Actes des Sixièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel

Catherine Dubois, Laurence Duchien, Nicole Levy

Actes des Sixièmes journées nationales du

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

Conservatoire National des Arts et Métiers

Laboratoire CEDRIC

11 au 13 juin 2014

Editeur : Catherine Dubois
Laurence Duchien
Nicole Levy

Impression : service de reprographie, Conservatoire National des Arts et Métiers

Table des matières

Préface

C'est avec grand plaisir que je vous accueille pour les Sixièmes Journées Nationales du GDR Génie de la Programmation et du Logiciel (GPL) au Conservatoire National des Arts et Métiers de Paris. Ces journées sont l'occasion de rassembler la communauté du GDR GPL. Les missions principales du GDR GPL sont l'animation scientifique de la communauté et la promotion de nos disciplines, notamment en direction des jeunes chercheurs. Cette animation scientifique est d'abord le fruit des efforts de nos groupes de travail, actions transverses et de l'Ecole des Jeunes Chercheurs en Programmation.

Le GDR GPL est maintenant dans sa septième année d'activité. Les journées nationales sont un temps fort de l'activité de notre GDR, l'occasion pour toute la communauté d'échanger et de s'enrichir des derniers travaux présentés. Plusieurs événements scientifiques sont co-localisés avec ces journées nationales : la 8ième édiction de la conférence francophone sur les architectures logicielles CAL 2014, la 3ième édition de la conférence en IngénieErie Logicielle CIEL 2014, ainsi que 13ème édition d'AFADL 2014, atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels.

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

Trois conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Roland Ducourneau (Université de Montpellier, LIRMM), de Christine Paulin (Université Paris XI, LRI) et de Gérard Morin (Esterel Technologies). Une table ronde, animée par Philippe Collet, Lydie du Bousquet et moi-même abordera les défis 2025. Nous aurons également un retour sur les actions spécifiques menées pendant l'année 2013. Finalement, nous accueillerons une session industrielle, qui sera l'occasion de faire le point sur leurs attentes vis-à-vis de notre domaine.

Le GDR GPL a à cœur de mettre à l'honneur les jeunes chercheurs. C'est pourquoi nous décernerons un prix de thèse du GDR pour la deuxième année consécutine. Nous aurons le plaisir de remettre ce prix de thèse GPL à Mathias Bourguoin pour sa thèse intitulée *Abstractions performantes pour cartes graphiques*. Le jury chargé de sélectionner le lauréat a été présidé par Dominique Méry. Que ce dernier soit ici remercié ainsi que l'ensemble des membres du jury, pour l'excellent travail de sélection.

Ces journées ont aussi pour objectif de préparer l'avenir en favorisant l'intégration des jeunes chercheurs dans la communauté et leur future mobilité. Dans cet esprit, nous les avons encouragés à proposer un poster ou une démonstration de leurs travaux. Une vingtaine ont répondu à cet appel. Un prix du meilleur poster sera égalemment remis par un jury présidé par Jean-Louis Giavitto.

Avant de clôturer cette préface, je tiens à remercier tous ceux qui ont contribué à l'organisation de

ces journées nationales : les responsables de groupes de travail ou d'actions transverses, les membres du comité de direction du GDR GPL et, tout particulièrement, le comité d'organisation de ces journées nationales présidé par Catherine Dubois et Nicolas Levy. Je remercie chaleureusement l'ensemble des collègues parisiens qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Laurence Duchien
Directrice du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2014 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Laurence Duchien (présidente), LIFL, Université Lille 1

Yamine Ait Ameur,LISI / ENSMA
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Florian Brandner, ENSTA
Yohan Boichut, LIFO, Université d'Orléans
Isabelle Borne, IRISA, Université de Bretagne Sud
Frédérice Dadeau, FEMTO-ST, Université de Franche-Comté
Catherine Dubois, CEDRIC, ENSIIE
Lydie Du Bousquet, LIG, Université Joseph Fourier
Frédéric Gava, LACL, Université Paris-Est
Jean-Louis Giavitto (Président du jury des posters), IRCAM, CNRS
Laure Gonnord, LIP (ENS Lyon), Université Lyon1
Gaetan Hains, LACL, Université Paris-Est
Pierre-Cyrille Heam, FEMTO-ST, Université Franche-Comté
Akram Idani, LIG, ENSIMAG
Claude Jard, AtlanSTIC, LINA, Université de Nantes
Thomas Jensen, IRISA, CNRS
Yves Ledru, LIG, Université Joseph Fourier
Pierre-Etienne Moreau, LORIA, INRIA
Pascal Poizat, LIP6, Université Paris-Nanterre
Marc Pouzet, Ecole Normale Supérieure, Université Pierre et Marie Curie, IUF
Fabrice Rastello, INRIA, ENS Lyon
Romain Rouvoy, LIFL, Université Lille 1
Olivier H. Roux, IRCCyN, Université de Nantes
Salah Sadou, VALORIA, Université Bretagne-Sud
Christel Seguin, ONERA Centre de Toulouse
Chouki Tibermacine, LIRMM, Université Montpellier II
Sarah Tucci, CEA LIST
Virginie Wiels, Onera, Centre de Toulouse

Comité scientifique du GDR GPL

Franck Barbier(LIUPPA, Pau)
Charles Consel (LABRI, Bordeaux)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Stéphane Ducasse (INRIA, Lille)
Jacky Estublier (LIG, Grenoble)
Nicolas Halbwachs (Verimag, Grenoble)
Marie-Claude Gaudel (LRI, Orsay)
Gatan Hains (LACL, Créteil)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christine Paulin (LRI, Orsay)

Comité d'organisation

Catherine Dubois (co-présidente), ENSIIE
Nicole Lévy, (co-présidente), CNAM
Tristan Crolard, CNAM
David Delahaye, CNAM
Frédéric Gava, Université de Paris-Est
Fabrice Kordon, Université Pierre et Marie Curie
Renaud Rioboo, ENSIIE

Conférenciers invités

Les talons d'Achille de la programmation par objets

Auteur : Roland Ducournau (Université Montpellier, LIRMM)

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

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

Preuves formelles d'algorithmes probabilistes

Auteur : Christine Paulin (Université Paris XI, LRI)

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

Biographie :
Christine Paulin-Mohring est professeur à l'université Paris-Sud 11 depuis 1997, après avoir été chargée de recherche CNRS au LIP à l'ENS Lyon.

Elle exerce ses activités au LRI dans le cadre du projet commun Inria Toccata. Ses recherches portent sur la théorie des types, les assistants de preuve et leur application au développement de programmes corrects par construction. Elle a contribué au développement de l'assistant de preuve Coq, en particulier en ce qui concerne l'extraction de programmes, les définitions inductives et plus récemment une bibliothèque pour raisonner sur les programmes aléatoires. Elle a coordonné le développement de Coq de 1996 à 2004. Coq a reçu en 2013 deux prix l'ACM SIGPLAN Programming Languages award et l'ACM Software System award.

C. Paulin-Mohring a été déléguée scientifique du centre INRIA Saclay Ile-de-France de 2007 à 2011. Elle est responsable depuis 2012 du labex DigiCosme dans le cadre de l'Idex Paris-Saclay. Elle a été directrice de l'ED Informatique Paris-Sud de 2005 à 2012 et dirige actuellement le collège des Ecoles Doctorales de Paris-Sud. Elle préside le département Informatique de Paris-Sud depuis février 2012.

SCADE Model-Based Requirements Engineering

Auteur : Gérard Morin, Head of Professional Services (Esterel Technologies)

Résumé :

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

Biographie :
Gérard Morin, Directeur des Services Professionnels, a rejoint Esterel Technologies en Octobre 2001 en tant que responsable marketing industriel. Entre 2002 et 2005, il a été directeur du marketing produit pour la ligne de produit SCADE. Depuis 2005, Il est directeur des Services Professionnels : il dirige ou conseille des projets développés avec SCADE avec nos clients, il manage le développement des formations SCADE (présentiel et e-learning). Gérard est spécialisé en ingénierie système et logicielle, en particulier dans le domaine des systèmes critiques (standards DO-178C, ARP-4754A, EN 50128, et IEC 61508).

Sessions des groupes de travail

Session de l'action AFSEC

Approches Formelles des Systémes Embarqués Communicants

Une analyse des boucles de causalité dans les modeleurs de systèmes hybrides

Marc Pouzet

DI, École normale supérieure, 45 rue d'Ulm, 75230 Paris cedex 05.

Les outils de modélisation de systèmes hybrides tels que Simulink/StateFlow,[1] sont vus à juste titre comme de véritables langages de programmation: les *modèles* ne sont plus seulement simulés mais ils servent à produire des séquences de test, à vérifier formellement des propriétés et à générer du code embarqué. Cette chaîne complète depuis de modèle de haut niveau jusqu'au code embarqué explique la nécessité de les appuyer sur une sémantique de référence [5].

Le modèle mathématique sous-jacent est la composition parallèle synchrone d'équations de suites, d'équations différentielles ordinaires (EDOs), d'automates hiérarchiques et de traits impératifs. Si chaque construction prise indépendemment est bien comprise, leur combinaison est la cause de comportements non reproductibles et de bugs dans les compilateurs. Un des points délicats à traiter est celui des boucles de rétroaction (*feedback loop*) non causales. En présence de telles boucles, l'existence de solution aux équations de point-fixe n'est plus garantie et les compilateurs ne savent pas produire du code séquentiel.

La solution la plus simple pour les systèmes à temps discret — celle utilisée dans le langage Lustre, par exemple — consiste à rejeter statiquement les boucles qui ne traversent pas un délai unitaire. Les programmes causalement corrects peuvent alors être compilés vers un code séquentiel qui réalise un pas de calcul. Qu'en est-il lorsque l'on combine des équations de suite, des équations différentielles ordinaires et des ré-initialisations de ces équations différentielles ? Considérons: [2]

```
der y = z init 4.0 and z = 10.0 - 0.1 * y and k = y + 1.0
```

Cette équation définit les signaux y, z et k, tels que pour tout $t \in \mathbb{R}^+$, $\frac{dy}{dt}(t) = z(t)$, $y(0) = 4$, $z(t) = 10 - 0.1 \cdot y(t)$, et $k(t) = y(t) + 1$.[3]

Ce programme est causal simplement parce qu'il est possible de produire une fonction séquentielle $derivative(y) = \texttt{let}\ z = 10 - 0.1 * y\ \texttt{in}\ z$ retournant la valeur courante de la dérivée de y à partir de la valeur courante de y ainsi qu'une valeur initiale 4 pour y. Données à un solveur numérique [4], celui-ci calcule une suite d'approximations $y(t_n)$ pour des valeurs croissantes du temps $t_n \in \mathbb{R}^+$ et $n \in \mathbb{N}$. En coupant les boucles de causalité, un intégrateur joue le rôle d'un délai unitaire en discret. Peut-on réutiliser le même critère que celui de Lustre? Pour cela, imaginons que y soit calculé par un solveur idéal effectuant à chaque instant un pas infiniment petit de longueur ∂ [2]. En écrivant ${}^\star y(n)$, ${}^\star z(n)$ et ${}^\star k(n)$ pour les valeurs de y, z et k à l'instant $n\partial$, avec $n \in {}^\star\mathbb{N}$ un entier non-standard, c'est-à-dire pouvant lui-même être infiniment grand:

$$\begin{aligned} {}^\star y(0) &= 4 & {}^\star z(n) &= 10 - 0.1 \cdot {}^\star y(n) \\ {}^\star y(n+1) &= {}^\star y(n) + {}^\star z(n) \cdot \partial & {}^\star k(n) &= {}^\star y(n) + 1 \end{aligned}$$

où ${}^\star y(n)$ est défini en fonctions des valeurs passées. Les valeurs successives de ${}^\star y(n)$ sont infiniment proches les unes des autres. On retrouve une interprétation classique des signaux en s'appuyant sur le principe de *standardisation* [6]. Ici, la valeur de $y(t)$ à $t \in \mathbb{R}$, est la valeur standard (réelle) de ${}^\star y(n)$ où la valeur standard de $n\partial$ est t (ils sont infiniment proches l'un de l'autre et t est unique).

Considérons maintenant la situation où une équation différentielle est réinitialisée en présence d'un événement. Le signal en dent de scie, par exemple, $y : \mathbb{R}^+ \mapsto \mathbb{R}^+$ tel que $\frac{dy}{dt}(t) = 1$ et $y(t) = 0$ si $t \in \mathbb{N}$, s'écrit:

```
der y = 1.0 init 0.0 reset up(y - 1.0) -> 0.0
```

[1] http://www.mathworks.com/products/simulink

[2] Les programmes sont écrits dans la syntaxe de Zélus (zelus.di.ens.fr).

[3] $\texttt{der}(y) = e\ \texttt{init}\ v_0$ correspond à $y = \frac{1}{s}(e)$ initialisé à v_0 en Simulink.

où `y` est initialisé à 0.0, sa dérivée est 1.0, et il est réinitialisé à 0.0 à chaque fois que `up(y - 1.0)` est vrai, c'est-à-dire que `y - 1.0` traverse 0.0 des négatifs aux positifs. Pour savoir si ce programme est causal, considérons la valeur qu'aurait `y` s'il était calculé par un solveur idéal effectuant des pas infinitésimaux de longeur ∂. La valeur de ${}^\star y(n)$ à l'instant $n\partial$, pour tout $n \in {}^\star\mathbb{N}$ serait:

$$\begin{aligned} {}^\star y(0) &= 0 & {}^\star y(n) &= \texttt{if}\ {}^\star z(n)\ \texttt{then}\ 0.0\ \texttt{else}\ {}^\star ly(n) \\ {}^\star ly(n) &= {}^\star y(n-1) + \partial & {}^\star c(n) &= ({}^\star y(n) - 1) \geq 0 \\ {}^\star z(0) &= \texttt{false} & {}^\star z(n) &= {}^\star c(n) \wedge \neg {}^\star c(n-1) \end{aligned}$$

Cet ensemble d'équations n'est pas causal: la valeur de ${}^\star y(n)$ dépend instantanément de ${}^\star z(n)$ qui dépend lui-même de ${}^\star y(n)$. Il y a deux manières le cycle de causalité: 1. considérer que l'effet du zero-crossing est retardé d'un pas, c'est-à-dire que le test concerne ${}^\star z(n-1)$ plutôt que $z(n)$, ou bien, 2. distinguer la valeur courante de ${}^\star y(n)$ de la valeur qu'il aurait eu s'il n'y avait pas eu de reset, c'est-à-dire ${}^\star ly(n)$. En remplaçant le test de traversée de zéro de y par le test de ly,

$${}^\star c(n) = ({}^\star ly(n) - 1) \geq 0,$$

on obtient un programme causal puisque ${}^\star y(n)$ ne dépend plus instantanément de lui-même. Nous l'écrivons:

```
der y = 1.0 init 0.0 reset up(last y - 1.0) -> 0.0
```

où `last`(y) vaut ly, c'est-à-dire la *limite à gauche* de y. En sémantique synchrone non-standard [2], c'est simplement la valeur précédente du signal et elle coincide avec la limite à gauche lorsque le signal est continu. Cet opérateur `last`(.) sert donc à couper les cycles de causalité et à définir la valeur du signal en fonction de sa valeur précédente, aux moment d'un *zero-crossing*. Lorsque y est définie par sa dérivée, `last`(y) correspond au 'state port' du bloc d'intégration $\frac{1}{s}$ de Simulink. Nous proposons ici d'en faire un usage plus général pour tous les signaux.

