GdR Génie de la Programmation et du Logiciel
Défis 2025

Philippe Collet (I3S), Lydie Du Bousquet (LIG),
Laurence Duchien (CRIStAL), Pierre-Etienne Moreau (Loria)

8 février 2015

1 Introduction

Le Groupe de Recherche en Génie de la Programmation et du Logiciel (GdR GPL) a lancé en janvier 2014 un appel à défis 2025, qui fait suite à l'appel à défis 2020 lancé en 2010. Les défis 2010 ont été publiés dans TSI en 2012 [12]. Nous reprenons dans ce document les réponses de l'appel 2014, tout en le complétant par d'autres travaux de façon à aborder l'ensemble des thèmes de la communauté GPL.

Les chercheurs du GDR GPL travaillent sur des abstractions logicielles et les fondements scientifiques associés, en vue de proposer des solutions bien fondées et pratiques permettant la production et la maintenance de logiciels de qualité. Ces abstractions et fondements interviennent à toutes les étapes du cycle de vie du logiciel, de sa conception à son exploitation et à sa maintenance, et prennent différentes formes dans des théories, techniques, outils et méthodes de modélisation, de validation, de vérification, dans les langages de programmation et dans les plates-formes d'exécution. Quelle que soit l'étape du cycle de vie étudiée, l'objectif est d'offrir des moyens de construction de logiciel satisfaisant les besoins exprimés tout en apportant une aide aux concepteurs et en maîtrisant coûts et délais.

Les problématiques sur lesquelles les chercheurs se positionnent sont renouvelées par de nouveaux domaines d'application dans tous les secteurs (informatique embarquée, intelligence ambiante, informatique dématérialisée), de nouveaux enjeux de société (développement durable, économies d'énergie, objets connectés), ainsi que par la diversité des fonctionnalités à fournir et à adapter aux besoins spécifiques de clients et aux modifications continuelles.

Les questions auxquelles les chercheurs du GdR GPL tentent de répondre sont :

- Comment concevoir et réaliser des systèmes qui soient, par exemple, personnalisés ou capables de s'adapter par eux-mêmes aux changements de leur environnement, pouvant facilement découvrir, sélectionner et intégrer des services disponibles à l'exécution ?
- Comment construire des langages, compilateurs, pilotes et supports d'exécution qui prennent en compte les nouvelles architectures, multiples, hétérogènes tout en maîtrisant les dépenses d'énergie ?
- Comment valider et vérifier ces logiciels qui vont évoluer dans des environnements de plus en plus imprévisibles ?

Ce document reprend ces questions et apporte un éclairage sous forme de défis en suivant les différentes étapes du cycle de vie du logiciel. La section 2 reprend les défis relevant des exigences à la réalisation des logiciels. La section 3 enchaîne sur les défis de la compilation jusqu'à l'exécution

des logiciels. La section 4 présente les défis nouveaux concernant les méthodes et outils de validation et de vérification. Finalement la section 5 conclut ce document. Finalement, les défis rédigés par les chercheurs de la communauté du GdR GPL en 2014 sont ajoutés en annexe à ce document.

2 Des exigences à la réalisation des logiciels

Modéliser et analyser des Systèmes de Systèmes

La complexité croissante de notre environnement socio-économique produit de très grands systèmes, le plus souvent concurrents, distribués à grande échelle et parfois composés d'autres systèmes. Cela engendre de nouveaux défis de l'expression des exigences jusqu'à la réalisation des systèmes.

On appelle Système de Systèmes (SdS) le résultat de l'intégration de plusieurs systèmes indépendants et interopérables, inter-connectés dans le but de faire émerger de nouvelles fonctionnalités. La complexité d'un SdS réside dans cinq caractéristiques intrinsèques qui sont : l'indépendance *opérationnelle* des systèmes constituants, l'indépendance *managériale* de ces mêmes systèmes, la distribution géographique, l'existence de comportements émergents (souhaités ou non), et enfin un processus de développement *évolutionnaire*.

La conception de SdS présente des différences importantes par rapport à la conception de systèmes classiques. Elle intègre les systèmes constituants existants, parfois traités comme des systèmes hérités (en anglais *legacy*), c'est-à-dire des boîtes noires. Il faut trouver un équilibre entre les exigences émergentes du SdS et celles préexistantes des systèmes intégrés, parfois contradictoires en terme de performances, sécurité ou encore tolérance aux pannes. Le cycle de vie d'un SdS peut être long. Il faut compter des dizaines d'années pour des SdS tels que ceux du domaine de la défense, par exemple. Au cours d'un cycle, le SdS n'est pas figé. A tout moment, de nouveaux systèmes peuvent être ajoutés ou supprimés, à la conception comme à l'exécution. De plus, chaque système constituant peut subir des évolutions, des mises à jour, qui peuvent avoir une incidence sur l'ensemble du SdS.

Dans ce contexte, il ne suffit pas de simplement adapter les méthodes et outils existants de développement. Il est nécessaire de proposer des concepts, des langages, des méthodes et des outils permettant la construction de SdS capables de s'adapter dynamiquement pour continuer d'assurer leur mission malgré les évolutions internes ou externes qu'ils subissent. Des cycles de vie itératifs et flexibles doivent être envisagés [8][15].

De plus, pour gérer l'indépendance opérationnelle des systèmes constituants, il faut être capable de définir une architecture et de construire un SdS qui pourra remplir sa propre mission, sans violer l'indépendance de ses systèmes constituants. Face à ce défi, il est envisagé de s'attaquer, d'une part, à la modélisation et l'analyse de propriétés non-fonctionnelles des SdS, en particulier aux défis liés à la sécurité et, d'autre part, à l'analyse, au développement et à l'évolution dynamique des architectures de SdS.

Gérer la diversité et la variabilité à grande échelle

La taille ou la composition d'un système (ou d'un SdS) n'est pas le seul facteur de complexité qu'il faut gérer. Un autre facteur concerne la *diversité* qui apparaît dans tous les domaines d'application et dans toutes les activités de développement. Cette diversité s'exprime dans les fonctionnalités à fournir à différents utilisateurs, dans les langages et les modèles à utiliser conjointement et dans les environnements d'exécution [1], allant des réseaux de capteurs intelligents aux grandes infrastructures

1. Le nombre de plates-formes Android distinctes est passé d'environ 11000 à plus de 18000 entre 2013 et 2014 (`http://opensignal.com/reports/2014/android-fragmentation/`).

de calcul et de stockage de données. Plusieurs défis sont liés à cette problématique.

Concevoir un logiciel en fonction des besoins d'un client et l'adapter aux besoins d'un autre, en réutilisant une partie du code, ne peut plus se faire de façon artisanale. L'approche proposée par les lignes de produits logiciels permet de généraliser, et donc de maîtriser, la variabilité des logiciels. Cependant, la gestion et la modélisation de la variabilité à grande échelle restent un défi. Dès lors qu'il faut gérer des milliers, voire des millions de configurations de produits logiciels, l'identification, la définition et la composition des variants se doivent d'être accompagnées de méthodes et d'outils permettant le passage à l'échelle [17].

Un autre enjeu réside dans la diversification et la spécialisation des langages utilisés par les acteurs impliqués dans la construction des logiciels. Ces langages sont spécifiques à un domaine (DSL ou *Domain-Specific Language*), permettant ainsi à un expert de développer rapidement, et de façon fiable, un logiciel adapté à son expertise. Actuellement, le développement de logiciels demande de nombreuses expertises. L'un des défis est de composer et coordonner ces langages de modélisation spécifiques de façon à construire des logiciels multi-domaines[1].

Utiliser intensivement et facilement l'Ingénierie Dirigée par les Modèles

Malgré de nombreuses avancées faites en *Ingénierie Dirigée par les Modèles* (IDM), son application dans bon nombre de domaines se limite à l'abstraction comme seule dimension de modélisation. Plusieurs problèmes cruciaux du point de vue de l'usage des modèles ont été identifiés [22]. L'un des problèmes concerne l'ergonomie et une part limitée des travaux et outils liés à l'IDM prend en compte les processus cognitifs généralement à l'œuvre lors de la conception et le développement. Le non-alignement est donc fréquent et entraîne généralement des processus cognitifs supplémentaires chez le concepteur pour adapter les modèles ou les outils [6]. Si la syntaxe diagrammatique classiquement utilisée pour représenter les modèles répond pour partie aux attentes et visées fonctionnelles, il reste que celle-ci n'est pas suffisamment efficiente. La contribution du *design* peut s'avérer d'une aide précieuse en vue d'inventer et de concevoir de nouveaux langages de représentation et de visualisation graphique des modèles. Les applications courantes et futures nécessitent que la *multiplicité* des différentes sources de complexité soit maîtrisée en même temps (personnalisation et réactivité, accès aux non-informaticiens, prédictions et analyse, prise en compte des points de vue et des niveaux de compéten-ces) [5, 18].

Partant de ce constat, un certain nombre de défis peuvent être identifiés en croisant les considérations sociétales et cognitives à plus ou moins longue échéance [4] :

1. *Représentativité.* Le caractère de plus en plus trans-disciplinaire des applications et des systèmes doit nous inciter à, non seulement anticiper les difficultés techniques, mais aussi profiter de cette richesse. Cela exige aussi à repenser les formalismes de représentation, de description et de visualisation des modèles tout autant que les mécanismes utilisés en IDM pour les notations visuelles.
2. *Accessibilité.* Tandis que le logiciel est devenu réalité pour le grand public, l'IDM prône de placer les modèles au cœur des développements. C'est aux chercheurs dans le domaine de l'ingénierie dirigée par les modèles de rendre l'utilisation, la manipulation et la réalisation de modèles accessibles au plus grand nombre. Ceci implique aussi des modes d'interaction efficaces et intuitifs, qui exploitent au maximum les mécanismes visuels répertoriés (associativité, sélection, imposition, abstraction), et complémentés par des périphériques adaptés, c'est-à-dire flexibles et multi-dimensionnels.
3. *Flexibilité.* Les modèles doivent être à l'image des applications modernes. Ils seront ainsi plus

faciles à utiliser et pourront être composés ou intégrés de différentes façons. Il s'agit de pouvoir tenir compte, dans un domaine d'application donné, et dans un contexte dynamique, de toutes les préoccupations, techniques ou relatives au domaine, avec plus de confiance et de prévisibilité dans le résultat d'une intégration. Ceci pourrait être facilité par l'utilisation de modèles de référence, facilement personnalisables et intégrables, ainsi que par des environnements adaptés lors des désynchronisations entre codes exécutables et modèles.

4. *Evolutivité.* Un modèle n'est jamais aussi riche que la réalité. Il est donc par définition incomplet, alors que les outils de modélisation se sont souvent intéressés au caractère complet et formel des modèles. Pour ne pas être un frein à la créativité, ces outils devraient au contraire permettre, par exemple, qu'un modèle ne soit pas conforme à son méta-modèle pendant la phase d'élaboration. Ainsi les aspects cognitifs liés aux utilisateurs devraient être pris en compte dans ces outils dès leur création tout en ayant un impact sur les aspects techniques de maintenance, partage et évolution.

L'augmentation croissante de la complexité et de la diversité des systèmes, ainsi que leur regroupement en systèmes de systèmes sont des problèmes que notre communauté scientifique se doit de résoudre. Les chercheurs vont proposer de nouvelles approches pour favoriser l'adaptation dynamique, la sécurité, la composition, mais aussi maîtriser la variabilité à grande échelle et fournir des modèles ouverts, évolutifs et plus accessibles.

3 De la compilation à l'exécution des logiciels

La plupart des systèmes informatiques, des smartphones jusqu'aux accélérateurs de calcul (i.e., GPUs, FPGAs, super-calculateurs des futurs centres de calcul) sont désormais pourvus de systèmes multi-cœurs. Ces nouveaux matériels mettent le parallélisme à la portée de tous, mais restent d'une grande complexité. Le défi est double. Il faut d'une part maîtriser la difficulté de programmation de ces architectures tout en assurant une certaine portabilité des codes et des performances. D'autre part, il faut offrir une meilleure interaction entre les utilisateurs et les compilateurs. Cela nécessite de revisiter les langages, notamment parallèles, les techniques de compilation (analyse et optimisation de codes), les systèmes d'exploitation (OS), mais également les moyens de construire les logiciels avec des techniques avancées du génie logiciel, tout en respectant des contraintes liées à l'espace, au temps, mais de façon plus récente à la consommation d'énergie.

Maitriser la difficulté de programmation des architectures multi-coeurs

Dans ce cadre, la communauté Compilation et Calcul Haute Performance a identifié un ensemble de défis [7].

1. *Les langages parallèles*. Avec la mise sur le marché de plates-formes de plus en plus hétérogènes, le gain en performance des applications se fait maintenant en utilisant des processeurs dédiés à des tâches spécifiques. La programmation de tels systèmes logiciels/matériels devient beaucoup plus complexe, et les langages parallèles existants (MPI, openMP) semblent peu adaptés à cette tâche. Pour diminuer le coût du développement et du déploiement de logiciels sur des plates-formes spécifiques toujours changeantes, nous avons besoin d'approches et de langages qui permettent l'expression du parallélisme potentiel des applications et la compilation vers une plate-forme parallèle spécifique.

De plus, la complexité et la diversité de ces plates-formes justifient le développement de langages ou approches de haut niveau (comme X10, Chapel, OpenAcc, CAF, ou encore OpenStream) et d'optimisations au niveau source, en amont des langages natifs ciblés, dialectes souvent encore proches du langage C. Ceci offrira plus de portabilité de performances, mais aussi, pour l'utilisateur, une meilleure compréhension des transformations effectuées par le compilateur. L'amélioration de l'interface entre le programmeur et le compilateur est une perspective fondamentale à améliorer. L'interaction langage/compilateur/OS/environnement d'exécution (*runtime*) est également un point clé à améliorer, la portabilité de la performance passant par l'adaptation à la plate-forme via cette interaction.

2. *La compilation optimisante pour les performances en temps et en mémoire*. La consommation d'énergie des systèmes (embarqués surtout) est maintenant en grande partie due au mouvement et au stockage des données. Pour s'adapter à cette problématique, les données et leurs mouvements doivent être exposés au programmeur, et les compilateurs doivent prendre en compte la trace mémoire et les mouvements des données par exemple entre deux calculs qui s'effectuent sur des unités de calcul différentes. Des approches existent déjà en compilation, notamment en ce qui concerne la compilation vers FPGA, mais ce n'est que le début vers une prise en compte plus générique des contraintes de mémoire. Dans un sujet connexe, la recherche de solutions dont le pire cas peut être garanti (WCET) est rendue encore plus difficile par la présence de parallélisme. À l'heure actuelle, la compilation à performances prédictives reste un problème largement ouvert.
3. *La validation théorique des analyses et optimisations*. Les analyses de programme (séquentiel ou parallèle), les transformations de code, la génération de code pour des architectures précises, doivent être définies et prouvées précisément. Si accompagner les algorithmes de leur preuve de correction reste essentiel, des techniques comme la *preuve assistée* et la *translation validation* permettent maintenant de valider des compilateurs entiers (par exemple Compcert). Le challenge ici réside dans l'application plus générale des méthodes formelles (*cf.* section 4), que ce soit pour définir précisément des algorithmes ou pour valider des optimisations dans le cadre du parallélisme en particulier.
4. *La mesure et la reproductivité des résultats*. Contrairement aux autres sciences expérimentales comme par exemple les sciences naturelles, la branche expérimentale de l'informatique souffre d'un manque de principe scientifique : la reproductibilité et la vérification des résultats expérimentaux en informatique ne sont pas encore entrées dans nos habitudes. Dans le domaine de la compilation optimisante en particulier, lorsque des performances sont publiées, il est très rare que ces performances puissent être vérifiées ou observées par une partie tiers. Définir des méthodes expérimentales de validation statistique de résultats et des modèles présupposés (modèles de coût, modèles de programmation, abstractions) reste un défi majeur pour notre communauté, même si certaines conférences de notre domaine commencent à proposer des évaluations expérimentales [2].

