/espace/KW/€. Un thème émergent est celui de la fusion de ces travaux "programmation" avec les technologies industrielles du monitoring (métrologie) des centres de calcul, celles-ci étant essentiellement au niveau système et pas encore middleware.

Adaptation diffuse des logiciels –Sylvie Vignes [Vignes 2010]

Un environnement diffus peut se composer d'objets hétérogènes, tels que des objets émettant et recevant des informations via des réseaux de capteurs ou d'autres éléments logiciels. Le fait de positionner un logiciel dans un environnement de ce type et être capable d'interagir avec celui-ci nécessite de prendre en compte à la fois l'environnement et le logiciel dans leur globalité, et ceci dans toutes les étapes du cycle de vie du logiciel. Les premières étapes, c'est-à-dire l'analyse des besoins, doit être revue pour permettre l'expression à la fois du comportement global, mais également des comportements individuels. Les phases de conception, de développement et d'exécution doivent être également revisitées pour proposer des méthodes et outils permettant la prise en compte des interactions avec l'environnement du logiciel. Ce travail sur l'élaboration de nouvelles approches de développement pour des environnements diffus doit permettre la prise en compte de l'incertain tel que décrit dans la section 6 et offrir des capacités d'évolution tel que décrit dans la section 2.

6. Modélisation de connaissances et de l'incertain

Identifier et intégrer les connaissances des développeurs dans les outils de développement – Mikal Ziane

Pourquoi est-il si difficile de développer un logiciel ? Peut-on être satisfait de l'état actuel du génie logiciel ? Imaginer qu'un non informaticien puisse un jour assembler lui-même une application simple sur son ordinateur pour créer un nouveau programme est-ce trop demander ? Pourquoi les logiciels sont-ils toujours si coûteux à créer et maintenir et si peu fiables ?

Les outils sont ignares. Les outils actuels d'aide au développement sont typiquement ignares en ce sens qu'ils ne partagent que très peu de connaissances avec les développeurs sur la manière de développer du logiciel (et *a fortiori* sur les domaines d'application des logiciels qu'ils aident à développer). La conséquence est que le dialogue entre les développeurs et leurs outils se situe trop souvent à un très bas niveau d'abstraction, au niveau du code, ce qui limite drastiquement les possibilités d'utilisation de ces outils. Ce constat a déjà été fait dans les années 1980 voire avant mais il est toujours d'actualité.

Une thématique a disparu. Dans les années 80 et jusqu'au milieu des années 1990 le génie logiciel à base de connaissances était un thème de recherche reconnu [KBSE 1990], à l'intersection du génie logiciel et de l'intelligence artificielle. Son objectif était d'identifier les connaissances sous-jacentes au développement au logiciel et à les intégrer dans des outils. Ce n'est plus le cas. Ainsi la conférence *Knowledge-Based Software Engineering* s'est elle renommée *Automated Software Engineering* en 1997. Pourquoi ?

Sans doute la tâche était-elle trop ambitieuse à l'époque. Pourtant des tentatives qui n'affichent pas forcément explicitement cette thématique vont de temps à autre dans la même direction. Citons par exemple la tentative de Microsoft avec le projet *Intentional Programming* [IP 1996]. Remarquons aussi que l'initiative MDA de l'OMG [MDA] visait à séparer « *business and application logic from the underlying platform technology* » sans d'ailleurs s'en donner les moyens scientifiques.

La situation est plus favorable aujourd'hui qu'il y a 20 ans. Aujourd'hui un ensemble de signes suggèrent qu'aujourd'hui pourrait être le bon moment pour à nouveau, explicitement, chercher à identifier et à intégrer à des outils, non seulement les connaissances propre au développement logiciel mais aussi les connaissances des domaines applicatifs :

- Les outils de transformations ou de manipulation de programmes sont plus mûrs (TOM, Stratego, ASF+SDF/Rascal, ANTLR, ...). Or les connaissances de développement sont en premier lieu des transformations.
- Le développement de langage de domaines (DSL) est en plein essor, parfois adossés à des ontologies, bien que combler le hiatus entre les deux reste un problème ouvert.
- Des plates-formes comme Eclipse facilitent l'intégration d'outils dans un ensemble relativement cohérent.
- Le développement de technologies à base de composants, les intergiciels, les architectures à bases de services, s'ils posent de nouveaux problèmes facilitent globalement le développement du logiciel : il ne s'agit plus nécessairement de synthétiser un programme mais d'assembler et déployer des éléments de solution.

Quels défis pour demain ? Vu la difficulté de la tâche (à la mesure de son ambition) il semble qu'il faut définir les défis sur cette thématique selon des critères suivants :

- Domain-specific : on sait que des connaissances de domaine sont nécessaires. Il faut se focaliser sur des domaines abordables pour lesquels on sait déjà faire du logiciel mais pas un domaine trivial,
- Connaissances opérationnelles : il ne s'agit pas de tomber dans l'écueil d'ontologies de domaines totalement inutilisables par un outil d'aide au développement,
- Test-driven : il faut un moyen automatique de vérifier/spécifier (partiellement) que le logiciel produit fonctionne. Pas directement une spécification formelle car un utilisateur ne saura pas la produire, plutôt des tests partiels,
- Connaissances explicites : les connaissances utilisées pour automatiser partiellement telle ou telle étape du cycle de vie doivent absolument être explicitées et on doit pouvoir raisonner dessus. Ceci implique qu'elles soient formalisées rigoureusement. C'est le défi mais c'est aussi le levier. Si on sait un peu exprimer des transformations de programmes on raisonne mal dessus et on sait mal exprimer le contrôle de ces transformations.

Auto-adaptation des logiciels, l'incertitude en tant qu'intrus dans le cycle de vie – Laurence Duchien

La volonté d'accéder à l'information en continu et le fait que les technologies soient de plus en plus intégrées dans nos lieux de vie invitent à repenser la manière dont nous construisons, gérons et exécutons les logiciels. Pour que les logiciels deviennent flexibles, configurables ou reconfigurables et paramétrables tout en étant résistants et fiables, il est nécessaire de revisiter leur cycle de vie complet. En effet, ceux-ci doivent s'adapter en prenant en compte les changements de leurs contextes opérationnels, de leurs environnements ou encore de leurs propres caractéristiques. Dès lors, l'auto-adaptation devient un élément prépondérant des différentes phases de vie des logiciels. Même si ce sujet est déjà traité dans le cas de la mise en place de plates-formes d'exécution ad-hoc dans de nombreux domaines tels que les systèmes autonomes, les systèmes embarqués, les réseaux ad hoc ou mobiles, la robotique, l'intelligence ambiante, les réseaux de capteurs ou encore les architectures orientées services, il est nécessaire de visiter les différentes phases de conception des applications associées, c'est-à-dire la prise en compte de l'évolution à tout moment.

La gestion de cette fonctionnalité d'auto-adaptation reste un défi majeur dans le domaine de l'ingénierie du logiciel, ceci malgré les avancées récentes dans les différents domaines cités ci-dessus. Les défis à relever concernent essentiellement la représentation de l'incertitude dans la modélisation, l'expression des besoins, l'ingénierie et la validation de ces systèmes. En modélisation, différentes dimensions devront permettre la représentation des propriétés associées au raisonnement ayant lieu à l'exécution et la prise de décision pour mettre en œuvre l'auto-adaptation. Il s'agit par exemple de l'évolution, de la flexibilité, de la multiplicité des évolutions, de la dépendance de ces évolutions ou encore de leur fiabilité. On représentera alors la fréquence des changements, leur anticipation et leur impact. Les effets de ces changements pourront être définis en termes de criticité, de prédictibilité, *d'overhead* ou encore de résilience. Le défi pour l'expression des besoins concerne la définition de nouveaux langages capables de représenter l'incertain à un haut niveau d'abstraction et de le gérer. Au niveau de l'ingénierie, la boucle de contrôle devient une entité de première classe tout au long du processus de développement et d'exécution. Pour cela, il est nécessaire de modéliser le support qui permet la réalisation de cette boucle et de prendre ce modèle en compte dans la réalisation des applications. Cela devrait permettre une réification des propriétés du système et faciliter le raisonnement sur les propriétés du système. Finalement, la validation des systèmes auto-adaptables passe par l'identification dynamique des nouveaux contextes et donc la prise en compte de l'incertitude que ce soit en termes de modèles et de leur validation.

7. Méthodes et outils de vérification et validation

Langages, Types et Preuves [Castéran, Dubois 2010]

Le groupe LTP (Langages, Types et Preuves) [Castéran, Dubois 2010] regrette la diffusion limitée des formalismes, langages et outils permettant de produire des logiciels sûrs, malgré leur existence en nombre significatif et s'interrogent sur leur diffusion limitée. Sa première proposition est de faire passer les preuves dans le quotidien du développeur en permettant aux techniques existantes de passer à l'échelle. Pour cela, il préconise d'automatiser le plus possible les preuves en utilisant des prouveurs automatiques, de faire collaborer les techniques (test, model checking, analyse statique et preuves) qui sont complémentaires et, finalement, de réutiliser des preuves existantes pour passer à l'échelle. La seconde proposition concerne la rupture entre le semi-formel et le formel qui est, à son avis, trop grande. La réduction de cette rupture est possible si l'on restreint les formalismes utilisés dans les outils tels que B ou Coq à des domaines d'application où un langage spécifique facilite la lecture des spécifications tout en étant accompagné d'une sémantique formelle très précise. Cette approche permettrait une

jonction entre DSL (langages dédiés à un domaine) (voir section 3) et les outils de preuves formelles.

Méthodes Formelles pour le Développement Logiciel [Potet, aït Ameer, Laleau 2010]

Le groupe MFDL (Méthodes Formelles pour le Développement Logiciel) [Potet, aït Ameer, Laleau 2010] s'intéresse à une intégration maîtrisée et raisonnée des méthodes formelles dans un cycle de développement de systèmes à logiciel prépondérant. Il est alors envisageable d'appliquer différentes méthodes et approches de modélisation et de vérification dans un continuum, de l'analyse du système à la production du code sur un matériel spécifique, de bout en bout. Une traçabilité formelle entre les différents niveaux du cycle de vie d'un processus est également l'un des défis à relever. Cela devrait permettre une production de logiciels certifiés. Comme pour le groupe LTP, le groupe MFDL souligne également l'importance du passage à l'échelle dans l'utilisation des méthodes formelles, ici dans les processus industriels et l'importance d'un meilleur enseignement de ces approches aux futurs ingénieurs.

Méthodes de Test pour la Validation et la Vérification [Gotlieb, Groz 2010]

Finalement, le groupe MTV2 (Méthodes de Test pour la Validation et la Vérification) [Gotlieb, Groz 2010] souligne que la complexité des systèmes à logiciel prépondérant demande à ce que les processus et techniques de tests soient mieux définis, fiables et capables de passer à l'échelle. Trois défis sont proposés par ce groupe. Le premier consiste à mieux intégrer les tests dans les méthodes de développement, ceci dès les phases de modélisation et de manière continue et traçable jusqu'à la production du code. La co-évolution modèle-code est également un aspect à prendre en compte dans la génération de tests. Le second défi consiste à réconcilier test et fiabilité. La grande dépendance de fonctionnement des entreprises, et plus généralement de la société, vis-à-vis des logiciels rend nécessaire une évaluation quantitative de leur fiabilité. Les procédures de tests reposent sur l'évaluation des défauts de fonctionnement. Il est alors nécessaire d'établir un lien fort entre test et fiabilité du logiciel en étudiant les notions de couverture liées à des profils d'utilisation et d'évaluer des probabilités d'événements rares à partir de techniques de simulation. Le dernier défi consiste à prendre en compte les environnements nouveaux d'exécution, tels que les systèmes mobiles ou ubiquitaires, ou encore des systèmes hybrides. Les caractéristiques de ces environnements posent de nouveaux défis en matière de tests et demandent à être étudiés. Finalement, comme pour les deux groupes précédents, le groupe MTV2 pose le problème de l'enseignement du test, et de la transformation du métier d'ingénieur informatique en France. En effet, ce métier, avec l'apparition des développements off-shore, est maintenant amené à contrôler plus avant la qualité du logiciel développé en dehors de nos frontières, et donc à maîtriser les techniques de test.

8. Une analyse transverse des 7 familles thématiques

A la suite des présentations et des discussions qui ont eu lieu à Pau en Mars 2010 et à Paris en juillet 2010, plusieurs thèmes non identifiés en tant que tels sont revenus dans plus d'une présentation et ont suscité des débats. Nous en donnons ici une liste.

- **Continuum dans le développement** : les frontières entre différentes phases du développement du logiciel sont devenues floues. On va vers un continuum qui s'étend jusqu'à l'exécution du logiciel et à sa maintenance. Ce continuum va permettre d'enrichir de façon continue le logiciel et devra s'appuyer sur une traçabilité forte de l'analyse jusqu'à la conception du logiciel. Il est également à remarquer que l'introduction des propriétés extra-fonctionnelles devrait se faire au plus tôt dans le cycle de développement.
- **Approches agiles** : le développement du logiciel par des méthodes agiles avec des cycles de développement courts. La notion de méthode agile s'étend au-delà de la

programmation. Ce type de méthode s'appliquerait également dans les différentes étapes du cycle de vie du logiciel. Il s'agit plus d'une méthode alliant la rapidité et une intégration à la fois continue et fiable des différents éléments constitutifs du logiciel.

- **Connaissances et développement** : l'appui sur des connaissances issues des réseaux sociaux (Web 2.0), mais également des forums, mails, etc. devrait apporter une aide à la fois lors du développement et lors de l'évolution du logiciel. En effet, il s'agit ici de constituer les documentations, le suivi du logiciel, mais également une aide à la conception par les informations échangées entre les développeurs, mais également par les utilisateurs. Ces connaissances ont besoin d'être structurées, traitées et enrichies de façon continue.
- **Usages et utilisateurs** : l'utilisateur est au centre du dispositif du cycle de vie. Il est important de dialoguer avec lui tout au long de la vie du logiciel. Ce défi rejoint la notion de méthode agile. Il s'agit de proposer un dialogue constant dans les phases d'élaboration du logiciel avec l'utilisateur final. Une autre voie d'échange avec les utilisateurs finaux est l'apprentissage des usages du logiciel et son optimisation via des approches expérimentales fondées sur des traces.
- **Développement et exécution ouverts** : les environnements de développement et d'exécution deviennent plus ouverts. Ils sont évolutifs jusqu'à aller vers l'autonomie. Par autonomie, on pense à la supervision, mais également à l'auto-gestion avec l'auto-optimisation, l'auto-protection, l'auto-guérison, l'auto-test ou encore l'auto-adaptation.
- **Complexité et passage à l'échelle dans l'utilisation des approches formelles** : dans l'utilisation des méthodes formelles, que ce soit pour vérifier des propriétés, pour accompagner des processus de développement ou encore pour tester des parties logicielles, il devient important de prendre en compte la complexité des logiciels actuels et de permettre le passage à l'échelle.
- **Remodularisation ou langage pivot** : il semble important d'avoir des vues sur des objets existants, quelque soit l'étape du cycle de vie et de pouvoir manipuler ces objets selon cette vue, aussi bien pour faire évoluer le logiciel que pour l'analyser. Par exemple, avec les outils de preuves tels que Coq ou B, il serait intéressant de réaliser des vues dynamiques.
- **Contractualisation et responsabilité face aux utilisateurs du logiciel** : les concepteurs des logiciels devraient être conscients des enjeux des logiciels qu'ils développent. Une forme de contractualisation est à trouver vis-à-vis des utilisateurs pour améliorer la qualité des logiciels, mais aussi pour responsabiliser les concepteurs de logiciels.