Dans cet expose, nous presenterons deux analyses des boucles de causalité pour un langage combinant des constructions d'un langage synchrone — équations de suites, automates hiérarchiques et composition parallèle synchrone — et des équations differentielles ordinaires (EDOs). Ces analyses sont décrites sous la forme de systèmes de type. Lorsque le programme est bien typé, il définit des signaux qui évoluent continuement en dehors des instants de *zero-crossing* déclarés avec la construction `up`(.). Nous illustrerons cette analyse a l'aide d'exemples ecrits dans le langage Zélus [3].

Ce travail a été réalisé en collaboration avec Albert Benveniste, Timothy Bourke, Benoit Caillaud et Bruno Pagano. Il reprend des résultats présentés dans [1].

References

1. Albert Benveniste, Timothy Bourke, Benoit Caillaud, Bruno Pagano, and Marc Pouzet. A Type-based Analysis of Causality Loops in Hybrid Systems Modelers. In *International Conference on Hybrid Systems: Computation and Control (HSCC)*, Berlin, Germany, April 15–17 2014. ACM.
2. Albert Benveniste, Timothy Bourke, Benoit Caillaud, and Marc Pouzet. Non-Standard Semantics of Hybrid Systems Modelers. *Journal of Computer and System Sciences (JCSS)*, 78(3):877–910, May 2012. Special issue in honor of Amir Pnueli.
3. Timothy Bourke and Marc Pouzet. Zélus, a Synchronous Language with ODEs. In *International Conference on Hybrid Systems: Computation and Control (HSCC 2013)*, Philadelphia, USA, April 8–11 2013. ACM.
4. Germund Dahlquist and Åke Björck. *Numerical Methods in Scientific Computing: Volume 1*. SIAM, 2008.
5. Edward A. Lee and Haiyang Zheng. Operational semantics of hybrid systems. In *Hybrid Systems: Computation and Control (HSCC)*, volume 3414, Zurich, Switzerland, March, 9-11 2005. LNCS.
6. T. Lindstrom. An invitation to non standard analysis. In N. Cutland, editor, *Nonstandard analysis and its applications*. Cambridge Univ. Press, 1988.

Precise Robustness Analysis of Time Petri Nets with Inhibitor Arcs*

Étienne André, Giuseppe Pellegrino**, and Laure Petrucci

Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France

Abstract. Quantifying the robustness of a real-time system consists in measuring the maximum extension of the timing delays such that the system still satisfies its specification. In this work, we introduce a more precise notion of robustness, measuring the allowed variability of the timing delays in their neighbourhood. We consider here the formalism of time Petri nets extended with inhibitor arcs. We use the inverse method, initially defined for timed automata. Its output, in the form of a parametric linear constraint relating all timing delays, allows the designer to identify the delays allowing the least variability. We also exhibit a condition and a construction for rendering robust a non-robust system.

Keywords: Time Petri nets, Quantitative robustness, Parameter synthesis

1 Introduction

Formalisms for modelling real-time systems, such as time Petri nets [Mer74] or timed automata [AD94], have been extensively used in the past decades, and led to useful and efficient implementations. Time Petri nets (TPNs for short) are an extension of Petri nets where firing conditions are given in the form of intervals $[a, b]$. Each transition can only fire at least a time units and at most b time units after it is enabled. ITPNs extend TPNs with inhibitor arcs, i.e. arcs that disable their outgoing transition if their incoming place is not empty.

However, these formalisms allow for modelling in theory delays arbitrarily close (or even equal) to zero; this implies that the real system must be arbitrarily fast, which may be unrealistic in practice, where response times may not be neglected. These formalisms also allow for simultaneous occurrence of events, which may not be realistic in practice either, due to slightly different clock rates of several processors. And similarly, they allow for arbitrary precision, which is unrealistic: For example, a system where some component performs an action for e.g. 2 seconds can be implemented with a delay greater but very close to 2 (e.g. 2.0001 s), in which case the formal guarantee may not hold anymore.

The implementation in practice of a real-time system (modelled, e.g. by an ITPN) can lead in particular to two kinds of undesired consequences: the occurrence of behaviours that were proven impossible in theory, and the unlikely occurrence of behaviours that were proven possible in theory.

Consider the simple TPN in Fig. 1a (from [AHJR12]). According to the semantics of TPNs (e.g. defined in [TLR09]), place C is unreachable, that is, there exists no reachable marking such that the number of tokens in C is greater than 0. Indeed, starting from marking A (i.e. a marking with 1 token in place A), t_1 can fire anytime between 1 and 2 time units after the system start. At time 2, t_1 must fire if it has not yet fired, because its associated interval is about to expire and no other transition is firable (t_2 will be firable right after time 2). Hence, C is unreachable. Now suppose that the upper bound of the firing interval of t_1 is increased, even by an infinitesimal duration. Then, t_2 is firable immediately after time 2, and C can be reached in some executions.

Now consider the ITPN in Fig. 1b. According to the semantics of ITPNs, place E is reachable. Indeed, starting from a marking AB (i.e. a marking with 1 token in place A and 1 token in place B),

* This is an author and slightly improved version of the paper of the same name accepted for publication at the 11th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS 2013). The final publication is available at www.springer.com.

** This work is partially supported by an Erasmus grant.

(a) Example of undesired reachability

(b) Example of unlikely reachability

Fig. 1: Examples of non-robust ITPNs

t_1 can fire at time 1, giving marking CB. Then, after a null duration, t_3 can fire due to the absence of token in D. This sequence of transitions is unlikely to happen in practice due to delays exactly equal to zero; if the bounds of t_1 or the lower bound of t_3 become slightly larger, or the bounds of t_2 becomes slightly smaller, E becomes unreachable.

In this work, we use techniques based on parameter synthesis to compute a precise quantitative analysis of the admissible variability of the timing bounds of an ITPN with respect to linear-time properties. We use PITPNs, that is extensions of ITPNs where timing bounds are *parameters*, i.e. unknown constants. Our contributions are as follows:

1. We define the notion of *covering constraint* for parametric time Petri nets with inhibitor arcs (PITPNs), and characterise it;
2. We extend the *inverse method* to PITPNs (initially defined in the setting of parametric timed automata [AS13]), and prove that it preserves linear-time properties, based on the notion of covering constraint; and
3. We exploit the constraint output to obtain a precise quantitative measure of the system robustness for linear-time properties.

Given in the form of a constraint on the timing bounds seen as parameters, our robustness condition allows a designer (i) to relate the variability of the timing bounds with each other, (ii) to exhibit the critical timing bounds that do not allow any variability, and (iii) to render a system robust under certain conditions.

Related Work. Robustness in the setting of timed automata has received much attention in the past decade (see [Mar11] for a survey). Most previous works (see e.g. [BLM+11,Mar11,JR11,AHJR12,Tra12,BMS12]) consider that all timing constraints can be enlarged by a single very small (but positive) variation Δ. This robustness condition considers a unique positive parameter Δ; hence, roughly speaking, the robustness is guaranteed as long as the different clocks remain in intervals $[a-\Delta, b+\Delta]$ instead of $[a,b]$. In a geometrical context, the admissible variability can be seen as a simple hypercube (called "Δ-cube" from now on) in $2*n$ dimensions, with n the number of timing constraints. In contrast, we give a precise measure of the robustness, by considering possible local variations of each lower and upper bound of the firing intervals of a time Petri net. This is given in the form of a *polyhedron* in $2*n$ dimensions, where n is the number of transitions. Hence, each bound can vary independently of the others. Our approach has the following advantages: (1) it identifies the most critical interval bounds, and helps the designer in tuning them (when possible) so that the system becomes robust; (2) it relates bounds in a parametric way, identifying bounds that should, for example, remain smaller than others; (3) it also outputs a constraint even when some bounds cannot tolerate any variation, whereas Δ-based approaches would just classify the system as non-robust (i.e. synthesise a $\Delta = 0$). Since parameter synthesis is undecidable for PITPNs [TLR09], our algorithm may not terminate in the general case; however, we give sufficient termination conditions for subclasses of PITPNs.

In [BMS12], it is shown that parameterised robust reachability in timed automata is decidable, again for a single Δ. In [JR11], computing the greatest acceptable variation Δ is proven decidable for flat timed automata with progressive clocks. In [Tra12], a counter-example refinement approach is used with parametric techniques to evaluate the greatest acceptable variation Δ for parametric timed automata (although not decidable in the general case). These works share similarities with ours in the problem addressed and in the use of parametric techniques. However, beyond the fact that these works consider (a restriction of) timed automata whereas we consider (an extension of) time Petri nets, the main difference lies in the number of dimensions, since they all consider a simple Δ.

Recent work also considered robustness issues in time Petri nets. In [AHJR12], the quantification of robustness is performed by considering that the firing intervals can be enlarged by a (positive) parameter. Two problems are considered: the robust boundedness of the net (a bounded net remains bounded even in presence of small time variations) and the robust untimed language preservation (the untimed language remains preserved in presence of small time variations). Our work is close to [AHJR12], with notable differences. First, we use here a technique based on parameter synthesis. Second, we give a condition for trace preservation, where traces are defined as alternating markings and actions. Hence, the robustness condition in our work is different from the boundedness and language preservation of [AHJR12]. Last but not least, the robustness condition in [AHJR12] again considers a unique positive parameter Δ, whereas we compute a polyhedron in $2*n$ dimensions. In [AHJ+12], a more general notion of robustness is used for time Petri nets, that includes not only a robustness with respect to time, but also with constraints on the resources (e.g. memory), scheduling schemes (in a multi-processor environment) and possible system failures.

Outline. Section 2 recalls PITPNs and related results. In Section 3, we introduce and characterise covering constraints. In Section 4, we introduce the inverse method for PITPNs and prove its correctness. In Section 5, we exhibit a precise quantitative measure of the system robustness, and use it to turn some non-robust systems robust. We give directions of future research in Section 6.

2 Preliminaries

We denote by $\mathbb{N}$, $\mathbb{Q}_+$ and $\mathbb{R}_+$ the sets of non-negative integers, non-negative rational and non-negative real numbers, respectively.

2.1 Firing Times, Parameters and Constraints

Throughout this paper, we assume a set $\{\theta_1, \theta_2, \dots\}$ of *firing times*. A *firing time* is a variable with value in $\mathbb{R}_+$, encoding the time remaining before a given transition fires. In the following, Θ will denote a finite set $\{\theta_1, \dots, \theta_H\}$ of firing times, for some $H \in \mathbb{N}$. A *firing time valuation* is a function $\nu : \Theta \to \mathbb{R}_+^H$ assigning a non-negative real value with each firing time.

We also assume a set $\{\lambda_1, \lambda_2, \dots\}$ of *parameters*, i.e. unknown constants. In the following, $\Lambda = \{\lambda_1, \dots, \lambda_l\}$ denotes a finite set of parameters for some $l \in \mathbb{N}$. A *parameter valuation* π is a function $\pi : \Lambda \to \mathbb{R}_+$ assigning with each parameter a value in $\mathbb{R}_+$. A valuation π can be seen as a point $(\pi(\lambda_1), \dots, \pi(\lambda_l))$.

Constraints are defined as a set of inequalities. A (linear) inequality over Θ and Λ is $lt \prec lt'$, where $\prec \in \{<, \leq\}$, and lt, lt' are two linear terms of the form $\sum_{1 \leq i \leq N} \alpha_i z_i + d$ where $z_i \in \Theta \cup \Lambda$, $\alpha_i \in \mathbb{Q}_+$, for $1 \leq i \leq N$, and $d \in \mathbb{Q}_+$. We define similarly inequalities over Θ (resp. Λ). A constraint is a conjunction of inequalities. In particular, a constraint over the parameters can be seen as a polyhedron in l dimensions. We denote by $\mathcal{L}(\Lambda)$ the set of all constraints over the parameters. In the sequel, J denotes an inequality over the parameters, E a constraint over the firing times, K a constraint over the parameters, and D a constraint over firing times and parameters. Often, given a PITPN transition t_i, we will denote its parametric lower and upper bounds by λ_i^- and λ_i^+, respectively.

Given an inequality J of the form $lt < lt'$ (respectively $lt \leq lt'$), the *negation* of J, denoted by $\neg J$, is the inequality $lt' \leq lt$ (respectively $lt' < lt$).

Given a constraint E and a firing time valuation ν, $[\![E]\!]_\nu$ denotes the expression obtained by replacing each firing time θ in E with $\nu(\theta)$. A firing time valuation ν *satisfies* constraint E (denoted by $\nu \models E$) if $[\![E]\!]_\nu$ evaluates to true.

Given a parameter valuation π and a constraint D, $[\![D]\!]_\pi$ denotes the constraint over Θ obtained by replacing each parameter λ in D with $\pi(\lambda)$. Likewise, given a firing time valuation ν, $[\![[\![D]\!]_\pi]\!]_\nu$ denotes the expression obtained by replacing each firing time θ in $[\![D]\!]_\pi$ with $\nu(\theta)$. We say that a parameter valuation π *satisfies* a constraint D, denoted by $\pi \models D$, if the set of firing time valuations that satisfy $[\![D]\!]_\pi$ is non-empty.

A parameter valuation π *satisfies* a constraint K over the parameters, denoted by $\pi \models K$, if the expression obtained by replacing each parameter λ in K with $\pi(\lambda)$ evaluates to true. Given two constraints K_1 and K_2, K_1 is *included in* K_2, denoted by $K_1 \subseteq K_2$, if $\forall \pi : \pi \models K_1 \Rightarrow \pi \models K_2$. We consider `true` as a constraint over Λ, corresponding to the set of all possible values for Λ.

We denote by $D{\downarrow_\Lambda}$ the constraint over Λ obtained by projecting D onto Λ, i.e. after elimination of the firing times. Formally, $D{\downarrow_\Lambda} = \{\pi \mid \pi \models D\}$.

We finally define intervals as in [TLR09]. An interval I of $\mathbb{R}_+$ is a $\mathbb{Q}_+$-interval if its left endpoint ${}^{\uparrow}I$ belongs to $\mathbb{Q}_+$ and its right endpoint $I^{\uparrow}$ belongs to $\mathbb{Q}_+ \cup \{\infty\}$. We denote by $\mathcal{I}(\mathbb{Q}_+)$ the set of $\mathbb{Q}_+$-intervals of $\mathbb{R}_+$. A parametric time interval is a function $J : {\mathbb{Q}_+}^{\Lambda} \to \mathcal{I}(\mathbb{Q}_+)$ that associates with each parameter valuation a $\mathbb{Q}_+$-interval. The set of parametric time intervals over Λ is denoted by $\mathcal{J}(\Lambda)$. As for I, we define ${}^{\uparrow}J$ and $J^{\uparrow}$ as the minimum and maximum bounds of J, respectively. They can both be represented using a constraint over Λ.

2.2 Parametric Time Petri Nets with Inhibitor Arcs

Parametric time Petri nets with inhibitor arcs (PITPNs) are a parametric extension of ITPNs, where the temporal bounds of the transitions can be parameters. We slightly adapt the notations defined in [TLR09] to fit our setting.