Ces problématiques orientées compilation/langages sont aussi liées aux problématiques générales du Génie Logiciel que sont la complexité et l'hétérogénéité des logiciels. La spécificité ici est que la même complexité est à prendre en compte aussi bien côté matériel (plates-formes hétérogènes, accélérateurs matériels) que côté logiciel dès lors que l'on veut faire coopérer différents matériels et logiciels.

2. `http://evaluate.inf.usi.ch/artifacts`

S'inspirer du domaine génomique pour des pilotes de périphériques adaptables

L'Internet des objets est entravé par le fait que le développement de pilotes de périphérique reste une tâche complexe ainsi qu'une source d'erreurs et exige un haut niveau d'expertise, liée à la multiplicité des systèmes d'exploitation cibles et au dispositif visé. Il est nécessaire de proposer de nouvelles méthodologies qui amènent une rupture forte avec les méthodes actuelles de développement de pilotes de périphériques.

Le défi consiste alors à considérer *une nouvelle méthodologie pour comprendre les pilotes de périphériques, inspirée par le domaine de la génomique* [16]. Plutôt que de se concentrer sur le comportement des entrées/sorties d'un dispositif, cette nouvelle méthodologie reposera sur l'étude du code existant du pilote de périphérique de manière à le faire muter vers une version adaptée à l' environnement cible. D'une part, cette méthodologie devrait permettre d'identifier les comportements de pilotes de périphériques réels, de mettre à disposition ou non les caractéristiques du dispositif et du système d'exploitation, et d'améliorer les propriétés telles que la sécurité ou la performance. D'autre part, cette méthodologie a pour objectif de capter les patrons actuels de code utilisé pour réaliser des comportements, ceci en élevant le niveau d'abstraction à partir d'opérations individuelles vers des collections d'opérations réalisant une seule fonctionnalité, appelées gènes. Parce que les exigences du pilotes sont figées, quel que soit le système d'exploitation, les gènes ayant des comportements communs dans des OS différents devraient pouvoir être mutualités, même lorsque leur structure interne diffère. Cela devrait conduire à des pilotes de périphériques réalisés par une composition de gènes, ouvrant ainsi la porte à de nouvelles méthodes pour résoudre les problèmes de pilotes lors de développement ou lors du portage de pilotes existants vers d'autres systèmes d'exploitation.

La nouveauté de ce défi réside dans l'élévation du niveau d'abstraction de la compréhension du code des pilotes de périphérique à partir des opérations individuelles sur des gènes. La description et l'analyse des codes en termes de gènes fournissent un cadre pour le raisonnement sur les opérations connexes. En outre, les spécifications utilisées par les outils de traitement de code peuvent devenir plus portables et adaptables lorsqu'elles sont exprimées en termes de gènes, permettant à une seule spécification d'être appliquée de façon transparente à des variants d'un gène unique, quel que soit le code réel cible.

Les différentes étapes nécessaires pour réaliser ce défi concernant le développement de pilotes périphériques par composition de gènes sont les suivantes :

1. *Identifier les gènes en interaction avec le système d'exploitation*, dans un premier temps à la main puis automatiquement. De tels gènes impliquent généralement des fonctions API OS, et ont une structure commune.
2. *Identifier les gènes impliquant une interaction avec l'appareil* dans un premier temps à la main puis automatiquement. De tels gènes impliquent généralement des opérations de bas niveau, qui sont spécifiques à chaque appareil.
3. *Développer des techniques pour la composition de gènes* en vue de construire de nouveaux pilotes de périphérique.

Un autre problème, déjà partiellement abordé dans la section précédente, concerne l'hétérogénéité des plates-formes considérées - des capteurs intelligents aux grandes infrastructures de calcul - et les formes multiples des fautes et pannes à considérer. Le défi est alors double : il faut fournir les abstractions pour modéliser, déployer et adapter de manière homogène les différentes couches d'architectures logicielles et les besoins multiples, tout en revisitant les problématiques de tolérance aux pannes et de sécurité.

Optimiser la consommation d'énergie lors d'exécutions sur architectures multi-coeurs

Un dernier défi concerne l'omniprésence des logiciels dans notre vie quotidienne. Ceux-ci engendrent une part importante de la consommation énergétique globale française (13%). La communauté du génie logiciel peut jouer un rôle important dans la diminution significative et durable de cette consommation. En particulier, la consommation des logiciels s'exécutant sur des architectures multi-cœurs, qui sont aujourd'hui omniprésentes, que ce soit dans les serveurs ou les téléphones portables, demeure mal exploitée à ce jour : une part importante de l'énergie consommée est utilisée à mauvais escient.

Il est donc nécessaire de pouvoir comprendre finement comment les logiciels pilotent la consommation et d'identifier quels leviers peuvent être exploités dans les différentes couches d'un système informatique pour optimiser cette consommation en continu. L'éco-conception des logiciels apparaît donc aujourd'hui comme un défi crucial pour l'informatique afin de sensibiliser et guider les développeurs dans la réalisation de logiciels efficients. Cependant des gains notables ne peuvent être obtenus qu'en considérant des optimisations tout au long du cycle de vie du logiciel, depuis le recueil des besoins, durant leur exécution et jusqu'à leur maintenance, voire même leur recyclage.

La prise en compte des nouvelles caractéristiques matérielles pour construire les compilateurs, les pilotes de périphériques ou encore réduire la consommation d'énergie des logiciels semble évidente. Ce constat amène de nombreuses questions auxquelles les chercheurs de ces domaines devront apporter des réponses, aussi bien en termes de nouvelles structurations des logiciels qu'en termes d'optimisation de ces structures.

4 Méthodes et outils de validation et de vérification

Les nouveaux domaines d'application (informatique embarquée, intelligence ambiante, Internet des objets) et les nouvelles architectures (informatique dans les nuages, *Software as a Service* ou système de systèmes) font émerger de nouvelles propriétés ou contraintes (architectures reconfigurables, dynamicité de l'environnement, évolution des usages), en plus des propriétés classiques, telles que la sûreté et la sécurité. Un défi majeur consiste à revoir les méthodes de *Validation* et de *Vérification* (V&V) pour prendre en compte ces nouvelles architectures et les nouvelles propriétés associées.

Créer des méthodes évolutives

Une des difficultés est que les techniques actuelles de V&V requièrent la connaissance de la logique applicative pour garantir la qualité avant déploiement. Pour ce faire, elles s'appuient sur des spécifications de référence, qui n'anticipent pas a priori les prochaines évolutions. Dans ces conditions, il faut garantir la qualité de systèmes, qui vont devoir s'adapter à des univers flexibles et ouverts, et dont on ne peut prévoir les nouvelles fonctionnalités, les nouveaux usages, et les évolutions environnementales. Une des pistes peut être de s'appuyer sur un apprentissage de l'évolution incluant un processus de validation dynamique (pendant le déploiement), par exemple sur la forme de test passif ou de *monitoring* [9].

Mieux intégrer les méthodes dans le monde industriel

Au delà de la prise en charge de ces nouvelles propriétés, un défi récurrent pour la V&V est de faire pénétrer plus amplement les approches et outils dans le monde industriel.

Dans le cadre de la vérification de modèles (*model-checking*), l'impact dans l'industrie est, à ce jour, principalement limité aux systèmes embarqués critiques. Deux raisons principales sont d'une part, la réponse binaire à des propriétés de satisfaction qui n'est pas suffisamment informative, et d'autre part l'abstraction insuffisante pour répondre au réglage et à l'évolutivité des systèmes. Il faudra surmonter ces limitations, par exemple, en offrant des méthodes formelles paramétriques pour la vérification et l'analyse automatisée du comportement des systèmes. L'enjeu est d'obtenir des garanties sur la qualité des systèmes en fonctionnement, dès la phase de conception [2].

Parvenir à une meilleure automatisation des preuves déductives est elle-aussi essentielle, mais ce n'est pas le seul levier. Réutiliser spécifications et preuves associées serait un atout important. Modularité, héritage, paramétrisation facilitent la réutilisation, mais ne suffisent pas. Des approches inspirées des lignes de produits logiciels ou des approches à la carte ont été proposées, mais ces mécanismes demandent à être simplifiés. Un autre mode de réutilisation serait possible en développant un standard permettant l'interopérabilité entre les différents outils et formalismes, permettant ainsi la réutilisation de preuves dans un formalisme donné (*in the small*) ainsi que la réutilisation des preuves dans différents formalismes (*in the large*) [10].

Le test à partir de modèles ou *model-based testing* (*MBT*) représente le moyen principal pour automatiser la génération et l'exécution de tests fonctionnels [21]. En plus de permettre la traçabilité des exigences jusqu'aux tests, cette approche fournit des métriques d'avancement du processus de validation. Mais l'adoption des approches MBT dans l'industrie se heurte au problème de la conception de modèles. Les travaux sur l'inférence automatique de modèles ont montré leur intérêt et doivent être poursuivis. Par ailleurs, l'utilisation massive des approches MBT dans le monde industriel passe par un accompagnement des équipes de validation, et la construction de formalismes et outils adaptés [9].

Améliorer le passage à l'échelle des méthodes

Le passage à l'échelle est aussi un enjeu majeur pour l'adoption des différentes approches de V&V dans le monde industriel, notamment pour faire face à l'augmentation de la taille des systèmes. La capacité d'exprimer de grands modèles avec une représentation et des outils adaptés est un défi qui rejoint les problématiques déjà présentées section 2. Cependant le passage à l'échelle ne se limite pas à la seule expression de la spécification sous la forme d'un modèle. Par exemple, avec le recours aux techniques de développement agiles, les tests passent au premier plan, parfois construits avant le code, et ré-exécutés à chaque mise à jour. L'augmentation de la taille du code (et donc du nombre de tests) rend problématique la ré-exécution systématique de tous les tests. La prioritisation des tests devient essentielle. Dans ce contexte où l'application est amenée à évoluer, la maintenance de nombreux tests est un défi à part entière avec par exemple l'invalidation des tests devenus obsolètes ou encore la mise à jour des tests encore pertinents.

Améliorer la gestion des fautes

La détection de fautes ou de défaillances (bug) est l'objectif premier des techniques de V&V. Dans la plupart des domaines, la complexité et la richesse des spécifications, des utilisations et des environnements d'exécution rendent caduc le désir d'un logiciel sans faute. Le but de l'ingénieur face à ce fait n'est donc pas d'éradiquer toutes les fautes mais de gérer au mieux celles qui demeurent.

Les pionniers Avizienis, Laprie et Randell [3] ont défini quatre axes principaux dans la gestion des fautes : la prévention, la tolérance, l'estimation, et la suppression des fautes. A l'heure actuelle, les approches proposées concernent des fautes simples, les fautes difficiles étant laissées à l'expertise et l'intelligence humaine. Pour 2025, un défi consiste à envisager ces quatre tâches relatives à la gestion de faute comme des tâches d'intelligence artificielle, où, pour un type de faute donné, la machine devient comparablement aussi bonne que l'expert humain.

Ainsi, l'étape ultime de la suppression de faute est une réparation automatique [13]. Un logiciel intelligent analyse un rapport de défaillance, identifie la faute et corrige automatiquement le code source. Attaquer ce défi nécessite une approche interdisciplinaire. Premièrement, des experts en logiciel (génie logiciel, programmation, système) doivent collaborer avec des experts en décision (apprentissage, fouille de données, contraintes) [23]. Deuxièmement, une approche interdisciplinaire de la résilience des systèmes avec des chercheurs en biologie, écologie, physique et théorie des systèmes devrait faire sauter des verrous conceptuels dus au cloisonnement actuel.

Face à de nouveaux domaines et de nouvelles architectures logicielles, les méthodes de V&V se doivent d'évoluer et de mieux pénétrer le monde industriel. Les chercheurs du domaine s'attaquent à cette problématique en améliorant les techniques actuelles de réutilisation, d'automatisation en prenant en compte le passage à l'échelle, tout en les faisant évoluer. La détection de fautes et leur correction seront également revisitées en s'appuyant largement sur des approches interdisciplinaires.

5 Conclusion

Les trois challenges décrits par I. Sommerville [20] comme étant les défis du 21ème siècle pour le domaine du génie logiciel et de la programmation restent d'actualité : *i)* l'héritage de logiciels existant depuis de nombreuses années et devant continuer à fonctionner et à évoluer, *ii)* la prise en compte de l'hétérogénéité des matériels et des systèmes d'exploitation sur lesquels s'exécutent les logiciels qui doivent être construits de façon souple pour être adaptables à ces environnements, tout en étant fiables, et finalement *iii)* la diminution des délais de livraison des logiciels, traditionnellement longs, sans compromettre la qualité du système. D'autres documents récents font également état de défis dans le domaine du génie logiciel et de la programmation. On peut citer le NESSI *white paper* intitulé *Software Eenginnering Key Enabler for Innovation* [19] et le papier de la commission européenne écrit par ISTAG et intitulé *Software Technologies, The Missing Key Enabling Technology* [14]. Dans ces deux documents, nous retrouvons les défis précédemment cités, mais également l'importance de la coopération des académiques avec les industries du logiciel, le fait que des dépôts de logiciels open source soient créés et maintenus, et qu'un enseignement de qualité en génie logiciel et programmation soit assuré dans les universités et écoles d'ingénieurs.

Les trois grandes questions qui ont permis de classer les différents défis proposés par les chercheurs du GDR GPL reposent quant à elles sur les différentes étapes de construction des logiciels, allant de l'expression des besoins jusqu'à l'exécution et au delà par une réflexion sur l'évolution et la maintenance du logiciel. L'importance des nouveaux domaines d'application, mais également l'évolution des architectures matérielles obligent à revoir ces étapes, en y intégrant les nouvelles propriétés liées à ces nouveaux contextes. La nécessité de construire ces logiciels de façon fiable, mais également en prenant en compte la dimension du temps de mise sur le marché oblige les chercheurs à revoir les méthodes et outils pour aller plus rapidement vers des logiciels ajustés au mieux à leur contexte, tout en étant adaptables.

6 Remerciements

Les présentations et les échanges lors de la table ronde sur les défis 2025 des Journées Nationales du GDR GPL à Paris en Juin 2014 et la journée sur le défis 2025 en septembre 2014 à Paris ont permis de rédiger cette synthèse. Les auteurs adressent leurs remerciements à l'ensemble des collègues ayant participé à ces événements et ayant rédigé des contributions, avec un remerciement particulier à Catherine Dubois pour sa relecture attentive.

Références

[1] Mathieu Acher, Olivier Barais, Benoit Baudry, Arnaud Blouin, Johann Bourcier, Benoit Combemale, Jean-Marc Jézéquel, and Noel Plouzeau. Software diversity : Challenges to handle the imposed, opportunities to harness the chosen. In Dubois et al. [11], page 239.

[2] Etienne André, Benoît Delahaye, Peter Habermehl, Claude Jard, Didier Lime, Laure Petrucci, Olivier H. Roux, and Tayssir Touili. Beyond model checking : Parameters everywhere. In Dubois et al. [11], page 239.

[3] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, et al. *Fundamental concepts of dependability*. University of Newcastle upon Tyne, Computing Science, 2001.

[4] David Bihanic, Jean-Michel Bruel, Philippe Collet, Benoît Combemale, Sophie Dupuy-Chessa, Xavier Le Pallec, and Thomas Polacsek. L'IDM de demain : un accès facilité, un usage intensif pour des performances accrues, 2014.

[5] David Bihanic, Max Chevalier, Sophie Dupuy-Chessa, Thierry Morineau, Thomas Polacsek, Xavier Le Pallec, et al. Modélisation graphique des si : Du traitement visuel de modèles complexes. In *Inforsid 2013*, 2013.

[6] David Bihanic, Sophie Dupuy-Chessa, Xavier Le Pallec, and Thomas Polacsek. Manipulation et visualisation de modèles complexes. In Dubois et al. [11], page 239.