9. Conclusion

La maîtrise du développement de logiciel est une technologie clé pour l'innovation et la performance économique d'un pays. Depuis sa création, l'informatique n'a pas cessé d'évoluer et de se révolutionner, tant dans le domaine du matériel que des paradigmes logiciels. Cette révolution permanente a permis à l'informatique de devenir moteur de progrès dans tous les domaines de l'activité humaine.

L'objet d'étude de la Science Informatique est un objet en mouvement, qui lève de nouveaux défis à chaque révolution, et remet en cause les solutions apportées aux défis du passé. Ces défis requièrent des avancées tant sur le plan fondamental que dans la mise en œuvre de résultats de la Recherche.

Dans ce document, nous avons essayé d'identifier plusieurs défis que des équipes du GDR GPL souhaitent relever dans les prochaines années. Ce document n'a pas l'ambition d'être

exhaustif, mais il nous semble représentatif de la diversité des compétences de la communauté française en GPL et de l'intérêt de soutenir ses recherches.

10. Références

[Anquetil et al. 2010] N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet (INRIA), R. Ducournau, R. Giroudeau, M. Huchard, J.C. König et D. Seriai (LIRMM / CNRS / Univ. Montpellier 2), Software (re)modularization : Fight against the structure erosion and migration préparation, Session défis, Journées du GDR GPL, Pau 2010

[Arévalo et al. 2010] Gabriela Arévalo (LIFIA / UNLP _ Argentine), Zeina Azmeh, Marianne Huchard, Chouki Tibermacine (LIRMM / CNRS / Univ. Montpellier 2), Christelle Urtado et Sylvain Vauttier (LGI2P / Ecole des Mines d'Alès) Component and Service Farms, Session défis, Journées du GDR GPL, Pau 2010

[Albert et al. 2010] Patrick Albert (IBM), Mireille Blay-Fornarino (I3S), Philippe Collet (I3S), Benoit Combemale (IRISA), Sophie Dupuy-Chessa (LIG), Agnès Front (LIG), Anthony Grost (ATOS), Philippe Lahire (I3S), Xavier Le Pallec (LIFL), Lionel Ledrich (ALTEN Nord), Thierry Nodenot (LIUPPA), Anne-Marie Pinna-Dery (I3S), et Stéphane Rusinek (Psitec), End-User Modelling, Session défis, Journées du GDR GPL, Pau 2010.

[Castéran, Dubois 2010] Pierre Castéran, Catherine Dubois, Proposition de défis – Groupe LTP, Appel à défis, Juin 2010.

[Consel 2010] Charles Consel (INRIA / Univ. Bordeaux) Towards Disappearing Languages, Session défis, Journées du GDR GPL, Pau 2010.

[Gotlieb, Groz 2010] Arnaud Gotlieb, Roland Groz, Test Logiciel à Grande Echelle, synthèse du groupe de travail MTV2 en réponse à l'appel à défi GDR-GPL, Appel à défis, Juin 2010.

[KBSE 1990] Knowledge-Based Software Engineering, in Avron Barr (Author), Paul R. Cohen (Author), Edward A. Feigenbaum (Editor) The Handbook of Artificial Intelligence, Addison-Wesley, 1990, vol 4, chap. XX.

[IP 1996] Charles Simonyi, Intentional Programming - Innovation in the Legacy Age Presented at IFIP WG 2.1 meeting, June 4, 1996

[MDA] <http://www.omg.org/mda/>

[Menaud et al. 2010] Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé Pierre Cointe, Rémi Douence et Mario Südholt (Ecole des Mines de Nantes / INRIA / LINA) Vers une réification de l'énergie dans le domaine du logiciel _ L'énergie comme ressource de première classe, Session défis, Journées du GDR GPL, Pau 2010.

[Potet, aït Ameur, Laleau 2010] Marie-Laure Potet, Yamine Aït-Ameur, Régine Laleau, Intégration maîtrisée et raisonnée des modèles et méthodes formelles dans un processus industriel de développement logiciel, Appel à défis, Juin 2010.

[Vignes 2010] Sylvie Vignes, Contribution à l'Ingénierie des Besoins spécifique à l'adaptation diffuse, Appel à défis, Juin 2010.

Défis pour le Génie Logiciel et la Programmation à l'échéance 2020

Ce document est un extrait des
« *Actes des deuxièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel* »,
Edités par Eric Cariou, Yves Ledru et Laurence Duchien,
Université de Pau
10 au 12 Mars 2010

Les actes des deuxièmes journées nationales sont disponibles sur le site web du GDR GPL

<http://gdr-gpl.cnrs.fr/>

Cet extrait regroupe les 5 textes qui ont été sélectionnés suite à notre appel à défis



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

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

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

Patrick Albert (IBM), Mireille Blay-Fornarino (I3S), Philippe Collet (I3S), Benoit Combemale
(IRISA), Sophie Dupuy-Chessa (LIG), Agnès Front (LIG), Anthony Grost (ATOS),
Philippe Lahire (I3S), Xavier Le Pallec (LIFL), Lionel Ledrich (ALTEN Nord), Thierry
Nodenot (LIUPPA), Anne-Marie Pinna-Dery (I3S) et Stéphane Rusinek (Psitec)
End-User Modelling285

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

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

Appel à Défis pour le Génie de la Programmation et du
Logiciel à échéance de 2020
Organisée dans le cadre des journées du GDR CNRS GPL
11 mars 2010
Pau

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du GPL? Cet appel vise à proposer une vision prospective de notre domaine et à souligner les points qui nous paraissent particulièrement cruciaux.

Ainsi, de nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique. Pour arriver à dégager de nouvelles visions dans notre domaine, des défis doivent être posés. L'objectif de cet appel est d'encourager tous les groupes du GDR GPL, mais aussi les équipes des laboratoires, des groupes informels ou des individus à proposer des défis dans leur domaine. Ces défis pourront ensuite déboucher sur la création de nouveaux réseaux, de création d'ateliers, de proposition de projets, etc.

Idéalement, un défi devra proposer une rupture par rapport à l'usage. Il pourra contenir une description du défi, les fondements scientifiques sur lesquels il repose, les verrous scientifiques et techniques à lever, les usages et impacts sociétaux qu'il adresse, ainsi qu'un ensemble de jalons qui correspondra aux étapes pour le réaliser et le valider et qui pourra servir d'agenda. Des domaines applicatifs pourront accompagner ce défi.

Un défi prendra la forme d'un texte de deux à quatre pages.

Les défis seront soumis sur le site <http://gdr-gpl.cnrs.fr>. Ils seront sélectionnés par le comité de programme en vue d'être inclus dans les actes des journées du GDR GPL et présentés au cours d'un panel lors des journées du GDR GPL du 8 au 12 mars 2010 à Pau.

Suite à ce panel, il est envisagé de publier ces défis sous la forme d'un livre ou d'un numéro spécial de revue. Des discussions sont en cours avec des éditeurs.

Dates importantes

- 10 janvier 2010 : soumission en ligne (2 à 4 pages)
- 25 janvier 2010 : notification aux auteurs
- 1 février 2010 : réception de la version à inclure dans les actes du GDR GPL
- 11 mars 2010 : panel aux journées du GDR à Pau

- 31 mai 2010 : soumission étendue en vue d'une publication dans une revue/livre (20 pages)
- 15 septembre 2010 : notification aux auteurs
- fin 2010 : publication

Comité de programme (provisoire)

- Laurence Duchien (LIFL/INRIA, Lille) (présidente du comité)
- Yamine Ait Ameer (LISI/ENSMA, Poitiers)
- Jean-Pierre Banâtre (INRIA, Rennes)
- Mireille Blay-Fornarino (I3S, Université de Nice-Sophia Antipolis)
- Pierre Castéran (LABRI, Bordeaux)
- Charles Consel (INRIA/LABRI, Bordeaux)
- Jean-Michel Couvreur (LIFO, Université d'Orléans)
- Catherine Dubois (CEDRIC, Evry)
- Hubert Dubois (CEA LIST, Saclay)
- Jean-Louis Giavitto (IBISC, Evry)
- Gaétan Hains (LACL, Créteil)
- Valérie Issarny (INRIA, Paris-Rocquencourt)
- Olga Kouchnarenko (LIFC, Université Franche-Comté)
- Philippe Lahire (I3S, Nice)
- Yves Ledru (LIG, Grenoble)
- Frédéric Loulergue (LIFO, Université d'Orléans)
- Mourad Oussalah (LINA, Nantes)
- Marie-Laure Potet (Vérimag, Grenoble)
- Salah Sadou (Valoria, Université Bretagne-Sud)
- Christel Seguin (ONERA, Toulouse)
- Mikal Ziane (LIP6, Université Paris Descartes)

Défis et table ronde pour le Génie de la Programmation et du Logiciel à échéance de 2020

L'omniprésence de l'informatique dans notre quotidien à l'échelle de l'embarqué et de l'intelligence ambiante, l'extension du web au niveau de la planète, mais également dans les objets du quotidien, le développement de grandes infrastructures de calcul ou des centres de traitement de grandes masses de données soulèvent de nombreuses questions pour le génie de la programmation et du logiciel. Parmi ces questions, quelles sont celles qui correspondent à des défis que devront relever les chercheurs dans le domaine du génie de la programmation et du logiciel à échéance de 5 à 10 ans ?

De nouveaux paradigmes, de nouveaux langages, de nouvelles approches de modélisation, de vérification, de tests et de nouveaux outils dans le domaine de la programmation et du logiciel devraient voir le jour dans les 5 à 10 ans à venir, que ce soit pour faciliter la vie des concepteurs de logiciel, pour modéliser et fiabiliser les logiciels ou encore pour devancer l'évolution technologique, mais également pour prendre en compte de nouveaux enjeux de société tels que le développement durable et les économies d'énergie.

Les cinq défis seront présentés lors de deux sessions. Ils portent sur la (re)modularisation du logiciel, la mise en place de fermes de composants et de services, l'évaporation des langages, la modélisation pour l'utilisateur final et la réification de l'énergie au niveau des systèmes et des langages. Une table ronde permettra ensuite de débattre de ces sujets. Lors de cette table ronde, Bertrand Braunschweig fera le point sur la perception de notre domaine au sein de l'ANR.

Laurence DUCHIEN

Software (re)modularization: Fight against the structure erosion and migration preparation

N. Anquetil, S. Denier, S. Ducasse, J. Laval, D. Pollet

RMoD INRIA

stephane.ducasse@inria.fr

R. Ducournau, R. Giroudeau, M. Huchard, J.C. König, D. Serial

LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)

huchard@lirmm.fr

I. CONTEXT

Software systems, and in particular, Object-Oriented systems are models of the real world that manipulate representations of its entities through models of its processes. The real world is not static: new laws are created, concurrents offer new functionalities, users have renewed expectation toward what a computer should offer them, memory constraints are added, etc. As a result, software systems must be continuously updated or face the risk of becoming gradually out-dated and irrelevant [34]. In the meantime, details and multiple abstraction levels result in a high level of complexity, and completely analyzing real software systems is impractical. For example, the Windows operating system consists of more than 60 millions lines of code (500,000 pages printed double-face, about 16 times the Encyclopedia Universalis). Maintaining such large applications is a trade-off between having to change a model that nobody can understand in details and limiting the impact of possible changes. Beyond maintenance, a good structure gives to the software systems good qualities for migration towards modern paradigms as web services or components, and the problem of architecture extraction is very close to the classical remodularization problem.

II. A SIGNIFICANT PROBLEM

Most of the effort while developing and maintaining a software system is spent in supporting its evolution [50]. It is well-known that up to 80% of the total cost of software development project is spent in maintenance and evolution of existing applications [15], [19]. Large IT applications including applications running central and critical business (e.g., command tracking, banking, railway) have to run and evolve over decades.

Such situations are crystallized in the following two laws of Lehman and Belady. These laws, known as the laws of software evolution, stress the fact that software *must* continuously evolve to stay useful and that this evolution is accompanied by an increase of complexity.

Continuous Changes. *“an E-type program—i.e., a software system that solves a problem or imple-*

ments a computer application in the real world—that is used must be continually adapted else it becomes progressively less satisfactory” [34]

Increasing Complexity. *“As a program is evolved its complexity increases unless work is done to maintain or reduce it.”*[34]

From a business perspective, maintenance is mandatory and vital. It is worth to note that the Year 2000 Bug was also revealing some business occasions. For example in France SOPRA which was mainly an SSII, has since the year 2000 developed its TMA (Tierce Maintenance Applicative) branch to get into the maintenance business. SOPRA TMA now represents 2500 developers working on maintenance on a total of 11000 developers.

If many organizations use third party software (e.g. COTS) which they don't have to maintain, this is less true for their core business applications where their market differential must be implemented in the applications that help running day to day activities. As highlighted by the law of Lehman and Belady, these in-house applications must evolve and in this process will extend far beyond their initial structure, in many independent (and sometimes even conflicting) directions. After some time, it becomes mandatory to restructure the application to federate these evolutions inside a more general and renewed architecture to foster possible future evolutions.

Supporting evolution and prepare migration of applications will be *always* mandatory. Different programming paradigms have been invented to cope with changes: late-binding, the cornerstone of object-oriented programming, is a typical illustration. But we think that no paradigm will eradicate the need for evolution and changes and that the only possible approach is to guide evolution or to repair the damages caused by an inevitable erosion.

III. THE FAILURE OF CURRENT SOLUTIONS

Whereas software (re-)modularization is a relatively old research field in the context of C or Cobol, it is still really important and requires *innovative approaches* to deal with the complexity of modern systems especially those

developed in OOP languages. Our analysis — also confirmed in the recent literature (e.g., [1], [7], [47]) — is that this failure is a direct consequence of: (1) the *complexity of the manipulated concepts and the variety of modular abstractions* (subsystems, packages, classes, class hierarchies, late-binding, aspects, various import relationships....) as well as (2) a monolithic approach: the use of *one* type of algorithms (clustering) and *one* kind of system representation (software components interactions found in the source code).

A. Modular Abstractions

Modular constructs have been the focus of a large body of research. Here we give a non-exhaustive list. A lot of work is currently underway in the context of aspect-oriented programming. Module and package systems have been the focus on a large amount of work. The recent work on Units [22], Jiazzi [42], Mixjuice [28], MJ [14], JAM [52], mixin layers [48] shows that this topic is a crucial research area. Bergel *et al.* conducted a survey on modular language constructs that reflects such a diversity [6].

Modular constructs have also been considered at a lower level than classes, e.g., with mixins [9], traits [17]. These constructs denote, here, sets of properties that somewhat represent the *differentia* in the Aristotelian *genus-differentia* definition. Overall, remodularization must address both modular construct levels, by clustering related properties for defining classes and related classes for defining packages. Recursively nested class models have also been proposed [20], [44]—however they cannot be considered as long as the point with non-recursive models is not fixed.