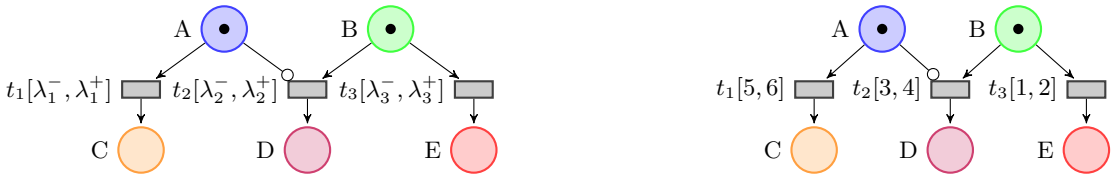
Definition 1. *A parametric time Petri nets with inhibitor arcs (PITPN) is a tuple* $\mathcal{N} = \langle P, T, \Lambda, {}^\bullet(.), (.)^\bullet, (.)^\circ, M_0, J_s, K_0 \rangle$ *where*

- $P = \{p_1, \ldots, p_m\}$ *is a non-empty finite set of* places,
- $T = \{t_1, \ldots, t_n\}$ *is a non-empty finite set of* transitions,
- $\Lambda = \{\lambda_1, \ldots, \lambda_l\}$ *is a finite set of* parameters,
- ${}^\bullet(.)$ *(resp.* $(.)^\bullet$*)* $\in (\mathbb{N}^P)^T$ *is the* backward *(resp.* forward*)* incidence function,
- $(.)^\circ \in (\mathbb{N}^P)^T$ *is the* inhibition function,
- $M_0 \in \mathbb{N}^P$ *is the* initial marking,
- $J_s \in \mathcal{J}(\Lambda))^T$ *is the function that associates a* parametric firing interval *with each transition, and*
- $K_0 \in \mathcal{L}(\Lambda)$ *is the* initial constraint *over* Λ.

K_0 is a constraint over Λ giving the initial domain of the parameters, and must at least specify that the minimum bounds of the firing intervals are lower than or equal to the maximum bounds. Additional linear constraints may of course be given. Sometimes, given a constraint K_0, we will denote a PITPN by $\mathcal{N}(K_0)$ when clear from the context, and to emphasise the value of K_0 in $\mathcal{N}$.

Given a PITPN $\mathcal{N}$ and a valuation π, we denote by $[\![\mathcal{N}]\!]_\pi$ the (non-parametric) ITPN where each occurrence of a parameter has been replaced by its constant value as in π. Formally, given $\mathcal{N} = \langle P, T, \Lambda, {}^\bullet(.), (.)^\bullet, (.)^\circ, M_0, J_s, K_0 \rangle$, then $[\![\mathcal{N}]\!]_\pi = \langle P, T, \Lambda, {}^\bullet(.), (.)^\bullet, (.)^\circ, M_0, J_s, K_0 \wedge K_\pi \rangle$, where $K_\pi = \bigwedge_{\lambda \in \Lambda} \big(\lambda = \pi(\lambda)\big)$. For example, the ITPN in Fig. 2b corresponds to the PITPN in Fig. 2a valuated with $\pi = \{\lambda_1^- \to 5, \lambda_1^+ \to 6, \lambda_2^- \to 3, \lambda_2^+ \to 4, \lambda_3^- \to 1, \lambda_3^+ \to 2\}$.

Semantics. We mostly reuse here the definitions and semantics from [TLR09]. The reachable states of a PITPN are *parametric state-classes* (or simply *classes*), i.e. pairs $c = (M, D)$ where M is a marking of the net and D is a parametric firing domain, that is, a constraint over Θ and Λ. Given a class $c = (M, D)$, a transition t is *enabled* in c if $M \geq {}^\bullet t$ (i.e. if the number of tokens in M in each input place of t is greater than or equal to the value on the arc between this place and



(a) A PITPN $\mathcal{N}$ (b) A valuated (P)ITPN $[\![\mathcal{N}]\!]_\pi$

Fig. 2: A PITPN and its valuation

the transition). Transition t is *inhibited* if the place connected to one of its inhibitor arc is marked with at least as many tokens than the weight of the considered inhibitor arc between this place and t. Transition t is *active* if it is enabled and not inhibited. Transition t is *firable* if it has been active for at least ${}^{\uparrow}J_s(t)$ time units.

For a given class, the firing times in Θ correspond to variables encoding the time remaining before an active transition can fire. Hence, these variables *decrease* with time. The initial class of $\mathcal{N}(K)$ is $c_0 = (M_0, D_0)$, with $D_0 = K \wedge \{\theta_k \in J_s(t_k) | (t_k \in \mathit{enabled}(M_0)\}$, where $\mathit{enabled}(M_0)$ denotes the enabled transitions in M_0. For example, suppose that $K = \lambda_1^- \leq \lambda_1^+ \wedge \lambda_2^- \leq \lambda_2^+ \wedge \lambda_3^- \leq \lambda_3^+$; then the initial class of $\mathcal{N}$ in Fig. 2a is:

$$c_0 = (AB, \lambda_1^- \leq \theta_1 \leq \lambda_1^+ \wedge \lambda_2^- \leq \theta_2 \leq \lambda_2^+ \wedge \lambda_3^- \leq \theta_3 \leq \lambda_3^+).$$

We consider a (classical) semantics where a transition must fire before its upper interval bound, unless another transition fires first and disables it; for example, in Fig. 2a, t_1 must fire before t_3 if $\lambda_1^+ < \lambda_3^-$, t_3 must fire before t_1 if $\lambda_3^+ < \lambda_1^-$, and both orders are possible otherwise. Given a class $c = (M, D)$ and a firable transition t_f, $c' = (M', D')$ can be reached from c in one step via transition t_f (denoted by $c \overset{t_f}{\Rightarrow} c'$) if the following holds:

- $M' = M - {}^\bullet t_f + t_f^\bullet$
- D' is computed along the following steps:
  1. intersection with the firability constraints: $\forall j$ s.t. t_j is active, $\theta_f \leq \theta_j$,
  2. variable substitutions for all enabled transitions t_j that are active, i.e. $\theta_j = \theta_f + \theta'_j$,
  3. elimination (using for instance the Fourier-Motzkin method) of all variables relative to transitions disabled by the firing of t_f,
  4. addition of inequalities relative to newly enabled transitions[1]: $\forall t_k \in \mathit{NewlyEnabled}(M, t_f), {}^{\uparrow}J_s(t_k) \leq \theta'_k \leq J_s(t_k)^{\uparrow}$, with $\mathit{NewlyEnabled}(M, t_f)$ denoting the set of transitions newly enabled by firing the transition t_f from marking M.

The full semantics can be found in [TLR09].

A *run* of $\mathcal{N}$ is a sequence $c_0 \overset{t_0}{\Rightarrow} \cdots \overset{t_{n-1}}{\Rightarrow} c_n$. Given a run r of $\mathcal{N}$ of the form $(M_0, D_0) \overset{t_0}{\Rightarrow} \cdots \overset{t_{n-1}}{\Rightarrow} (M_n, D_n)$, the *trace associated with* r is the alternating sequence of markings and actions $M_0 \overset{t_0}{\Rightarrow} \cdots \overset{t_{n-1}}{\Rightarrow} M_n$. The *trace set* of $\mathcal{N}$ is the set of all traces associated with the runs of $\mathcal{N}$. This corresponds to the discrete (or time-abstract) behaviour of $\mathcal{N}$. $\mathit{Post}_{\mathcal{N}(K)}(C)$ (resp. $\mathit{Post}^i_{\mathcal{N}(K)}(C)$) is the set of classes reachable from a set C of classes in exactly one step (resp. i steps) in $\mathcal{N}(K)$. Furthermore, we define $\mathit{Post}^*_{\mathcal{N}(K)}(C)$ as $\bigcup_{i \geq 0} \mathit{Post}^i_{\mathcal{N}(K)}(C)$. We define $\mathit{Reach}(\mathcal{N}(K))$ as the set of reachable classes of $\mathcal{N}(K)$, that is $\mathit{Post}^*_{\mathcal{N}(K)}(\{c_0\})$. Finally, we define $\mathcal{G}(\mathcal{N}(K))$ as the parametric reachability graph of $\mathcal{N}(K)$, that is the set of reachable parametric state-classes with the transition relation $\Rightarrow$.

Results. The following lemma, recalled from [TLR09], states that the projection onto the parameters of the constraint associated with a class always gets stronger (i.e. more restricted) along a run of the system.

[1] For sake of simplicity, we only consider here closed intervals of the form $[a, b]$. For open intervals (e.g. $(2, 3]$ in Fig. 1a), one should use strict instead of large inequalities.

Lemma 1 (Lemma 14 in [TLR09]). *Given a PITPN $\mathcal{N}$, let $c = (M, D)$ and $c' = (M', D')$ be two classes in $\mathcal{G}(\mathcal{N})$. If $c \overset{t}{\Rightarrow} c'$, then $D'{\downarrow_\Lambda} \subseteq D{\downarrow_\Lambda}$.*

The following result states that the valuation with π of a class c of $\mathcal{N}$ belongs to the graph of $\mathcal{N}$ valuated with π if and only if π belongs to the constraint associated with c.

Theorem 1 (Theorems 12 and 13 in [TLR09]). *Given a PITPN $\mathcal{N}(K)$ and a valuation $\pi \models K$, let $c = (M, D)$ be a class in $\mathcal{G}(\mathcal{N}(K))$. Then: $[\![c]\!]_\pi \in \mathcal{G}([\![\mathcal{N}]\!]_\pi)$ iff $\pi \models D{\downarrow_\Lambda}$.*

3 Covering Constraint

We introduce the notion of covering constraint as the constraint resulting from the intersection of the projection onto the parameters of the constraints associated with all the reachable classes of a PITPN.

Definition 2. *Let $\mathcal{N}$ be a PITPN. The* covering constraint *of $\mathcal{N}$ is:* $\bigcap_{(M,D)\in Reach(\mathcal{N})} D{\downarrow_\Lambda}$.

In the general case, it is possible that the covering constraint of a PITPN will be empty, due to the intersection of disjoint constraints over the parameters. But in the setting of the inverse method (see Section 4), it will not be.

The following lemma relates parametric and non-parametric runs, and derives from Theorem 1.

Lemma 2. *Let $\mathcal{N}$ be a PITPN, let π be a parameter valuation. Let r be a run of $\mathcal{N}$ reaching a class (M, D) in $\mathcal{G}(\mathcal{N})$. Then there exists an equivalent run in $[\![\mathcal{N}]\!]_\pi$ reaching class $(M, [\![D]\!]_\pi)$ in $\mathcal{G}([\![\mathcal{N}]\!]_\pi)$ iff $\pi \models D{\downarrow_\Lambda}$.*

Proof. Let $(M_0, D_0) \overset{t_0}{\Rightarrow} \dots \overset{t_{k-1}}{\Rightarrow} (M_k, D_k)$ be a run of $\mathcal{N}$. From Theorem 1, we have that $[\![(M_k, D_k)]\!]_\pi \in [\![\mathcal{G}(\mathcal{N}(K))]\!]_\pi$ iff $\pi \models D_k{\downarrow_\Lambda}$. Now consider transition $(M_{k-1}, D_{k-1}) \overset{t_{k-1}}{\Rightarrow} (M_k, D_k)$ in $\mathcal{G}(\mathcal{N})$. Then, from the semantics of PITPNs, for all $\pi \models [\![D_k]\!]_\pi$, then $(M_{k-1}, [\![D_{k-1}]\!]_\pi) \overset{t_{k-1}}{\Rightarrow} (M_k, [\![D_k]\!]_\pi) \in \mathcal{G}([\![\mathcal{N}]\!]_\pi)$. The result then derives from a reasoning by induction on k, with $(M, D) = (M_k, D_k)$. □

Conversely, the following lemma states that, given a PITPN $\mathcal{N}$, a run in a valuation of $\mathcal{N}$ always has an equivalent run in $\mathcal{N}$.

Lemma 3. *Let $\mathcal{N}(K)$ be a PITPN, let π be a parameter valuation such that $\pi \models K$. Let r be a run of $[\![\mathcal{N}]\!]_\pi$. Then there exists an equivalent run in $\mathcal{N}(K)$.*

Proof. $[\![\mathcal{N}]\!]_\pi$ can be seen as a PITPN (hence parametric) with an initial constraint K_π. Since $K_\pi \subseteq K$, from the semantics of PITPNs, the set of behaviours of $\mathcal{N}(K)$ includes the behaviours of $\mathcal{N}(K_\pi)$. Hence any run in $\mathcal{N}(K_\pi)$ has an equivalent in $\mathcal{N}(K)$. □

We now state below a general result that will be used to prove Lemma 5.

Lemma 4. *Let $\mathcal{N}(K)$ be a PITPN. Then for all $(M, D) \in \mathcal{G}(\mathcal{N}(K)), D{\downarrow_\Lambda} \subseteq K$.*

Proof. By induction on Lemma 1, with $K_0 \subseteq K$ as the base case. □

The following result states that, for a PITPN with its own covering constraint K_{cov} as initial constraint, the projection onto the parameters of the constraint associated with a reachable class is always the same, and equal to K_{cov}.

Lemma 5. *Let $\mathcal{N}(K)$ be a PITPN, let K_{cov} be the covering constraint of $\mathcal{N}(K)$.*
Then for all $(M, D) \in \mathcal{G}(\mathcal{N}(K_{cov})) : D{\downarrow_\Lambda} = K_{cov}$.

Algorithm 1: *IMPN*$(\mathcal{N}, \pi)$

input : PITPN $\mathcal{N}$ of initial class c_0 and initial constraint K_0, valuation π
output: Constraint K_r

$i \leftarrow 0$; $K_c \leftarrow K_0$; $C \leftarrow \{c_0\}$
while `true` **do**
  while $\exists$ π-*incompatible classes in* C **do**
    Select a π-incompatible class (M, D) of C
    Select a π-incompatible J in $D{\downarrow_\Lambda}$
    $K_c \leftarrow K_c \wedge \neg J$; $C \leftarrow \bigcup_{j=0}^{i} Post^j_{\mathcal{N}(K_c)}(\{c_0\})$
  if $Post_{\mathcal{N}(K_c)}(C) \subseteq C$ **then return** $K_r \leftarrow \bigcap_{(M,D)\in C} D{\downarrow_\Lambda}$;
  $i \leftarrow i + 1$; $C \leftarrow C \cup Post_{\mathcal{N}(K_c)}(C)$

Proof. If K_{cov} is empty, $\mathcal{G}$ is empty too and the result trivially holds. Suppose K_{cov} is non-empty. Let $c = (M, D) \in \mathcal{G}(\mathcal{N}(K_{cov}))$. Let $\pi \models D{\downarrow_\Lambda}$. By Lemma 4, $D{\downarrow_\Lambda} \subseteq K_{cov}$. By construction of K_{cov}, we have that $K_{cov} \subseteq K$. Hence $\pi \models D{\downarrow_\Lambda} \Rightarrow \pi \models K$. Since $\pi \models D{\downarrow_\Lambda}$, from Lemma 2, there exists an equivalent run in $[\![\mathcal{N}]\!]_\pi$ reaching class $(M, [\![D]\!]_\pi)$ in $\mathcal{G}([\![\mathcal{N}]\!]_\pi)$. Since $\pi \models K$, from Lemma 3, there exists an equivalent run in $\mathcal{N}(K)$ reaching class (M, D') for some D'.