[7] Florian Brandner, Albert Cohen, Alain Darte, Paul Feautrier, Grigori Fursin, Laure Gonnord, and Sid Touati. Défis en complain, horizon 2025 , 2014.

[8] Vanea Chiprianov, Laurent Gallon, Manuel Munier, Philippe Aniorte, and Vincent Lalanne. The systems-of-systems challenge in security engineering. In Dubois et al. [11], page 239.

[9] Frédéric Dadeau and Hélène Waeselynck. Les défis du test logiciel - bilan et perspectives. In Dubois et al. [11], page 239.

[10] Gilles Dowek and Catherine Dubois. Réutiliser les spécifications et les preuves , 2014.

[11] Catherine Dubois, Laurence Duchien, and Levy Nicole, editors. *Actes des Sixièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel*, France, June 2014. Conservatoire National des Arts et Métiers.

[12] Laurence Duchien and Yves Ledru. Défis pour le Génie de la Programmation et du Logiciel GDR CNRS GPL. *Technique et Science Informatiques (TSI)*, 31(3) :397–413, March 2012.

[13] Claire Goues, Stephanie Forrest, and Westley Weimer. Current challenges in automatic software repair. *Software Quality Control*, 21(3) :421–443, September 2013.

[14] ISTAG. Software technologies, the missing key enabling technology- digital agenda for europe, 2012.

[15] LIUPPA Laboratoire and IRISA Laboratoire. Défis dans l'ingénierie logiciel des Systèmes de Systèmes , 2014.

[16] Julia Lawall and Gilles Muller. The future depends on the low-level stuff. In Dubois et al. [11], page 239.

[17] Andreas Metzger and Klaus Pohl. Software product line engineering and variability management : achievements and challenges. In *Proceedings of the on Future of Software Engineering*, pages 70–84. ACM, 2014.

[18] Gunter Mussbacher, Daniel Amyot, Ruth Breu, Jean-Michel Bruel, Betty HC Cheng, Philippe Collet, Benoît Combemale, Robert B France, Rogardt Heldal, James Hill, et al. The relevance of model-driven engineering thirty years from now. In *Model-Driven Engineering Languages and Systems*, pages 183–200. Springer International Publishing, 2014.

[19] Networked European Software NESSI White Paper and Services Initiative. Software engineering key enabler for innovation, 2014.

[20] Ian Sommerville. *Software Engineering : (Update) (8th Edition) (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[21] Mark Utting and Bruno Legeard. *Practical Model-Based Testing : A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[22] Jon Whittle, John Hutchinson, and Mark Rouncefield. The state of practice in model-driven engineering. *Software, IEEE*, 31(3) :79–85, 2014.

[23] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *Computer*, 42(8) :55–62, August 2009.

The Systems-of-Systems Challenge in Security Engineering

Vanea Chiprianov, Laurent Gallon, Manuel Munier, Philippe Aniorte, Vincent Lalanne
LIUPPA, Univ Pau & Pays Adour, France
Email: name.surname@adelaide.edu.au

***Abstract*—Systems of systems (SoS) are large-scale systems composed of complex systems with difficult to predict emergent properties. One of the most significant challenges in the engineering of such systems is how to model and analyse their Non-Functional Properties, such as security. In this paper we identify, describe, analyse and categorise some challenges to security engineering of SoS. This catalogue of challenges offers a roadmap of major directions for future research activities, and a set of requirements against which present and future solutions of security for SoS can be evaluated.**

I. Introduction

Strategic attacks on a nation's infrastructure represent a great risk of disruption and loss of life and property. As the National Security Advisor, Condoleezza Rice, noted on 22 March 2001: 'US businesses, which own and operate more than 90% of the nation's banks, electric power plants, transportation systems, telecommunications networks, and other critical systems, must be as prepared as the government for the possibility of a debilitating attack in cyberspace.' Compounding the vulnerability of such systems is their interdependencies, with the result that impacts of attacks on one system can cascade into other systems [16].

As critical infrastructures are getting more and more dependant on Information Communication Technologies (ICT), the protection of these systems necessitates providing solutions that consider the vulnerabilities and security issues found in computers and digital communication technologies. However, the ICT systems that support these critical infrastructures are ubiquitous environments of composed heterogeneous components, and diverse technologies. These systems exhibit a variety of security problems and expose critical infrastructures to cyber attacks. Theses security challenges spread computer networks, through different ICT areas such as: cellular networks, operating systems, software, etc.

II. Engineering of System-of-Systems

Critical infrastructures have been considered a type of a larger class of systems, called Systems-of-Systems (SoS). SoS are large-scale concurrent and distributed systems that are comprised of complex systems [13]. Several definitions of SoS have been advanced, some of them are historically reviewed in [11] for example. SoS are complex systems themselves, and thus are distributed and characterized by interdependence, independence, cooperation, competition, and adaptation [7].

Examples of SoS comprise critical infrastructures like: electric grid interconnected with other sectors [23], the urban transportation sector interconnected with the wireless network [2], but also home devices integrated into a larger home monitoring system, interoperability of clouds [27], maritime security [22], embedded time-triggered safety-critical SoS [24], federated health information systems [6], communities of banks [3], self-organizing crowd-sourced incident reporting [20]. For example, a systematic review of SoS architecture [15] identifies examples of SoS in different categories of application domains: 58 SoS in defence and national security, 20 in Earth observation systems, 8 in Space systems, 6 in Modelling and simulation, 5 in Sensor Networking, 4 in Healthcare and electric power grid, 3 in Business information system, 3 in Transportation systems.

Characteristics that have been proposed to distinguish between complex but monolithic systems and SoS are [17]:

- *Operational Independence of the Elements*: If the SoS is disassembled into its component systems the component systems must be able to usefully operate independently. The SoS is composed of systems which are independent and useful in their own right.
- *Managerial Independence of the Elements*: The component systems not only *can* operate independently, they *do* operate independently. The component systems are separately acquired and integrated but maintain a continuing operational existence independent of the SoS.
- *Evolutionary Development*: The SoS does not appear fully formed. Its development and existence is evolutionary with functions and purposes added, removed, and modified with experience.
- *Emergent Behaviour*: The SoS performs functions and carries out purposes that do not reside in any component system. These behaviours are emergent properties of the entire SoS and cannot be localized to any component system. The principal purposes of the SoS are fulfilled by these behaviours.
- *Geographic Distribution*: The geographic extent of the component systems is large. Large is a nebulous and relative concept as communication capabilities increase, but at a minimum it means that the components can readily exchange only information and not substantial quantities of mass or energy.

Taking into account these characteristics specific to SoS needs specific engineering approaches. Most researchers agree that the SoS engineering approaches need to be different from

the traditional systems engineering methodologies to account for the lack of holistic system analysis, design, verification, validation, test, and evaluation [13], [5]. There is consensus among researchers [4], [18] and practitioners [1] that these characteristics necessitate treating a SoS as something different from a large, complex system. Therefore, SoS is treated as a distinct field by many researchers and practitioners.

III. Challenges in Security Engineering of Systems-of-Systems

Security engineering within SoS and SoS security life-cycle are influenced by SoS engineering and the SoS life-cycle. They need to take into account the characteristics specific to SoS, and how they impact security of SoS. At a general, abstract level, these impacts include [25]:

- *Operational Independence*: In an SoS, the component systems may be operated separately, under different policies, using different implementations and, in some cases, for multiple simultaneous purposes (i.e. including functions outside of the SoS purpose under consideration). This can lead to potential incompatibilities and conflict between the security of each system, including different security requirements, protocols, procedures, technologies and culture. Additionally, some systems may be more vulnerable to attack than others, and compromise of such systems may lead to compromise of the entire SoS. Operational independence adds a level of complexity to SoS that is not present in single systems.
- *Managerial Independence*: Component systems may be managed by completely different organisations, each with their own agendas. In the cyber security context, activities of one system may produce difficulties for the security of another system. What rights should one system have to specify the security of another system for SoS activities and independent activities? How can systems protect themselves within the SoS from other component systems and from SoS emerging activities? Does greater fulfilment require a component system to allow other component systems to access it?
- *Evolutionary Development*: An SoS typically evolves over time, and this can introduce security problems that the SoS or its components do not address, or are not aware of. Therefore, the security mitigations in place for an evolving SoS will be difficult to completely specify at design time, and will need to evolve as the SoS evolves.
- *Emergent Behaviour*: SoS are typically characterised by emerging or non-localised behaviours and functions that occur after the SoS has been deployed. These could clearly introduce security issues for the SoS or for its component systems, and therefore the security of the SoS will again need to evolve as the SoS evolves. In addition, responsibility for such behaviours could be complex and shared, leading to difficulties in deciding who should respond and where responses are needed.
- *Geographic Distribution*: An SoS is often geographically dispersed, which may cause difficulties in trying to secure the SoS as a whole if national regulations differ. These may restrict what can be done at different locations, and how the component systems may work together to respond to a changing security situation.

Identifying challenges to security engineering within SoS is the first step in engineering security within SoS. As highlighted by [18], a very desirable research direction would be an integrated description and analysis method that can express and guarantee user level security, reliability, and timeliness properties of systems built by integrating large application layer parts - SoS. Moreover, systems engineering of defence systems and critical infrastructure must incorporate consideration of threats and vulnerabilities to malicious subversion into the engineering requirements, architecture, and design processes; the importance and the challenges of applying System Security Engineering beyond individual systems to SoS has been recognized [8]. Additionally, secure cyberspace has been recognized as one of the major challenges for 21st century engineering [26], [14].

Starting from the challenges related to characteristics specific to SoS, we further identify, describe and analyse challenges to security engineering of SoS. We organise them according to the activity of the security process in which they have the most impact. Of course, most challenges impact several activities, but for clarity purposes, we present them in the activity in which we consider they have the most impact.

A. Challenges impacting all Activities

Long life of SoS How to approach constraints associated with these legacy systems? Consequently, will most SoS be composed of systems with uneven levels of system protection?

B. Requirements Challenges

Identifying SoS security requirements How to identify these SoS overarching security requirements?

Security requirements modelling How can security be integrated into requirements modelling? How can a balance between near-term and long-term security requirements be achieved?

Ownership Who should have the ultimate ownership responsibility for the SoS? Who will be responsible for dealing with issues arising from the SoS, for example if the system was used for malicious purposes, who would be legally culpable? Who will be responsible for testing and proving the system is running as expected and fulfilling its security requirements?

Risk management How to identify and mitigate risks associated with end-to-end flow of information and control, without, if possible, focusing on risks internal to individual systems?

Holistic security Information security comprises: 1) Physical software systems security based on applying computer cryptography and safety or software criticality implementation; 2) Human / personnel security based on the procedure, regulations, methodologies that make an organisation / enterprise / system safe; 3) Cyber / Networking level that is mainly concerned with controlling cyber attacks and vulnerabilities and reducing their effects [19]. How can such holistic standards

be extended to encompass SoS? How can they be applied and enforced in the context of SoS?

Requirements as source of variability How to adequately identify and allocate requirements to constituent systems for their respective teams to manage?

Security metrics for SoS What could be security-specific metrics and measures for an SoS? Is it possible to define a set of metrics which can be evaluated on the entire SoS, or are some security assessments limited to subparts of the SoS? Is it possible to define probability-theoretic metrics that can be associated with prediction models? How the mix of deterministic and uncertain phenomena, that come into play when addressing the behaviour of a SoS faced with malicious attacks, can be represented?

C. Design Challenges

Bridging the gap between requirements and design How to breach the gap between frameworks and implementation? How to assure a level of system and information availability consistent with stated requirements?

Designing security How can security be integrated into the SoS architecture? How to represent an exchange policy specification so as to verify some properties like: completeness, consistency, applicability and minimality?

Interdependency analysis How to identify threats that may appear insignificant when examining only first-order dependencies between composing systems of a SoS, but may have potentially significant impact if one adopts a more macroscopic view and assesses multi-order dependencies? How to asses the hidden interdependencies? How to represent the interdependencies existing among a group of collaborating systems? How such an approach can be integrated in a risk assessment methodology in order to obtain a SoS risk assessment framework? How to understand dependencies of a constituent system, on systems that are external to the formal definition of the SoS, but that nonetheless have security-relevant impacts to SoS capabilities?

New architectural processes Which would be the best suited process for architecting SoS and its security? Should it contain iterative elements, should it be agile, or model-based, etc? How does the type of dependencies between the development of SoS and the development of its constituent systems influence the design process of the SoS?

Design for evolution It is not sensible to assume that present security controls will provide adequate protection of a future SoS. Should there be a transition from system design principles based on establishing defensive measures aimed at keeping threats at bay, to postures that maintain operations regardless of the state of the SoS, including compromised states?

Scalability of security A larger number of users can interact with the SoS than with any of its composing systems. This means a possible increased number and/or scale of attacks. How can the security mechanisms for SoS be scaled up consequently?

Multiplicity of security mechanisms There are different security mechanisms at different levels. Defensive capabilities include for example physical security measures, personnel security measures, configuration control, intrusion detection, virus and mal-ware control, monitoring, auditing, disaster recovery, continuity of operations planning [10], cryptography, secure communications protocols, and key management methods that are time tested, reviewed by experts, and computationally sound [9]. How to use together effectively and efficiently all these mechanisms?

D. Implementation Challenges

Authentication The confirmation of a stated identity is an essential security mechanism in standalone systems, as well as in SoS. To achieve system interoperability, authentication mechanisms have to be agreed upon among systems to facilitate accessing resources from each system. How and when can this agreement be reached?

Authorisation In a SoS, users with different backgrounds and requirements should be granted accesses to different resources of each composing system. Therefore, a proper authorization mechanism is necessary for the composing systems to cooperate together and provide the best user experience possible for the SoS users [27]. How would delegation of rights be handled? Who would be responsible for it?

Accounting / Auditing In conjunction to security, accounting is necessary for the record of events and operations, and the saving of log information about them, for SoS and fault analysis, for responsibility delegation and transfer, and even digital forensics. Where will this information be tracked and stored and who will be responsible for the generation and maintenance of logs?

Non-Repudiation How can an evidence of the origin of any change to certain pieces of data be obtained in the context of an SoS? Who should collect these data, who can be trusted?

Encryption Encryption mechanisms should be agreed upon in order for SoS users from different endpoints to access the resources of a SoS. Cryptographic keys must be securely exchanged, then held and protected on either end of a communications link. This is challenging for a utility with numerous composing systems [9].

Security classification of data How to provide the ability to securely and dynamically share information across security domains while simultaneously guaranteeing the security and privacy required to that information? How to define multiple security policy domains and ensure separation between them?

Meta-data What kind of data should meta-data contain? What kind of meta-data should be legally-conformant to collect and employ? What kind of meta-data would technically be available? Should meta-data tags include data classification to provide controlled access, ensure security, and protect privacy? Should meta-data be crypto-bound to the original data to ensure source and authenticity of contents?

Heterogeneity and multiplicity of platforms How to detect cross-protocol, cross-implementation and cross-infrastructure vulnerabilities? How to correlate information across systems to identify such vulnerabilities and attacks?

E. Verification Challenges

Verifying the implementation satisfies the requirements When multiple, interacting components and services are involved, verifying that the SoS satisfies chosen security controls

increases in complexity over standalone systems. This complexity is because the controls must be examined in terms of their different applications to the overall SoS, the independent composing systems, and their information exchange [12].

F. Release/Response Challenges

Configuration Who will be responsible for investigating any configuration issues and performing changes?

Monitoring Who will be responsible for monitoring addressing any faults or issues that may occur?

Runtime re-engineering In some cases, the SoS is only created at runtime, and the exact composition may not be known in advance. However, security currently takes time to establish, and there are many interrelated security issues that could create delay or loss of critical information. For some applications, runtime delays will have a big impact. Balance is therefore required in order to ensure security doesn't have a negative impact on operational effectiveness [21].

IV. Conclusions

In this paper we provided a catalogue of challenges that have been identified in the literature regarding the subject of security engineering for Systems-of-Systems (SoS). Organised according to the security process activities, they represent an easy to consult, clear roadmap of major directions for future research. Future research can position their research questions according to the challenges identified here. Moreover, these challenges can serve as a set of requirements against which existing and future solutions to security engineering of SoS can be evaluated.