Component-based software approach proposed to build software systems by assembling prefabricated reusable components [54]. Assembly consists in connecting matching interfaces of the components: in a connection, a required interface (describing the services a component needs) is connected to a provided interface (describing the services another component offers). Extracting a component architecture from an object-oriented software has many in common with remodularization because classes need to be grouped based on their dependencies to form the components [11].

Current Issues – *The class notion is the only universally accepted modular abstraction. Higher and lower level abstractions are often still in flux. Different languages use different concepts such as modules, packages, namespaces. Hence, current issues involve both identifying adequate abstractions and adapting remodularization algorithms to the various alternative abstractions. Moreover, assessing the modularity of software requires specific tools (e.g., metrics, visualization) that must be adapted to each modular abstraction.*

B. Remodularization Approaches

Class hierarchy analysis: Class hierarchy analysis has been largely investigated, for restructuring purposes or find-

ing separate concerns (aspects, traits). As it has been shown in [26], most of the refactorization approaches use explicitly or implicitly substructures of those obtained by Formal Concept Analysis (FCA).

The application of clustering algorithms for software remodularization has been intensively studied [2]. Thousands of experiments were conducted to compare different clustering algorithms, different representation schemes and different coupling metrics between files. Although the experiments used procedural systems, many conclusions may be applied to OO systems as well. The extraction of class views based on Formal Concept Analysis has been proposed in [3]. They evaluated how FCA supports the identification of traits in existing hierarchies [8], [35]. Godin [24] developed FCA algorithms to infer a non-redundant form for implementation and interface hierarchies and carried out experiments on several Smalltalk applications. For dealing with UML models, Relational Concept Analysis, an extension of FCA, takes the relations into account [27]. Other approaches analyze class hierarchies using access or usage information. [49], [51] analyze the *usage* of the hierarchy by a set of client programs. Mining aspects has been considered in the context of FCA [10].

Current Issues – *Factorization is by nature a combinatorial process. Recent studies [21] show that Relational Concept Analysis, applied to rich UML descriptions including references between concepts, produces a huge number of artefacts which is quite impossible to analyze by hand. Execution time can become a problem for large size software, but the actual difficulty is the result size.*

Other modular construct discovery: Clustering approaches are, by far, the preferred algorithmic approach to the problem. They have been proposed to identify modules in applications that are not specifically object-oriented (e.g. [29], [30], [38], [43]).

Finding components in object-oriented software is proposed in [12]. Simulated annealing is used to gather classes into components by optimizing metrics measuring cohesion and coupling. The problem has similarity with package mining.

It is a well-known practice to layer applications with bottom layers being more stable than top layers [41]. Until now few works have been done in practice to identify layers: Mudpie [55] is a first cut at identifying cycles between packages as well as package groups potentially representing layers. DSM (dependency structure matrix) [53], [46] are adapted for such a task but there is a lack of detail information. From the side of remodularization algorithms, a lot of them were defined for procedural languages [31]. However object-oriented programming languages bring some specific problems linked with late-binding and the fact that a package does not have to be systematically cohesive since it can be an extension of another one [18], [57].

Current Issues – *These approaches are often not customized for object-oriented applications. Existing solutions propose modules at a very low level of abstraction that do not reduce enough the size of the system comprehension problem. Solutions that may offer larger (potentially more abstract) modules, result in modules that have no meaning for the software engineers.*

C. Techniques for module assessment

Software Metrics: Re-modularization of software systems is geared toward producing highly cohesive and loosely coupled modules. Many different cohesion/coupling metrics were proposed (including a study by [2]). In the more specific case of object-oriented programming, assessing cohesion and coupling has been the focus of several metrics. However their success is rather mitigated as the number of critics raised. For example, LCOM [13] has been highly criticized [4]. Other approaches have been proposed such as RFC and CBO [13] to assess coupling between classes. However, many other metrics have not been the subject of careful analysis such as Data Abstraction Coupling (DAC) and Message Passing Coupling (MPC) [5], or some metrics are not clearly specified (MCX, CCO, CCP, CRE) [36]. New cohesion measures were proposed [40], [45] taking class usage into account. The Cohesion/Coupling dogma, however, started to receive critics in recent times [1], [47]. People argue that software engineers do not base clustering on this criterion but rather use more semantical approaches.

Software Visualization: There is a significant effort to create efficient software visualizations to support the understanding and analyses of applications [32], [37], [39], [56]. Lanza and Ducasse worked on system level understanding combining metrics and visualization [33] and class understanding support [16].

Current Issues – *Existing cohesion and coupling metric resulting values are difficult to map back to the actual situation, they lead to packages that seem artificial and are not understood by experts of the systems. There is a lack for package cohesion and coupling software metrics in presence of late-binding promoted by object-oriented programming. There is a need for program visualization to support the understanding of packages and procedural code. In addition, there is a need for new metrics that would yield more “natural” packages.*

IV. SCIENTIFIC CHALLENGES AND TRACKS OF RESEARCH

The main scientific challenges are as follows.

Abstraction Diversity: One of the major problems to solve when tackling modularization of object-oriented systems is the choice of good abstractions and the appropriate relations between them.

Complementary Remodularization Algorithms: There is a need for a global remodularization infrastructure in terms of analyses (algorithms, information presentation, metrics) that can take into account the diversity of the abstractions in presence (different module semantics, different abstractions and relationships including different levels, functions, classes, packages, etc.).

Complexity and approximation: In the point of view of graph theory, the central problem seems to be close to the classic k -cuts problem [23], [25]. Nevertheless, we must add several new criteria, like the *quality* of the proposed solution. The first step of the research will be the characterization of an optimization criterion. We also think useful to study and analyse the sensibility of the problem with respect to the operations: add/delete of vertices/edges. It is a major challenge to be able to propose robust approximations.

Scalability: Computational complexity of algorithms has a limit, already known for Formal Concept Analysis (FCA/RCA) and foreseeable for exact methods. Checking the scalability of these algorithms is thus an additional challenge. Another issue is the combinatorial explosion which may occur in the modularization results. Because of the size of current applications, presenting these results to the engineers and guiding them to take a decision is an additional challenge.

Reengineer inputs and quality of the solution: Engineers should drive the modularization. Fully automated approaches are applicable only to a very limited context. In reality, external constraints have to be specified and taken into account by the modularization algorithms (such as the inclusion of a class in a specific package). Software engineers should guide the process possibly confronted to different solutions and their relative impacts. Often favoring minimum impact on existing code has to be considered. Finally the quality of the resulting modularizations has to be taken into account.

As a conclusion: The problems of software evolution are many and varied, in this proposal we plan to consider one of these problems: software modularization. We think urgent to drive such a complete study of the problem, both “vertically” by studying all the aspects of the modularization problem (modeling of the software, modularization quality metrics, modularization algorithms, presentation of the results), and, “horizontally”, by considering different modularization approaches. The solution will not apply one single method, but a combination of various skills in different research domains. Such a research would also be guided by platforms for testing ideas on real-world applications.

REFERENCES

- [1] F. B. Abreu and M. Goulao. Coupling and cohesion as modularization drivers: are we being over-persuaded? In *Fifth European Conference on Software Maintenance and Reengineering*, pages 47–57, Mar. 2001.

- [2] N. Anquetil and T. Lethbridge. Comparative study of clustering algorithms and abstract representations for software modularization. *IEEE Proceedings - Software*, 150(3):185–201, 2003.
- [3] G. Arévalo, S. Ducasse, and O. Nierstrasz. X-Ray views: Understanding the internals of classes. In *Proceedings of 18th Conference on Automated Software Engineering (ASE'03)*, pages 267–270. IEEE Computer Society, Oct. 2003. Short paper.
- [4] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):491–506, 2004.
- [5] J. Bansiya, L. Etzkorn, C. Davis, and W. Li. A class cohesion metric for object-oriented designs. *Journal of Object-Oriented Programming*, 11(8):47–52, Jan. 1999.
- [6] A. Bergel, S. Ducasse, and O. Nierstrasz. Analyzing module diversity. *Journal of Universal Computer Science*, 11(10):1613–1644, Nov. 2005.
- [7] P. Bhatia and Y. Singh. Quantification criteria for optimization of modules in oo design. In *Software Engineering Research and Practice*, pages 972–979, 2006.
- [8] A. P. Black, N. Schärli, and S. Ducasse. Applying traits to the Smalltalk collection hierarchy. In *Proceedings of 17th International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*, volume 38, pages 47–64, Oct. 2003.
- [9] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '90, ACM SIGPLAN Notices*, volume 25, pages 303–311, Oct. 1990.
- [10] M. Ceccato, M. Marin, K. Mens, L. Moonen, P. Tonella, and T. Tourwé. Applying and combining three different aspect mining techniques. *Software Quality Journal*, 14(3):209–231, 2006.
- [11] S. Chardigny, A. Seriai, M. Oussalah, and D. Tamzalit. Extraction of component-based architecture from object-oriented systems. In *WICSA*, pages 285–288. IEEE Computer Society, 2008.
- [12] S. Chardigny, A. Seriai, M. C. Oussalah, and D. Tamzalit. Extraction d'Architecture à Base de Composants d'un Système Orienté Objet. In *INFORSID*, pages 487–502, 2007.
- [13] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.
- [14] J. Corwin, D. F. Bacon, D. Grove, and C. Murthy. MJ: a rational module system for Java and its applications. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 241–254. ACM Press, 2003.
- [15] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, 1995.
- [16] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.
- [17] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A mechanism for fine-grained reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, Mar. 2006.
- [18] A. Dunsmore, M. Roper, and M. Wood. Object-oriented inspection in the face of delocalisation. In *Proceedings of ICSE '00 (22nd International Conference on Software Engineering)*, pages 467–476. ACM Press, 2000.
- [19] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.
- [20] E. Ernst. Higher-order hierarchies. In *Proceedings European Conference on Object-Oriented Programming (ECOOP 2003)*, LNCS, pages 303–329, Heidelberg, July 2003. Springer Verlag.
- [21] J.-R. Falleri, M. Huchard, and C. Nebut. A generic approach for class model normalization (short paper). In *ASE 2008: 23th IEEE/ACM International Conference on Automated Software Engineering*, 2008.
- [22] M. Flatt and M. Felleisen. Units: Cool modules for hot languages. In *Proceedings of PLDI '98 Conference on Programming Language Design and Implementation*, pages 236–248. ACM Press, 1998.
- [23] A. Freize and M. Jerrum. Improved approximation algorithm for max $-k$ -cut and max bisection. *Algorithmica*, 18:67–81, 1997.
- [24] R. Godin, H. Mili, G. W. Mineau, R. Missaoui, A. Arfi, and T.-T. Chau. Design of Class Hierarchies based on Concept (Galois) Lattices. *Theory and Application of Object Systems*, 4(2):117–134, 1998.
- [25] O. Goldschmidt and D. Hochbaum. Polynomial algorithm for the k -cut problem. In I. C. Society, editor, *Proc. 29th Annual IEEE Symposium on Foundations of Computer Science*, pages 444–451, 1988.
- [26] M. Huchard, H. Dicky, and H. Leblanc. Galois lattice as a framework to specify building class hierarchies algorithms. *ITA*, 34(6):521–548, 2000.
- [27] M. Huchard, M. R. Hacene, C. Roume, and P. Valtchev. Relational concept discovery in structured datasets. *Ann. Math. Artif. Intell.*, 49(1-4):39–76, 2007.
- [28] Y. Ichisugi and A. Tanaka. Difference-based modules: A class independent module mechanism. In *Proceedings ECOOP 2002*, volume 2374 of LNCS, Malaga, Spain, June 2002. Springer Verlag.
- [29] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data*. Prentice Hall, Englewood Cliffs, 1988.
- [30] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

- [31] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. PhD thesis, Universität Stuttgart, 2000.
- [32] G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 214–223, New York, NY, USA, 2005. ACM.
- [33] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.
- [34] M. Lehman. Laws of software evolution revisited. In *European Workshop on Software Process Technology*, pages 108–124, Berlin, 1996. Springer.
- [35] A. Lienhard, S. Ducasse, and G. Arévalo. Identifying traits with formal concept analysis. In *Proceedings of 20th Conference on Automated Software Engineering (ASE'05)*, pages 66–75. IEEE Computer Society, Nov. 2005.
- [36] M. Lorenz and J. Kidd. *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall, 1994.
- [37] J. I. Maletic and A. Marcus. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 103–112, May 2001.
- [38] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A Clustering Tool for the Recovery and Maintenance of Software System Structures. In *Proceedings of ICSM '99 (International Conference on Software Maintenance)*, Oxford, England, 1999. IEEE Computer Society Press.
- [39] A. Marcus, L. Feng, and J. I. Maletic. 3D representations for software visualization. In *Proceedings of the ACM Symposium on Software Visualization*, pages 27–ff. IEEE, 2003.
- [40] A. Marcus and D. Poshyvanyk. The conceptual cohesion of classes. In *Proceedings International Conference on Software Maintenance (ICSM 2005)*, pages 133–142, Los Alamitos CA, 2005. IEEE Computer Society Press.
- [41] R. C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice-Hall, 2002.
- [42] S. McDirmid, M. Flatt, and W. Hsieh. Jiazi: New age components for old fashioned Java. In *Proceedings OOPSLA 2001, ACM SIGPLAN Notices*, pages 211–222, Oct. 2001.
- [43] B. S. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, 2006.
- [44] N. Nystrom, X. Qi, and A. C. Myers. J&: nested intersection for scalable software composition. In *OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 21–36, New York, NY, USA, 2006. ACM.
- [45] L. Ponisio and O. Nierstrasz. Using context information to re-architect a system. In *Proceedings of the 3rd Software Measurement European Forum 2006 (SMEF'06)*, pages 91–103, 2006.
- [46] N. Sangal, E. Jordan, V. Sinha, and D. Jackson. Using dependency models to manage complex software architecture. In *Proceedings of OOPSLA'05*, pages 167–176, 2005.
- [47] R. Sindhgatta and K. Pooloth. Identifying software decompositions by applying transaction clustering on source code. In *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference*, pages 317–326, Washington, DC, USA, 2007. IEEE Computer Society.
- [48] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Soft. Eng. Meth.*, 11(2):215–255, 2002.
- [49] G. Snelling and F. Tip. Reengineering Class Hierarchies using Concept Analysis. In *ACM Trans. Programming Languages and Systems*, 1998.
- [50] I. Sommerville. *Software Engineering*. Addison Wesley, fifth edition, 1996.
- [51] M. Streckenbach and G. Snelling. Refactoring class hierarchies with KABA. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*, pages 315–330, New York, NY, USA, 2004. ACM Press.
- [52] R. Strniša, P. Sewell, and M. Parkinson. The java module system: core design and semantic definition. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications*, pages 499–514, New York, NY, USA, 2007. ACM.
- [53] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE 2001*, 2001.
- [54] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [55] D. Vainsencher. Mudpie: layers in the ball of mud. *Computer Languages, Systems & Structures*, 30(1-2):5–19, 2004.
- [56] R. Wetzel and M. Lanza. Program comprehension through software habitability. In *Proceedings of ICPC 2007 (15th International Conference on Program Comprehension)*, pages 231–240. IEEE CS Press, 2007.
- [57] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, SE-18(12):1038–1044, Dec. 1992.