Let $\pi' \models K_{cov}$. By construction, $K_{cov} \subseteq D'{\downarrow_\Lambda}$, hence $\pi' \models D'{\downarrow_\Lambda}$. By Lemma 4, $D{\downarrow_\Lambda} \subseteq K$, hence $\pi' \models K$. Since $\pi' \models K$ and $\pi' \models D'{\downarrow_\Lambda}$, applying Theorem 1 to $\mathcal{N}(K)$ gives that $[\![c]\!]_{\pi'} \in \mathcal{G}([\![\mathcal{N}]\!]_{\pi'})$. Since $\pi' \models K_{cov}$ by hypothesis, and $[\![c]\!]_{\pi'} \in \mathcal{G}([\![\mathcal{N}]\!]_{\pi'})$, then applying Theorem 1 to $\mathcal{N}(K_{cov})$ gives that $\pi' \models D{\downarrow_\Lambda}$. Hence $K_{cov} \subseteq D{\downarrow_\Lambda}$. (Lemma 4 gives the other direction.) □

Finally, Theorem 2 states that the trace set of a PIPTN valuated with any parameter valuation satisfying its covering constraint K_{cov} is the same as the trace set of this PITPN with K_{cov} as initial constraint.

Theorem 2. *Let $\mathcal{N}$ be a PITPN, let K_{cov} be the covering constraint of $\mathcal{N}$. Let $\pi \models K_{cov}$. Then the trace sets of $\mathcal{N}(K_{cov})$ and $[\![\mathcal{N}]\!]_\pi$ are equal.*

Proof. Let $\pi \models K_{cov}$. Consider a run of $\mathcal{N}(K_{cov})$ reaching a class (M, D) in $\mathcal{G}(\mathcal{N}(K_{cov}))$. By Lemma 5, it holds that $D{\downarrow_\Lambda} = K_{cov}$. Since $\pi \models K_{cov}$, then $\pi \models D{\downarrow_\Lambda}$. Hence, by Lemma 2, there exists an equivalent run in $\mathcal{G}([\![\mathcal{N}]\!]_\pi)$. Conversely, since $\pi \models K_{cov}$, by lemma 3, any run in $[\![\mathcal{N}]\!]_\pi$ has an equivalent run in $\mathcal{N}(K_{cov})$. □

We can derive from Theorem 2 that the trace set of a PIPTN with any parameter valuation satisfying its covering constraint is always the same. This result will be used to prove the correctness of the inverse method (see Section 4).

Corollary 1. *Let $\mathcal{N}$ be a PITPN, let K_{cov} be the covering constraint of $\mathcal{N}$.*
Then for all $\pi, \pi' \models K_{cov}$, the trace sets of $[\![\mathcal{N}]\!]_\pi$ and $[\![\mathcal{N}]\!]_{\pi'}$ are equal.

4 The Inverse Method for Time Petri Nets

We extend to PITPNs the inverse method initially proposed for timed automata [AS13]. The algorithm relies on the following definition of π-compatibility.

Definition 3. *Given a parameter valuation π, a class (M, D) is said to be* π-compatible *if $\pi \models D{\downarrow_\Lambda}$, and π-incompatible otherwise.*

4.1 Principle

We introduce in Algorithm 1 *IMPN* (i.e. the Inverse Method for time Petri Nets with inhibitor arcs). It uses 3 variables: an integer i measuring the depth of the state space exploration, the current constraint K_c, and the set C of explored classes. Starting from the initial class c_0, *IMPN* iteratively computes classes. When a π-incompatible class is found, an incompatible inequality is non-deterministically selected within the projection of the constraint onto Λ (line 5); its negation is then added to K_c (line 6). The set of reachable classes is then updated. When all successor classes have already been reached (line 7), *IMPN* returns the intersection K_r of the projection onto Λ of the constraints associated with all the reachable classes.

4.2 Results

Lemma 6. *Let $\mathcal{N}$ be a PITPN, and π be a parameter valuation. Suppose that algorithm IMPN$(\mathcal{N}, \pi)$ terminates with output K_r. It holds that $\pi \models K_r$.*

Proof. By construction, at the end of the inner **while** loop, all classes of C are π-compatible, that is for all $(M, D) \in C, \pi \models D$. As a consequence, $\pi \models D{\downarrow_\Lambda}$. Recall that $K_r = \bigcap_{(M,D)\in C} D{\downarrow_\Lambda}$. Hence $\pi \models K_r$. □

The correctness of *IMPN* mainly relies on the fact that K_r is the covering constraint of $\mathcal{N}$. Hence, the results of Section 3 can be applied.

Theorem 3 (Correctness). *Let $\mathcal{N}$ be a PITPN, and π be a parameter valuation. Suppose IMPN$(\mathcal{N}, \pi)$ terminates with output K_r. Then:*

1. *$\pi \models K_r$, and*
2. *$\forall \pi' \models K_r$, $[\![\mathcal{N}]\!]_{\pi'}$ and $[\![\mathcal{N}]\!]_\pi$ have the same trace set.*

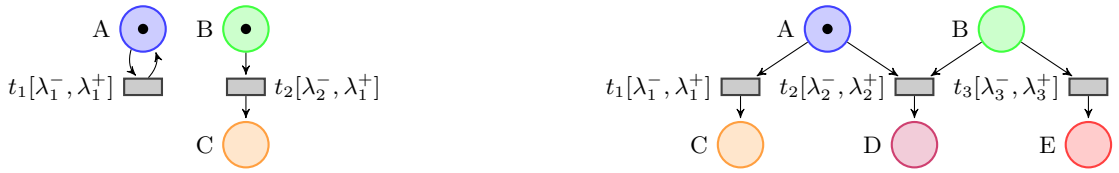
Proof. Item 1 comes from Lemma 6. For item 2, since K_r is the covering constraint of $\mathcal{N}$, then we can apply Corollary 1, which gives the result. Also note that the covering constraint cannot be empty since $\pi \models K_r$. □

Non-termination. Parameter synthesis is undecidable for PITPNs [TLR09] and *IMPN* may not always terminate. Consider the PITPN $\mathcal{N}$ in Fig. 3a; then, *IMPN* applied to $\mathcal{N}$ and a reference valuation with all parameters equal to 0 will generate an infinite set of classes with constraints of the form $i * \lambda_1^- \leq \lambda_2^+$, with i infinitely growing. Intuitively, t_1 can fire an arbitrary number of times before t_2 fires. Of course, this is a typical Zeno-behaviour (an infinite number of transitions within a null duration) and, in the case of non-null reference parameter valuations, an inequality $i*\lambda_1^- \leq \lambda_2^+$ will eventually be π-incompatible, thus ensuring termination. Also note $\mathcal{N}$ is a bounded L/U (lower/upper bounds) PTPN [TLR09], showing that termination of *IMPN* is not guaranteed for general bounded L/U PTPNs (although emptiness and reachability problems are decidable in theory). Studying the decidability of this problem, and adapting *IMPN* to ensure termination in this case is the subject of ongoing work.

We can exhibit subclasses for which *IMPN* terminates. This is obviously the case of loopless PITPNs (in which no syntactical loop exists in the model). This is also the case of parametric sequential TPNs [AHJR12]; this subclass of TPNs is such that each time a discrete transition is fired, each transition that is enabled in the new/resulting marking is newly enabled. Hence, the problem of infinitely concurrent loops such as in Fig. 3a cannot happen.

Non-confluence and Non-completeness. Due to the non-deterministic selection of an inequality, *IMPN* is non-confluent (i.e. different applications of the algorithm can yield different outputs). As a consequence, it is also non-complete (i.e. the resulting constraint may not be the maximal one). Formally:

Proposition 1 (Non-completeness). *There may exist $\pi' \not\models K_r$ such that $[\![\mathcal{N}]\!]_{\pi'}$ and $[\![\mathcal{N}]\!]_\pi$ have the same trace set.*



(a) Non-termination (b) Non-completeness

Fig. 3: Counter-examples PITPNs

An example for non-completeness is the PITPN $\mathcal{N}$ in Fig. 3b, with the reference parameter valuation $\pi = \{\lambda_1^- \to 5, \lambda_1^+ \to 6, \lambda_2^- \to 1, \lambda_2^+ \to 3, \lambda_3^- \to 2, \lambda_3^+ \to 4\}$. In $[\![\mathcal{N}]\!]_\pi$, either t_2 or t_3 can fire first, but not t_1, due to the fact that we have both ${}^\uparrow I(t_1) > I^\uparrow(t_2)$ and ${}^\uparrow I(t_1) > I^\uparrow(t_3)$.

When applying the inverse method to $\mathcal{N}$ and π, a class will be generated with BC as a marking, and an associated constraint containing in particular inequalities $\lambda_1^- \leq \lambda_2^+ \wedge \lambda_1^- \leq \lambda_3^+$. Since both are π-incompatible, the algorithm can add to the current constraint either $\lambda_1^- > \lambda_2^+$ or $\lambda_1^- > \lambda_3^+$. Either of them is sufficient to prevent BC to be reachable. Then the result of the application of *IMPN* to $\mathcal{N}$ and π is both non-confluent and non-complete. Also note that, due to the absence of inhibitor arc in $\mathcal{N}$, the non-completeness of *IMPN* also holds for PTPNs.

Nevertheless, it can be shown (as it was the case for timed automata [AS13]) that a sufficient (but non-necessary) condition for completeness is that *IMPN* does not perform non-deterministic selections of inequalities, i.e. at most one π-incompatible class is met at each iteration.

5 Precise Robustness Analysis

5.1 Local Robustness

Throughout this section, we assume an ITPN N, as well as a parameterised version $\mathcal{N}$ of N where each lower (resp. upper) bound of a transition t_i is replaced with a fresh parameter λ_i^- (resp. λ_i^+). Let π be the reference valuation such that $[\![\mathcal{N}]\!]_\pi = N$. We assume that $\mathit{IMPN}(\mathcal{N}, \pi)$ terminates with output K_r.

We will exploit K_r to characterise the precise robustness of the system, i.e. the admissible variability of each timing bound. The original trace set is preserved by any valuation satisfying K_r. Hence, any linear-time (LTL) property that is true in $[\![\mathcal{N}]\!]_\pi$ is also true in $[\![\mathcal{N}]\!]_{\pi'}$, for $\pi' \models K_r$. Thus, if the correctness is given in the form of an LTL property, the timing delays can safely vary as long as they satisfy K_r.

We use here several examples in order to better illustrate the notions. For the PITPN in Fig. 2a, with $\pi = \{\lambda_1^- \to 5, \lambda_1^+ \to 6, \lambda_2^- \to 3, \lambda_2^+ \to 4, \lambda_3^- \to 1, \lambda_3^+ \to 2\}$ as a reference valuation, *IMPN* outputs the constraint $K_r = \lambda_1^- \leq \lambda_1^+ \wedge \lambda_2^- \leq \lambda_2^+ \wedge \lambda_3^- \leq \lambda_3^+ \wedge \lambda_3^+ < \lambda_1^-$. For a parameterised version of the ITPN in Fig. 1a, *IMPN* outputs the constraint $K_r = \lambda_1^- \leq \lambda_1^+ \wedge \lambda_2^- \leq \lambda_2^+ \wedge \lambda_2^- \geq \lambda_1^+$. For a parameterised version of the ITPN in Fig. 1b, *IMPN* outputs the constraint $K_r = \lambda_1^- \leq \lambda_1^+ \wedge \lambda_2^- \leq \lambda_2^+ \wedge \lambda_3^- = 0 \wedge 0 \leq \lambda_3^+ \wedge \lambda_1^+ = \lambda_2^-$.

Definition 4. *An ITPN N is* robust with respect to linear-time properties *(or* LT-robust*) if there exists $\gamma > 0$ such that for any linear time property φ, $N' \models \varphi$ if and only if $N \models \varphi$, where N' is an ITPN similar to N where each timing bound c can be replaced with any value within $[c - \gamma, c + \gamma]$.*

For example, the ITPN in Fig. 2a is LT-robust (with e.g. $\gamma = 1$), whereas the ITPNs in Fig. 1 are not.

Local Robustness. The resulting constraint K_r is given in the form of a convex (possibly unbounded) polyhedron. For each interval bound λ_i in $\mathcal{N}$, its *local robustness* $LR(\lambda_i)$ is defined as the distance between $\pi(\lambda_i)$ and the closest border of the polyhedral representation of K_r. For example, in Fig. 2a, $LR(\lambda_1^-) = 1$. In Fig. 1a, $LR(\lambda_1^-) = 1$ whereas $LR(\lambda_1^+) = 0$, showing that this

latter bound renders the system non-robust. The following lemma follows from Definition 4, from the definition of LR and the correctness of $IMPN$.

Lemma 7. *If for each parameter λ in $\mathcal{N}$, $LR(\lambda) > 0$, then N is LT-robust.*

Ranging Interval. For each interval bound λ_i in $\mathcal{N}$, its *ranging interval* $RI(\lambda_i)$ is defined as its minimum and maximum admissible values within K_r. It is computed by valuating all parameters but λ_i in K_r, and converting the resulting inequality in the form of an interval. For example, in Fig. 2a, $RI(\lambda_1^-) = (2, 6]$. In Fig. 1a, $RI(\lambda_1^+) = [1, 2]$.

The local lower (resp. upper) variability is defined as the distance between the parameter valuation and the lower (resp. upper) bound of RI; formally, given $RI(\lambda_i) = (a, b)$, $LLV(\lambda_i) = \pi(\lambda_i) - a$ and $LUV(\lambda_i) = b - \pi(\lambda_i)$. Note that the local robustness can be obtained from the local variability: $LR(\lambda_i) = \min(LLV(\lambda_i), LUV(\lambda_i))$.

Computation of Δ. Our approach also allows to retrieve the value of the "Δ" of Δ-based approaches. It is defined as the minimum over the set of parameters of the distance between a parameter and the closest border of the polyhedron. Formally, $\Delta = \min\big(\min_{i\in\Delta^-} LLV(\lambda_i), \min_{i\in\Delta^+} LUV(\lambda_i)\big)$, where Δ^- (resp. Δ^+) denotes the set of parameters appearing in an interval lower (resp. upper) bound. This distinction is necessary, since Δ-based approaches only consider the positive enlarging of intervals. For the ITPN in Fig. 2a, the maximum possible Δ is 1.5 (see Section 5.3). And, obviously, $\Delta = 0$ for the ITPNs in Fig. 1.

5.2 Improving the System Robustness

Identifying Critical Timing Bounds. Our approach allows to exhibit *critical* timing bounds: critical timing bounds are those rendering the system non-robust, i.e. with a null local robustness. For example, in Fig. 1a, λ_1^+ and λ_2^- are the critical timing bounds. In Fig. 1b, λ_1^+, λ_2^- and λ_3^- are the critical timing bounds.

Relaxing Bounds. For some systems, it is possible to refine the values of the critical timing bounds so that the system becomes robust, with the same discrete behaviour. In practice, this may in particular be the case of hardware systems, where the timing bounds come from the traversal time of micro components: One can change the timing bounds by replacing a component with another one. In software, one can also refine the values of some timers if needed.

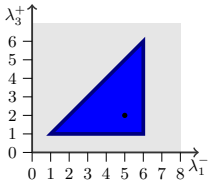
In that case, one can exploit the precise robustness analysis to synthesise values for the timing bounds so that the system is robust. A system is said to be potentially robust if all timing bounds λ_i have a ranging interval non-reduced to a point (even if their local robustness may possibly be null, i.e. $LR(\lambda_i) = 0$).