References

[1] Systems engineering guide for systems of systems, version 1.0., 2008.

[2] C. Barrett, R. Beckman, K. Channakeshava, Fei Huang, V.S.A. Kumar, A. Marathe, M.V. Marathe, and Guanhong Pei. Cascading failures in multiple infrastructures: From transportation to communication network. In *Critical Infrastructure (CRIS), 2010 5th International Conference on*, pages 1–8, Sept 2010.

[3] Walter Beyeler, Robert Glass, and Giorgia Lodi. Modeling and risk analysis of information sharing in the financial infrastructure. In Roberto Baldoni and Gregory Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 41–52. Springer Berlin Heidelberg, 2012.

[4] J. Boardman and B. Sauser. System of systems - the meaning of OF. In *System of Systems Engineering, 2006 IEEE/SMC International Conference on*, pages 6 pp.–, April 2006.

[5] Roland T. Brooks and Andrew P. Sage. System of systems integration and test. *Information, Knowledge, Systems Management*, 5:261–280, 2006.

[6] Mario Ciampi, Giuseppe Pietro, Christian Esposito, Mario Sicuranza, Paolo Mori, Abraham Gebrehiwot, and Paolo Donzelli. On securing communications among federated health information systems. In Frank Ortmeier and Peter Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *Lecture Notes in Computer Science*, pages 235–246. Springer Berlin Heidelberg, 2012.

[7] Cihan H. Dagli and Nil Kilicay-Ergin. *System of Systems Architecting*, pages 77–100. John Wiley & Sons, 2008.

[8] J. Dahmann, G. Rebovich, M. McEvilley, and G. Turner. Security engineering in a system of systems environment. In *Systems Conference (SysCon), 2013 IEEE International*, pages 364–369, April 2013.

[9] Michael Duren, Hal Aldridge, Robert K. Abercrombie, and Frederick T. Sheldon. Designing and operating through compromise: Architectural analysis of ckms for the advanced metering infrastructure. In *Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop*, CSIIRW '13, pages 48:1–48:3, New York, NY, USA, 2013. ACM.

[10] D.L. Farroha and B.S. Farroha. Agile development for system of systems: Cyber security integration into information repositories architecture. In *IEEE Systems Conference*, SysCon, pages 182 –188, April 2011.

[11] A. Gorod, R. Gove, B. Sauser, and J. Boardman. System of systems management: A network management approach. In *System of Systems Engineering, 2007. SoSE '07. IEEE International Conference on*, pages 1–5, April 2007.

[12] J. Hosey and R. Gamble. Extracting security control requirements. In *6th Annual Workshop on Cyber Security and Information Intelligence Research*, CSIIRW, 2010.

[13] M. Jamshidi. System of systems - innovations for 21st century. In *Industrial and Information Systems, 2008. ICIIS 2008. IEEE Region 10 and the Third international Conference on*, pages 6–7, Dec 2008.

[14] Roy S. Kalawsky. The next generation of grand challenges for systems engineering research. *Procedia Computer Science*, 16(0):834 – 843, 2013. 2013 Conference on Systems Engineering Research.

[15] John Klein and Hans van Vliet. A systematic review of system-of-systems architecture research. In *Proceedings of the 9th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '13, pages 13–22, New York, NY, USA, 2013. ACM.

[16] S.J. Lukasik. Vulnerabilities and failures of complex systems. *Int. J. Eng. Educ.*, 19(1):206–212, 2003.

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

[18] M.W. Maier. Research challenges for systems-of-systems. In *Systems, Man and Cybernetics, 2005 IEEE International Conference on*, volume 4, pages 3149–3154, Oct 2005.

[19] E.I. Neaga and M.J. de C Henshaw. Modeling the linkage between systems interoperability and security engineering. In *5th International Conference on System of Systems Engineering*, SoSE, June 2010.

[20] Craig Nichols and Rick Dove. Architectural patterns for self-organizing systems-of-systems. *Insight*, 4:42–45, 2011.

[21] Charles E. Phillips, Jr., T.C. Ting, and Steven A. Demurjian. Information sharing and security in dynamic coalitions. In *Proceedings of the Seventh ACM Symposium on Access Control Models and Technologies*, SACMAT '02, pages 87–96, New York, NY, USA, 2002. ACM.

[22] Nicola Ricci, Adam M. Ross, and Donna H. Rhodes. A generalized options-based approach to mitigate perturbations in a maritime security system-of-systems. *Procedia Computer Science*, 16(0):718 – 727, 2013. 2013 Conference on Systems Engineering Research.

[23] S.M. Rinaldi, J.P. Peerenboom, and T.K. Kelly. Identifying, understanding, and analyzing critical infrastructure interdependencies. *Control Systems, IEEE*, 21(6):11–25, Dec 2001.

[24] Florian Skopik, Albert Treytl, Arjan Geven, Bernd Hirschler, Thomas Bleier, Andreas Eckel, Christian El-Salloum, and Armin Wasicek. Towards secure time-triggered systems. In *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security*, SAFECOMP'12, pages 365–372, Berlin, Heidelberg, 2012. Springer-Verlag.

[25] A. Waller and R. Craddock. Managing runtime re-engineering of a system-of-systems for cyber security. In *System of Systems Engineering (SoSE), 2011 6th International Conference on*, pages 13–18, June 2011.

[26] W. A. Wulf. Great achievements and grand challenges. Technical report, National Academy of Engineering, 2000.

[27] Zhizhong Zhang, Chuan Wu, and David W.L. Cheung. A survey on cloud interoperability: Taxonomies, standards, and practice. *SIGMETRICS Perform. Eval. Rev.*, 40(4):13–22, April 2013.

Software Diversity: Challenges to handle the imposed, Opportunities to harness the chosen

Mathieu Acher, Olivier Barais, Benoit Baudry, Arnaud Blouin, Johann Bourcier, Benoit Combemale, Jean-Marc Jézéquel, and Noel Plouzeau

DiverSE team at INRIA / IRISA

1 Context

Diversity emerges as a critical concern that spans all activities in software engineering (from design to verification, from deployment to runtime resilience) and appears in all sorts of domains, which rely on software intensive systems, from systems of systems to pervasive combinations of Internet of Things and Internet of Services. If these domains are apparently radically different, we envision a strong convergence of the scientific principles underpinning their construction and validation towards **flexible and open yet dependable systems**.

In this paper, we discuss the software engineering challenges raised by these requirements for flexibility and openness, focusing on four dimensions of diversity: the **diversity of functionalities** required by the different customers (Section 2); the **diversity of languages** used by the stakeholders involved in the construction of these systems (Section 3); the **diversity of runtime environments** in which software has to run and adapt (Section 4); the **diversity of failures** against which the system must be able to react (Section 5). In particular, we want to emphasize the **challenges for handling imposed diversity**, as well as the **opportunities to leverage chosen diversity**. The main challenge is that software diversity imposes to integrate the fact that software must adapt to changes in the requirements and environment – in all development phases and in unpredictable ways. Yet, exploiting and increasing software diversity is a great opportunity to allow the spontaneous exploration of alternative software solutions and proactively prepare for unforeseen changes. Concretely, we want to provide software engineers with the ability:

- to characterize an 'envelope' of possible variations;
- to compose 'envelopes' (to discover new macro envelopes in an opportunistic manner);
- to dynamically synthesize software inside a given envelop.

The major scientific challenge we foresee for software engineering is elicited below

> Automatically **compose and synthesize software diversity** from design to runtime to **address unpredictable evolutions of software intensive systems**.

2 Diversity of functionalities

2.1 Imposed diversity: diversity of requirements and usages

The growing adoption of software in all sectors of our societies comes with a growing diversity of usages (from pure computation in its early days, to a variety ranging from transportation, energy, economy, communication, games and manufacturing today). This variety of usages and users puts pressure on software development companies, who aim at reusing as much code as possible from one customer to

another, yet who want to build the product that fits the user specific requirements. *Software Product Lines* (SPL) have emerged as a way to handle this challenge (reuse, yet be specific) [1]. Central to both processes is the management and modeling of variability across a product line of software systems. Variability is usually expressed in terms of *features*, originally defined by Kang et al. as: *"a prominent or distinctive user-visible aspect, quality or characteristic of a software system or systems"* [2].

A fundamental problem is that the number of variants can be exponential in the number of features: 300 boolean optional features lead to approximately 10^{90} configurations. Practitioners thus face the challenge of developing billions of variants. It is easy to forget a necessary constraint, leading to the synthesis of unsafe variants, or to under-approximate the capabilities of the software platform. Scalable modelling techniques are therefore crucial to specify and reason about a very large set of variants.

Challenge #1: scalable management of variability

2.2 Chosen diversity: adaptive systems in evolving environments

Software systems now need to dynamically evolve to fit changes in their requirements (e.g., change of environment, user, or platform) at runtime. The growing adoption and presence of software is a factor of chosen diversity that can answer this problem. Such a diversity is composed available software services developed and deployed by third parties that software systems can exploit at runtime to fit their current requirements. The challenge is to develop (self-)adaptive systems that can smoothly discover, select, and integrate available services at runtime.

Opportunity #1: exploiting ambient functionalities within adaptive systems

3 Diversity of languages

3.1 Imposed diversity: diversity of views and paradigms in systems engineering

Past research on modeling languages focused on technologies for developing languages and tools that allow domain experts to develop system solutions efficiently, i.e., domain-specific modeling languages (DSMLs) [3, 4]. A new generation of complex software-intensive systems, for example, smart health, smart grid, building energy management, and intelligent transportation systems, presents new opportunities for leveraging modeling languages. The development of these systems requires expertise in diverse domains.

Consequently, different types of stakeholders (e.g., scientists, engineers and end-users) must work in a coordinated manner on various aspects of the system across multiple development phases. DSMLs can be used to support the work of domain experts who focus on a specific system aspect, but they can also provide the means for coordinating work across teams specializing in different aspects and across development phases. The support and integration of DSMLs leads to what we call the globalization of modeling languages, i.e., the use of multiple languages for the coordinated development of diverse aspects of a system. One can make an analogy with world globalization in which relationships are established between sovereign countries to regulate interactions (e.g., travel and commerce related interactions) while preserving each countryś independent existence.

Challenge #2: globalization of domain-specific languages

3.2 Chosen diversity: proactive diversification of computation semantics

We see an opportunity for the automatic diversification of program's computation semantics, for example through the diversification of compilers or virtual machines. The main impact of this artificial diversity is to provide flexible computation and thus ease adaptation to different execution conditions. A combination

of static and dynamic analysis, could support the identification of what we call "plastic computation zones" in the code. We identify different categories of such zones: (i) areas in the code in which the order of computation can vary (e.g. the order in which a block of sequential statements is executed); (ii) areas that can be removed, keeping the essential functionality [5] (e.g., skip some loop iterations); (iii) areas that can replaced by alternative code (e.g., replace a try-catch by a return statement). Once we know which zones in the code can be randomized, it is necessary to modify the model of computation to leverage the computation plasticity. This consists in introducing variation points in the interpreter to reflect the diversity of models of computation. Then, the choice of a given variation is performed randomly at runtime.

Opportunity #2: flexible computation

4 Diversity of runtime environments

4.1 Imposed diversity: diversity of devices and execution environments

Flexible yet dependable systems have to cope with heterogeneous hardware execution platforms ranging from smart sensors to huge computation infrastructures and data centers. Evolutions range from a mere change in the system configuration to a major architectural redesign, for instance to support addition of new features or a change in the platform architecture (new hardware is made available, a running system switches to low bandwidth wireless communication, a computation node battery is running low, etc).

In this context, we need to devise formalisms to reason about the impact of an evolution and about the transition from one configuration to another [6, 7]. The main challenge is to provide new homogeneous architectural modelling languages and efficient techniques that enable continuous software reconfiguration to react to changes. The main challenge is to handle the diversity of runtime infrastructures, while managing the cooperation between different stakeholders. This requires abstractions (models) to (i) systematically define predictable configurations and variation points – see also the challenge of Section 2 – through which the system will evolve ; (ii) develop behaviors necessary to handle unpredicted evolutions.

Challenge #3: effective deployment and adaptation over heterogeneous platforms

4.2 Chosen diversity: diversity of distribution and deployment strategies

Diversity can also be an asset to optimize software architecture. Architecture models must integrate multiple concerns in order to properly manage the deployment of software components over a physical platform. However, these concerns can contradict each other (*e.g.*, accuracy and energy). This context, provides new opportunities to investigate solutions, which systematically explore the set of possible architecture models and establish valid trade-offs between all concerns in case of changes.

Opportunity #3: continuous exploration and improvement of software architecture

5 Diversity of failures

5.1 Imposed diversity: diversity of accidental and deliberate faults

One major challenge to build flexible and open yet dependable systems is that current software engineering techniques require architects to foresee all possible situations the system will have to face. However, openness and flexibility also mean unpredictability: unpredictable bugs, attacks, environmental evolutions, etc. Current fault-tolerance [8] and security [9] techniques provide software systems with the capacity of detecting accidental and deliberate faults. However, existing solutions assume that the set of bugs or vulnerabilities in a system do not evolve. This assumption does not hold for open systems, thus

it is essential to revisit fault-tolerance and security solutions to account for diverse and unpredictable faults.

Challenge #4: adaptive software resilience

5.2 Chosen diversity: diversity of and redundancy of software components

Current fault-tolerance and security are based on the introduction software diversity and redundancy in the system. There is an opportunity to enhance these techniques in order to cope with a wider diversity of faults, by multiplying the levels of diversity in the different software layers that are found in software intensive systems (system, libraries, frameworks, application). This increased diversity must be based on artificial program transformations and code synthesis, which increase the chances of exploring novel solutions, better fitted at one point in time. The biological analogy also indicates that diversity should emerge as a side-effect of evolution, to prevent over-specialization towards one kind of diversity.

Opportunity #4: synthetic, emergent software diversity

References

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag, 2005.

[2] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (foda) feasibility study," tech. rep., Carnegie-Mellon University Software Engineering Institute, November 1990.

[3] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in *Proc. of Future of Software Engineering*, pp. 37–54, 2007.

[4] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen, "Empirical assessment of MDE in industry," in *Proc. of the Int. Conf. on Software Engineering (ICSE)*, pp. 471–480, 2011.

[5] Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. C. Rinard, "Randomized accuracy-aware program transformations for efficient approximate computations," in *Proc. of the Symp. on Principles of Programming Languages (POPL)*, pp. 441–454, 2012.

[6] B. H. Cheng, R. De Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, *et al.*, "Software engineering for self-adaptive systems: A research roadmap," in *Software engineering for self-adaptive systems*, pp. 1–26, 2009.

[7] G. Blair, N. Bencomo, and R. B. France, "Models@run.time," *IEEE Computer*, vol. 42, no. 10, pp. 22–27, 2009.

[8] B. Randell, "System structure for software fault tolerance," *IEEE Trans. on Software Engineering*, vol. 1, no. 2, 1975.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proc. of the Symp. on Security and Privacy (S&P)*, pp. 120–128, 1996.

Beyond Model Checking: Parameters Everywhere

Étienne André[1], Benoît Delahaye[2], Peter Habermehl[3], Claude Jard[2], Didier Lime[4], Laure Petrucci[1], Olivier H. Roux[4], Tayssir Touili[3]

[1] Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, Villetaneuse, France
[2] LINA/Université de Nantes, France
[3] LIAFA, Université Paris Diderot – Paris7, France
[4] IRCCyN, École Centrale de Nantes, France

Keywords: parameter synthesis, parametric model checking, discrete parameters, continuous parameters

Beyond Model Checking... After many years of academic research on model checking, its impact in industry is mostly limited to critical embedded systems, and thus is somewhat disappointing w.r.t. the expectations. Two major reasons are the binary response to properties satisfaction, which is not informative enough, and the insufficient abstraction to cater for tuning and scalability of systems.