Component and Service Farms

Gabriela Arévalo

LIFIA - Facultad de Informática (UNLP) - La Plata (Argentina)
garevalo@sol.info.unlp.edu.ar

Zeina Azmeh, Marianne Huchard, Chouki Tibermacine

LIRMM - CNRS UMR 5506 - Université de Montpellier II - Montpellier (France)
{azmeh, huchard, tibermacin}@lirmm.fr

Christelle Urtado, Sylvain Vauttier

LGI2P - Ecole des Mines d'Alès - Nîmes (France)
{Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

I. CONTEXT

Software components and web services are software building blocks that are used in the composition of modern software applications. They both provide functionalities that require to be advertised by registries in order to be discovered and reused during software building processes¹ [1], [2], [3], [4], [5]. Building or evolving existing software implies assembling software components. This task is not trivial because it requires to select the adequate component or service that provides some part of the desired application functionality and connects easily (with minimum adaptations) to other selected components.

Within this context, we identified two main issues: (1) finding appropriate components from huge databases, and (2) creating and maintaining distributed applications.

Finding appropriate components from huge databases:

A huge number of components already exist. For example, the Seekda² web service search engine has a catalog of more than 28.000 references, and the OW2 consortium³ groups around 40 open-source projects in the domain of middleware technology. In Seekda, functionalities are rather directly exposed, but in projects, components are sometimes buried into application code repositories and should be properly extracted before being publicly advertised. However, in both cases the task of finding and adapting an appropriate component is hard and time-consuming, and this will surely worsen when more components become available.

Creating and maintaining distributed applications:

There is a growing need for being able to create and maintain distributed applications assembled from third party components as network, middleware and deployment technologies now are mature enough. Reduced cost, increased quality and decreased dependence to a given provider also

contribute to popularize this trend. Paradigms that are based on this principle are numerous: component-based development, web 2.0, mashups, cloud computing, software as a service, etc. State-of-the-practice technologies such as OSGi⁴, that enables the deployment and redeployment of software (for example, in remotely administered internet boxes), or upcoming technologies such as Google Chrome OS⁵, that aims to put web-apps at the center of net-books' operating system, also illustrate this trend.

II. INNOVATIVE REGISTRIES

In this context, it is necessary to design innovative software component registries to advertise components and assist users when they need to search, select, adapt and connect a component to others [6], [7]. We even could think of the automation of these tasks as much as possible, in order to adapt to open and dynamic contexts (remotely administered embedded devices, pervasive computing, open and extensible applications, mobile computing, etc.).

Then, the challenge consists in proposing an online architecture for a component service registry, with two functionalities (a) gives efficient access to adequate components, and (b) provides life-cycle long support to developers and applications (in automatic mode). This registry will be a platform for developers to share their knowledge and experiences on the components they use (what runs and what does not, what are ideal contexts for running some given component, what adaptations have already been performed or tested on some component, what are user ratings on components, etc.), and improve development efficiency as well as running software reliability or liveliness.

As an extension of the component search engine, we think of a *component farm*, where components could be either physically located (component repository model) or solely referenced in an adequately organized component

¹In the following, we will often use the term *components* as a generic name for both web services and software components.

²<http://webservices.seekda.com/>

³<http://www.ow2.org/>

⁴<http://www.osgi.org>

⁵<http://googleblog.blogspot.com/2009/07/introducing-google-chrome-os.html>

advertisement directory). The component farm (*cf.* Fig. 1) should offer multiple views on components and tailored efficient retrieval mechanisms.

Developed software applications that use components from the farm could either simply use the farm's public facilities or register to benefit from increased community-related capabilities. In the latter case, the added value would be provided by the exchange of component-related data between registered software applications and the farm, in both directions:

- *from software applications to the farm.* Applications could register components developed for their own purposes into the farm directory or share usage information on components they have previously retrieved from the farm, etc. This means that developers should contribute as in collaborative web.
- *from the farm to software applications.* The farm should provide several operations supported by efficient tools for searching and selecting components, providing usage feedback on components, informing the developer of other developers' experiences regarding new products, found problems with components, problem solving issues, possible adaptations, etc.

Our proposal focuses on two challenging issues, each of which is developed in one of the following sections: (1) the adequacy and efficiency of the organization of the component directory, (2) the proposal of tailored support to applications.

III. EFFICIENT ORGANIZATION

Many components exist but are not always available in public servers/directories. Providing abstract views and adequate services, a global organization could help to stimulate their sharing and be profitable increasing cross-fertilization between projects. The first aim of the registry is to efficiently organize components for several development and maintenance tasks. The components can be physically contained in the repository or just accessible from the directory, through hypertext links, from other external locations. An efficient organization is based on an efficient classification of the components. For example, some approaches [8], [9], [10], [11], [12], [13] have already investigated formal concept analysis (FCA) [14], [15], an approach that rigorously classifies data in structures that have strong mathematical properties. Other approaches should nevertheless be studied.

Several aspects can be used to classify components: syntactical, semantic and pragmatic [16], [17]. All of them could be used complementarily. At the syntactical level, component external descriptions include ports, interfaces, functionality signatures and parameter types. At the semantic level, components can be documented (with plain or structured text, with interaction protocols, etc.), described by keywords (either normalized through ontologies, for example, or not), and by any information that conveys the meaning

of the component (for what purpose it has been developed, by whom, etc.). At the pragmatic level, components are documented by information that provides feedback from its usage, in assemblies, choreographies or orchestrations: which functionalities are used, which functionalities are never used, how components are connected (with which adaptations), good and bad experiences, etc.

IV. APPLICATION SUPPORT

The second aim of the registry is software application support. Applications using the components from the registry are invited to register so as the directory can receive feedback. Thus, the directory stores information on which application uses which components and documentation about this use, in order to share usage information among developers.

The registry memorizes the manual or automatic adaptations that are necessary when a faulty component is replaced by another. This information is used to optimize future replacements and is capitalized to be used by other registered applications.

The registry prepares backups (sets of possible substitutes) for each used component in order to ensure rapid repairing of a software application in case one of its components fails [18]. These backups are organized in small classifications for efficiency purposes. They can be used manually or sometimes automatically in restricted cases where either exact matching between the used component and its potential substitute exists, or automatic adaptations are known and can be applied. These backups can be stored inside the repository or uploaded on demand by applications. Backups can also be used by designers to make the software evolve when the application designer wants to add extra functionalities or improve the software quality attributes.

Candidate backup components and possible adaptations are dynamically updated, as components become unavailable or are upgraded. Software applications that use them are also dynamically informed of these updates.

V. STARTING POINT

In previous work, we have started to imagine partial solutions to the issues identified here.

- We have developed an automated process for classifying components from their external descriptions. This process is based on type-theory (we only use syntactic information) and uses FCA to iteratively build lattices that provide functionality signature classifications, interface classifications and component classifications [8], [9], [10].
- We have prototyped the CoCoLa tool [10] that implements the aforementioned process. Thanks to a pivot meta-model, component descriptions from various formats are translated into comparable models (instances of the common meta-model). These descriptions are then processed to build context tables and lattices.

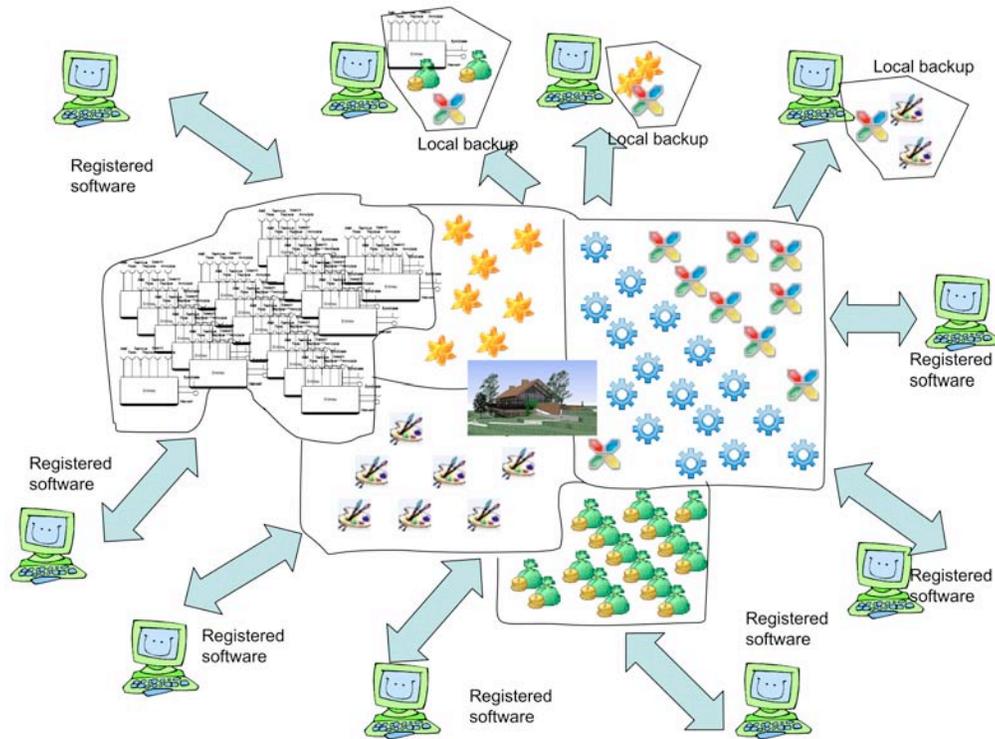


Figure 1. Overview of the component farm

Experiments have been run on the Dream⁶ component library (that comes from a real-world component-based framework). They show the feasibility of our approach as the produced lattices enables us to identify possible component substitutions and provides a readable component classification.

- We also have proposed an approach based on formal concept analysis (FCA) for classifying web services [19], [18]. A web service lattice reveals the invisible relations between web services in a certain domain, showing the services that are able to replace other ones. This facilitates service browsing, selection and identification of possible substitutions. We explained how to exploit the resulting lattices to build web services orchestrations and support them with backup services.

VI. MAIN CHALLENGES

There still are numerous challenges to overcome in order to build such a component farm:

- combine syntactical, semantic and pragmatic views of components in order to be able to efficiently search and select appropriated components (pertinence of the classification).

- automate component indexing and classification as well as feedback collection in order not to put the burden on component developers or component users. This task could need to exploit automatically all the available semantic information available (such as functionality names, documentations, interaction protocols, etc.) using text mining techniques [20].
- try and capitalize coarser grained software parts by classifying whole component assemblies,
- think about building techniques and browsing techniques for each of the provided classifications but also inter-classifications.
- propose replacement scenarios and tools,
- define the services that would motivate users in collaboratively providing usage feedback information,
- find means to conduct early experimentations of the component farm. This is not easy as very few component directories are available online, as for now.

As a response to the globalization challenge, software engineering has the capability to positively react in providing software component catalogs, thus increasing software quality while decreasing costs and time-to-market. As an immaterial product market-place, the software component industry benefits from great assets: software components must not be stocked, they can be transported at no cost and instantaneously, they are not perishable. To take advantage of this, efforts should be focused not only on development,

⁶<http://dream.ow2.org/>

as traditionally done, but also on better diffusion, sharing, deployment and maintenance. We believe that component farms could contribute to this objective.

REFERENCES

- [1] W. Hoschek, “The web service discovery architecture,” in *Int’l. IEEE/ACM Supercomputing Conference (SC 2002)*, I. C. S. Press, Ed., 2002.
- [2] OMG, “Trading Object Service Specification (TOSS) v1.0,” 2000, <http://www.omg.org/cgi-bin/doc?formal/2000-06-27>.
- [3] *ebXML Registry Services Specification (RS) v3.0*, <http://www.oasis-open.org/>, May 2005.
- [4] L. Clement, A. Hately, C. von Riegen, and T. Rogers, “Uddi version 3.0.2. uddi spec technical committee draft , dated 20041019. http://uddi.org/pubs/uddi_v3.htm,” Tech. Rep. [Online]. Available: <http://uddi.org/pubs/uddiv3.htm>
- [5] Information Technology Open Distributed Processing, “ODP Trading Function Specification ISO/IEC 13235-1:1998(E),” December 1998, http://webstore.iec.ch/preview/info_isoiec13235-1%7Bed1.0%7Den.pdf.
- [6] L. Iribarne, J. M. Troya, and A. Vallecillo, “A trading service for COTS components,” *The Computer Journal*, vol. 47, no. 3, pp. 342–357, 2004.
- [7] S. Dustdar and M. Treiber, “A view based analysis on web service registries,” *Distributed and Parallel Databases*, vol. 18, pp. 147–171, 2005.
- [8] G. Arévalo, N. Desnos, M. Huchard, C. Urtado, and S. Vauttier, “Precalculating component interface compatibility using FCA,” in *Proceedings of the 5th international conference on Concept Lattices and their Applications (CLA 2007)*, *CEUR Workshop Proceedings Vol. 331*, J. Diatta, P. Eklund, and M. Liquière, Eds., Montpellier, France, October 2007, pp. 241–252. [Online]. Available: <http://ceur-ws.org/Vol-331/>
- [9] —, “FCA-based service classification to dynamically build efficient software component directories,” *Int. Journ. of General Systems*, vol. 38, no. 4, pp. 427–453, May 2009.
- [10] N. A. Aboud, G. Arévalo, J.-R. Falleri, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, “Automated architectural component classification using concept lattices,” in *In proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture 2009 (WICSA/ECISA’09)*. Cambridge, UK: IEEE Computer Society Press, September 2009.
- [11] A. M. Zaremski and J. M. Wing, “Specification matching of software components,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 4, pp. 333–369, 1997.
- [12] B. Fischer, “Specification-based browsing of software component libraries,” in *Proc. of the 13th IEEE int. conf. on Automated Software Engineering (ASE’98)*, 1998, pp. 74–83.
- [13] B. Sigonneau and O. Ridoux, “Indexation multiple et automatisée de composants logiciels orientés objet,” in *AFADL — Approches Formelles dans l’Assistance au Développement de Logiciels*, J. Julliand, Ed. Besançon, France: RTSI, Lavoisier, Juin 2004.
- [14] M. Barbut and B. Monjardet, *Ordre et Classification*. Hachette, 1970.
- [15] B. Ganter and R. Wille, *Formal Concept Analysis: Mathematical Foundations*. Springer, 1999.
- [16] M. Á. Corella and P. Castells, “Semi-automatic semantic-based web service classification,” in *Business Process Management Workshops*, ser. LNCS 4103, J. Eder and S. Dustdar, Eds. Springer, 2006, pp. 459–470.
- [17] M. Bruno, G. Canfora, M. D. Penta, and R. Scognamiglio, “An approach to support web service classification and annotation,” in *Proc. of the IEEE Int. Conf. on e-Technology, e-Commerce and e-Service (EEE’05)*, 2005, pp. 138–143.
- [18] Z. Azmeh, M. Huchard, C. Tibermacine, C. Urtado, and S. Vauttier, “Using concept lattices to support web service compositions with backup services,” in *To appear in Proceedings of the 5th International Conference on Internet and Web Applications and Services (ICIW 2010)*, Barcelona, Spain, May 2010.
- [19] —, “WSPAB: A tool for automatic classification & selection of web services using formal concept analysis,” in *Proceedings of the 6th IEEE European Conference on Web Services (ECOWS 2008)*. Dublin, Ireland: IEEE, November 2008, pp. 31–40.
- [20] J.-R. Falleri, Z. Azmeh, M. Huchard, and C. Tibermacine, “Automatic tag identification in web service descriptions,” in *To appear in proceedings of the International Conference on Web Information Systems and Technology (WEBIST’10)*, Valencia, Spain, April 2010.