Definition 5. *An ITPN N is* potentially robust *if, for all timing bounds λ_i, $LLV(\lambda_i) > 0$ or $LUV(\lambda_i) > 0$.*

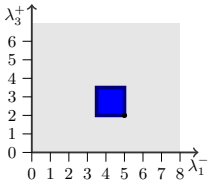
This notion of potential robustness is a sufficient condition so that an ITPN becomes robust with the same discrete behaviour.

Theorem 4. *If N is potentially robust, then there exists π_R such that $[\![\mathcal{N}]\!]_{\pi_R}$ is LT-robust, and has the same trace set as N.*

Proof. By Lemma 7, only the timing bounds λ_i such that $LR(\lambda_i) = 0$ render $\mathcal{N}$ non-LT-robust. For all λ_i such that $LR(\lambda_i) > 0$, we set $\pi_R(\lambda_i) = \pi(\lambda_i)$. Now consider a λ_i such that $LR(\lambda_i) = 0$. By definition of LR, either $LLV(\lambda_i) = 0$ or $LUV(\lambda_i) = 0$. Consider the former case (the latter case is dual). Let $(a, b) = RI(\lambda_i)$. Let $\pi_R(\lambda_i) = \pi(\lambda_i) + \big(a + b)\big)/2$. Since $\mathcal{N}$ is potentially robust, and since $LLV(\lambda_i) = 0$, then $LUV(\lambda_i) > 0$. By definition of LUV, $LUV(\lambda_i) = b - \pi(\lambda_i)$, hence $b > \pi(\lambda_i)$. Hence $a < \pi_R(\lambda_i) < b$. As a consequence, in π_R, we have $LR(\lambda_i) > 0$. By construction, and from the convexity of K_r, $\pi_R(\lambda_i)$ is in K_r; hence, from Theorem 3, $[\![\mathcal{N}]\!]_{\pi_R}$ and $[\![\mathcal{N}]\!]_{\pi}$ have the same trace set. □



(a) Representation of K_r



(b) Representation of the Δ-cube

Fig. 4: Graphical comparison for the example in Fig. 2a

Note that this is a sufficient but non-necessary condition, since the notion of potential robustness is based on LLV and LUV, that come from K_r, which is non-complete. Furthermore, one can find further conditions (and constructions) to render a system robust. For example, the ITPN in Fig. 1b is not potentially robust; but it can be made robust with the same discrete behaviour, e.g. by replacing the intervals associated with both t_1 and t_2 with $[0, 1]$.

5.3 Comparison with Δ-based Approaches

The main drawback of our approach is that it does not terminate in the general case, although we exhibited cases for which termination is guaranteed (see Section 4.2). In contrast, related work show that deciding only whether a system is robust is decidable in most cases. However, beside the fact that we give a quantitative measure of the robustness in the form of a constraint in $2 * n$ dimensions (with n the number of transitions), our approach is particularly interesting in the case of a non-robust system. First, we exhibit which timing bounds are responsible for the non-robustness. Second, we give a condition to render the system robust without changing its discrete behaviour.

Furthermore, our approach may output a significantly larger constraint than the Δ-cube output by Δ-based approaches. Actually, when the result of *IMPN* is complete, the resulting polyhedron is necessarily at least as large as the Δ-cube. Consider again the example in Fig. 2a. In order to enable a graphical comparison in 2 dimensions, we assign all parameters but λ_1^- and λ_3^+ to their value as in π. Hence the constraint becomes $\lambda_1^- \leq 6 \wedge 1 \leq \lambda_3^+ \wedge \lambda_3^+ < \lambda_1^-$. This constraint is depicted in Fig. 4a. As of Δ-based approaches, they cannot compute a value for Δ greater than 1.5 in this situation. Indeed, with $\Delta = 1.5$, λ_3^+ becomes $\lambda_3^+ + \Delta = 3.5$, λ_1^- becomes $\lambda_1^- - \Delta = 3.5$, in which case the discrete behaviour becomes different (t_1 can fire before t_3). This Δ is given in Fig. 4b.

The interpretation of the much larger parametric domain covered by K_r compared to the Δ-cube can be explained as follows: (1) The parametric domain below $\lambda_3^+ = 2$ and above $\lambda_1^- = 5$ is not covered by the Δ-cube, because Δ-based approaches consider a positive parameter $\Delta \geq 0$. Hence, it is not possible to study, e.g. by how much an upper bound can be decreased. (2) The constraint K_r allows to relate parameters. Whereas the value of Δ prevents λ_1^- and λ_3^+ to vary by more than 1.5, the inequality $\lambda_3^+ < \lambda_1^-$ states that λ_1^- may vary by more than 1.5, as long as λ_3^+ varies less (i.e. $\lambda_3^+ < \lambda_1^-$). This is of particular interest in systems where some bounds are more likely to vary than others. (3) This small example is a "good" example for Δ-based approaches. In the case where at least one parameter cannot vary, Δ would be inevitably equal to 0, whereas K_r would still give an output for other dimensions. This is the case of the ITPNs in Fig. 1.

6 Final Remarks

In this paper, we extended the inverse method to PITPNs and showed how to exploit its output to obtain a precise quantitative measure of the system robustness for linear-time properties. This paper considers the quantification of the system robustness with respect to linear-time (hence time-abstract) properties only. Nevertheless, timed properties can also be considered, by adding

an observer net. This observer synchronises with the system ITPN, and can reduce timed properties to time-abstract properties.

Our algorithms should be implemented and compared with similar tools, such as Shrinktech [San13]. Finally, we only addressed here the variability of the timing delays (Δ), but not the admissible variations of the clock speed (usually called "ϵ"). Our approach could be extended to this setting using extensions of the inverse method for parameterised hybrid systems [FK13], by adding for each clock two additional parameters ϵ_i^- and ϵ_i^+ measuring the admissible decrease and increase speed rate.

Acknowledgment. We are grateful to an anonymous reviewer for his/her very detailed comments.

References

AD94. Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

AHJ+12. S. Akshay, Loïc Hélouët, Claude Jard, Didier Lime, and Olivier H. Roux. Robustness of time Petri nets under architectural constraints. In *FORMATS*, volume 7595 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2012.

AHJR12. S. Akshay, Loïc Hélouët, Claude Jard, and Pierre-Alain Reynier. Robustness of time Petri nets under guard enlargement. In *RP*, volume 7550 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2012.

AS13. Étienne André and Romain Soulat. *The Inverse Method*. FOCUS Series in Computer Engineering and Information Technology. ISTE Ltd and John Wiley & Sons Inc., 2013.

BLM+11. Patricia Bouyer, Kim G. Larsen, Nicolas Markey, Ocan Sankur, and Claus R. Thrane. Timed automata can always be made implementable. In *CONCUR*, volume 6901 of *Lecture Notes in Computer Science*, pages 76–91. Springer, 2011.

BMS12. Patricia Bouyer, Nicolas Markey, and Ocan Sankur. Robust reachability in timed automata: A game-based approach. In *ICALP*, volume 7392 of *Lecture Notes in Computer Science*, pages 128–140. Springer, 2012.

FK13. Laurent Fribourg and Ulrich Kühne. Parametric verification and test coverage for hybrid automata using the inverse method. *IJFCS*, 24(2):233–249, 2013.

JR11. Rémi Jaubert and Pierre-Alain Reynier. Quantitative robustness analysis of flat timed automata. In *FoSSaCS*, volume 6604 of *Lecture Notes in Computer Science*, pages 229–244. Springer-Verlag, 2011.

Mar11. Nicolas Markey. Robustness in real-time systems. In *SIES*, pages 28–34. IEEE Computer Society Press, 2011.

Mer74. Philip Meir Merlin. *A study of the recoverability of computing systems.* PhD thesis, University of California, Irvine, CA, USA, 1974.

San13. Ocan Sankur. Shrinktech: A tool for the robustness analysis of timed automata. In *CAV*, Lecture Notes in Computer Science. Springer, 2013. To appear.

TLR09. Louis-Marie Traonouez, Didier Lime, and Olivier H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009.

Tra12. Louis-Marie Traonouez. A parametric counterexample refinement approach for robust timed specifications. In *FIT*, volume 87 of *EPTCS*, pages 17–33, 2012.

Synthèse de pilotes de périphériques pour systèmes temps-réel embarqués

Julien Tanguy

See4sys Technologie,
Espace Performance La Fleuriaye,
44481 Carquefou CEDEX, France
julien.tanguy@see4sys.com

1 Introduction

Le développement de pilotes de périphériques embarqués est une tâche ardue et sujette à erreurs. À l'interface entre le matériel, l'applicatif et le système d'exploitation, un pilote doit respecter les contraintes imposées par les trois éléments, ce qui implique que les concepteurs doivent posséder une bonne connaissance de tous ces éléments pour développer des pilotes efficaces et sûrs de fonctionnement. L'aspect sûreté est renforcé par la criticité du contexte d'exécution des pilotes: pour pouvoir interagir avec le matériel, ces derniers doivent être exécutés en mode superviseur; un bug peut avoir des conséquences néfastes sur *l'ensemble du système*.

De plus, les interfaces sont généralement mal définies. Le côté applicatif est en général hors contexte lors de la conception d'un pilote, et l'interface matérielle, définie dans la *datasheet* du composant, est parfois incomplète. Pour pallier tous ces problèmes, un certain nombres de méthodes de vérification ont été mises en place [1,4]. Cependant le nombre de cas d'utilisation et a fortiori de configurations possible rend en pratique toute vérification exhaustive impossible.

Une alternative à la vérification formelle est de dériver les pilotes directement depuis une modélisation formelle. Les bonnes propriétés de sûreté peuvent être vérifiées exhaustivement par des techniques de *model checking*.

Ces travaux s'inscrivent dans un contexte temps-réel embarqué critique, avec des contraintes temporelles fortes. Ces systèmes possèdent habituellement des contraintes fortes en terme de sûreté de fonctionnement, mais ne disposent que peu de ressources systèmes (mémoire, puissance de calcul, etc.) pour y parvenir.

Pour répondre à cette problématique, chaque communauté a développé un standard d'architecture et de méthodologie. Par exemple, la communauté automobile a développé le standard AUTOSAR[2], un standard permettant une plus grande interopérabilité des différents composants logiciels, réduisant ainsi le risque de bugs liés à une mauvaise utilisation de telle ou telle brique logicielle. Cette architecture définit un ensemble de modules configurables à différents niveaux d'abstraction. Cependant cette configurabilité rend la tâche d'intégration plus ardue, et bien souvent il est nécessaire d'avoir recours à un outil de configuration afin de correctement paramétrer une plateforme logicielle.

Nous proposons une approche différente, spécifique à chaque application et permettant de réduire le nombre de niveaux d'abstraction entre l'application et le pilote. Cette approche, basée sur une modélisation formelle des différents composants, permet d'élever le problème de contrôle du périphérique au niveau du modèle et de générer le code minimal nécessaire à nos besoins.

1.1 Travaux précédents

L'idée de générer des pilotes de périphériques n'est pas nouvelle et a déjà été proposée pour des ordinateurs de bureau.

Wang et Malik[7] proposent un modèle permettant de générer des pilotes de périphériques complets et de vérifier des propriétés sur ces derniers. Cependant, ce modèle vise à générer des pilotes pour des machines de bureau, plus puissantes, avec une architecture conforme au modèle traditionnel de pilote pour des raisons de compatibilité et d'échangeabilité.

L'outil Termite[6] se base sur un framework particulier, *Dingo*, afin de simplifier les interactions pilote/environnement. Ce framework, bien que très simple dans son fonctionnement, possède un surcoût de ressources à l'exécution, surcoût trop important pour la plupart des systèmes embarqués.

1.2 Notre contribution

Nous proposons une méthodologie de génération de pilotes de périphériques basée sur une modélisation formelle. Le but est de générer automatiquement des pilotes spécifiques à une application donnée, en utilisant une particularité des systèmes embarqués: leur comportement est entièrement défini et connu à la conception. Ces informations nous permettent de réaliser des optimisations au niveau du modèle et de ne générer que le nécessaire pour assurer le bon fonctionnement du pilote généré.

Ceci nous permet de réduire la taille de l'exécutif final, réduisant ainsi l'empreinte mémoire, tout en assurant la sûreté du système en éliminant tout code mort. L'interface pilote/applicatif est également simplifiée car générée spécifiquement aux besoins de ce dernier.

2 Définitions

On note $\mathbb{N}$ l'ensemble des nombres entiers positifs. Soit E un ensemble. On note 2^E l'ensemble de ses sous-ensembles. On définit également une logique propositionnelle γ_P sur un ensemble de prédicats atomiques P de la forme

$$\varphi := p | \neg\varphi | \varphi \wedge \varphi, \text{où } p \in P$$

Pour $A \subseteq P$, la sémantique de cette logique est définie par :

- $A \vDash p \Leftrightarrow p \in A$;
- $A \vDash \neg\varphi \Leftrightarrow A \nvDash \varphi$

– $A \vDash \varphi \wedge \psi \Leftrightarrow A \vDash \varphi$ et $A \vDash \psi$

On définit également une notion de recouvrement de formules: Soir $g, g' \in \gamma_P$. g et g' se recouvrent si

$$\exists A \subseteq P, \text{ tel que } A \vDash g \text{ et } A \vDash g'$$

Definition 1 (Système de transition étiqueté avec gardes). *Un système de transition étiqueté avec gardes (Guarded Labeled Transition System, ou GLTS) est le tuple*

$$(Q, Q_0, A, P, E, l)\text{, avec}$$

– Q *est un ensemble d'états;*
– Q_0 *est un ensemble d'états initiaux;*
– A *est un ensemble d'actions;*
– P *est un ensemble de propositions atomiques;*
– $E \subseteq Q \times \gamma_P \times A \times Q$ *est un ensemble d'arcs entre les états;*
– $l \subseteq Q \times 2^P$ *est une fonction d'étiquetage.*

Similairement aux systèmes de transitions étiquetés classiques, un GLTS (Q, Q_0, A, P, E, l) est dit déterministe si - $|Q_0| = 1$. On note alors cet état q_0. - si (q, a, g', q') et $(q, a, g'', q'') \in E,$ alors g', g'' ne se recouvrent pas si $q' = q''$. Par la suite, on considèrera des GLTS déterministes uniquement.

On définit également des notions de produits asynchrones de plusieurs GLTS.

Definition 2 (Sémantique d'un GLTS). *La sémantique d'un GTLS*

$$(Q, q_0, A, P, E, l)$$

est un système de transition étiqueté $(Q, q_0, A, \rightarrow)$*, où*

$$\forall (q, a, g, q') \in E, (q, a, q') \in \rightarrow \iff l(q) \vDash g$$

.

3 Méthodologie

Le pilote généré est issu de plusieurs modèles différents. À la base, le modèle du périphérique qui en représente le comportement dans tous les cas[1]. Vient s'ajouter le modèle de configuration du pilote. Ce modèle s'appuie sur celui du pilote, et vient en restreindre les comportements conformément à son utilisation par l'application. Le modèle des objectifs du pilote, quant à lui, représente les fonctions que doit remplir ce dernier: empêcher un buffer de déborder, rester dans un mode de fonctionnement du périphérique nominal, etc. Ces trois modèles servent à générer le modèle du pilote configuré. Ce dernier représente le comportement abstrait du pilote final, et est transformé en code grâce au modèle d'implémentation.