A major challenge is to overcome these limitations by providing parametric formal methods for the verification and automated analysis of systems behaviour.

...are Parameters The challenge is clearly to obtain guarantees on the quality of the systems in operation, quality being evaluated during the design phase. For any given level of abstraction, we want to maintain the formal description of the behaviour of the system together with its expected properties. The current verification techniques ensure that the properties are true for all possible behaviours of a given instance and environment of the system. Hence the utmost importance of a characterisation of the conditions under which the properties are guaranteed to hold, in particular since systems are often incompletely specified or with an environment unknown *a priori*.

In order to broaden the applicability of formal modelling methods within the wide range of digital world that is being built, a key point is the control of abstraction in the models. A main challenge is to develop the theory and implementation of the verification of parameterised models. This area of research is still in its infancy and a significant advance should be performed, by a coordinated study of several types of parameters: discrete (e.g. number of threads, size of counters), timed (deadlines, periods), continuous costs (energy, memory), and probabilistic (redundancy, reliability).

Being able to treat these parameterised models constitutes a scientific breakthrough in two ways:

- It significantly increases the level of abstraction in models. It will be possible to handle a much larger and therefore more *realistic class of models*.
- The existence of parameters can also address more relevant and *realistic verification issues*. Instead of just providing a binary response to the satisfaction of expected properties, constraints on the parameters can be synthesised. These constraints can either ensure satisfaction of the expected properties when this is possible, or provide quantitative information in order to *optimise* satisfaction of *some* properties w.r.t. parameter values. Such information are highly valuable to designers for the proper behaviour of the systems they develop.

Towards a Safe Digital Society With the booming broadening of software and hardware devices in our lives, the need for safe, secure and predictable systems becomes higher and higher. Hence, methods for formally verifying these systems are strongly needed. Model checking techniques used in the design phase of a system prove the system either correct or incorrect, in which case the design phase may have to restart from the beginning, thus implying a high cost. This binary answer is certainly one of the key reasons explaining why formal methods are not as widespread as they could be. Parameter synthesis overcomes this drawback by directly providing the designer with sufficient working conditions, hence allowing to consider systems only partially specified, or with an only partially known environment. Efficient and effective parameter synthesis techniques shall broaden the use of formal methods in the software and hardware industry towards a safe digital society. The modelling and derivation of formal conditions ensuring a good behaviour is a clear step towards a digital and software industry able to guarantee and ensure its products, thus becoming a more mature industry. This is in particular of utmost importance for the development of the open source software industry.

Challenges One of the key challenges in the area of parameter synthesis, that we hope to be solved in 2025, is the definition of *decidable* subclasses of existing formalisms and problems. Almost all interesting parameter synthesis problems are known to be undecidable. However, in the past few years, some problems were shown to be decidable, in particular integer parameter synthesis, or synthesis of the system robustness, which are subproblems of the main parameter synthesis issue.

Another key challenge is the ability to combine parameters, viz., discrete parameters and continuous parameters, altogether. This also implies the

definition of adequate formalisms, either decidable, or with efficient semi-decidable algorithms.

Applications Beyond classical applications (hardware verification, process management, embedded systems), typical applications in the near future are *smart homes*, in particular catering for elderly or disabled people in a safe manner. Parameterisation there characterises the adaptation of the system to a specific subject, either in a static manner (list of parameters to be instantiated when the managing software is installed for a specific person) or in a dynamic manner (parameters regularly improved following new living conditions). Additionally, the use of costs in parameter synthesis typically addresses the reduction of energy consumption, either by managing the home, or performing medical surveillance through sensor networks.

Les défis du Test Logiciel - Bilan et Perspectives

Réponse à l'appel à défis GDR-GPL 2025 – Mars 2014

Frédéric Dadeau – FEMTO-ST Hélène Waeselynck – LAAS

Résumé

Ce document dresse un bilan des défis identifiés par le groupe de travail Méthodes de Test pour la Vérification et la Validation (MTV2) lors de l'appel lancé par le GDR GPL en 2010. Pour chaque défi initialement identifié, nous évaluons si des réponses ont été apportées durant ces 4 dernières années, nous présentons les éventuelles avancées réalisées, et proposons le cas échéant de nouveaux défis liés aux technologies émergentes.

1 Le défi des techniques de test

1.1 Test à partir de modèles et de code

Lors du précédent appel à défi, nous partions du constat que code et modèles, deux artefacts de la génération de test, étaient réalisés indépendamment. Un défi consistait à chercher une plus grande intégration entre *code-based testing* (CBT) et *model-based testing* (MBT). Différentes approches ont mise en avant l'utilisation de langages d'annotations, notamment au CEA, avec la définition du langage ACSL (ANSI-C Specification Language) et le couplage entre la plateforme Frama-C et l'outil de génération de test PathCrawler. En déportant les éléments de modèle au sein du code, les langages d'annotation apportent également une réponse au besoin de faire évoluer conjointement code et modèle lors du cycle de vie du logiciel. Néanmoins, l'expressivité des langages d'annotations restreint leur utilisation à la génération de tests unitaires, et n'adresse pas directement la problématique du test de recette, comme pourraient le faire d'autres formalismes. La conception conjointe de modèle pour le test et de code n'est donc pas encore d'actualité.

1.2 Passage à l'échelle de technologies

L'un des enjeux techniques majeurs est le passage à l'échelle des techniques et des technologies de génération de test. Le défi ici est d'être capable de traiter des modèles, ou des systèmes, de taille industrielle, qui présentent un très grand nombre de comportements, et un espace d'états quasiment infini. De nombreux progrès ont été faits ces dernières années, avec l'avènement des techniques symboliques, et notamment les solveurs SMT (Z3, CVC4, etc.) dont les performances s'améliorent sans cesse. Similairement, l'exploration utilisant des techniques issues du model-checking, notamment basées sur des aspects aléatoires ou probabilistes permettait d'améliorer le passage à l'échelle et de parcourir de vastes espaces d'états. Pour finir, les techniques d'algorithmique distribuée dans des centres de calcul ou en cloud-computing ont permis de repousser la limite technologique des architectures matérielles sur lesquelles s'exécutaient les générateurs de test.

Pour autant, le passage à l'échelle reste encore d'actualité.

2 Le défi des attentes sociétales : tester la sécurité logicielle

La sécurité logicielle est au coeur des préoccupations actuelles. Les dernières années ont vu grandir le phénomène des Software-as-a-Service (SaaS), accélérés par la démocratisation des "clouds". Par ailleurs, le déploiement des smartphones, et des applications mobiles, a entraîné l'apparition

de nouvelles formes d'exploitation des vulnérabilités (par exemple, l'accès aux données personnel de l'utilisateur) qui impactent directement les citoyens. Aussi, les technologies web représentent un domaine en constante évolution, dans lequel les problèmes de sécurité se règlent le plus souvent en aval, par le biais de mises à jour. Les techniques de test actifs, comme le test de pénétration permettent désormais d'assurer en amont de la sécurité du système en termes d'absences de vulnérabilités exploitables. De nombreux travaux ont également émergé autour des vulnérabilités logiques (notamment le test de protocoles de sécurité).

Dans ce contexte, un défi est lié au développement des techniques de test actif pour la sécurité, telle que le test de pénétration, qui requiert la connaissance de la logique applicative pour permettre d'explorer exhaustivement les points de vulnérabilités potentiels. Par ailleurs, un aspect non négligeable de ces approches est lié à la testabilité des applications considérées lors de la recherche de failles logiques.

3 Le défi des environnements

3.1 Test de systèmes mobiles/ubiquitaires

Le défi précédent identifiait une montée en puissance des appareils mobiles, et les besoins associés en termes de modélisation d'infrastructures mobiles, d'exécutions de tests ciblant la topologie d'un réseau de terminaux, Un système mobile inclut des dispositifs qui se déplacent dans le monde physique tout en étant connectés aux réseaux par des moyens sans fil. Des travaux récents ont défini une approche de test passif pour de tels systèmes vérifiant des propriétés les traces d'exécution prenant en compte à la fois les configurations spatiales des nœuds du système et leurs communications. Pour compléter ces vérifications macroscopiques relatives aux interactions entre nœuds, un défi serait l'observation fine de l'exécution au niveau d'un dispositif (ex : tablette ou téléphone mobile), qui pose le problème des moyens d'observation dans un système enfoui. Selon les propriétés à vérifier, il est possible de combiner des instrumentations matérielles et logicielles pour enregistrer les données d'exécution pertinentes. L'idée serait d'utiliser ces enregistrements non seulement lors des tests, mais également après déploiement pour effectuer un suivi des problèmes opérationnels.

3.2 Test d'architectures reconfigurables

Une approche de conception actuellement en vogue consiste à utiliser une bibliothèque de micro-mécanismes, qui sont composés pour construire des mécanismes de tolérance aux fautes et attachés au code applicatif. L'objectif est de permettre des manipulations à grain fin de l'architecture résultante, pour qu'un intégrateur ou un administrateur puisse facilement la reconfigurer à des fins d'adaptation à un nouveau contexte opérationnel. Plusieurs technologies logicielles sont actuellement étudiées pour composer le code applicatif et les mécanismes de tolérance aux fautes, comme la programmation orientée-aspect et des technologies basées sur des composants et des services. Dans tous les cas, le défi est de valider le comportement émergeant de la composition des mécanismes avec le code applicatif. La conception du test va alors dépendre de la technologie considérée, en particulier des opérateurs de composition offerts. Pour les technologies permettant des reconfigurations à l'exécution, des problèmes additionnels concernent la validation des transitions entre configurations. Cette problématique se rapproche de celle du test de lignes de produits, qui représente un domaine d'application émergeant, dans lequel le problème de la réutilisation de cas de test va également se heurter aux problèmes de variabilité, et de nouveaux comportements issus de combinaisons inattendues.

3.3 Données et monde aléatoires

Dans le cadre de la génération aléatoire, un défi concerne la notion même de domaine d'entrée, dans le cas de systèmes autonomes évoluant dans un environnement incertain. Si l'on considère le test de services de base d'un système autonome –par exemple, le test de la navigation d'un robot– le domaine de génération est un espace de mondes dans lequel le système est susceptible d'évoluer !

En pratique, les services sont testés en simulation sur une poignée d'exemple de mondes, ce qui est tout à fait insuffisant. Pour assurer plus de diversité, on peut envisager de mettre en œuvre des techniques issues de la génération procédurale de mondes, utilisées notamment dans la création de scènes pour des jeux vidéo. Se posent alors des problèmes amont de modélisation de l'espace des mondes, incluant des caractéristiques stressantes pour le service testé (en liaison avec des analyses de sécurité), et sur la définition de critères de couverture pour guider l'échantillonnage de l'espace.

3.4 Objets connectés et Internet des objets

Les objets connectés commencent à apparaître dans les foyers. Téléviseurs, imprimantes, réfrigérateurs sont désormais connectés en permanence à Internet, et répondent à des standards ou à des normes qu'ils doivent satisfaire. Ces objets représentent un challenge évident du point de vue de la sécurité et des questions de protection de la vie privée des citoyens, déjà abordé précédemment. Au delà de ces problèmes se pose la question de la validation des normes, et du respect des standards, par le biais de techniques de test de *compliance* (conformité au standard et compatibilité des interfaces). Cette problématique existe déjà pour certains systèmes, comme les cartes à puce, et sera amenée à s'accroître dans le futur (on estime à 80 milliards le nombre d'objets connectés d'ici 2020, contre 15 milliards aujourd'hui). En ce sens, le test à partir de modèle peut apparaître comme une solution pertinente pour s'assurer de la cohérence du standard (lors de la construction du modèle) et de la conformité des applications à la norme.

4 Le défi des pratiques du test

4.1 IDM et méthodes agiles pour le test

Lors de l'appel à défi précédent, le groupe MTV2 avait identifié un défi lié aux méthodes agiles et aux aspects IDM, alors émergeantes dans les pratiques de développement. Les méthodes agiles ont bousculé les pratiques des équipes de développement ces dernières années. Une large majorité des équipes se sont (ré)organisées autour de ces méthodes, telles que SCRUM par ailleurs outillées, qui remettent le développeur au sein des décisions et font une part importante aux phases de validation. Par ailleurs, la démocratisation d'environnements d'intégration continue, tels que Jenkins, qui permettent un couplage entre un gestionnaire de version et un environnement d'exécution de tests, offrent un support de qualité à ces pratiques. Ainsi le test prend une place de plus en plus centrale dans les processus de développement actuels.

Dans ce contexte, deux challenges se posent alors. Le premier concerne l'accroissement intrinsèque du nombre de tests, rendant problématique la ré-exécution systématique de tout le référentiel de tests. Ainsi des techniques de prioritisation des cas de tests doivent être mises en oeuvre, pour maintenir un bon niveau de service des outils. Le second challenge est lié aux évolutions constantes du logiciel en cours de développement, qui entraine à la fois la nécéssité de tests de non-régression, et l'invalidation de tests devenus obsolètes. Il est donc nécessaire trouver des solutions pour gérer l'évolution du code et du référentiel de test, en particulier dans le cadre des méthodes de développement agiles.

Une autre idée serait d'exploiter la connaissance de tests déjà existants pour suggérer de nouveaux tests, généraliser les tests existants ou les faire évoluer. Pour cela, on pourra s'inspirer de techniques issues d'un domaine de recherche très actif, le software repository mining en les adaptant à la fouille de tests.

4.2 Démocratiser le test à partir de modèles

Le test à partir de modèles (Model-Based Testing – MBT) représente le moyen principal pour automatiser la génération et l'exécution de tests fonctionnels. En outre, cette approche permet d'assurer la traçabilité entre les exigences informelles exprimées au niveau du modèle, et les tests produits, fournissant ainsi des métriques séduisantes d'un point de vue industriel (notamment dans le cadre de certifications de type Critères Communs). Néanmoins, l'adoption du MBT dans

l'industrie se heurte à deux problèmes majeurs. Le premier concerne la conception de modèle, qui constitue un effort conséquent demandé à l'ingénieur validation. Par ailleurs, l'ingénieur validation se heurtera également au problème corrolaire de valider le modèle de test, pour s'assurer que celui-ci représente fidèlement le système modélisé. Le second problème concerne le passage à l'échelle des techniques de génération de test qui peinent à convaincre les industriels. Outre ces challenges techniques, le défi consiste à faire en sorte que, dans les cas les plus adaptés, des approches de type MBT soient favorisées et par des industriels. Cela passe par de l'accompagnement des équipes de validation vers ces pratiques, ainsi que la construction de formalismes et d'outils adaptés, qui réduisent le difficulté d'apprentissage (par exemple, en passant par des DSL) et apportent des solutions adaptées aux problèmes du passage à l'échelle.

4.3 Découverte de propriétés de services externes

On peut attribuer un autre rôle au test et à la surveillance en-ligne que la validation de systèmes, notamment, ces techniques peuvent être utilisées pour la découverte de propriétés. Cette approche est notamment pertinente dans le cadre de systèmes ouverts, caractérisés par la composition de services développés et maintenus en dehors des applications cibles. Plusieurs travaux ont déjà porté sur l'inférence de modèles comportementaux à partir de l'observation de traces d'exécution. Dans ce contexte, un défi serait de définir de nouvelles méthodes d'inférence active, où l'information apportée par des traces existantes serait complétée en sélectionnant des tests additionnels. Les critères de sélection de test pourraient alors être liés à la structure du modèle déjà inféré et à d'éventuelles hypothèses sur des généralisations/abstractions possibles des comportements observés.