End-User Modelling

Albert Patrick (*IBM, Paris*), Blay-Fornarino Mireille (*I3S, Sophia-Antipolis*), Collet Philippe (*I3S, Sophia-Antipolis*), Combemale Benoit (*IRISA, Rennes*), Dupuy-Chessa Sophie (*LIG, Grenoble*), Front Agnès (*LIG, Grenoble*), Gros Anthony (*ATOS, Lille*), Lahire Philippe (*I3S, Sophia-Antipolis*), Le Pallec Xavier (*LIFL, Lille*), Ledrich Lionel (*ALTEN Nord, Lille*), Nodenot Thierry (*LIUPPA, Pau*) et Pinna-Dery Anne-Marie (*I3S, Sophia-Antipolis*) et Rusinek Stéphane (*Psittec, Lille*)

1 Contexte et domaines applicatifs

En 2006, le *Standish Group* indiquait que l'amélioration du taux de réussite des projets informatiques (passant de 16 à 32%) provenait de différents facteurs, dont le plus important était l'implication grandissante des utilisateurs finaux. Le mouvement syndicaliste scandinave (début 70) nommé "conception participative" avait déjà influencé l'ingénierie logicielle dans les années 80 dans ce sens: l'adhésion des employés pour un nouvel outil peut s'obtenir par leur participation à sa conception. Ce sont les démarches agiles, officialisées en 2001, qui ont popularisé ce principe d'implication des utilisateurs dans la conception logicielle. Parallèlement, l'augmentation de la production logicielle et de la taille des applications informatiques a accentué la préoccupation de réutilisation logicielle. Une réponse à celle-ci a été d'accorder plus d'importance à la modélisation dans la conception logicielle : un modèle fournit un meilleur support pour la discussion, la compréhension d'une architecture ou d'un composant et demeure moins impacté par les évolutions technologiques.

L'utilisation de plus en plus intensive des « wiki » et la mise à disposition d'outils **google** témoignent de l'intérêt de donner aux utilisateurs les moyens de « programmer » leurs propres outils. Cette thématique se retrouve au sein d'éditeurs d'applications *mash up* comme Yahoo Pipes, MS Pop Fly. Ces éditeurs, très en vogue actuellement, sont souvent basés sur des éditeurs de modèles graphiques très accessibles et en principe au moins lisible par un grand nombre d'utilisateurs "avertis". De manière plus industrielle, la construction d'usines logicielles et les environnements de modélisation dédiés à des domaines spécifiques entrent également dans ce cadre, en donnant à l'utilisateur des langages de modélisation adaptés à son métier. Le défi proposé ici est donc : *comment donner aux experts d'un domaine les moyens de construire leur propre système informatique en utilisant des techniques de modélisation ?*

Les enjeux sont une plus grande productivité et une meilleure adéquation des produits aux problèmes. En particulier en permettant aux utilisateurs d'adapter le logiciel, une plus grande réactivité/agilité est obtenue. Ces enjeux ne seront atteints que si les artefacts de modélisation donnés sont adéquats et si les produits résultants sont fiables et opérationnels.

Les risques sont la définition de méga-métamodèles non exploitables, l'obtention de produits inutiles, non performants, non réutilisables.

Nous proposons d'aborder ce défi par la construction d'environnements de modélisation dédiés à des domaines spécifiques. Des systèmes tels que CENTAUR [1] dans les années 80-90 avaient des objectifs similaires mais étaient orientés langages de programmation. Le défi que nous proposons en s'appuyant sur ces acquis envisage une appréhension de la "programmation" dirigée non pas par la syntaxe mais par les concepts, les usages et les contraintes du métier. Le défi est de faciliter la construction de ces environnements en utilisant des techniques de modélisation, en particulier la modélisation sous la forme de diagrammes.

2 Verrous

Les verrous identifiés se situent au niveau (i) des représentations à donner aux multiples profils d'utilisateurs qui doivent être en adéquation avec leur usage, (ii) de l'interprétation de ces notations par le système pour assurer l'aide, la validation et la collaboration et l'exécution éventuelle des systèmes produits, (iii) de l'impact sur les démarches usuelles de développement.

1. Représentation métier :
 - comment éviter la surcharge cognitive au niveau des syntaxes concrètes dédiées aux utilisateurs ?
 - comment supporter la montée en compétences (connaissances du système) d'un utilisateur ?
 - comment supporter les multi-représentations lors de travaux collaboratifs ?
2. Interprétation des modélisations métier :
 - comment supporter plusieurs représentations d'un même système (multi-utilisateurs, multi-niveaux) ?
 - comment rendre opérationnelles ces représentations en limitant les coûts de développements et en assurant la cohérence des systèmes construits ? (intégration de l'existant, incrément, agilité des développements ?
 - comment permettre à l'utilisateur final d'interagir avec le système de manière à en adapter le comportement (e.g., systèmes adaptables) ?
 - Un des verrous à lever est le juste appariement entre des approches dites "formelles" indispensables à ancrer le procédé dans une logique de fiabilité, et des approches de modélisation plus flexibles qui supportent une expression plus "naturelle" des domaines métiers.
3. Démarches de développement :
 - comment supporter l'agilité de la construction, la robustesse des environnements construits tout en acceptant les incohérences nécessaires au développement collaboratif, ...

3 Fondements

L'IDM apporte un ensemble de fondements permettant d'aborder la problématique soulevée par ce défi [2]. En effet, la montée en abstraction supportée par l'IDM pour la construction de systèmes complexes réduit l'écart entre la spécification et la conception/développement d'un système. En premier lieu, la prise en compte des différents profils d'utilisateurs finaux nécessitent de définir des représentations adéquates pour chacun d'entre eux. Cette adéquation nécessite une étude des propriétés cognitives des formalismes utilisés. Pour cela, nous pourrions nous baser sur les synthèses de Bernard Morand [16] sur les propriétés cognitives des diagrammes (du point de vue de la psychologie cognitive, de la linguistique...) comme sur les travaux de Gerald Lohse [17] qui sont plus spécifiques aux diagrammes représentant des systèmes informatiques. D'autres travaux ont porté sur la compréhensibilité globale d'une notation et ont permis d'aboutir à des protocoles expérimentaux réutilisables et utilisées dans le monde industriel [18]. Dans tous ces cas, l'approche utilisée est celle des expériences utilisateurs.

Mais une autre approche, plus automatisable, est possible pour évaluer la qualité d'un modèle ou d'un langage : elle se base sur l'utilisation de métriques qui peuvent être définies sur les modèles [19] ou les métamodèles [20]. Il existent donc de nombreuses possibilités pour appréhender ce qui a été définie dans les frameworks de qualité [21][22] comme la qualité pragmatique, c'est-à-dire la qualité dépendant de l'interprétation des utilisateurs et des concepteurs.

La définition d'un langage dédié (*Domain Specific Language*, ou DSL), et plus récemment de langage de modélisation dédié (*Domain Specific Modeling Language*, ou DSML) a été étudié au travers de techniques telles que les profils UML (consistant à spécialiser UML pour des besoins particuliers), ou la métamodélisation offrant toute l'ingénierie pour la définition d'un nouveau langage [9]. L'outillage de celui-ci peut par ailleurs être facilité à l'aide d'approches génératives ou génériques, telle que par exemple GMF¹ qui permet de générer automatiquement des outils de modélisation métier graphique pour un langage dédié donné. Le traitement automatique, à base de modèles, de langue naturelle ou de règles métiers est également un axe de recherche intéressant pour la résolution de ce défi. La publication récente par l'OMG de la norme *Semantics for Business Vocabulary and Rules* (SBVR)² permet par exemple d'interpréter formellement sous la forme de modèle un anglais structuré [23]

Les travaux sur les approches par points de vues devront également être pris en considération de manière à bien préciser l'intention de chaque type d'utilisateurs [15], de même que les travaux sur les usines logicielles [8]. D'autres travaux sur l'extension des métamodèles, tant syntaxique que comportementale (e.g., la modélisation orientée aspect [3,4]) offre également des moyens d'adapter ou de préciser un langage existant pour des besoins spécifiques.

¹ Cf. <http://www.eclipse.org/gmf/>

² *Semantics of Business Vocabulary and Business Rules* 1.0 specification (2008), <http://www.omg.org/spec/SBVR/1.0/>

D'autre part, des techniques de composition [5,6,10] et de transformation de modèles [7] permettent de prendre en compte les modèles issus de différentes préoccupations de manière à obtenir un modèle unique du système complet. Ces techniques permettent ainsi de rendre productif les modèles métiers et de les considérer comme des artefacts terminaux du processus de développement. Le couplage transparent des représentations métiers avec les méthodes formelles [13] devra également être pris en considération afin d'assurer la fiabilité des systèmes construits.

L'utilisation de ces techniques doit toutefois s'intégrer dans une démarche agile favorisant un cycle court de manière à permettre à l'utilisateur de voir très rapidement le résultat de son travail. Des adaptations dynamiques de l'environnement doivent également être envisagées [24,25]. La nécessité d'intégrer un processus itératif est donc indispensable et doit s'appuyer pour cela sur la traçabilité entre les modèles métiers et le système final obtenu. Cette traçabilité peut par exemple être utilisée pour indiquer les impacts sur le système final de chaque modification d'un modèle métier.

Enfin, l'assistance à la modélisation peut s'appuyer d'une part sur les modèles paramétrés pour permettre une modélisation par la réutilisation de briques génériques [14], et d'autre part sur le typage de modèle [11] pour permettre la généralité de certain traitement [12].

4 Usages et impacts sociétaux

- La programmation des applications est donnée aux experts des domaines, la construction des environnements est donnée aux experts logiciels. Mais les environnements eux-mêmes doivent pouvoir évoluer par des cycles courts.
- Les services informatiques représentent une activité économique importante (> 700Md \$ en 2008). C'est pourtant un secteur dont la majorité des projets sont des échecs. L'implication des utilisateurs est le facteur de réussite préconisé mais sa mise en œuvre reste difficile. Un objectif majeur du défi est de diminuer voir supprimer cette difficulté.

5 Jalons, démarche

- Circonscrire la complexité par des études de cas :
 - Expérimentations sur des classes d'applications pour lesquelles les langages/formalismes sont maîtrisables par les "personnes métier" (ex. : Business Rules, modélisation de workflows en analyse d'images médicales, modélisation de systèmes de diffusions d'informations en milieu scolaire, domotique),
- Définition des artefacts de modélisation (formalismes) cognitivement adaptés aux différents rôles (ou personnes) impliquées
 - Processus de construction attendu, validations, tests : les artefacts se définissent relativement à leur usage.
 - Les experts d'un domaine peuvent adapter ou faire évoluer l'application uniquement sur la partie métier.
- Corrélation entre modélisation métier et mise en œuvre au niveau des plates-formes :
 - Dans un premier temps, des experts en programmation définissent les méthodologies, processus, outils de tests et de validation qui permettent de construire, modifier, valider des "modèles métiers" et d'assurer la cohérence des systèmes construits : "*Controlled Agility*".
 - Dans un second temps, des patrons (transformations, mécanismes de profiling) sont utilisés pour, sur la base de la définition de métamodèles métiers, obtenir les outils associés (validation, tests, ...). Partiellement atteint dans le cadre des langages de programmation, cet objectif devrait bénéficier des avancées en matière de métamodélisation.
 - En obtenant une certaine automatisation de la production des outils, les utilisateurs finaux devraient pouvoir faire évoluer leur propre environnement de modélisation dans la limite des contraintes nécessaires pour assurer la cohérence des environnements et des systèmes construits.

References

1. P. Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual, *Centaur: the system*, Proceedings of SIGSOFT'88, Third Annual Symposium on Software Development Environments (SDE3), Boston, USA, 1988.
2. Jean-Marie Favre, Jacky Establier, and Mireille Blay-Fornarino, editors. *L'ingénierie dirigée par les modèles : au-delà du MDA*. Hermes-Lavoisier, Cachan, France, February 2006.
3. Jacques Klein, Franck Fleurey, and Jean Marc Jézéquel. Weaving multiple aspects in sequence diagrams. *Transactions on Aspect-Oriented Software Development (TAOSD)*, LNCS 4620 :167–199, 2007.
4. Jean-Marc Jézéquel. – Model driven design and aspect weaving. – *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, may 2008.
5. Robert France, Franck Fleurey, Raghu Reddy, Benoit Baudry, and Sudipto Ghosh. Providing support for model composition in metamodels. In *EDOC'07 (Entreprise Distributed Object Computing Conference)*, Annapolis, MD, USA, 2007.
6. Olivier Barais, Jacques Klein, Benoit Baudry, Andrew Jackson, Siobhan Clarke, *Composing Multi-View Aspect Models*, 7th IEEE International Conference on Composition-Based Software Systems (ICCBSS) (2008)
7. Jean Bézivin, Salim Bouzitouna, Marcos Didonet Del Fabro, Marie-Pierre Gervais, Frédéric Jouault, Dimitrios S. Kolovos, Ivan Kurtev, and Richard F. Paige. A canonical scheme for model composition. In *Arend Rensink and Jos Warmer, editors, ECMDA-FA*, volume 4066 of *Lecture Notes in Computer Science*, pages 346–360. Springer, 2006.
8. Carlos Parra, Rafael Leano, Xavier Blanc, Laurence Duchien, Nicolas Pessemier, Chantal Taconet and Zakia Kaziaoul, *Dynamic Software Product Lines for Context-Aware Web Services.*, in *Enabling Context-Aware Web Services: Methods, Architectures, and Technologies*. Chapman and all, July 2009
9. F. Barbier, *UML 2 et MDE - Ingénierie des modèles avec études de cas*, Dunod, 2005
10. Mireille Blay-Fornarino, “*Interprétations de la composition d'activités*”, HDR Thesis University of Nice-Sophia Antipolis, 1-201 pages, Sophia Antipolis, France, apr 2009
11. Jim Steel and Jean-Marc Jézéquel. – On model typing. – *Journal of Software and Systems Modeling (SoSyM)*, 6(4):401–414, December 2007.
12. Naouel Moha, Vincent Mahé, Olivier Barais, and Jean-Marc Jézéquel. – Generic Model Refactorings. – In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Denver, Colorado, USA, Oct 2009.
13. B. Combemale, X. Crégut, P.-L. Garoche and X. Thirioux – Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification – *Journal of Software (JSW)*, 4(6), December 2009
14. Alexis Muller, Olivier Caron, Bernard Carré, Gilles Vanwormhoudt, *On Some Properties of Parameterized Model Application* in *Proceedings of ECMDA-FA'2005: First European Conference on Model Driven Architecture - Foundations and Applications*, Nuremberg, November 2005, LNCS 3748, A. Hartman and D. Kreisliche Eds.
15. Adil Anwar, Sophie Ebersold, Bernard Coulette, Mahmoud Nassar, Abdelaziz Kriouile. A Rule-Driven Approach for composing Viewpoint-oriented Models. Dans : *Journal of Object Technology*, ETH Swiss Federal Institute of Technology, p. 1-26, 2010
16. B. Morand, *Logique de la Conception, Figures de sémiotique générale d'après Charles S. Peirce*, L'Harmattan, Collection L'Ouverture Philosophique, 294 p.
17. G. L. Lohse, D. Minb and J. R. Olsonc, Cognitive evaluation of system representation diagrams, *Information & Management*, Volume 29, Issue 2, August 1995, Pages 79-94
18. (Patig 2008) : S. Patig, A practical guide to testing the understandability of notations, *Conferences in Research and Practice in Information Technology Series*; Vol. 325, *Proceedings of the fifth on Asia-Pacific conference on conceptual modelling - Volume 7*, 2008, pp 49-58
19. (Lange 2005) : C. Lange, M. Chaudron, *Managing Model Quality in UML-Based Software Development*, Proc. of the 13th workshop on software Technology and Engineering Practice (STEP'05), 2005, pp. 7-16.
20. (Rossi 1996) M. Rossi et S. Brinkkemper, Complexity metrics for systems development methods and techniques, *Information Systems*, Vol. 21, num 2, 1996, pp. 209-227.
21. (Lindland 1994) O. Lindland, G. Sindre, A. Solvberg, Understanding Quality in Conceptual Modeling, *IEEE Software*, Vol. 11, 1994, pp. 42-49.
22. (Krogstie 1998) J. Krogstie, Integrating the Understanding of Quality in Requirements Specification and Conceptual Modeling, *Software Engineering Notes*, ACM SIGSOFT, Vol 23, num 1, 1996, pp. 86-91
23. Mathias Kleiner, Patrick Albert and Jean Bézivin, Parsing SBVR-based controlled languages, In *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, empirical track, Denver, Colorado, USA, Oct 2009, pages 122-136.
24. Reza Razavi, Noury Bouraqadi, Joseph W. Yoder, Jean-François Perrot, Ralph E. Johnson: *Language support for adaptive object-models using metaclasses*. *Computer Languages, Systems & Structures* 31(3-4): 199-218 (2005)
25. Razavi, Reza, Kirill Mechtov, Gul Agha, Jean-Francois Perrot. "Ambiance: A Mobile Agent Platform for End-User Programmable Ambient Systems," J.C. Augusto and D. Shapiro (eds.), *Advances in Ambient Intelligence*, *Frontiers in Artificial Intelligence and Applications (FAIA)*, vol. 164, IOS Press, 2007