La méthodologie présentée figure 1 est la suivante:

[1] Il n'est pas nécessaire que le modèle soit représentatif de *tous* les comportements du périphérique, mais il faut que les comportements présents dans le modèle soient aussi présents dans le périphérique

1. Modélisation du périphérique matériel;
2. Modélisation de la configuration du matériel en accord avec son utilisation par l'application;
3. Définition des objectifs du pilote;
4. Génération du modèle du pilote configuré;
5. Transformation du modèle configuré en code.

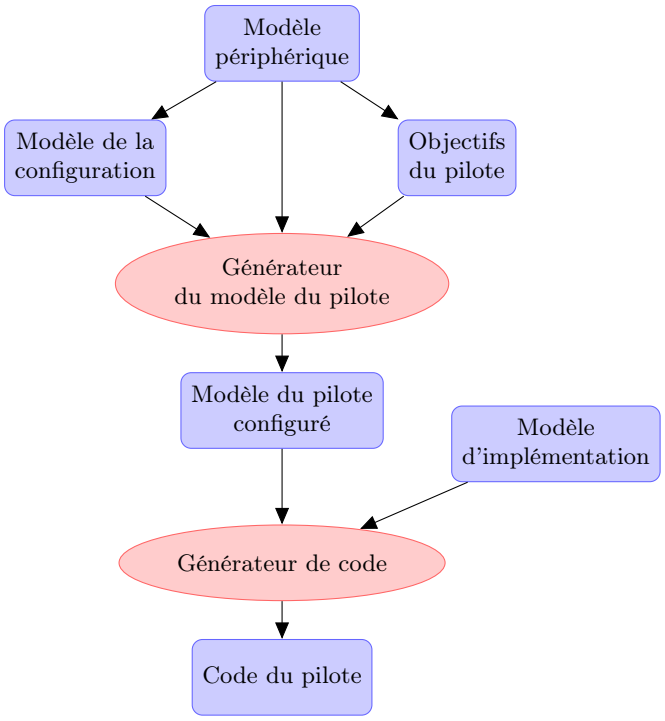


Fig. 1. Méthodologie de génération

3.1 Modélisation des différentes parties

Modélisation du matériel Le premier modèle — le modèle du périphérique — représente le comportement du matériel à bas niveau, au niveau des interactions via les registres: écriture dans les registres de configuration et de contrôle, lecture dans les registres de statut et de données, ainsi que les interruptions.

Ce modèle est basé uniquement sur le périphérique, et est indépendant de toute application. Il est réutilisable d'une application à une autre, pourvu que le matériel soit le même.

À l'issue de cette modélisation, le modèle du périphérique offre au concepteur:

- un ensemble d'actions abstraites A, qui représentent les différentes actions que le périphérique effectue et celles que le pilote peut effectuer.
- un ensemble de paramètres de configuration, sous la forme de propriétés atomiques P_{cfg}. Ces propriétés peuvent être arrangées sémantiquement grâce à la logique propositionnelle.
- un ensemble de propriétés de synchronisation P^{sync}. Ces propriétés représentent des conditions nécessaires pour effectuer certaines actions du périphérique (comme par exemple un changement d'horloge qui doit être effectué périphérique en veille).
- un ensemble de propriétés informelles P_{info} sur l'état tu périphérique, comme *PowerDown*, *Idle*, *Busy*, *Waiting*, etc.

Modèle de configuration Une fois le modèle du périphérique défini, le concepteur doit définir la façon dont l'application va utiliser ce dernier. Il peut s'agir de modes de fonctionnement *Nominal*, *Veille*, *Basse consommation*, de propriétés de configuration statiques, comme un type de trame, une configuration de broches d'entrées/sorties, ou bien encore un protocole complet. La seule contrainte est que ce modèle respecte les propriétés de synchronisation (sous la forme de gardes dans certaines actions).

3.2 Génération du pilote final

Une fois les différents modèles d'entrée définis, on génère le modèle du pilote configuré. Il met à profit la théorie des jeux[5,3] pour générer un contrôleur (le pilote) respectant les objectifs sus définis.

Anatomie d'un pilote Dans notre modèle, un pilote $\mathcal{D}$ répond à un ensemble d'objectifs $\mathcal{O}$. Un objectif représente un ensemble de propriétés que le pilote doit satisfaire, comme par exemple *Ne jamais être dans un état de dépassement de buffer*, ou *Aller dans l'état de veille*.

Pour chaque objectif, le pilote possède une stratégie, c'est-à-dire une séquence d'actions à effectuer pour arriver dans un certain état ou pour éviter certains états. Ces stratégies sont générées automatiquement à partir des différents modèles du périphérique et de sa configuration.

À tout instant, le pilote n'a qu'un seul objectif actif, et effectue les actions nécessaires pour le satisfaire.

Génération du modèle du pilote Pour générer le modèle du pilote configuré en utilisant la théorie des jeux, il faut tout d'abord mettre en place le *plateau*. Le plateau est un système de transitions étiqueté modélisant tous les états possibles du pilote et du périphérique, ainsi que les actions possibles de ces derniers. Il est obtenu en pratique en réalisant le produit asynchrone du modèle matériel et du modèle de configuration.

Le problème se résume alors à un jeu non temporisé à deux joueurs, avec d'un côté le pilote effectuant les actions contrôlables, et le périphérique et son

environnement effectuant le actions incontrôlables. La résolution de ce problème est décidable, et des algorithmes ont été développés pour générer des stratégies.

Un des plus communément utilisé utilise un point fixe sur l'opérateur *prédécesseur contrôlable*[3].

Intuitivement, cette méthode calcule de façon itérative l'ensemble des états gagnants, pour lesquels il existe une stratégie pour respecter l'objectif. Elle part des états pour lesquels ces objectifs sont vérifiés, et effectue une recherche en arrière pour déterminer les prédécesseurs contrôlables de ces états, c'est-à-dire les états pour lesquels le pilote a la possibilité de forcer le périphérique à aller dans un état gagnant. À chaque itération, les nouveaux états pour lesquels on a une stratégie sont ajoutés aux états gagnants. L'algorithme s'arrête quand on a atteint un point fixe, et qu'on ne peut plus ajouter d'état gagnant. Les états restants sont les états perdants, états dans lesquels le pilote n'a aucun moyen sûr (voire aucun moyen du tout) de satisfaire les objectifs.

4 Conclusion

Nous avons développé une méthodologie générique, ainsi que les modèles associés pour générer des pilote de périphériques embarqués. Cette méthode est spécifique aux systèmes embarqués, et permet de générer des pilotes minimaux tout en garantissant le bon fonctionnement de l'ensemble.

Grâce aux informations disponibles à la conception, il est en effet possible de déterminer l'ensemble des états dans lesquels *peut* se trouver le périphérique et son pilote, assurant ainsi l'absence de code mort dans l'exécutif généré.

Perspectives Nous envisageons l'ajout de données explicites au sein du modèle, afin de gérer et modéliser des comportements et des pilotes plus complexes.

References

1. T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S.K. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. *ACM SIGOPS Operating Systems Review*, 40(4):73–85, 2006.
2. Frank Kirschke-Biller. Autosar – A worldwide standard current developments, roll-out and outlook. www.autosar.org, 2011.
3. A Pnueli, E Asarin, O Maler, and J Sifakis. Controller synthesis for timed automata. In *Proc. System Structure and Control. Elsevier*. Citeseer, 1998.
4. H. Post and W. Küchlin. Integrated static analysis for linux device driver verification. In *Integrated Formal Methods*, pages 518–537. Springer, 2007.
5. Peter JG Ramadge and W Murray Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, 1989.
6. Leonid Ryzhyk. *On the Construction of Reliable Device Drivers*. PhD thesis, University of New South Wales, 2009.
7. Shaojie Wang, Sharad Malik, and Reinaldo A Bergamaschi. Modeling and integration of peripheral devices in embedded systems. In *Proceedings of the conference on Design, Automation and Test in Europe*, volume 1, pages 136–141. IEEE Computer Society, 2003.

Session du groupe de travail COSMAL

Composants Objets Services : Modèles, Architectures et Langages

Component-based Autonomic Managers for Coordination *

Soguy Mak Karé Gueye[1], Noël de Palma[1], and Eric Rutten[2]

[1] LIG / UJF, Grenoble, France,
soguy-makkare.gueye@imag.fr, noel.depalma@imag.fr
[2] INRIA, Grenoble, France, eric.rutten@inria.fr

Abstract. This paper presents our work addressing the coordination of multiple autonomic managers. We focus on the component-based approach which provides supports to build systems with introspection, adaptivity and reconfiguration features, and reactive models and discrete control techniques for the synthesis of a control logic for the Coordination of the managers constituting a component-based autonomic management system.

1 Introduction

The increasing complexity of computing systems has motivated the automation of their administration functions in the form of autonomic managers. Today many autonomic managers are available for use but they are usually dedicated for few management aspects. An autonomic management system requires several autonomic managers addressing different aspects to ensure a global management. However coordinating the managers activities is necessary to avoid incoherent and conflicting decisions.

Discrete control. We investigate the use of reactive models and discrete control techniques to build a hierarchical controller enforcing coherency properties on the autonomic managers at runtime. One specificity and novelty of our approach is that discrete controller synthesis (DCS) performs the automatic generation of the control logic, from the specification of an objective, and automata-based descriptions of managers behaviours.

Component-based assembly. We use the component-based approach [1] for building a management system made of several autonomic managers with the hierarchical controller for their coordination. The latter approach supports building systems with introspection, adaptivity and reconfiguration features.

2 Design approach

In this section we give a short description of our approach, however more details are available in [3]. In our approach we use the Heptagon/BZR language [2] for modelling of managers behaviours, and for applying DCS.

* This work is supported by the ANR INFRA project Ctrl-Green.

Modelling manager control. As shown in Fig. 1, automata are used to model the different execution states of the managers which are relevant for their coordination – their activity as well as their ability to be controlled. The automata are equipped with controllable variables making it possible to prevent, when not allowed, states transition leading to the triggerring of the management processes inherent to their management decisions.

In components wrapping the managers, the automata are associated with the component controller in the membrane as in Fig. 2, and are updated at runtime to reflect the current state of component.

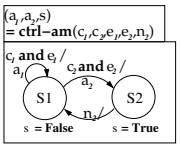


Fig. 1. AM model

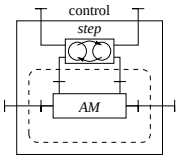


Fig. 2. Component

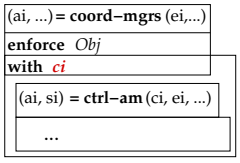


Fig. 3. Coordination

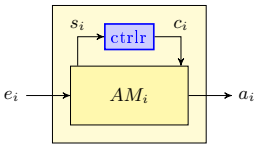


Fig. 4. Assembly

Coordinated assembly of controllable AMs. Once the managers are modelled, the latter models are composed in parallel to model their uncoordinated coexistence. The controllable variables in the automata are used as control points to enforce the coordination strategy. In BZR the specification of the coordination strategy (control objectives) is done by associating a BZR contract to the node modelling the coexistence as shown in Fig. 3. Then the BZR compiler, and its associated DCS tool, solves the control problem by automatically computing the controller ensuring, by automated formal computation, a correct coordination of the managers respecting the contract as shown in Fig. 4.

3 Conclusion

We address the problem of coordination of multiple autonomic managers using a component-based approach to build the complex management system, and reactive models and discrete controller synthesis for modeling managers behaviors and for the construction of the coordination controller.

References

1. E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The Fractal component model and its support in java. *Software – Practice and Experience (SP&E)*, 36(11-12), sep 2006.
2. G. Delaval, H. Marchand, and E. Rutten. Contracts for modular discrete controller synthesis. In *Proc. Conf. LCTES*, 2010.
3. S. M.-K. Gueye, N. de Palma, and E. Rutten. Coordination control of component-based autonomic administration loops. In *Proc. Conf. Coordination*, 2013.

Efficient High-Level Abstractions for Web Programming

Julien Richard-Foy, Olivier Barais, and Jean-Marc Jézéquel

IRISA, Université de Rennes 1 {first}.{last}@irisa.fr

Abstract. Writing large Web applications is known to be difficult. One challenge comes from the fact that the application's logic is scattered into heterogeneous clients and servers, making it difficult to share code between both sides or to move code from one side to the other. Another challenge is performance: while Web applications rely on ever more code on the client-side, they may run on smart phones with limited hardware capabilities. These two challenges raise the following problem: how to benefit from high-level languages and libraries making code complexity easier to manage and abstracting over the clients and servers differences without trading this ease of engineering for performance? This article presents high-level abstractions defined as deep embedded DSLs in Scala that can generate efficient code leveraging the characteristics of both client and server environments. We compare performance on client-side against other candidate technologies and against hand written low-level JavaScript code. Though code written with our DSL has a high level of abstraction, our benchmark on a real world application reports that it runs as fast as hand tuned low-level JavaScript code.

Keywords: Heterogeneous code generation, Domain-specific languages, Scala, Web

1 Introduction

Web applications are attractive because they require no installation or deployment steps on clients and enable large scale collaborative experiences. However, writing large Web applications is known to be difficult [1,2]. One challenge comes from the fact that the business logic is scattered into heterogeneous client-side and server-side environments [3,4]. This gives less flexibility in the engineering process and requires a higher maintenance effort: there is no way to move a piece of code targeting the server-side to target the client-side – the code has to be rewritten. Even worse, logic parts that run on both client-side and server-side need to be duplicated. For instance, HTML fragments may be built from the server-side when a page is requested by a client, but they may also be built from the client-side to perform an incremental update subsequent to a user action. How could developers write HTML fragment definitions once and render them on both client-side and server-side?

The more interactive the application is, the more logic needs to be duplicated between the server-side and the client-side, and the higher is the complexity of the client-side code. Developers can use libraries and frameworks to get high-level abstractions on client-side, making their code easier to reason about and to maintain, but also making their code run less efficiently due to *abstraction penalty*.

Performance is a primary concern in many Web applications, because they are expected to run on a broad range of devices, from the powerful desktop personal computer to the less powerful smart phone [5,6].

Using the same programming language on both server-side and client-side could improve the software engineering process by enabling code reuse between both sides. Incidentally, the JavaScript language – which is currently the most supported action language on Web clients – can be used on server-side. Conversely, an increasing number of programming languages or compiler back-ends can generate JavaScript code (*e.g.* Java/GWT [7], SharpKit[1], Dart [8], Kotlin[2], ClojureScript [9], Fay[3], Haxe [10] or Opa[4]).

However, using the same programming language is not enough because the client and server programming environments are not the same. For instance, DOM fragments can be defined on client-side using the standard DOM API, but this API does not exist on server-side. How to define a common vocabulary for such concepts? And how to make the executable code leverage the native APIs, when possible, for performance reasons?

Generating efficient code for heterogeneous platforms is hard to achieve in an extensible way: the translation of common abstractions like collections into their native counterpart (JavaScript arrays on client-side and standard library's collections on server-side) may be hard-coded in the compiler, but that approach would not scale to handle all the abstractions a complete application may use (*e.g.* HTML fragment definitions, form validation rules, or even some business data type that may be represented differently).

On one hand, for engineering reasons, developers want to write Web applications using a single high-level language, abstracting over the target platforms differences and reducing code complexity. But on the other hand, for performance reasons, they want to keep control on the way their code is compiled to each target platform. We propose to solve this dilemma by providing high-level abstractions in compiled domain-specific embedded languages (DSELs) [11,12]. Compiled DSELs allow the definition of domain-specific languages (DSLs) as libraries on top of a host language, and to compile them to a target platform. Their deep embedding gives the opportunity to control the code generation scheme for a given abstraction and target platform.