4.4 Formation des étudiants et des professionnels au test logiciel

Le dernier défi initialement identifié concernait l'enseignement du test. En effet, l'activité de test devient de plus en plus importante suite à la délocalisation du développement des logiciels dans des pays à faibles coûts de main d'oeuvre. De fait, il revient aux équipes de désormais valider le code développé hors des frontières. Ainsi, le métier d'ingénieur validation devient de plus en plus central au sein des équipes de développement et il est nécessaire de former les étudiants et les professionnels au métier de testeur. L'arrivée dans le paysage universitaire des Cursus Master en Ingénierie (CMI) apparaît comme une solution au besoin de formation d'ingénieurs validation. Les CMIs visent en effet à former des diplômés à Bac+5 (niveau Master) qui acquièrent une solide connaissance de l'état de l'art des pratiques et des théories issues du monde académique, au travers une interaction forte entre formation et recherche. Ils fournissent ainsi un moyen, dans le cadre d'une spécialité "test de logiciels" d'initier les futurs diplômés aux techniques et concepts les plus avancés dans ce domaine, issus des laboratoire de recherche à la pointe de ces thématiques. Par ailleurs, la généralisation des cours en ligne ouverts et massifs (MOOC) offre à tous un accès à des enseignements des différentes techniques du test logiciel. Pour finir, les certifications proposées par l'International Software Testing Qualification Board (ISTQB) offrent une formation ad hoc et permettent de valider un certain niveau de connaissance des pratiques du test.

Manipulation et visualisation de modèles complexes

David Bihanic[1], Sophie Dupuy-Chessa[2], Xavier Le Pallec[3] and Thomas Polacsek[4]

[1] Université de Valenciennes
david.bihanic@univ-valenciennes.fr
[2] LIG, Université Grenoble Joseph Fourrier
sophie.dupuy@imag.fr
[3] LIFL, University Lille 1
xavier.le-pallec@univ-lille1.fr
[4] ONERA, Toulouse
Thomas.Polacsek@onera.fr

1 Contexte

Pour beaucoup de chercheurs en Génie Logiciel (GL), l'Ingénierie Dirigée par les Modèles (IDM) est maintenant largement intégrée à l'Ingénierie Logicielle. Pourtant, en y regardant de plus près, la réalité semble différente. Dans bon nombre de domaines (comme les IHM ou l'ingénierie des besoins), les travaux qui adoptent une démarche dirigée par les modèles se limitent souvent l'abstraction comme seule dimension de modélisation. On reste dans une vision OMG-MDA [1] [14] de l'IDM. Il est plus question ici d'une version améliorée des outils CASE [2] que d'une vraie approche dirigée par les modèles [12, 13] où différentes perspectives de modélisation sont utilisées. Dans l'industrie, l'intégration de l'IDM est encore plus problématique, car les professionnels du logiciel sont encore peu nombreux à avoir sauté le pas [10]. En y prêtant plus d'attention [19, 6], il semble que ces mêmes professionnelles rencontrent des obstacles autres que ceux traités par les travaux scientifiques sur l'IDM (tissage de modèles, génération de code...) : il est question de déphasage entre modèles et code mais surtout de notation inadaptée, de modèles décontextualisés/complexes et d'outils proposant des modes d'interaction inefficaces (dans la pratique ils sont délaissés pour Powerpoint ou des éditeurs de dessin). L'IDM peut-elle tomber aux oubliettes comme ce fut le cas des outils CASE car certains aspects de l'approche furent totalement occultés [20] ? Une décennie riche en résultats scientifiques sera-t-elle perdue pour des raisons de possibilités pratiques d'utilisation face à des modèles complexes ? C'est ce constat qui est à la base de notre défi : augmenter les efforts sur les aspects notations et interaction pour que l'essai IDM soit transformé et apportent une véritable évolution dans les pratiques du développement logiciel.

2 Verrous

L'inflation de la taille, de l'hétérogénéité et de l'évolutivité des systèmes semblent avoir rendu les modèles qui les représentent impossibles à appréhender par leurs utilisateurs. Les langages se révèlent inopérants renvoyant notamment divers problèmes de formalisation et de représentation. Et c'est donc précisément au carrefour de ces difficultés nouvelles que se situe

1. Model Driven Architecture
2. Computer Aided Software Engineering

ce défi visant à clarifier ce qu'est une bonne notation et ce qu'est un bon outil de modélisation permettant la manipulation, l'exploration, l'édition de modèles que nous qualifierons de complexes [2, 3]. Face au défi de la manipulation de modèles complexes, les verrous identifiés se situent au niveau de :

— la notation graphique utilisée pour représenter le modèle. Il faut définir ce qu'est une bonne notation (syntaxe concrète), sans en passer par une révision des formalismes et concepts qui sous-tendent la modélisation (syntaxe abstraite). Cette définition doit se faire au travers d'un prisme interdisciplinaire interrogeant les fondements épistémologiques de l'écriture de modèles développés dans le monde du GL ;
— les techniques d'ingénierie des modèles pour gérer abstractions et points de vue ;
— les techniques d'interaction pour visualiser et manipuler facilement les modèles.

L'objectif de ce défi est d'engager un débat sur les différentes approches en vue de la conception d'une vision "appréhendable" des modèles, laquelle repose nécessairement sur une adaptation aux différents contextes d'emploi. Plus largement, nous souhaitons ici établir un dialogue interdisciplinaire entre notamment l'ingénierie dirigée par les Modèles (IDM), les sciences cognitives, le design informatique et l'interaction homme-machine.

3 Fondement

3.1 Modèle complexe appréhendable

S'il est déjà difficile de s'accorder sur ce qu'est un modèle [18], l'absence de consensus est encore plus marquée pour la notion de modèle conceptuel appréhendable. La norme ISO 9000 [16] en précise quant à elle les qualités générales : "l'ensemble des propriétés et caractéristiques d'un modèle conceptuel portent sur sa capacité à satisfaire des besoins à la fois explicites et implicites". Pour autant, rien n'indique distinctement ce qui fait les qualités propres d'un modèle lesquelles varieraient donc selon l'angle d'appréciation de chacun : du point de vue des outils, des concepteurs, etc. Dans le cas des modèles complexes, le problème majeur n'est autre que leur appréhension et manipulation par des opérateurs humains. En effet, si les machines sont en capacité de gérer techniquement la complexité des modèles, elles ne parviennent pas pour autant à en faciliter leur compréhension et leur manipulation pour l'utilisateur.

3.2 Les pistes

C'est cette qualité pragmatique des modèles [15] qu'il faut étudier plus en profondeur. [9] ont, depuis 1996, proposé un cadre conceptuelle appelé les Dimensions Cognitives visant à améliorer cette qualité pour les langages de programmation visuelle ou outils CASE. Toutefois, la notation visuelle, élément essentiel pour comprendre et manipuler des modèles n'intervenait que ponctuellement dans les 13 dimensions proposées. Il a fallu attendre les travaux de Moody [17] et sa physique des notations pour que l'accent soit mis sur la notation. Moody énonce neuf principes pour concevoir des notations visuelles cognitivement efficaces. Nous pouvons citer, à titre d'exemple, la transparence sémantique, qui définit dans quelle mesure la signification d'un symbole peut être déduite de son apparence. Les symboles doivent donc fournir des indices sur leur sens : la forme exprime le contenu. Ce concept est proche de celui d'affordance en interaction homme-machine ; l'affordance cherche la transparence dans les actions possibles pour l'utilisateur alors que la transparence sémantique vise la facilité de compréhension des concepts. Les principes de Moody sont un début de réponse au problème des notations visuelles : [5, 22] montrent que de nombreux points sont encore à éclaircir et que tels quels ces principes sont

peu utilisables. Par exemple, sur les aspects perceptuels, Moody, comme [4] ou [7], renvoie à la sémiologie graphique (SG). La SG définie par J. Bertin[1] vise à structurer l'espace de conception graphique, et donc à produire des représentations graphiques - telles que celles que l'on trouve en IDM - plus efficaces. Pour Bertin, l'objectif d'un diagramme est de transcrire graphiquement une ou plusieurs informations. Le concepteur doit être conscient du fait que lorsqu'une personne procède à la lecture de son diagramme, celui-ci s'engagera dans un processus de recherche d'information. Aussi, le concepteur doit donc s'adresser au lecteur en mettant en avant, dans sa transcription graphique, la question principale à laquelle répond son diagramme. Ensuite, si le concepteur juge qu'une partie de son diagramme retranscrit des éléments relatifs à des questions secondaires, les éléments graphiques concernés devront être eux-aussi mis en second plan. Cette logique vaut de manière récursive pour tous les niveaux inférieurs de questionnement. Pour ce faire, Jacques Bertin propose d'exploiter les capacités de notre système visuel à percevoir la profondeur de plan des objets dans l'espace. En exploitant ainsi le système perceptif, le travail cognitif à réaliser en sera allégé et les différents niveaux de lecture pourront apparaître spontanément. Les variables visuelles que la SG définit n'ont pas le même pouvoir de mise en perspective en ce qui concerne l'association, la dissociation et la sélectivité. C'est dans cette hétérogénéité que la SG se révèle importante : elle est une très longue liste de bonnes pratiques pour une utilisation optimale de ces variables. Et une très grande majorité de ces règles ne sont pas abordées dans la physique des notations.

Enfin, il nous semble primordiale de tenir compte des avancées dans le cadre du design d'interface et de l'interaction-homme machine. Ceux-ci visent à l'aménagement de formes et symboles graphiques, l'ajout de couleurs en passant par la composition de vues et points de vue jusqu'à l'élaboration d'interacteurs. La conception de nouveaux processus de visualisation offriraient d'autres solutions et modalités de traitement, associant alors à chaque variable issue des modèles une variable graphique(telle que celle de Bertin) [21] : l'évolution de ces objets dans le temps et dans l'espace renvoyant à une modification dynamique de variables permettrait ainsi de parer à une complexité croissante des modèles [23]. Mobilisant plus fortement la capacité de traitement humain par le couplage de la vision et de l'action, il en résulterait une meilleure adaptation perceptivo-cognitive de l'utilisateur aux aléas de l'environnement, c'est-à-dire à la variabilité des modèles, à leur évolutivité et dynamisme au-delà des seuils repérés : surcharge et désorientation cognitive. A cette plus-value, s'ajoute celle de la création d'interacteurs d'un registre tout à fait nouveau offrant une saisie directe des objets à l'écran ("touch/drag") pour une meilleure continuité ou contiguïté de la perception en direction de l'action (tel que [11, 8]).

4 Conclusion

Nous avons dégagé certains nombres problèmes parmi les plus persistants en matière de manipulation, visualisation et exploration de modèles, que nous qualifions de complexes, pour lesquels une réelle résolution se fait désespérément attendre. Nous pensons qu'une réponse à ces problèmes ne pourra venir de l'arrangement de solutions existantes mais oblige notamment à refonder les paradigmes d'écriture des modèles. Aussi, une telle rupture paradigmatique devait occasionner un rapprochement du monde de l'Ingénierie des Modèles avec celui des sciences cognitives, de l'IHM et du design d'interface. Car c'est bien de cette rencontre que naîtront des pistes de résolution et de développement nouvelles remédiant aux principaux points d'achoppements que recouvrent l'appréhension de modèles complexes.

Références

[1] Jacques Bertin. *Semiology of Graphics. Diagrams, Networks and Maps.* University of Wisconsin Press, 1983.

[2] D. Bihanic and T. Polacsek. Models for visualisation of complex information systems. In *Information Visualisation (IV), 2012 16th International Conference on*, pages 130–135, July 2012.

[3] David Bihanic, Max Chevalier, Sophie Dupuy-Chessa, Thierry Morineau, Thomas Polacsek, and Xavier Le Pallec. Modélisation graphique des SI : Du traitement visuel de modèles complexes. In *Inforsid 2013*, Paris, France, May 2013.

[4] Alan Blackwell and Yuri Engelhardt. A meta-taxonomy for diagram research. In Michael Anderson, Bernd Meyer, and Patrick Olivier, editors, *Diagrammatic Representation and Reasoning*, pages 47–64. Springer London, 2002.

[5] Patrice Caire, Nicolas Genon, Patrick Heymans, and Daniel Laurence Moody. Visual notation design 2.0 : Towards user comprehensible requirements engineering notations. In *RE*, pages 115–124. IEEE, 2013.

[6] MichelR.V. Chaudron, Werner Heijstek, and Ariadi Nugroho. How effective is uml modeling ? *Software & Systems Modeling*, 11(4) :571–580, 2012.

[7] Stéphane Conversy, Stéphane Chatty, and Christophe Hurter. Visual scanning as a reference framework for interactive representation design. *Information Visualization*, 10(3) :196–211, July 2011.

[8] Randall Davis. Magic paper : Sketch-understanding research. *Computer*, 40(9) :34–41, September 2007.

[9] T.R.G. Green and M. Petre. Usability analysis of visual programming environments : A cognitive dimensions framework. *Journal of Visual Languages & Computing*, 7(2) :131 – 174, 1996.

[10] John Hutchinson, Mark Rouncefield, and Jon Whittle. Model-driven engineering practices in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 633–642, New York, NY, USA, 2011. ACM.

[11] Hiroshi Ishii and Brygg Ullmer. Tangible bits : Towards seamless interfaces between people, bits and atoms. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, CHI '97, pages 234–241, New York, NY, USA, 1997. ACM.

[12] Jean-Marc Jezequel. Model driven design and aspect weaving. *Software & Systems Modeling*, 7(2) :209–218, 2008.

[13] Stuart Kent. Model driven engineering. In *Proceedings of the Third International Conference on Integrated Formal Methods*, IFM '02, pages 286–298, London, UK, UK, 2002. Springer-Verlag.

[14] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained : The Model Driven Architecture : Practice and Promise.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[15] Odd I. Lindland, Guttorm Sindre, and Arne Solvberg. Understanding Quality in Conceptual Modelling. *IEEE Software*, 11(2) :42–49, March 1994.

[16] Daniel L. Moody. Theoretical and practical issues in evaluating the quality of conceptual models : current state and future directions. *Data and Knowledge Engineering*, 55(3) :243 – 276, 2005. Quality in conceptual modeling Five examples of the state of art The International Workshop on Conceptual Modeling Quality 2002 and 2003.

[17] Daniel L. Moody. The "Physics" of Notations : Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. *IEEE Transactions on Software Engineering*, 35 :756–779, 2009.

[18] Pierre-Alain Muller, Frdric Fondement, Benot Baudry, and Benot Combemale. Modeling modeling modeling. *Software and Systems Modeling*, 11(3) :347–359, 2012.

[19] Marian Petre. Uml in practice. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

[20] D.C. Schmidt. Guest editor's introduction : Model-driven engineering. *Computer*, 39(2) :25–31, Feb 2006.

[21] B. Shneiderman. The eyes have it : a task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343, Sep 1996.

[22] Harald Störrle and Andrew Fish. Towards an operationalization of the "physics of notations" for the analysis of visual languages. In Ana Moreira, Bernhard Schätz, Jeff Gray, Antonio Vallecillo, and Peter J. Clarke, editors, *MoDELS*, volume 8107 of *Lecture Notes in Computer Science*, pages 104–120. Springer, 2013.

[23] R. Wettel and M. Lanza. Visual exploration of large-scale system evolution. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 219–228, Oct 2008.

The Future Depends on the Low-Level Stuff

Julia Lawall and Gilles Muller
Inria/LIP6/UPMC/Sorbonne University

1 Description of the Challenge

Device drivers are essential to modern computing, to provide applications with access, via the operating system (OS), to devices such as keyboards, disks, networks, and cameras. Development of new computing paradigms, such as the internet of things, is hampered because device driver development is challenging and error-prone, requiring a high level of expertise in both the targeted OS and the specific device. Furthermore, implementing just one driver is often not sufficient; today's computing landscape is characterized by a number of OSes, *e.g.*, Linux, Windows, MacOS, and BSD, and each is found in a wide range of variants and versions. All of these factors make the development, porting, backporting, and maintenance of device drivers a critical problem for device manufacturers, industry that requires specific devices, and even ordinary users.