Towards Disappearing Languages

Charles Consel,

INRIA / University of Bordeaux

Charles.Consel@inria.fr

This paper explores challenges that need to be addressed to make the Domain-Specific Language (DSL) approach successful. In particular, we argue that a DSL should be blended with a domain process and used by domain experts in support of their job. We call such languages *disappearing languages*.

The DSL approach has long been used with great success in both historical domains, such as telephony, and recent ones, such as Web application development. And yet, from software engineering to programming languages, there is a shared feeling that there is still much work to do to make the DSL approach successful.

Unlike General-Purpose programming Languages (GPLs) that target trained programmers, a DSL revolves around a domain: it originates from a domain and targets members of this domain. Thus, a successful DSL should be some kind of a *disappearing language*; that is, one that is blended with some domain process. In doing so, programming expands its scope to reach end users. That is, users for which writing programs come in support of achieving their primary job function. Well-known examples include Excel and MatLab.

Creating a disappearing language critically relies on an analysis phase of the target domain. The result of this analysis then fuels a design phase of the language. Practical experience shows that these two phases are time consuming, human intensive and high risk.

- What kind of tools could assist in performing these two phases?
- How much improvement can we expect from a tool-assisted process?

A successful DSL is above all one that is being used. To achieve this goal, the designer may need to downgrade, simplify and customize a language. In doing so, DSL development contrasts with programming language research where generality, expressivity and power should characterize any new language. As a consequence, programming language experts may not be the perfect match for developing a DSL.

- Does this mean that, for a given domain, its members should be developing their own DSL?
- Or, should there be a new community of *language engineers* that bridge the gap between programming language experts and members of a domain?

In many respects, a DSL is often an over-simplified version of a GPL: customized syntactic constructs, simple semantics, and by-design verifiable properties. These key differences may raise concerns about a lack of tool support for DSL development. Yet, there are many program manipulation tools (parser generators, editors, IDEs) that can be easily customized for new languages, whether textual or graphical. Furthermore, for a large class of DSLs,

compilation amounts to producing code for a domain-specific programming framework, and enabling the use of high-level transformation tools. Lastly, properties can often be checked by generic verification tools.

- Then, what is missing to develop DSLs?
- Do we need to have an integrated environment for DSL development, orchestrating a library of tools?
- Should there be a new breed of compiler and analysis generators matching the requirements of DSLs?

The research community in programming languages has always overwhelmingly focused on the design and implementation of languages. Very little effort has been devoted to understanding how humans use programming languages. This avenue of research is a key to expanding programming beyond computer scientists.

- How do we design languages that are easy to use?
- How do we measure the productivity benefit of the use of a language?
- How do we assess the software quality of DSL programs?

Vers une réification de l'énergie
dans le domaine du logiciel
L'énergie comme ressource de première classe

Jean-Marc Menaud, Adrien Lèbre, Thomas Ledoux, Jacques Noyé
Pierre Cointe, Rémi Douence, Mario Südholt

Équipe ASCOLA (EMNantes-INRIA, LINA)
prénom.nom@emn.fr

Résumé

En quelques années, le problème de la gestion de l'énergie est devenu un enjeu de société. En informatique, les principaux travaux se sont concentrés sur des mécanismes permettant de maîtriser l'énergie au niveau du matériel. Le renforcement du rôle de l'informatique dans notre société (développement des centres de données, prolifération des objets numériques du quotidien) conduit à traiter ces problèmes aussi au niveau du logiciel.

Dans ce papier, nous nous posons la question de la réification de l'énergie comme fut posée en son temps celle de la réification de la mémoire (l'espace) et de l'interpréteur (la machine d'exécution). Le défi est d'abord de sensibiliser l'utilisateur final au problème de la consommation énergétique en visualisant, grâce à des mécanismes d'introspection, sa consommation, à l'image de ce qui se fait aujourd'hui dans le domaine automobile (consommation instantanée d'essence). Il s'agira ensuite de proposer aux développeurs des mécanismes d'intercession leur permettant de contrôler cette consommation énergétique. Ces mécanismes réflexifs devront concerner l'ensemble du cycle de vie du logiciel.

1 L'énergie comme nouvelle préoccupation ?

En 2008, 2% de la production électrique mondiale a été consommée par les centres de données utilisés pour l'hébergement de l'Internet. La problématique est d'importance car, d'un point de vue mondial, et selon l'*Environmental Protection Agency* (EPA), il a été estimé qu'entre 2007 et 2012 les serveurs et les centres de données vont doubler leurs besoins en énergie pour atteindre 100 milliards de kWh. Cette augmentation des besoins est liée à l'accroissement du nombre d'internautes¹, des terminaux d'accès², des données disponibles³, et à l'apparition de nouveaux usages comme le *Cloud Computing*, ou l'Internet des Objets⁴. À ce rythme, dans 25 ans, Internet consommera autant d'énergie que l'humanité tout entière en 2008.

D'un point de vue plus global, l'impact des TIC

1. Principalement dans les pays du BRIC : Brésil, Russie, Inde, Chine.

2. Le nombre d'ordinateurs est estimé à 1 milliard en 2008, 2 milliards en 2013, sans compter le nouveau marché des téléphones intelligents (77 millions vendus sur les trois premiers mois de 2009 soit 14 par seconde).

3. 161 exaoctets de données ont été créés en 2006, soit approximativement 3 millions de fois l'information contenue dans tous les livres jamais écrits.

4. D'ici à quelques années, le nombre d'appareils communicants pourrait être de plusieurs dizaines de milliards, soit plus que le nombre d'êtres humains.

dans la consommation électrique est de l'ordre de 10 à 15% dans les pays développés. Pour la France, l'estimation de la consommation électrique est comprise entre 55 et 60 TWh par an, soit 13,5% de la consommation française. Au niveau des particuliers, les TIC dans leur globalité représentent 30%⁵ de l'électricité consommée. Depuis 10 ans, cette consommation a triplé. Cette progression risque d'augmenter fortement dans les prochaines années avec l'apparition d'équipements toujours plus « énergivores » (écrans LCD, passerelles domestiques, etc.).

Par conséquent, une rupture technologique est essentielle pour relever le défi majeur de la maîtrise énergétique des équipements informatiques. Les fondateurs, constructeurs et installateurs de matériels informatiques ont déjà travaillé sur ce problème. Par exemple, dans les centres de données, l'unité d'accueil de nouvelles machines n'est pas le nombre de serveurs à installer ou leur type, mais la puissance électrique consommée. Si des modèles de consommation d'énergie au niveau matériel sont aujourd'hui disponibles, il n'existe pas l'équivalent au niveau logiciel.

Dans ce papier, nous explorons les possibilités d'intégrer l'énergie en tant que ressource de première classe pour le logiciel, et donc de disposer de tels modèles dans le domaine du logiciel, manipulables par le logiciel. Cette réification permettra de maîtriser l'empreinte énergétique d'une application tout au long de son cycle de vie.

2 L'énergie, nouvelle ressource physique ?

En matière de gestion explicite des ressources physiques et depuis la fin des années 40, l'informatique est passée d'un extrême à l'autre. Pendant longtemps, le programmeur a été comptable de toutes les ressources qu'il utilisait : mémoire, nombre de cycles du processeur, nombre d'accès au disque, etc. Ainsi sous DOS, la mémoire était organisée en blocs de 64 Ko et cela expliquait la taille maximale d'un tableau dans

5. Étude REMODECE 2007, ADEME, électricité spécifique consommée.

des langages comme Pascal. Pour le langage Lisp, les programmeurs attachaient une grande importance au contrôle de la consommation de la mémoire qui se traduisait dans le nombre de doublets alloués et la taille de la pile. D'où l'utilisation des fonctions de chirurgie (par exemple `nconc` versus `append` pour la concaténation) permettant d'altérer physiquement les listes. « *Ces fonctions sont d'un maniement délicat faisant passer LISP d'un langage d'expressions à un langage de manipulation de pointeurs où les programmes peuvent s'automodifier* » (chapitre 2 de [2]). Avec l'usage des macros, ces fonctions ont donné lieu aux travaux sur la métaprogrammation et la réflexion, dont 3LISP, dans le but initial d'autodécrire des mécanismes de base de l'interpréteur dont la gestion de la mémoire et celle des continuations [4].

À partir des années 80, la mise en œuvre de mécanismes de gestion des ressources, comme les ramasse-miettes, a permis de décharger les développeurs de la contrainte d'une gestion fine des ressources. Ce détachement du programmeur vis-à-vis des ressources a été amplifié par la montée en puissance des capacités des machines suivant la loi de Moore. Les méthodes de programmation et de génie logiciel se sont affranchies (presque) complètement des notions physiques de la machine pour traiter de la rapidité de développement (objets, programmation agile, patrons de conception, usine logicielle) et la facilité de maintenance et d'évolution (architectures à base de composants, d'aspects et de services).

Grâce à ces techniques logicielles, et à la loi de Moore, nous avons pu construire des applications complexes à forte fonctionnalité ajoutée. Mais, alors que la demande continue à croître, il est impératif d'enrayer l'envolée de la consommation énergétique associée. On va donc rapidement se retrouver dans la situation des programmeurs des années 70, avec des contraintes énergétiques plutôt que des contraintes sur la consommation de la mémoire. Peut-on appliquer à l'énergie les mêmes stratégies que celles utilisées pour gérer la consommation de la mémoire ? Contrairement à la mémoire, la consommation énergétique propre d'une application n'est pas directement accessible. Le premier verrou à lever consiste à réifier cette ressource particulière.

3 Comment réifier l'énergie ?

Pour réifier l'énergie, il faut dans un premier temps la quantifier, donc disposer d'outils logiciels ou matériels permettant de mesurer la consommation électrique d'un quantum d'exécution (instruction de la machine, pseudo-code, exécution d'un service, suivant la granularité choisie). Cependant mesurer la consommation énergétique d'un quantum d'exécution complexe, comme un service, est une tâche ardue. En effet, l'énergie consommée par un quantum d'exécution est la somme des énergies consommées par les ressources physiques utilisées par ce quantum d'exécution (CPU, mémoire, disque, réseau, etc.).

Dans un deuxième temps, il est nécessaire de construire un modèle théorique de la consommation énergétique qui permette d'attacher des quanta d'énergie aux quanta d'exécution. Pour être utilisable, ce modèle doit être compositionnel et réaliste, c'est-à-dire qu'il doit être capable de prédire avec suffisamment de précision la consommation d'un programme par composition simultanée des quanta d'exécution et des quanta d'énergie. Ces besoins peuvent conduire à choisir des quanta de grain assez gros. Travailler à un niveau de pseudo-code semble prometteur [3]. Une première difficulté peut être, suivant le modèle de programmation, la prise en compte de la composition parallèle de programmes. Certains modèles de programmation, comme le modèle de programmation synchrone, peuvent être plus favorable que d'autres. Une deuxième difficulté est, dans un contexte distribué, de prendre en compte les coûts de communication et plus généralement d'infrastructure (assurant la disponibilité des services).

Cette quantification énergétique des applications sera utilisée à la fois statiquement par le programmeur et le compilateur, assistés de systèmes d'analyse statique quantitative (voir, par exemple, [5]), et dynamiquement par le système d'exécution.

Réifier l'énergie au niveau langage permettra à un programmeur d'introspecter son code sous l'angle énergétique, et de le modifier en changeant ses fonctionnalités ou son implémentation, par exemple en développant des algorithmes de type *anytime* [6]. Les algorithmes *anytime* sont des algorithmes dont

la qualité des résultats augmente avec le temps de calcul. Ce type d'algorithme permet de contrôler l'énergie affectée à un calcul en interrompant ce calcul lorsque son quota est épuisé.

Réifier l'énergie au niveau du système d'exécution permettra d'attribuer l'énergie disponible aux applications et d'arbitrer ces allocations en cas de pénurie (énergétique) comme cela est fait pour d'autres ressources [1]. Des stratégies d'allocations spécifiques à la gestion énergétique seront probablement à développer.

4 Conclusion

L'informatique au sens large a et aura un impact significatif sur la consommation électrique mondiale. La poursuite de son développement passe par une prise en charge généralisée de la contrainte énergétique au-delà des domaines habituels du matériel et de certains secteurs de l'informatique embarquée. Pour ce faire, il est nécessaire de réifier l'énergie en faisant de celle-ci une ressource accessible statiquement et dynamiquement au niveau de chaque composant applicatif. Il sera alors possible, dans un premier temps, de sensibiliser de manière efficace à la fois les utilisateurs et les programmeurs finaux aux coûts énergétiques et, dans un deuxième temps, de fournir des concepts⁶ et des outils de développement ainsi que des systèmes d'exécution susceptibles de significativement réduire ces coûts.

Références

- [1] Luc Morau and Christian Queinnec. Resource aware programming. *ACM Transactions on Programming Languages and Systems*, 27(3) :441–476, May 2005.
- [2] Christian Queinnec. *LISP Mode d'emploi*. Eyrolles, 1984.
- [3] Chiyoung Seo, Sam Malek, and Nenad Medvidovic. Component-Level Energy Consumption Esti-

⁶ On peut par exemple penser au développement, dans la continuité des algorithmes *anytime*, d'une algorithmique « verte ».