Kossakowski *et al.* introduced *js-scala*, a compiled embedded DSL defined in Scala that generates JavaScript code, making it possible to write the client-

[1] http://sharpkit.net
[2] http://kotlin.jetbrains.org/
[3] http://fay-lang.org/
[4] http://opalang.org/

side code of Web applications using Scala [13]. However, the authors did not discuss any specific optimization and did not consider performance issues of their approach. Our paper shows how js-scala has been extended to support a set of specific optimizations allowing our high-level abstractions for Web programming to be efficiently compiled on both client and server sides[5].

We validate our approach with a case study implemented with various candidate technologies and discuss the relative pro and cons of them. We also measured the individual impact of each of our optimizations using micro-benchmarks. Though the code written in our DSL is high-level and can be shared between clients and servers, it has the same runtime performance on client-side as hand-tuned low-level JavaScript code.

The remainder of this paper is organized as follows. The next section overviews the existing approaches defining high-level languages for Web programming. Section 3 presents the framework we used to define our DSLs. Section 4 presents our contribution. Section 5 compares our solution to common approaches. Section 6 discusses our results and section 7 concludes.

2 Related Work

We classified existing approaches providing high-level abstractions for Web programming in four categories, as shown in Figure 1.

Fat Languages The first approach for defining a cross-platform language consists in hard-coding, in the compiler, the code generation scheme of each language feature to each target platform. Figure 1 (a) depicts this process. In order to support a feature related to a specific domain, the whole compiler pipeline (parser, code generator, *etc.*) may have to be adapted. This approach gives *fat* languages because a lot of concepts are defined at the language level: general programming concepts such as naming, functions, classes, as well as more domain-specific concepts such as HTML fragment definition. Thus, implementing a fat language may require a high effort and adding support for these languages in development environments may require a even higher effort. Examples of such languages for Web programming are Links [14], Opa, Dart [8].

Domain-Specific Languages Another approach consists in defining several independent domain-specific languages [15], each one focusing on concerns specific to a given problem domain, and then to combine all the source artifacts written with these language into one executable program, as shown in Figure 1 (b). Defining such languages requires a minimal effort compared to the previous approach because each language has a limited set of features. On the other hand, it is difficult to have interoperability between DSLs. [16] gave an example of such a domain-specific language for defining Web applications.

[5] The code is available at http://github.com/js-scala

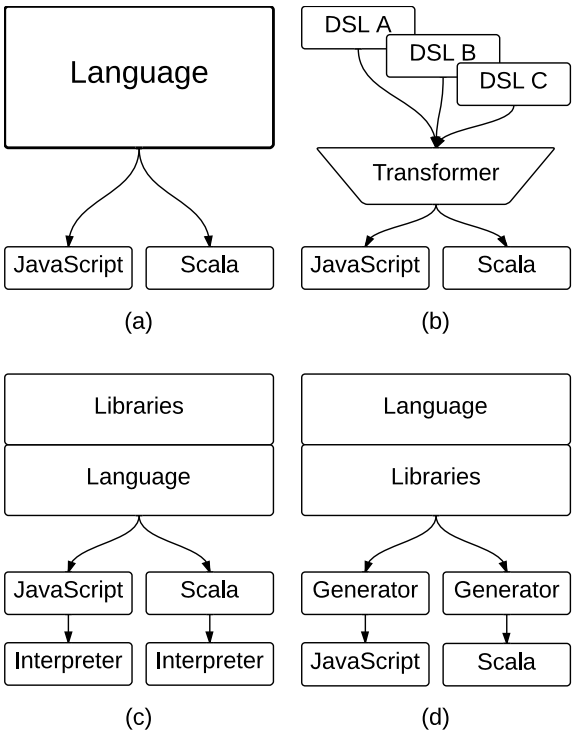


Fig. 1. Language engineering processes

Thin Languages Alternatively, one can define concepts relative to a specific domain as a library on top of a thin general purpose language (it is also referred to as a domain-specific *embedded* language [11]). Figure 1 (c) depicts this approach. Defining such a library requires minimal effort (though the syntax of the DSL is limited by the syntax flexibility of the host language) and several DSLs can interoperate freely within the host language. However, this approach gives no opportunity to efficiently translate a concept according to the target platform characteristics because the compiler has no domain-specific knowledge (though some compilers hard-code the translation of some common abstractions such as arrays to leverage the target platform characteristics). Examples of languages following this approach are Java/GWT, Kotlin, HaXe and SharpKit. Libraries written in JavaScript (*e.g.* jQuery [17]) also match this category though most of them do not support both client and server sides.

Deeply Embedded Languages The last approach, shown in Figure 1 (d), can be seen as a middle-ground between the two previous approaches: DSLs are embedded in a host language but use a code generation process. This approach shares the same benefits and limitations as embedded DSLs for defining language units. However, the code generation process is specific to each DSL and gives the opportunity to perform domain-specific optimizations. In other words deeply

embedded DSLs bring domain-specific knowledge to the compiler. Js-scala [13] is an example of deeply embedded DSL in Scala for Web programming. It makes it possible to produce JavaScript programs from Scala code that uses basic language concepts like arrays and control structures (`if` and `while`) as well as mechanisms specific to the Scala compiler like delimited continuations to handle asynchronous computations. Paper [13] presented the implementation of js-scala using staging, but did not discuss any specific optimization and did not consider performance issues of this approach. In this paper, we show how js-scala has been extended to support a set of specific optimizations allowing our high-level abstractions for Web programming to be efficiently compiled on heterogeneous platforms.

3 Lightweight Modular Staging

This section gives background material on the framework used to define js-scala.

Lightweight Modular Staging [18,19] (LMS) is a framework for defining deeply embedded DSLs in Scala. It has been used to define high-performance DSLs for parallel computing [20] and to define JavaScript as an embedded DSL in Scala [13].

LMS is based on staging [21]: a program using LMS is a regular Scala program that evaluates to an intermediate representation (IR) of a final program. This IR is a graph of expressions that can be traversed by code generators to produce the final program code. Expressions evaluated in the initial program and those evaluated in the final program (namely, staged expressions) are distinguished by their type: a `Rep[Int]` value in the initial program is a staged expression that generates code evaluating to an `Int` value in the final program. An `Int` computation in the initial program is evaluated during the initial program evaluation and becomes a constant in the final program.

Defining a DSL with LMS consists in the following steps:

- writing a Scala module providing the DSL vocabulary as an abstract API,
- implementing the API in terms of IR nodes,
- defining a code generator visiting IR nodes and generating the corresponding code.

3.1 LMS technical overview

In LMS, a DSL is split into two parts, its interface and its implementation. Both parts can be assembled from components in the form of Scala traits. DSL programs are written in terms of the DSL interface only, without knowledge of the implementation.

Part of each DSL interface is an abstract type constructor `Rep[_]` that is used to wrap types in the DSL programs. The DSL implementation provides a concrete instantiation of `Rep` as IR nodes. When the DSL program is staged, it produces an intermediate representation (IR), from which the final code can

be generated. In the DSL program, wrapped types such as `Rep[Int]` represent staged computations while expressions of plain unwrapped types (`Int`, `Bool`, etc.) are evaluated at staging time as in [?,?].

Consider the difference between these two programs:

```
def prog1(b: Bool, x: Rep[Int]) = if (b) x else x+1
def prog2(b: Rep[Bool], x: Rep[Int]) = if (b) x else x+1
```

The only difference in these two programs is the type of the parameter `b`, illustrating that staging is purely type-driven with no syntactic overhead as the body of the programs are identical.

In `prog1`, `b` is a simple boolean, so it must be provided at staging time, and the **if** is evaluated at staging time. For example, `prog1(true, x)` evaluates to `x`. In `prog2`, `b` is a staged value, representing a computation which yields a boolean. So `prog2(b, x)` evaluates to an IR node for the **if**: `If(b, x, Plus(x, Const(1)))`.

For `prog2`, notice that the **if** got transformed into an IR node. To achieve this, LMS uses Scala-Virtualized [?], a suite of minimal extensions to the regular Scala compiler, in which control structures such as **if** can be reified into method calls, so that alternative implementations can be provided. In our case, we provide an implementation of **if** that constructs an IR node instead of acting as a conditional. In addition, the + operation is overloaded to act on both staged and unstaged expressions. This is achieved by an implicit conversion from `Rep[Int]` to a class `IntOps`, which defines a + method that creates an IR node `Plus` when executed. Both of `Plus`'s arguments must be staged. We use an implicit conversion to stage constants when needed by creating a `Const` IR node.

3.2 Example: a DSL program and its generated JavaScript code

The following DSL snippet creates an array representing a table of multiplications:

```
def test(n: Rep[Int]): Rep[Array[Int]] =
  for (i <- range(0, n); j <- range(0, n)) yield i*j
```

Here is the JavaScript code generated for this snippet:

```
function test(x0) {
  var x6 = []
  for(var x1=0;x1<x0;x1++){
    var x4 = []
    for(var x2=0;x2<x0;x2++){
      var x3 = x1 * x2
      x4[x2]=x3
    }
    x6.splice.apply(x6, [x6.length,0].concat(x4))
  }
  return x6
}
```

The generated code resembles single-assignment form. The nested **for**-loop is desugared into a `flatMap` which generates the nested **for**-loop and the `splice`

pattern concatenating the inner x4 arrays into one x6 array in the JavaScript code.[6]

3.3 Walkthrough: defining a DSL component

To conclude the introduction to LMS, we show how to add a component for logging in a DSL, generating JavaScript code which calls `console.log`.

We start by defining the interface:

```
trait Debug extends Base {
  def log(msg: Rep[String]): Rep[Unit]
}
```

The Base trait is part of the core LMS framework and provides the abstract type constructor Rep.

Now, we define the implementation:

```
trait DebugExp extends Debug with EffectExp {
  case class Log(msg: Exp[String]) extends Def[Unit]
  def log(msg: Exp[String]): Exp[Unit] = reflectEffect(Log(msg))
}
```

The EffectExp trait is part of the core LMS framework. It inherits from BaseExp which instantiates Rep as Exp. Exp represents an IR via two subclasses: Const for constants and Sym for named values defining a Def. Def is the base class for all IR nodes. In our DebugExp trait, we extend Def to support a new IR node: Log.

IR nodes are defined as Defs but they are never referenced explicitly as such. Instead each Def has a corresponding symbol (an instance of Sym). IR nodes refer to each other using their symbols. This is why, in the code shown, the msg parameter is of type Exp (not Def). The method `log` returns an Exp. Calling `reflectEffect` is what creates this symbol from the Def.

In general, the framework provides an implicit conversion from Def to Exp, which performs common subexpression elimination by re-using the same symbol for identical definitions. We do not use the automatic conversion here, because `log` is a side-effecting operation, and we do not want to (re)move any such calls even if their message is the same.

The framework schedules the code generation from the graph of Exps and their dependencies through Defs. It chooses which Sym/Def pairs to emit and in which order. To implement code generation to JavaScript for our logging IR node, we simply override emitNode to handle Log:

```
trait JSGenDebug extends JSGenEffect {
  val IR: DebugExp
  import IR._
  override def emitNode(sym: Sym[Any], rhs: Def[Any])(
    implicit stream: PrintWriter) = rhs match {
      case Log(s) => emitValDef(sym, "console.log(" + quote(s) + ")")
      case _ => super.emitNode(sym, rhs)
```

[6] Obviously, the generated code can be optimized further.

```
    }
}
```

Notice that in order to compose nicely with other traits, the overridden method just handles the case it knows and delegates to other traits, via **super**, the emitting of nodes it doesn't know about.

4 Efficient High-Level Abstractions for Web Programming

This section presents some tasks typically performed in Web applications, either on client-side or server-side or on both, generalizes them in terms of high-level abstractions, and shows how they are implemented in js-scala to generate efficient code.

4.1 Selectors API

In a Web application, the user interface is defined by a HTML document that can be updated by the JavaScript code. A typical operation consists in searching some "interesting" element in the document, in order to extract its content, replace it or listen to user events triggered on it (such as mouse clicks). The standard API provides several functions to search elements in a HTML document according to their name or attribute values. Figure 2 summarizes the available functions and their differences.

Function	Description
`querySelector(s)`	First element matching the CSS selector s
`getElementById(i)`	Element which attribute id equals to i
`querySelectorAll(s)`	All elements matching the CSS selector s
`getElementsByTagName(n)`	All elements of type n
`getElementsByClassName(c)`	All elements which **class** attribute contains c

Fig. 2. Standard selectors API. The `querySelector` and `querySelectorAll` are the most general functions while the others handle special cases.

```
function getWords() {
  var form = document.getElementById('add-user');
  var sections =
    form.getElementsByTagName('fieldset');
  var results = [];
  for (var i = 0 ; i < sections.length ; i++) {
    var words = sections[i]
      .getElementsByClassName('word');
    results[i] = words;
  }
  return results
}
```

Listing 1.1. Searching elements using the native selectors API

Listing 1.1 gives an example of use of various functions from the native selectors API to retrieve a list of input fields within a form. The `getWords` function first finds in the document the HTML element with id `add-user`, then collects all its `fieldset` children elements, and for each one returns the list of its children elements having class `word`. The existence of several specialized functions in the API makes it possible to write efficient code, but forces users to think at a low abstraction level.

```
function getWords() {
  var form = $('#add-user');
  var sections = $('fieldset', form);
  return sections.map(function () {
    return $('.word', this)
  })
}
```

Listing 1.2. Searching elements using jQuery

A high-level abstraction for searching elements in a document could be just one function finding all elements matching a given CSS selector. In fact, most JavaScript developers[7] use the jQuery library that actually provides only one function to search for elements. Listing 1.2 shows an equivalent JavaScript program as Listing 1.1, but using jQuery. The code is both shorter and simpler, thanks to its higher level of abstraction. jQuery provides an API that is simpler to master because it has fewer functions, but this benefit comes at the price of a decrease in runtime performance.

Instead, we propose a solution that has a high-level API but generates JavaScript code using the specialized native API, when possible, in order to

[7] According to http://trends.builtwith.com/javascript, jQuery is used by more than 40% of the top million sites.

get both ease of engineering and performance. We achieve this by analyzing, during the first evaluation step, the selector that is passed as parameter and, when appropriate, by producing JavaScript code using the specialized API, and otherwise producing code using `querySelector` and `querySelectorAll`.

```
def find(selector: Rep[String]) =
  getConstIdCss(selector) match {
    case Some(id) if receiver == document =>
      DocumentGetElementById(Const(id))
    case _ =>
      SelectorFind(receiver, selector)
  }
```

Listing 1.3. Selectors optimization

Our API has two functions: `find` to find the first element matching a selector and `findAll` to find all the matching elements. Listing 1.3 gives the implementation of the `find` function. It is a Scala function that returns an IR node representing the JavaScript computation that will search the element in the final program. The `getConstIdCss` function analyzes the selector: if it is a constant `String` value containing a CSS ID selector, it returns the value of the identifier. So, if the `find` function is applied to the `document` and to an ID selector, it returns a `DocumentGetElementById` IR node (that is translated to a `document.getElementById` call by the code generator), otherwise it returns a `SelectorFind` IR node (that is translated to a `querySelector` call).