Recent years have seen a number of approaches directed towards easing device driver development. Merillon *et al.* propose Devil [10], a domain-specific language for describing the low-level interface of a device. Chipounov *et al.* propose RevNic [3], a template-based approach for porting device drivers from one OS to another. Ryzhyk *et al.* propose Termite [14], an approach for synthesizing device driver code from a specification of an OS and a device. Currently, these approaches have been successfully applied to only a small number of toy drivers. Indeed, Kadav and Swift [5] observe that these approaches make assumptions that are not satisfied by many drivers; for example, that a driver involves little computation other than the direct interaction between the OS and the device. At the same time, a number of tools have been developed for finding bugs in driver code. These tools include SDV [1], Coverity [4], Coccinelle [12], CP-Miner [8], and PR-Miner [9]. These approaches, however, focus on analyzing existing code, and do not provide guidelines on structuring drivers.

Our thesis is that the weaknesses of previous methods for easing device driver development arise from an insufficient understanding of the range and scope of driver functionality, as required by real devices and OSes. In this challenge, we propose to consider a new methodology for understanding device drivers, inspired by the biological field of *genomics*. Rather than focusing on the input/output behavior of a device, we propose to take the radically new methodology of studying existing device driver code itself. On the one hand, this methodology makes it possible to identify the behaviors performed by real device drivers, whether to support the features of the device and the OS, or to improve properties such as safety or performance. On the other hand, this methodology makes it possible to capture the actual patterns of code used to implement these behaviors, raising the level of abstraction from individual operations to collections of operations implementing a single functionality, which we refer to as *genes*. Because the requirements of the device remain fixed, regardless of the OS, we expect to find genes with common behaviors across different OSes, even when those genes have a different internal structure. This leads to a view of a device driver as being constructed as a composition of genes, thus opening the door to new methodologies to address the problems faced by real driver developers. Among these, we have so far identified the problems of developing drivers, porting existing drivers to other OSes, backporting existing drivers to older OS versions, and long-term maintenance of the driver code.

2 Applications and Societal Impact

Innovations in areas such as health and autonomy, intelligent cities, energy and intelligent networks, and global security increasingly rely on the use of powerful, special-purpose devices. These devices are, however, useless without the availability of device drivers. Furthermore, for the companies that produce these devices, minimizing the time-to-market of these drivers is essential to the company's reputation and long-term viability.

Today, the era of the Personal Computer as the main form of computing is over. Smartphones are used for playing games, listening to music, and surfing the web, and are increasingly finding application in more specialized areas such as care of the elderly and the disabled. Appliances such as washing machines use computers to adapt to current conditions and deliver their services as efficiently as possible. Smart homes, automobiles, and airplanes, rely on highly diverse networked computing entities to provide a configurable and adaptable user experience. All of these applications require an ever increasing array of devices, which in turn require device drivers.

Furthermore, simply having a functioning driver for a given device is no longer enough. New constraints are emerging across the computing spectrum, in terms of security and energy usage. Applications integrate more and more of our personal information, and are becoming more critical to our health and safety, while at the same time they are becoming more dependent on unreliable battery power. Making device drivers secure and energy aware requires that they be developed according to well-tested strategies and be easy to fix when problems arise.

It is our belief that genes address these design issues. Genes found in mature driver code encapsulate well-tested development strategies. New drivers that incorporate well-known genes will be easily understandable and maintainable by developers. The study of genes in driver code will thus make it possible to develop device drivers more quickly, ensure aspects of the resulting driver code quality, and improve the usability and maintainability of the driver in the long term.

3 Scientific Background

The novelty of our proposal lies in raising the level of abstraction of our understanding of device driver code from the level of individual operations to genes. In recent years, due to the importance of device driver code, numerous tools have been proposed for problems such as finding bugs [4, 13], verification [7], and automating software evolution [12] in such code. These tools, however, are designed in terms of individual operations, stripped of their semantic interrelationships, and thus risk false positives, when requirements are arbitrarily imposed that do not correspond to the actual genetic structure, and false negatives, when such requirements are overlooked. In contrast, the description and analysis of code in terms of genes provides a framework for accurately reasoning about related operations. Furthermore, specifications used by code processing tools can become more portable and adaptable when expressed in terms of genes, allowing a single specification to be transparently applied to instances of all variants of a single gene, regardless of the actual code involved.

Our notion of a gene is related to that of a feature in feature-oriented programming (FOP) [2]. FOP is a form of software development in which an instance of a software product is constructed by selecting and composing code fragments chosen according to a desired set of properties. The Linux kernel has been extensively studied by the FOP community [11, 15], but primarily in terms of the configuration options exposed by its build system, rather than its code structure. Instead, we focus on understanding the use of genes within device driver C code. Our work can benefit from the experience of the FOP community on designing feature composition strategies. Complementarily, our work may suggest new techniques for feature mining, *i.e.*, identifying features in existing software, that can be of use to the FOP community.

Our notion of a gene is also related to that of an aspect in aspect-oriented programming (AOP) [6]. Aspect-oriented programming allows a developer to modularize the implementation of a so-called crosscutting concern and to specify how code fragments from this module should be distributed across a code base. Genes, on the other hand, are intrinsic to the modules in which they appear. Our goal for genes is to guide the construction and analysis of a code base, rather than to provide a means of augmenting an existing code

base with new functionalities.

4 Agenda and Research Challenges

The understanding and exploiting of driver genomics will require expertises in areas ranging from programming languages and software engineering to OSes and hardware. Expertise in programming languages and software engineering will be needed to devise methodologies for extracting information from existing code and recomposing the extracted into new device drivers. Expertise in OSes and hardware will be needed to understand the existing code, and to test the generated code in realistic scenarios. We envision a project with the following steps:

1. Identifying genes involving interaction with the OS, first manually and then automatically. Such genes typically involve OS API functions, and have a common structure.

   Challenges. Driver code exhibits many variations, and thus our preliminary studies have shown that it does not fit well with the assumptions of most existing specification mining tools. During the manual study, it will be necessary to carefully observe the properties of these variations, to be able to subsequently select the most appropriate automatic mining techniques. Furthermore, separating one gene from another requires understanding what driver code does and why it is written in the way that it is, which will require a high degree of expertise in OSes and hardware.

2. Identifying genes involving interaction with the device, first manually and then automatically. Such genes typically involve low-level bit operations, which are specific to each device.

   Challenges. Individual bit operations are untyped and themselves give little hint of their semantics. Understanding of auxiliary material, such as comments, via techniques from natural language processing, may be necessary to identify these genes.

3. Developing techniques for composing genes, to construct new device drivers.

   Challenges. Constructing new device drivers requires not only composing the genes that provide the desired features, but also constructing the glue code to hold them together. If a composition is needed that has not previously been explored, the relevant genes might not fit together well, making the construction of this glue code difficult to automate. Furthermore, to have practical impact, compositions must be easy to construct. Techniques from feature modeling may be helpful to address this issue.

4. Applying the gene-based methodology for constructing device drivers to address issues of porting and backporting. Porting refers to using a driver for one OS as the basis of the implementation of a driver for another OS. Backporting refers to using a driver for one version of an OS as the basis of the implementation of a driver for another version of the OS, typically an earlier one. The latter is particularly important in the context of an OS that evolves frequently, such as Linux, as it enables a company to stay with one version of the OS while still being able to access the latest devices.

   Challenges. It is essential that the resulting driver should be structured in a way that is compatible with the coding strategies of the target OS, to facilitate the subsequent maintenance of the generated code. This requires identifying corresponding genes across OSes or OS versions. In practice, the overall functionalities may be decomposed in different ways across the different systems, so it will be necessary to find the right level of abstraction at which commonalities can be found. This is likely to require a deep understanding of OS design.

5 Conclusion

The ability to quickly and easily develop robust and maintainable device drivers is critical to many aspects of modern computing. We have proposed a new direction that brings together a range of expertises with the goal of producing effective changes in how device drivers are designed and implemented.

References

[1] Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S. K., and Ustuner, A. Thorough static analysis of device drivers. In *EuroSys* (2006).

[2] Batory, D., and OMalley, S. The design and implementation of hierarchical software systems with reusable components. *TOSEM* (1992).

[3] Chipounov, V., and Candea, G. Reverse engineering of binary device drivers with RevNIC. In *EuroSys* (2010).

[4] Engler, D. R., Chelf, B., Chou, A., and Hallem, S. Checking system rules using system-specific, programmer-written compiler extensions. In *OSDI* (2000).

[5] Kadav, A., and Swift, M. M. Understanding modern device drivers. In *ASPLOS* (2012).

[6] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of AspectJ. In *ECOOP* (2001).

[7] Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: formal verification of an operating-system kernel. *Commun. ACM* (2010).

[8] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. CP-Miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI* (2004).

[9] Li, Z., and Zhou, Y. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE* (2005).

[10] Mérillon, F., Réveillère, L., Consel, C., Marlet, R., and Muller, G. Devil: An IDL for hardware programming. In *OSDI* (2000).

[11] Nadi, S., and Holt, R. The Linux kernel: A case study of build system variability. *Journal of Software Maintenance and Evolution: Research and Practice* (2012).

[12] Padioleau, Y., Lawall, J., Hansen, R. R., and Muller, G. Documenting and automating collateral evolutions in Linux device drivers. In *EuroSys* (2008).

[13] Rubio-González, C., and Liblit, B. Defective error/pointer interactions in the Linux kernel. In *ISSTA* (2011).

[14] Ryzhyk, L., Chubb, P., Kuz, I., Le Sueur, E., and Heiser, G. Automatic device driver synthesis with Termite. In *SOSP* (2009).

[15] Tartler, R., Lohmann, D., Sincero, J., and Schröder-Preikschat, W. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *EuroSys* (2011).

L'IDM de demain : un accès facilité, un usage intensif pour des performances accrues

David Bihanic, Jean-Michel Bruel, Philippe Collet,
Benoît Combemale, Sophie Dupuy-Chessa, Xavier Le Pallec, Thomas Polacsek

1er novembre 2014

1 Contexte

Pour beaucoup de chercheurs en Génie Logiciel, l'Ingénierie Dirigée par les Modèles (IDM) est maintenant largement intégrée à l'ingénierie logicielle. Pourtant, en y regardant de plus près, la réalité semble différente. Dans bon nombre de domaines, comme les Interfaces Homme-Système (IHS) ou l'ingénierie des besoins, les travaux qui adoptent une démarche dirigée par les modèles se limitent souvent à l'abstraction comme seule dimension de modélisation. Industriellement, on observe des gains par l'application de l'IDM, mais plus par la seule explicitation des abstractions (documentation) que par les autres bénéfices attendus (maîtrise de la complexité, génération d'applications). La communauté de l'IDM a pourtant réalisé un certain nombre d'avancées significatives sur des aspects complexes, mais spécifiques comme notamment la composition, l'adaptation, le temps-réel ou bien encore le passage à l'échelle.

2 Problématique

Cependant plusieurs problèmes cruciaux persistent du point de vue de l'usage des modèles.

Ergonomie. Une part limitée des travaux et outils liées à l'IDM prennent en compte les processus cognitifs généralement à l'œuvre lors de la conception et le développement. Le non-alignement est donc fréquent et entraîne généralement des processus cognitifs supplémentaires chez le concepteur pour adapter les modèles et/ou leurs outils à son contexte de travail. Cela implique malheureusement une augmentation de l'effort à fournir, du nombre d'erreurs possibles et une diminution de la vitesse de travail. Cet impact important sur l'utilisabilité des modèles et outils est un frein important à l'adoption de l'IDM et à son passage à l'échelle.

Design. Si la syntaxe visuo-graphique/diagrammatique classiquement usitée pour décrire ou représenter les modèles répond pour partie aux attentes et visées fonctionnelles (qualités opératoires), reste que celle-ci n'est pas suffisamment efficiente. En effet, les apories et difficultés de traitement 'utilisateur/concepteur' sont aujourd'hui nombreux. Aussi, la contribution du design peut s'avérer d'un aide précieuse en vue d'inventer, de concevoir de nouveaux langages de représentation et de visualisation graphique des modèles (cf. data design ou design de données). Ces langages, ouvrant à une réelle plus-value de traitement, seraient également en capacité de dénouer nombre de problèmes ou écueils d'ordre logique et conceptuel auxquels

doivent faire face les utilisateurs, concepteurs et opérateurs. A noter que cette contribution, touchant ici à la représentativité des modèles (langage visuo-graphique), appelle à être complétée sur le plan des interfaces.

Multiplicité. Les applications courantes et futures nécessitent que les différentes sources de complexité soient maîtrisées en même temps (personnalisation et réactivité, accès aux non-informaticiens, prédictions et analyse, prise en compte des points de vue et des niveaux de compétences, ...).

Ces problèmes entravent grandement l'adoption de l'IDM dans de nombreux contextes industriels. L'IDM peut-elle tomber aux oubliettes comme ce fut le cas des outils CASE, car certains aspects de l'approche furent totalement occultés ? Une décennie riche en résultats scientifiques sera-t-elle perdue car l'IDM n'est pas assez facile d'accès, ne supporte pas l'usage intensif du génie logiciel actuel ?

3 Challenges identifiés

Partant de ce constat, un certain nombre de challenges peuvent être identifiés en croisant les considérations sociétales et cognitives avec des défis à plus longue échéance, comme illustré par la table 1. Ces défis à long terme sont issus d'un travail de réflexion à l'échelle internationale sur les grands problèmes à résoudre par la communauté IDM dans le cadre de l'évolution de notre société [1]. Quatre points de ruptures précis sont ainsi obtenus.

Représentativité. Le caractère de plus en plus trans-diciplinaire des applications et des systèmes doivent nous inciter à, non seulement anticiper les difficultés techniques, mais aussi profiter de cette richesse. Notre discipline reste trop souvent cantonnée à l'ingénierie des logiciels ou des systèmes sans s'intéresser à ce que peuvent apporter des disciplines comme l'intelligence artificielle, le design, le web sémantique, mais aussi la médecine, les sciences cognitives, sociales, etc. Les efforts actuels sur la fusion de modèles doivent intégrer cette dimension multi-disciplinaire. Cela exige aussi à repenser les formalismes de représentation, de description et de visualisation des modèles tout autant que les mécanismes utilisés en IDM pour les notations visuelles. Ces dernières sont souvent limitées aux réseaux - terme employé en sémiologie graphique -, alors que l'espace de conception graphique est beaucoup plus riche.

Accessibilité. Le logiciel est devenu quelque chose de palpable dans le grand public depuis l'avènement des *AppStore* et autre *GooglePlay* où les applications s'installent, se partagent, s'évaluent de manière intuitive et abordable au plus grand nombre. L'étape suivante sera de faciliter le développement de ces logiciels. L'IDM prône de placer les modèles au cœur des développements. À nous de rendre l'utilisation, la manipulation et la réalisation de modèles accessibles au plus grand nombre. Ceci implique aussi des modes d'interaction efficaces et intuitifs. Les notations utilisées devront mettre en avant les informations pertinentes pour la préoccupation du concepteur lambda et donc exploiter au maximum les mécanismes visuelles répertoriés (associativité, sélection, imposition, abstraction...) afin de maîtriser la complexité

1. Gunter Mussbacher et. al. *The Relevance of Model-Driven Engineering Thirty Years from Now.* In IEEE/ACM Int. Conf. MoDELS 2014, Valence.

des modèles logiciels. Les périphériques d'interaction utilisés par les outils d'édition ne sont pas naturels au regard de la modélisation : un travail important est aussi à fournir afin de créer/déterminer des périphériques adaptés.

Flexibilité. Les modèles doivent être à l'image des applications modernes : flexibles pour faciliter leur manipulation et intégration de manière multiple et dynamique. Il s'agit de pouvoir tenir compte, dans un domaine d'applications donné, de toutes les préoccupations, techniques ou relatives au domaine, avec plus de confiance et de prévisibilité dans le résultat d'une intégration. Ceci doit être aussi valable dans un contexte dynamique, pour s'adapter à la volée aux changements de leur environnement (outils, besoins utilisateurs, interfaces, ...). Ceci pourrait être facilité par l'utilisation de modèles de référence, facilement customisable et intégrable. Par ailleurs, si les informations contenus dans un modèle ne sont pas en phase avec le reste du projet, cela exige un travail cognitif important du concepteur pour réactualiser mentalement le modèle. La désynchronisation entre le code et le modèle est un exemple récurrent en IDM. Ce travail cognitif devrait être facilité par des environnements adaptés .