- mation for Distributed Java-Based Software Systems. In *Component-Based Software Engineering*, pages 97–113. Springer-Verlag, 2008.
- [4] Brian Cantwel Smith. Reflection and semantics in LISP. Technical report, Palo Alto Research Center, 1984.
- [5] Pascal Sotin, David Cachera, and Thomas Jensen. Quantitative static analysis over semirings : Analysing cache behaviour for Java Card. In Alessandra Di Pierro and Herbert Wiklicky, editors, *Quantitative Aspects of Programming Languages (QAPL '06)*, 2006.
- [6] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3) :73–83, 1996.

Appel à défi émanant du groupe MFDL :

Intégration maîtrisée et raisonnée des modèles et méthodes formelles dans un processus industriel de développement logiciel.

1. Le groupe MFDL

Ce défi émane du groupe MFDL (Méthodes formelles pour le développement logiciel) initié dans le cadre du GdR Alp, puis reconduit dans le GdR GPL. Ce groupe, à l'origine de l'atelier AFADL qui vient de tenir sa 10^{ième} édition à Poitiers, s'intéresse aux méthodes formelles dans le processus de développement logiciel. Le dernier atelier AFADL a permis de faire le point sur les pratiques actuelles et le défi proposé ici est issu de cette réflexion et de celles des signataires de ce texte.

2. Contexte actuel et nouveaux besoins

Depuis les premiers rapports recensant les débuts de l'utilisation des méthodes formelles dans des processus industriels (CGR93, OFTA 97,BC99) il est indéniable que les techniques formelles au sens large (allant des modèles formels et de leur validation aux techniques d'analyse et de vérification de code) pénètrent de nouveaux types d'applications et de domaines. Dans certains domaines, et pour certaines étapes plus ciblées d'un développement, les approches formelles sont maintenant intégrées dans le processus de développement logiciel (Scade et son générateur de code certifié par exemple, les développements ferroviaires en B utilisant l'animation, le raffinement automatique, la preuve et le model-checking, ...). Basés sur ces nouvelles pratiques, on peut situer les nouveaux besoins sur deux axes.

- Le besoin de construire des véritables chaînes de développement, pour un domaine particulier, pouvant être réutilisées et adaptées en fonction des nouveaux projets et contraintes (aspects industrialisation) (rejoint en cela les préoccupations des approches IDM-MDE en incluant la vérification)
- Le besoin de pouvoir apporter des assurances sur les produits développés, vis-à-vis des normes de certification, mais aussi vis-à-vis des utilisateurs, en terme d'engagement de responsabilités.

3. Les défis

Le défi proposé est l'intégration **maîtrisée** et **raisonnée** des MF dans un cycle de développement de systèmes à logiciel prépondérant. Ceci signifie de pouvoir appliquer différentes méthodes et approches, à différents niveaux du cycle de spécification et développement, tout en mesurant les apports et les gains et de pouvoir qualifier les assurances de bout en bout d'un tel développement. Quatre axes sont à considérer :

1. L'étude d'un continuum, en terme de modélisation et de vérification, de l'analyse système à la production de code sur du matériel spécifique. En plus de la prise en compte du développement

logiciel de bout en bout, la complémentarité de différentes approches et de leur intégration dans un cycle de développement doit être finement étudiée.

2. Un défi important est de pouvoir garantir la traçabilité formelle entre les différents niveaux d'un cycle de spécification et développement. La traçabilité est incontournable pour la production de logiciels certifiés, mais s'avère nécessaire dès qu'un processus de vérification est introduit. La traçabilité repose sur l'établissement de liens sémantiques entre plusieurs niveaux, liens qui doivent aussi être justifiés.

3. L'étude de l'intégration au plus fin de l'apport des méthodes formelles (modélisation **et** vérification) dans de réels processus industriels ou en accord avec les normes des domaines (ex la nouvelle mouture de la DO-178C pour le domaine aéronautique). En particulier la question de quels passages à l'échelle sont réellement envisageables devra se poser. En effet, bien que les méthodes et outils aient acquis en performance et robustesse, l'évolution simultanée de la taille et de la complexité des systèmes fait que le problème du passage à l'échelle se pose toujours.

4. La formation qui doit être dispensée aux futurs ingénieurs, mais aussi aux personnes en place, pour les sensibiliser et les préparer à la mise en œuvre de ces techniques .

Nous ne nous fixons pas de méthode de travail ni de benchmarks mais le groupe possède des expériences variées en études de cas et domaines d'applications des méthodes formelles. Enfin la complémentarité d'approches et leur étude en terme de garanties globales peut s'envisager autour de plates-formes ouvertes telles que Frama-C [Frama-C], dédiée à l'analyse de programmes C et permettant de faire collaborer différentes formes d'analyses, ou la plate-forme Rodin [Rodin], intégrant différents outils comme la preuve, le raffinement, l'animation et le model-checking.

4. Les signataires

Les porteurs de la proposition :

- Marie-Laure Potet – Vérimag – Grenoble
- Yamine aït-Ameur – LISI - Poitiers
- Régine Laleau – LACL – Paris Est Créteil

Agnès Arnould – XLIM - Poitiers

Christian Attiogbé – LINA - Nantes

Yves Ledru – LIG - Grenoble

Georges Mariano- INRETS - Lille

Dominique Mery – Loria - Nancy

Jacques Julliard – LIFC - Besançon

Catherine Dubois – Cedric – Cnam Paris

Christel Seguin, Virgine Wiels – ONERA - Toulouse

Philippe Dhaussy – LISyC - Brest

[CGR93] «An international Survey of Industrial Applications of Formal Methods”.

Technical Report NIST GCR 93/626. National Institute of Standards and Technology, 1993.

[OFTA 97] « Applications des techniques formelles au logiciel ».
Observatoire Français des Techniques Avancées, 1997.

[BC99] « Formal Method Diffusion: Past Lessons and Future Prospects. »
R.E. Blomfield, D. Craigen and all. LNCS 1943, 2000.
Computer Safety, Reliability and Security.

[Frama-C] <http://frama-c.com/>

[Rodin] <http://www.event-b.org/index.html>

Nouvelles dimensions du Test Logiciel

Synthèse du groupe de travail MTV2 en réponse à l'appel à défi GDR-GPL

Arnaud Gotlieb (INRIA) et Roland Groz (LIG)
avec des contributions de

Ana Cavalli (Inst. TELECOM), Richard Castanet (LABRI),
Marie-Claude Gaudel (LRI), Jacques Julliand (LIFC), César Viho (IRISA),
Hélène Waeselynck (LAAS), Burkhart Wolff (LRI), Fatiha Zaidi (LRI)

6 juillet 2010

1 Contexte

La complexité croissante des systèmes à base de logiciels appelle la définition de processus et techniques de test mieux étayés, plus fiables, et capables de faire passer à l'échelle la confiance que nous portons dans ces systèmes. D'une part, il est attendu des approches de test logiciel qu'elles permettent aux systèmes critiques à base de logiciel d'atteindre le même très haut niveau de fiabilité qui a été atteint pour les systèmes électroniques. Cet objectif est ambitieux dans un contexte où ces systèmes, qui doivent respecter de fortes contraintes à l'exécution, sont de plus en plus dépendants de leur environnement, et évoluent en permanence. D'autre part, le défi que pose le test de systèmes mobiles et ubiquitaires, ou le test des larges systèmes pair-à-pair, doit être abordé avec de nouvelles approches à cause de la grande volatilité des dispositifs et de la taille de plus en plus importante des composants individuels du système. L'absence de contrôlabilité des environnements d'exécution ainsi que l'introduction des mécanismes de concurrence dans les langages ont rendu le test des systèmes plus difficile, même au niveau unitaire. Plus largement, l'évolution des méthodes de développement logiciel, mêlant codes et modèles, agilité du développement et la répartition mondiale des productions de logiciel amène à revisiter les priorités dans la recherche et l'enseignement des méthodes de validation du logiciel.

2 Le défi des méthodes de développement

2.1 Tests à partir de modèles et de code

Les processus et les techniques de test doivent évoluer car ils reposent essentiellement sur des méthodes de développement structurées à partir de spécifications et de modèles d'un côté, et de productions de code de l'autre. La génération de tests à partir de modèles (Model-Based Testing) a connu ces dernières années un intérêt considérable de part la prise de conscience de l'importance des modèles dans le développement logiciel, mais indépendamment les techniques de génération automatique de données de test à partir du code (Code-Based Testing) se sont développées au point de connaître de nombreux grands succès tant académiques que industriels. Un axe important sera donc de trouver des moyens de marier efficacement les deux approches. Un autre aspect lié concerne la prise en compte de l'évolution conjointe des modèles et du code dans la durée de vie du logiciel et ce pour la génération des tests. En effet, les processus de test qui prendront en compte cette évolution, non nécessairement synchronisée, restent à bâtir.

2.2 IDM et méthodes Agiles pour le test

Si l'ingénierie à base de modèles (IDM) constitue une tendance croissante dans le secteur de l'informatique embarqué, les approches de type Agile pourront être adaptées pour la prise en compte des systèmes distribués. Là aussi, il s'agira de concilier les approches plutôt descendantes de type IDM autour de la définition de modèles de test, et ascendantes de type Agile autour du pivot que constituera l'exploitation conjointe des tests comme spécifications et des modèles. L'évaluation quantitative de la fiabilité des systèmes opérées conjointement avec le test devra également conduire à l'amélioration des processus de développement.

3 Le défi des attentes sociétales

3.1 Réconcilier test et fiabilité

La dépendance de plus en plus grande du fonctionnement des entreprises et de la société par rapport aux composants informatiques rend nécessaire une évaluation quantitative de la fiabilité des systèmes à base de logiciels. L'évaluation des défauts de fonctionnement repose sur des processus de test. Les relations entre les deux domaines sont loin d'être à la hauteur des besoins de la société. Le couplage entre les tests et l'analyse de fiabilité nécessite d'étudier des notions de couverture liées à des profils d'utilisation, d'évaluer les probabilités d'événements rares à partir de techniques de simulation, de mieux relier les approches de test à des modèles quantitatifs, de corrélérer les informations à différents stades de test (test actif en phase de développement, test passif en phase d'exploitation).

3.2 Le Test au service de la Sécurité Logicielle

Le test de la sécurité est un thème qui sera d'actualité dans les années à venir dans le domaine de la sécurité réseaux et la sécurité de systèmes d'information. Bien que, par exemple, dans le domaine des réseaux, les différents protocoles qui existent à l'heure actuelle dans les réseaux mobiles (ad hoc, mesh) offrent des performances satisfaisantes, leur grand point faible est qu'ils ne prennent pas en compte la problématique de la sécurité. Les techniques de test passif (ou de monitoring basé sur le test) se basent sur l'observation et l'évaluation des propriétés sur les traces d'exécution d'un système. L'avantage que procure le test passif par rapport au test actif est qu'il n'influe pas sur le système dans la mesure où, contrairement au test actif, l'implantation n'est pas stimulée par le testeur. Il est ainsi qualifié de non intrusif car le comportement interne de l'implantation n'est pas influencé. De plus, le testeur n'as pas besoin d'avoir un accès direct à l'implantation. Ces techniques seront très utilisées dans le futur, par exemple, pour le test des composants d'un système embarqué, car souvent dans ce type des systèmes les composants sont imbriqués et le testeur ne dispose pas d'un accès direct. Un défi pour ce type de techniques reste cependant le passage à l'échelle.

3.3 Méthodes probabilistes pour le test de logiciel basé sur des modèles

Le test de logiciel basé sur les modèles est confronté au même problème d'explosion de la taille des modèles que le model-checking. Les approches probabilistes d'exploration de modèles ont déjà été étudiées, mais de nouvelles opportunités sont apparues qu'il faut considérer. Les recherches dans ce domaine peuvent bénéficier de certains résultats du model-checking (Bounded model-checking, Monte-Carlo model-checking,etc.), mais un travail reste à faire sur les aspects contrôle et observabilité qui n'ont pas à être pris en compte pour le model-checking. Par ailleurs, il faut noter des progrès significatifs obtenus pour : - le tirage aléatoire dans des structures complexes, - les méthodes d'optimisation (recuit simulé, algorithmes génétiques,etc.), - la simulation des événements rares. Leur intégration dans les méthodes existantes, ou dans de nouvelles méthodes devrait permettre d'une part des passages à l'échelle spectaculaires, d'autre part des évaluations probabilistes de la confiance obtenue quand on les applique. Ce genre de recherche participe à la " réconciliation de la fiabilité et du test " mentionnée plus haut, mais pas seulement la fiabilité, car selon les modèles de fautes considérés on va pouvoir aborder tous les aspects de la sûreté de fonctionnement : disponibilité, sécurité, intégrité, etc.

4 Le défi des environnements

4.1 Test de systèmes mobiles/ubiquitaires

Les avancées technologiques du monde "sans fil" ont conduit au développement d'applications dédiées à l'informatique mobile. Les caractéristiques de ces applications (mobilité physique de dispositifs, dynamique de la structure du système, communication avec des partenaires inconnus dans le voisinage, dépendance vis-à-vis du contexte) posent de nouveaux défis pour le test. D'un point de vue conceptuel, un défi est de déterminer les abstractions adéquates pour modéliser de telles applications, et définir des stratégies de test basées sur des modèles. En particulier, la sélection des configurations expérimentales à tester reste un problème ouvert. D'un point de vue technologique, l'implémentation des tests peut nécessiter des fonctionnalités de simulation avancées pour prendre en compte les déplacements des dispositifs, gérer leurs communications selon la topologie de connexion, et produire des données contextuelles relatives à l'environnement physique. Ces problèmes conceptuels et technologiques sont amplifiés dans le cas de systèmes ubiquitaires, qui combinent la mobilité avec un haut niveau d'instrumentation de l'environnement, rendant ainsi la notion de "contexte" potentiellement très complexe.

Les architectures orientées services promeuvent le principe de réutilisation d'entités logicielles distribuées ou non encapsulées sous forme de services, et le principe de leur composition en applications à valeur ajoutée pour satisfaire les exigences fonctionnelles ou les besoins utilisateurs. Ce type de système s'inscrit dans la catégorie de système hautement dynamique car les processus de compositions et de recompositions peuvent avoir lieu au cours de l'exécution de ces services en fonction de l'évolution de l'environnement avec apparition/disparition de services ou évolution opportuniste du besoin utilisateur. De ce fait, ce qui est prévu au "*design time*" ne correspond pas forcément à ce qui se passe en réel au "*runtime*". Face à ce type de système, le processus de test doit être un processus en continu qui repose sur des techniques de test au "*runtime*". Ces dernières reposent sur des techniques de monitoring et sur une analyse "*on-line*" ou à posteriori des traces d'exécution du système. Les challenges à relever pour ce type de test vont être de déceler les changements (niveau utilisateur, niveau environnement) et de localiser la partie affectée par ce changement afin de répercuter ces changements sur les parties à tester pour s'assurer de la bonne conformité du système ainsi recomposé à ces besoins. Des techniques combinant le test et le diagnostique doivent être proposées pour la localisation et la détection des erreurs. Pour des approches de test basées modèles, il faudra également être capable de répercuter ces changements au niveau modèle pour que celui corresponde au modèle réel d'exécution.