Evolutivité. Un modèle n'est jamais aussi riche que la réalité. Il est donc par définition incomplet. Néanmoins, le modèle d'un système complexe se doit souvent d'être aussi complet que possible, notamment si on veut pouvoir effectuer des analyses et prédictions de propriétés. Les outils de modélisation se sont donc souvent intéressés au caractère complet, formel des modèles. Ceci est souvent perçu comme un frein à la créativité et à l'activité même d'abstraction. Il nous semble que les outils de modélisation devraient permettre, par exemple, qu'un modèle ne soit pas conforme à son méta-modèle pendant la phase d'élaboration. Ainsi les aspects cognitifs liés par exemple aux utilisateurs devraient être pris en compte dans les outils de modélisation dès leur création tout en impactant les aspects techniques de maintenance, partage, évolution, etc. Il devrait même être possible de générer le système modélisé ou tout au moins de le simuler lorsque le modèle est incomplet ou non conforme : ceci permettrait au concepteur d'avoir un retour concret de l'état d'avancement de son modèle et lui éviterait un effort important de projection mentale de ce dernier.

Challenge	Aspects sociétaux	Aspects techniques
Représentativité	relations cognitives inter-domaines ; exploitation de représentations graphiques non utilisées en GL	relations entre langages et outillages hétérogènes ; facilité de fusion de modèles
Accessibilité	modèles ouverts ; mécanismes visuels appropriés, interactions mieux alignés avec les gestes de schématisation	outillages ouverts ; modèles comme éléments essentiels du développement logiciel
Flexibilité	environnements facilitant le travail cognitif lors de l'intégration	modèles de référence exposants hypothèses et limites, et pouvant être facilement customisés et intégrés
Evolutivité	rendre compte de l'incomplétude ; simulation de modèles incomplets	outillage agile pour supporter le processus naturel de conception

TABLE 1 – Aspects sociétaux et techniques des points de rupture retenus

Défis dans l'ingénierie logiciel des Systèmes de Systèmes

Laboratoire LIUPPA, Université de Pau et Pays de l'Adour
Archware, Laboratoire IRISA, Université de Bretagne-Sud

2 novembre 2014

La complexité croissante de notre environnement socio-économique se traduit en génie logiciel par une augmentation de la taille des systèmes et par conséquent de leur complexité . Les systèmes actuels sont le plus souvent concurrents, distribués à grande échelle et composés d'autre systèmes. Ils sont alors appelés Systèmes de Systèmes (SdS). Un système de systèmes est une intégration d'un certain nombre de systèmes constituants indépendants et inter opérables, interconnectés pour un période de temps donnée dans le but d'assurer une certaine finalité. Dans la pratique, les systèmes de systèmes sont des systèmes complexes à logiciel prépondérant. Les systèmes constituants, non seulement pré existent au besoin à couvrir, plus encore, ils ont été conçus à des fins indépendantes de la finalité recherchée. Ils continuent par ailleurs à être opérés, maintenus, gérés par les différentes organisations dont ils sont issus, voire évoluent de manière indépendante. Mais, ils doivent, pour un temps donné, de manière non anticipée, partager des informations, inter opérer, pour faire émerger la finalité recherchée.

Après des systèmes dont l'unité de composition de base était des fonctions, puis des composants, maintenant il s'agit de développer des systèmes dans lesquels les unités de composition de base sont des systèmes eux-mêmes. La complexité des SdS réside dans leurs cinq caractéristiques intrinsèques qui sont : l'indépendance *opérationnelle* des systèmes constituants, l'indépendance *managériale* de ces mêmes systèmes, la distribution géographique, l'existence de comportements émergents, et enfin un processus de développement *évolutionnaire*. Clairement il ne suffira pas de simplement adapter les méthodes et outils existants de développement, le défi soulevé est la proposition de concepts, de langages, de méthodes et d'outils permettant la construction de SdS capables de s'adapter dynamiquement pour continuer d'assurer leur mission malgré les évolutions internes ou externes qu'ils subissent.

La caractéristique de *développement évolutif* de SdS signifie que la conception originale d'un SdS doit tenir compte des systèmes constituants existants, qui sont parfois traités en génie logiciel en tant que des systèmes hérités (legacy). En outre, à tout moment, de nouveaux systèmes peuvent être ajoutés ou supprimés, et pas seulement au moment de la conception, mais également à l'exécution. De plus, chaque système constituant subira des évolutions, des mises à jour, qui pourront avoir une incidence sur l'ensemble du SdS. Les cycles de vie seront longs, des dizaines d'années pour des SdS tels que ceux de la défense, par exemple. Par conséquent, les cycles de vie itératifs flexibles seraient probablement les plus adéquats.

Du point de vue de l'analyse des besoins, l'évolution continue des besoins est encore plus accentuée par les *comportements émergents* des SdS. Certains de ces comportements sont conçus, mais certains ne sont ni conçus, ni voulus. Comment peut-on analyser les besoins des SdS ? Comment les mesurer ? Comment trouver un équilibre entre les différentes exigences de SdS, notamment les propriétés non-fonctionnelles qui sont souvent contradictoires, telles que la performance, la sécurité, la tolérance aux pannes ? Comment équilibrer les exigences des différents systèmes constituants ?

Face à l'*indépendance opérationnelle* des systèmes constituants il faut être capable d'architecturer et construire un SdS qui pourra remplir sa propre mission, sans violer l'indépendance de ses systèmes constituants qui sont autonomes et ont leur propre mission à remplir. De plus, il devra d'adapter pour répondre aux changements de caractéristiques de son environnement et de ses systèmes constituants.

Les équipes du GDR GPL impliquées dans ce défi proposent de s'attaquer, d'une part, à la modélisation et l'analyse de propriétés non-fonctionnelles des SdS en particulier aux défis liés à la sécurité et, d'autre part, à l'analyse, au développement et à l'évolution dynamique des architectures de SdS.

Défis en compilation, horizon 2025

Florian Brandner, Albert Cohen, Alain Darte, Paul Feautrier
Grigori Fursin, Laure Gonnord, Sid Touati

13 octobre 2014

1 Contexte

La communauté compilation, de part l'émergence des processeurs multi-coeurs qui équipent tous les systèmes informatiques, des smartphones jusqu'aux accélérateurs de calcul (GPUs, FPGAs), ainsi que les super-calculateurs des centres de calcul a vu émerger de nouveaux problèmes tant matériels que logiciels. Le parallélisme est maintenant à la portée de tous, mais sans qu'il soit accessible à tous de part sa complexité.

2 Problématique

Maîtriser la difficulté de programmation de ces architectures tout en offrant une certaine portabilité des codes et des performances, et offrir une meilleure interaction entre les utilisateurs et les compilateurs (collaboration dans les deux sens), nécessite des recherches sur les langages, notamment parallèles, en compilation (analyse et optimisation de codes), et en systèmes d'exploitation (OS). Ces thèmes de recherche nécessitent des travaux parfois théoriques, mais surtout pratiques (développements d'outils, analyses d'applications, analyses de performances et validation expérimentale, en lien avec les domaines applicatifs).

3 Challenges identifiés

Voici trois challenges que nous relevons, parmi ceux de la communauté Compilation/Calcul Haute Performance :

1. **Les langages parallèles** : le défi est bien résumé dans la *road-map* d'HIPEAC, paragraphe 1.4.3 [1].

   Avec la mise sur le marché de plateformes de plus en plus hétérogènes, le gain en performance des applications se fait maintenant en utilisant des processeurs dédiés à des tâches spécifiques. La programmation de tels systèmes logiciels/matériels devient beaucoup plus complexe, et les langages parallèles existants (MPI, openMP) semblent peu adaptés à cette tâche. Pour diminuer le coût du développement et du déploiement de logiciels sur des plateformes spécifiques toujours changeantes, nous avons besoin d'approches et de langages qui permettent l'expression du parallélisme potentiel des applications et la compilation vers une plateforme parallèle spécifique.

   De plus, la complexité et diversité de ces plateformes justifient le développement de langages ou approches de haut niveau (comme X10, Chapel, OpenAcc, CAF, OpenStream, etc.) et d'optimisations au niveau source, en amont des langages natifs cible,

1. `http://www.hipeac.net/system/files/hipeac_roadmap1_0.pdf`

dialectes souvent proches du langage C. Ceci offrira plus de portabilité de performances mais aussi, pour l'utilisateur, une meilleure compréhension des transformations effectuées par le compilateur. L'amélioration de l'interface entre le programmeur et le compilateur est un également un point fondamental à améliorer. L'interaction langage/compilateur/OS/environnement d'exécution (*runtime*) est également un point clé à améliorer, la portabilité de la performance passant par l'adaptation à la plateforme via cette interaction.

2. **La compilation optimisante** pour les performances en temps et en mémoire. Même source, paragraphe 1.4.1. La consommation d'énergie des systèmes (embarqués surtout) est maintenant en grande partie due au mouvement et au stockage des données. Pour s'adapter à cette problématique, les données et leur mouvement doivent être exposées au programmeur, et les compilateurs doivent prendre en compte la trace mémoire et les mouvement des données par exemple entre deux calculs qui s'effectuent sur des unités de calcul différentes. Des approches existent déjà en compilation, notamment en ce qui concerne la compilation vers FPGA, mais ce n'est que le début vers une prise en compte plus générique des contraintes de mémoire. Dans un sujet connexe, la recherche de solutions dont le pire cas peut être garanti (WCET) est rendu encore plus difficile par la présence de parallélisme. À l'heure actuelle, la compilation à performances prédictibles reste un problème largement ouvert.
3. **La validation théorique des analyses et optimisations** que nous réalisons au sein des compilateurs. Les analyses de programmes (séquentiels ou parallèles), les transformations de code, la génération de code pour des architectures précises, doivent être définies et prouvées précisément. Si accompagner les algorithmes de leur preuves reste essentiel, des techniques comme la *preuve assistée* et la *translation validation* permettent maintenant de valider des compilateurs entiers (Compcert). Le challenge ici réside dans l'application plus générale des méthodes formelles, que ce soit pour définir précisement des algorithmes ou pour valider des optimisations dans le cadre parallèle en particulier.
4. **La mesure et la reproductivité des résultats**, notamment dans le domaine de l'optimisation du logiciel. Contrairement aux autres sciences expérimentales (sciences naturelles), la branche expérimentale de l'informatique souffre d'un manque de principe scientifique : la reproductibilité et la vérification des résultats expérimentaux en informatique ne sont pas entrés dans nos habitudes. Dans le domaine de la compilation optimisante en particulier, lorsque des chiffres de performances sont publiés, il est très rare que ces chiffres puissent être vérifiés ou observés par une partie tiers. Définir des méthodes expérimentales de validation statistique de résultats et des modèles présupposés (modèles de coût, modèles de programmation, abstractions ...) reste un défi majeur pour notre communauté, même si certaines conférences de notre domaine commencent à proposer des évaluations expérimentales [2].

Ces problématiques orientées compilation/langages sont aussi liées aux problématiques générales de Génie Logiciel que sont la complexité et l'hétérogénéité des logiciels. La spécificité ici est que la même complexité est à prendre en compte aussi du côté matériel (plateformes hétérogènes, accélérateurs matériels, ...) où se rajoutent en plus des contraintes énergétiques (consommation) et technologiques.

2. `http://evaluate.inf.usi.ch/artifacts`

Réutiliser les spécifications et preuves

Gilles Dowek, Catherine Dubois

8 décembre 2014

1 Contexte

On constate que l'utilisation de méthodes déductives pour produire des logiciels sûrs progresse mais reste encore à diffusion limitée. Elle progresse car de nombreux formalismes, langages et outils sont proposés mais elle reste encore trop limitée et réservée à certaines niches. L'investissement est souvent jugé lourd, en particulier pour les nouveaux domaines, et le passage à l'échelle reste difficile. On constate également qu'il manque de passerelles ente les outils de preuve (qu'ils soient automatiques ou non) même si la coopération entre outils a fait des progrès récents.

2 Problématique

Une façon de progresser est de pouvoir réutiliser les spécifications et preuves formelles d'un contexte à un autre, mais aussi d'un outil à un autre.

3 Challenges identifiés

Les challenges identifiés concernent la proposition de techniques de réutilisation intra (*in the small*) et inter-outils (*in the large*).

La plupart des assistants à la preuve offre des techniques architecturales, pour aider à la réutilisation : modules, type classes, paramètres, foncteurs, contextes, héritage, redéfinition. La preuve est en général modulaire. Néanmoins ces techniques, efficaces dans certaines situations, conduisent dans d'autres cas à des formulations peu naturelles (par exemple il faut introduire trop de paramètres) et demandent parfois de reprendre le code (spécifications et/ou preuves). Le challenge identifié ici est de proposer des techniques de réutilisation *in the small* qui permettent une réutilisation plus sémantique et mettent en œuvre des techniques d'adaptation des spécifications et preuves formelles.

Le deuxième challenge concerne la réutilisation *in the large*, c'est-à-dire la possibilité de réutiliser un développement formel (spécifications et preuves) réalisé dans un formalisme donné dans un autre développement réalisé dans un autre formalisme. Quelques tentatives existent entre des systèmes de la même famille (par exemple, Open Theory pour les assistants de la famille HOL [4]) ; des traductions pair à pair ont également été proposées (par exemple HOL Light vers Coq [5], SMT vers Coq [1]). Il s'agit ici de proposer un standard pour les spécifications et preuves, comme il existe des standards dans d'autres domaines (réseaux, web, etc).

4 Jalons

Pour la réutilisation *in the small*, il est intéressant d'étudier les techniques provenant du monde de la programmation orientée objet et des lignes de produits. Des travaux embryonnaires existent dans cette voie : introduction de l'héritage et de la redéfinition dans Focalize, utilisation des techniques issues des lignes de produit pour définir la sémantique des langages de programmation (Meta-theory à la carte) [3].

Nous proposons de lever le verrou de la réutilisation *in the large*, en mettant en place un formalisme qui permettrait d'exprimer les spécifications et les preuves provenant de formalismes divers (logique du premier ordre, théorie des types, théorie des ensembles, logique d'ordre supérieur par exemple). Il s'agirait ensuite de fournir des outils de traduction des formalismes existants vers ce standard de formalisation. Une première piste concerne l'étude du $\lambda\Pi$-calcul modulo, λ-calcul typé avec types dépendants paramétré par un système de réécriture. Ce formalisme, utilisé avec un système de réécriture particulier, peut encoder d'autres formalismes [2]. Il existe actuellement des outils de traduction : Coq vers $\lambda\Pi$-calcul modulo, HOL vers $\lambda\Pi$-calcul modulo, Focalize vers $\lambda\Pi$-calcul modulo (voir `http://dedukti.gforge.inria.fr/`). Un premier jalon est d'étudier le $\lambda\Pi$-calcul modulo, de l'étendre, de l'évaluer comme standard de formalisation. De nombreux problèmes théoriques sont liés à cette approche, en particulier des problèmes de cohérence, de confluence, de terminaison des systèmes de réécriture nécessaires. Le chemin est cependant long jusqu'à la proposition d'un standard reconnu par la communauté.

Références

[1] Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to coq through proof witnesses. In *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 2011.

[2] Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In *TLCA*, pages 102–117, 2007.

[3] Benjamin Delaware, Bruno C. d. S. Oliveira, and Tom Schrijvers. Meta-theory à la carte. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 207–218. ACM, 2013.

[4] Joe Hurd. The opentheory standard theory library. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2011.

[5] Chantal Keller and Benjamin Werner. Importing HOL Light into Coq. In Matt Kaufmann and Lawrence C. Paulson, editors, *ITP - Interactive Theorem Proving, First International Conference - 2010*, volume 6172 of *Lecture Notes in Computer Science*, pages 307–322, Edimbourg, Royaume-Uni, 2010. Springer.