4.2 Test de systèmes hybrides

À la frontière du discret et du continu, les systèmes hybrides posent un véritable défi en matière de vérification puisque les outils mathématiques utilisés

sont assez différents dans les deux domaines. Les techniques de génération de tests actuellement utilisées pour le test des systèmes hybrides sont encore assez préliminaires car ils échantillonnent les espaces continus de sorte à retomber dans le domaine du discret. Cette approche n'est pas sans défaut et il s'agira donc de mettre au point des outils riches en expression sémantique, capable de tester des propriétés portant directement sur les domaines continus. La génération de tests pourra ainsi tirer partie de l'analyse mathématique pour vérifier ces propriétés.

5 Enseignement du test

Depuis plusieurs années, une tendance forte dans le monde des télécoms et des banques, consiste en la délocalisation du développement logiciel des systèmes d'information vers des pays où les coûts sont moins élevés. Cette tendance engendre la transformation du métier d'ingénieur logiciel formé en France, celui-ci étant de plus en plus amené à contrôler des logiciels produit ailleurs. Dans ce contexte, il apparaît important que l'enseignement du test logiciel devienne un point capital de la formation des ingénieurs logiciels et doive être adapté pour prendre la mesure de ce changement radical. Des aspects du test logiciel, tantôt négligés, devront probablement être enseignés dans un avenir proche tels que l'intégration du test dans un processus de développement délocalisé, test de recette, test de sécurité, test de performance, etc.

Au-delà des techniques de test, et en relation avec les défis méthodologiques et sociétaux mentionnés plus haut, cet enseignement pourra aussi s'ouvrir à de nouvelles dimensions, en intégrant aussi bien des évaluations quantitatives, que des liens vers les sciences humaines (aspects juridiques, réglementaires, culturels...).

En introduction, problématique de l'adaptation

L'explosion de l'information disponible, l'intégration des technologies, les systèmes logiciels pénétrants à grande échelle, ces évolutions des systèmes informatiques innovants pour un proche futur ont en commun une préoccupation centrale : l'adaptation. Ces nouveaux systèmes doivent s'adapter à leur contexte opérationnel et à leur environnement. Ils doivent être changeants, flexibles, résilients, fiables, robustes, auto-configurables, auto-optimisés, économes en énergie. Les systèmes auto-adaptatifs ont été étudiés dans divers domaines de recherche d'Ingénierie Du Logiciel, essentiellement les architectures du logiciel, les intergiciels, les développements par composants sans qu'il ne se dégage une approche unifiée. D'autres communautés ont apporté des propositions pertinentes ponctuellement, essentiellement la tolérance aux fautes, les systèmes répartis, l'informatique inspirée de la biologie, l'intelligence artificielle répartie, les systèmes à base de connaissances, la théorie du contrôle, etc. Les technologies disponibles favorisent l'évolution des domaines d'application concernés par l'adaptation : systèmes embarqués, réseaux mobiles ad hoc, architectures orientées services, IHM adaptables, robotique, informatique ubiquitaire.

C'est au logiciel de par sa nature flexible de mettre en œuvre les diverses adaptations, pour ces divers domaines d'application. De façon générale, les concepteurs de systèmes logiciels s'appuient sur les activités amont du Génie Logiciel : l'Ingénierie des Besoins pour capturer les fonctionnalités et propriétés non fonctionnelles, en assurer la traçabilité et les faire évoluer pendant le processus de développement. Dans le cas des systèmes pour lesquels l'adaptation est une propriété prépondérante et composite, cette étape essentielle du processus n'a pas fait l'objet de proposition pour des approches spécifiques. Il y a là un défi général à prendre en considération.

Dans ce papier, on se focalisera sur l'« adaptation diffuse » pour traduire l'expression « Pervasive Adaptation », c'est à dire l'adaptation propre à l'informatique ambiante.

L'adaptation diffuse

Le domaine émerge d'une communauté large dont le support est assuré par le projet PerAda¹ de la Communauté Européenne, une Action Spécifique FET. PerAda assure la coordination des actions de six projets européens à partir desquels on peut cerner les spécificités de cette adaptation. Il s'agit de systèmes qui se définissent comme des sociétés de très petits artefacts qui soient capables de s'adapter automatiquement, en permanence, par eux-mêmes, en lien avec l'environnement, dans un contexte d'utilisation fortement dynamique, en prenant en compte l'émergence de comportement du système, en préservant la sécurité et la confiance.

L'inspiration et les références viennent des domaines des réseaux, de la biologie, de la sociologie, de l'interaction et de la cognition. Pour le projet ALLOW, il s'agit de développer un nouveau paradigme de programmation pour des applications de type workflow associant humains et objets mobiles, par exemple un système d'aide à la décision e-santé en milieu hospitalier. Pour le projet ATRACO, il s'agit de contribuer à la réalisation de systèmes d'écologies ambiantes et de sphères d'activités, basées sur des ontologies, par exemple en domotique. Pour le projet Fronts, il s'agit de créer des sociétés d'objets hétérogènes, tous n'étant pas informatisés (capteur infra-rouge par ex.) en réseaux sans fil pour leur confier une mission comportementale, par exemple une surveillance. Pour le projet REFLECT, il s'agit de nouveaux concepts et moyens pour la

¹ Futur and Emerging Technologies (FP7) - Proactive Initiative on Pervasive Adaptation (2007-2010)
<http://www.perada.eu/>

collaboration diffuse entre environnement et humain, par exemple un système embarqué dans une voiture s'adaptant aux émotions, aux conditions physiques ou encore un système de reconnaissance d'activités d'un humain pour agir sur son environnement domotique. Pour SocialNets, il s'agit de réseaux sociaux dans un contexte de mobilité, par exemple liés aux déplacements urbains. Pour le projet Symbion, il s'agit de concevoir une plateforme de robotique, d'essaim de micro-robots dont la capacité d'évolution collective leur permet de s'adapter à l'environnement et se configurer pour une mission globale.

Les démonstrateurs de ces projets mettent en évidence un besoin d'exprimer des propriétés pouvant conditionner des effets comportementaux de différentes natures : créatifs, souvent simples individuellement et sophistiqués lors de leur collaboration, localisés mais à influence globale, concernant à la fois des territoires concrets et virtuels dans une transition « sans couture » ... L'adaptation diffuse est la capacité du système à modifier les comportements individuels de ses artefacts pour faire émerger un comportement global en interaction avec son environnement.

Pour une contribution à une approche Génie Logiciel

Le développement de tels systèmes de sociétés de très petits artefacts peut-il être mené top-down en partant des artefacts individuellement ou bottom-up en considérant l'aspect coopératif des artefacts? Le comportement global émerge de comportements locaux. Pourtant il est difficile de déduire le comportement global de ces comportements locaux. La conception de tels systèmes est un compromis entre les deux approches auquel il n'est pas aisé de donner un cadre. Sans doute serait-il intéressant que ces systèmes s'appuient sur des buts de haut niveau, optimaux pour des motifs d'auto-régulation interne, d'interaction avec l'environnement, classés selon leur niveau de criticité, avec une part d'incertain liée à la variabilité de l'environnement ... encore faut-il que ces buts soient élicités préalablement et qu'ils puissent évoluer y compris au run-time. L'Ingénierie des Besoins propose dans le cadre de développement de systèmes « classiques » des méthodes orientées par les buts [1]. Des travaux ont été publiés concernant l'application d'une telle méthode en amont de la spécification de systèmes autonomiques [2] ou de systèmes adaptatifs [3].

Mais le principal défi lors de la conception d'un système à « adaptation diffuse » est que l'on ne peut pas anticiper pour cause d'incertain sur l'environnement, des évolutions au run-time. Le défi commence par l'Ingénierie des Besoins pour disposer d'un langage permettant l'expression dans les buts de l'incertain et de l'adaptation, langage moins prescriptif autorisant les alternatives. Le défi pourrait se poursuivre par le raffinement de ces modèles vers des spécifications d'architectures supportant l'adaptation diffuse ce qui serait ambitieux à court terme. En revanche, la traçabilité de propriétés des systèmes à adaptation diffuse serait une piste plus réaliste. Cela pourrait mettre en évidence que pour de tels systèmes l'Ingénierie des Besoins ne sera plus uniquement une activité en amont du développement mais incluse dans une boucle avec une rétro-action.

[1] A. van Lambsweerde; Requirements Engineering in the Year 00: A research Perspective. Invited Keynote paper, proceedings of ICSE'2000- In *22nd International Conference On Software Engineering*, ACM Press, 5-19

[2] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H.C.Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008

[3] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *CASCON'06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, page 7, New York, NY, USA, 2006. ACM.

Proposition de défis - groupe LTP

Pierre Castéran, Catherine Dubois

Il existe un certain nombre de formalismes, langages et outils pour produire des logiciels sûrs. Néanmoins force est de constater que leur diffusion reste limitée, tout particulièrement dans le monde de l'entreprise. A cela on peut avancer différentes raisons parmi lesquelles le manque de formation, la difficulté de prise en main des outils de preuve. L'investissement pour adopter un langage au typage plus sûr mais plus complexe ou prendre en main un outil de vérification est souvent jugé lourd, voire trop lourd.

Un autre problème réside aussi dans le fait que les techniques de preuve ont du mal à passer à l'échelle : code complexe, obligations de preuve trop nombreuses à décharger manuellement. Dans ce dernier cas, la réponse est par exemple d'augmenter la base de règles du prouveur ou d'introduire des propriétés supplémentaires qui permettront aux prouveurs de faire passer automatiquement un plus grand nombre de preuves. Mais le problème est alors d'assurer la correction de ces règles de preuve ou propriétés ajoutées.

Reste aussi un autre problème de taille, qui est souvent un point critique dans l'approche formelle, à savoir la validation des spécifications et hypothèses de départ. Ces spécifications et hypothèses forment le point de départ du développement formel. Il est crucial de pouvoir valider ces spécifications et hypothèses (appelées spécifications initiales dans la suite) par le client (ceci étant vrai dans toute méthode de développement, fut-elle informelle, semi-formelle ou formelle. Ces spécifications initiales peuvent apparaître des formulations complexes : en Coq des définitions (co)-inductives, en B le recours à des fermetures transitives, l'utilisation de sétoïdes quasi systématique en Focalize. Ces notions, si elles sont usuelles pour l'informaticien, ne le sont pas nécessairement pour l'ingénieur, généralement encore moins pour le client. Pour un informaticien formé à une méthode de spécification il est également souvent difficile de changer son mode de pensée pour passer à une autre méthode de spécification. On peut citer l'exemple assez caractéristique des graphes qui sont représentés de manière différente en Coq et en B alors que la notion de graphe vue comme un ensemble de sommets et d'arcs exhibant des chemins est une vue très classique et compréhensible par tous. On peut mettre en évidence ici une rupture entre le monde du semi-formel (on fait référence ici à une description des spécifications assez claire, épurée : le formalisme peut être un langage naturel "épuré", une notation semi-formelle par exemple).

Ce défi concerne les problèmes énoncés plus haut.

1. Faire passer la preuve dans le quotidien du développeur (au moins dans celui qui développe des applications critiques) en permettant à ces techniques de passer à l'échelle. Et pour cela "il faut faire feu de tout bois" :
 - automatiser les preuves le plus possible en utilisant les prouveurs automatiques dès que possible. Nous pensons que l'approche multi-prouveur adoptée par why [4] est une approche à généraliser mais en gardant à l'esprit que les prouveurs automatiques sont des outils à l'algorithmique complexe et qu'ils peuvent donc se tromper dans la réponse qu'ils fournissent. Dans un objectif de certification/vérification formelle, construire, à partir d'un résultat fourni par un prouveur automatique, une preuve qu'un assistant à la preuve pourra ensuite vérifier automatiquement et de manière la plus légère possible est un objectif important et légitime. Quelques prouveurs automatiques (Zenon [2] et Cime3 [3] par exemple) sont capables de faire ce travail. Un vrai défi est de généraliser l'approche à d'autres prouveurs automatiques et assistants à la preuve.

- faire collaborer les techniques : test, model checking, analyse statique et preuve sont des techniques complémentaires. Comment tirer profit du test par exemple pour faire une preuve ? Une approche proposée actuellement et implantée dans quelques environnements de développement formel est de tester une propriété avant de la prouver pour avoir une certaine confiance avant de se lancer dans un travail éventuellement difficile et/ou fastidieux ou trouver un contre-exemple (par génération automatique de test ou recherche de modèles). Le défi posé ici est d’aller plus loin dans cette collaboration entre test et preuve.
 - réutiliser des preuves est aussi un moyen de passer à l’échelle, ne refaisons pas ce qui a déjà été fait. Ceci peut se révéler difficile à mettre en oeuvre. En effet la formulation de la propriété à démontrer peut être différente quoique équivalente logiquement. Il peut aussi s’agir de réutiliser en adaptant ou généralisant la preuve initiale. Il peut aussi s’agir d’une preuve faite dans un autre assistant à la preuve, et dans ce cas en supposant que l’on dispose d’un mécanisme de traduction ou d’importation, comment s’assurer que l’on repose sur les mêmes hypothèses. Des travaux ont été entrepris sur la réutilisation de preuves (entre Focal et Coq par exemple ou de Coq à Coq dans le cadre de spécifications inductives incrémentales [1]) par le passé mais il s’agissait d’approches somme toute assez expérimentales et assez restreintes. Peut-on passer à l’échelle ? Quelle approche transversale mettre en oeuvre pour répondre à ces diverses formes de réutilisation ?
2. Eviter la trop grande rupture entre le semi-formel (selon la définition donnée plus haut) et le formel, pour permettre une meilleure validation des spécifications initiales. Nous pensons que le fossé qui sépare le semi-formel des formalismes utilisés dans des outils tels que B, Coq et Focalize peut se combler si l’on se restreint à des domaines d’application où un langage spécifique facilite la lecture des spécifications tout en étant accompagné d’une sémantique formelle très précise servant de point d’entrée aux formalismes et outils de preuve considérés.

Ce travail ferait la jonction entre la conception de langages dédiés (DSL : domain specific languages) et les outils de preuve formelle. Les domaines visés peuvent être par exemple les politiques de sécurité et réglementation, les langages permettant d’exprimer les calculs distribués, la géométrie, ou tout autre domaine d’application des DSL.

L’avantage de cette approche par rapport à l’existant serait de rendre accessible le langage de spécifications aux spécialistes du domaine considéré, et que les définitions les plus techniques demandant une réelle compétence en méthodes formelles soient à vérifier une fois pour toutes au cours de la validation de la traduction du DSL.

References

- [1] O. Boite. Proof reuse with extended inductive types. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Theorem Proving in Higher Order Logics, 17th International Conference, TPHOLs 2004, Park City, Utah, USA, September 14-17, 2004, Proceedings*, volume 3223 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2004.
- [2] R. Bonichon, D. Delahaye, and D. Doligez. Zenon: an Extensible Automated Theorem Prover Producing Checkable Proofs. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 4790 of *Lecture Notes in Computer Science (LNCS)/Lecture Notes in Artificial Intelligence (LNAI)*, pages 151–165, Yerevan (Armenia), Oct. 2007. Springer.
- [3] E. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3pat, an approach for certified automated termination proofs. In *PEPM ’10: Proceedings of the 2010 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 63–72, New York, NY, USA, 2010. ACM.
- [4] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, *Lecture Notes in Computer Science*, Berlin, Germany, July 2007. Springer-Verlag.