The same applies to the implementation of `findAll`: the selector passed as parameter is analyzed and the function returns a `SelectorGetElementsByClassName` in case of a CSS class name selector, a `SelectorGetElementsByTagName` in case of a CSS tag name selector, and a `SelectorFindAll` otherwise.

```
def getWords() = {
  val form = document.find("#add-user")
  val sections = form.findAll("fieldset")
  sections map (_.findAll(".word"))
}
```

Listing 1.4. Searching elements in js-scala

Figure 3 shows the IRs returned by the evaluation of *document .find("#add − userbutton")* and *document.find("#add − user")*.

In the former case, the selector is parsed and does not match an ID selector (it is a composite selector matching button elements within the element having

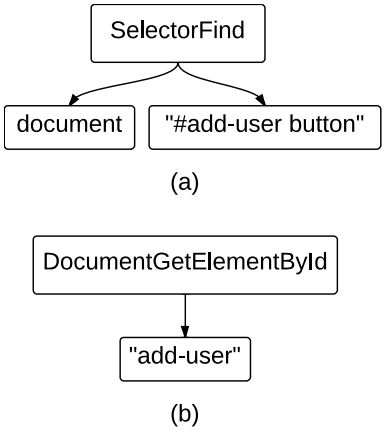


Fig. 3. Intermediate representations returned by the evaluation of (a) `document.find("#add-user button")` and (b) `document.find("#add-user")`

the add-user id), so a `SelectorFind` node is returned, then translated into a call to the general `querySelector` function. In the latter case, the selector matches an ID selector so a `DocumentGetElementById` node is returned, then translated into a call to the specialized `getElementById` function.

Finally, Listing 1.4 shows how to implement Listing 1.2 in Scala using js-scala. The code has the same abstraction level as with jQuery, however it generates a JavaScript program identical to Listing 1.1: the high-level abstractions (the `find` and `findAll` functions) exist only in the initial program, not in the final JavaScript program.

4.2 Monads Sequencing

This section presents an abstraction to handle **null** references and shows how this abstraction can be shared between client and server code.

null references are a known source of problems in programming languages [22,23]. For example, consider Listing 1.5 finding a particular widget in the page and then a particular button within the widget. The native `querySelector` method returns **null** if no node matched the given selector in the document. If we run this code in a page where the widget is not present, it will throw an error and stop further JavaScript execution. Defensive code can be written to handle **null** references, but leads to very cumbersome code, as shown in Listing 1.6.[8]

[8] However, one could alleviate the syntax burden by using a language such as CoffeeScript [24], that suppports a special notation for optional values dereferencing and desugars directly to JavaScript.

```
var loginWidget =
  document.querySelector("div.login");
var loginButton =
  loginWidget.querySelector("button.submit");
loginButton.addEventListener("click", handler);
```

Listing 1.5. Unsafe code

```
var loginWidget =
  document.querySelector("div.login");
if (loginWidget !== null) {
  var loginButton =
    loginWidget.querySelector("button.submit");
  if (loginButton !== null) {
    loginButton.
      addEventListener("click", handler);
  }
}
```

Listing 1.6. Defensive programming to handle null references

Some programming languages encode optional values with a monad (*e.g.* `Maybe` in Haskell and `Option` in Scala). In that case, sequencing over the monad encodes optional value dereferencing. If the language supports a convenient syntax for monad sequencing, it brings a convenient syntax for optional value dereferencing, alleviating developers from the burden of defensive programming.

In our DSL, we encode an optional value of type `Rep[A]` using a `Rep[Option[A]]` value, which can either be a `Rep[Some[A]]` (if there is a value) or a `Rep[None.type]` (if there is no value). An optional value can be dereferenced using the **for** notation, as shown in Listing 1.7, that implements in js-scala a program equivalent to Listing 1.6. The `find` function returns a `Rep[Option[Element]]`. The **for** expression contains a sequence of statements that are executed in order, as long as the previous statement returned a `Rep[Some[Element]]` value.

```
for {
  loginWidget <- document.find("div.login")
  loginButton <- loginWidget.find("submit.button")
} loginButton.on(Click)(handler)
```

Listing 1.7. Handling null references in js-scala

Such a monadic API brings both safety and expressiveness to developers manipulating optional values but usually involves the creation of an extra container object holding the optional value. In our case, the monadic API is used in the

initial program but generates code that does not wrap values in container objects but instead checks if they are **null** or not when dereferenced. So the extra container object exists only in the initial program and is removed during code generation: Listing 1.7 produces a code equivalent to Listing 1.6.

```
override def emitNode(sym: Sym[Any], rhs: Def[Any]) =
  rhs match {
    case OptionIsEmpty(o) =>
      emitValDef(sym, q" $o === null")
    case OptionForeach(o, b) =>
      stream.println(q"if ($o !== null) {")
      emitBlock(b)
      stream.println("}")
    case _ =>
      super.emitNode(sym, rhs)
  }
```

Listing 1.8. JavaScript code generator for null references handling DSL

Listing 1.8 shows the JavaScript code generator for methods isEmpty (that checks if the optional value contains a value) and `foreach` (that is called when the **for** notation is used, as in Listing 1.7). The `emitNode` method handles `OptionIsEmpty` and `OptionForeach` nodes returned by the implementations of isEmpty and `foreach`, respectively. In the case of the `OptionIsEmpty` node, it simply generates an expression testing if the value is **null**. In the case of the `OptionForeach` node, it wraps the code block dereferencing the value within a **if** checking that the value is not **null**.

The IR nodes are not tied to the JavaScript code generator, so we are able to make this abstraction available on server-side by writing a code generator similar to the JavaScript code generator, but targeting Scala. So the same abstraction is efficiently translated on both server and client sides.

4.3 DOM Fragments Definition

This section shows how we define an abstraction shared between clients and servers, as in the previous section, but that has different native counterparts on client and server sides. The challenge is to define an API providing a common vocabulary that generates code using the target platform native APIs.

A common task in Web applications consists in computing HTML fragments representing a part of the page content. This task can be performed either from the server-side (to initially respond to a request) or from the client-side (to update the current page). As an example, Listing 1.9 defines a JavaScript function `articleUi` that builds a DOM tree containing an article description. Listing 1.10 shows how one could implement a similar function on server-side using the standard Scala XML library. The reader may notice that the client-side and

```
var articleUi = function (article) {
  var div = document.createElement('div');
  div.setAttribute('class', 'article');
  var span = document.createElement('span');
  var name =
    document.createTextNode(article.name + ': ');
  span.appendChild(name);
  div.appendChild(span);
  var strong = document.createElement('strong');
  var price = document.createTextNode(article.price);
  strong.appendChild(price);
  div.appendChild(strong);
  return div
};
```

Listing 1.9. JavaScript DOM creation native API

```
def articleUi(article: Article) =
  <div class="article">
    <span>{ article.name + ": " }</span>
    <strong>{ article.price }</strong>
  </div>
```

Listing 1.10. Scala XML API

server-side APIs are very different and that the client-side native API is very low-level and inconvenient to use. We could use a library on client-side to get a higher level API for DOM fragment creation, but that would decrease the run-time performance. Instead, we want to define a high-level API that compiles to code as efficient as if it was written using the native APIs on both platforms.

Our first step consists in capturing, in a high-level API, the concepts common to the JavaScript and Scala APIs. Though they are different, both APIs define HTML elements with attributes and content. We propose to have a function `el` to define an HTML element, eventually containing attributes and children elements. Any children of an element that is not an element itself is converted into a text node. Listing 1.11 shows how to implement our example with our DSL. The children elements of an element can also be obtained dynamically from a collection, as shown in Listing 1.12.

```
def articleUi(article: Rep[Article]) =
    el('div, 'class -> 'article)(
        el('span)(article.name + ": "),
        el('strong)(article.price)
    )
```

Listing 1.11. DOM definition DSL

```
def articlesUi(articles: Rep[Seq[Article]]) =
    el('ul)(
        for (article <- articles)
        yield el('li)(articleUi(article))
    )
```

Listing 1.12. Using loops

The el function returns an Element IR node that is a tree composed of other Element and Text nodes. The JavaScript and Scala code generators traverse this tree and produce code building an equivalent DOM tree and XML fragment, respectively. When the children of an element are constant values (as in Listing 1.11) rather than dynamically computed (as in Listing 1.12), the code generators unroll the loop that adds children to their parent, for better performance. As a result, Listing 1.11 generates a code equivalent to Listing 1.9 on client-side and equivalent to Listing 1.10 on server-side.

```
case Tag(name, children, attrs) =>
  emitValDef(sym, q"document.createElement('$name')")
  for ((n, v) <- attrs) {
    stream.println(q"$sym.setAttribute('$n', $v);")
  }
  children match {
    case Left(children) =>
      for (child <- children) {
        stream.println(q"$sym.appendChild($child);")
      }
    case Right(children) =>
      val x = fresh[Int]
      stream.println(q"for (var $x = 0; $x < $children.length; $x++) {")
      stream.println(q"$sym.appendChild($children[$x]);")
      stream.println("}")
  }
case Text(content) =>
  emitValDef(sym, q"document.createTextNode($content)")
```

Listing 1.13. JavaScript code generator for the DOM fragment definition DSL

Listings 1.13 and 1.14 show the relevant parts of the code generators for this DSL. They basically follow the same pattern: they visit Tag and Text IR nodes and produce the corresponding elements in the target language.

5 Evaluation

Our goal is to evaluate the level of abstraction provided by our solution and its performance, by comparing it with common approaches. We take the number

```
case Tag(name, children, attrs) =>
  val attrsFormatted =
    (for ((name, value) <- attrs)
     yield q" $name={ $value }").mkString
  children match {
    case Left(children) =>
      if (children.isEmpty) {
        emitValDef(sym, q"<$name$attrsFormatted />")
      } else {
        emitValDef(sym,
          q"<name$attrsFormatted>{ ${children.map(quote)} }</$name>"
        )
      }
    case Right(children) =>
      emitValDef(sym, q"<$name$attrsFormatted>{ $children }</$name>")
  }
case Text(content) =>
  emitValDef(sym, q"{xml.Text(content)}")
```

Listing 1.14. Scala code generator for the DOM fragment definition DSL

of lines of code as an inverse approximation of the level of abstraction. We also evaluate the ability to share code between client and server sides.

We realized two micro-benchmarks involving programs using the selectors DSL and the optional value DSL, and we benchmarked a real world program. In each case we have written several implementations of the program, using plain JavaScript, Java/GWT, HaXe and js-scala (in each case we tried to write the application in an idiomatic way). The performance benchmarks measured the execution time of the generated JavaScript code. The tests were executed on a DELL Latitude E6430 laptop with 8 GB of RAM, on the Google Chrome v27 Web browser.

All our charts show three kinds of measures: the first group is the speed execution in operations per second (higher is better), the second group is the number of lines of code (lower is better) and the last group is the execution speed to number of lines of code ratio (higher is better). We normalized the values so the three groups can be shown within a same chart without scale issue.

5.1 Micro-Benchmarks

The micro-benchmarks measure the performance of our implementation of the selectors and optional value abstractions[9].

Selectors We could not implement this abstraction in GWT or HaXe as efficiently as we did in js-scala because it relies on the staging mechanism:

[9] The source code of the benchmarks is available at https://github.com/js-scala/js-scala/tree/master/papers/gpce2013/benchmarks

the best we could do in GWT or Haxe is to expose the native high-level API (`querySelector` and `querySelectorAll`). So we directly compared the execution time of the JavaScript code generated by Listing 1.4 with a JavaScript program equivalent to Listing 1.1 but using the high-level native API (`querySelector` and `querySelectorAll`) instead. The code was executed in a Web page containing a few elements: 4 `fieldset` elements, each containing 0 to 2 elements with class `word`.

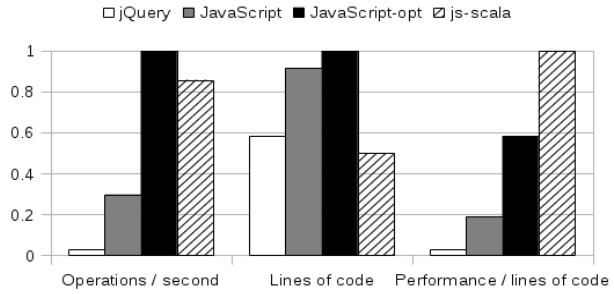


Fig. 4. Micro-benchmark on the selectors abstraction

Figure 4 shows the benchmark results. The JavaScript-opt version is Listing 1.1, which uses low-level native APIs, the JavaScript version is the equivalent listing using the high-level native API, and the jQuery version is Listing 1.2. The js-scala version is slightly slower than the JavaScript-opt (by 14%), but is 2.88 times faster than the JavaScript version, and 28.6 times faster than the jQuery version. Finally, the js-scala version has a performance to lines of code ratio more than 1.72 times higher than others.

Optional Value We reimplemented the optional value abstraction in plain JavaScript, Java and HaXe and wrote a small program manipulating optional values. Listing 1.15 shows the js-scala version of this program. The `maybe` function is a function partially defined on `Int` values.

Figure 5 shows the benchmark results. The js-scala version of the program runs between 3 to 10 times faster than other approaches. This version also takes less lines of code than others (this result is almost due to the special **for** notation, that has no equivalent in other benchmarked languages). Finally, the js-scala program has a performance to lines of code ratio more than 4 times higher than others.

```
val maybe = fun { (x: Rep[Int]) =>
  some(x + 1)
}

def benchmark = for {
  a <- maybe(0)
  b <- maybe(a)
  c <- maybe(b)
  d <- maybe(c)
} yield d
```

Listing 1.15. Micro-benchmark code for the optional values abstraction

Fig. 5. Micro-benchmark on the optional values abstraction

5.2 Real World Application

Chooze [10] is an existing complete application for making polls. It allows users to create a poll, define the choice alternatives, share the poll, vote and look at the results. It contains JavaScript code to handle the dynamic behavior of the application: double-posting prevention, dynamic form update and rich interaction with the document. The size of the whole application (server and client sides) is about one thousand lines of code.

The application was initially written using jQuery. We rewrote it in vanilla JavaScript (low-level hand-tuned code without third-party library), js-scala, GWT and HaXe.

Performance The benchmark code simulates user actions on a Web page (2000 clicks on buttons, triggering a dynamic update of the page and involving the use of the optional value monad, the selectors API and the HTML fragment definition API). Figure 6 shows the benchmark results.

The runtime performance of the vanilla JavaScript, HaXe and js-scala versions are similar (though the js-scala version is slightly slower by 6%). It is worth

[10] Source code is available at http://github.com/julienrf/chooze, under the branches `vanilla`, `jquery`, `gwt`, `haxe` and `js-scala`

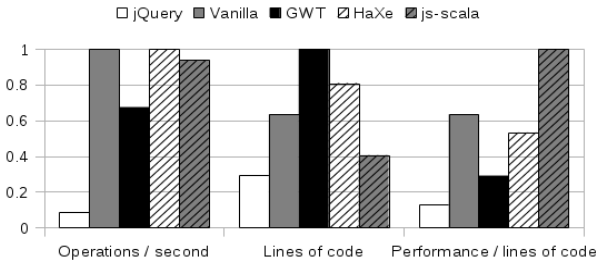


Fig. 6. Benchmarks on a real application

noting that the vanilla JavaScript and the HaXe versions use low-level code compared to js-scala, as shown in the middle of the figure (lines of code): the js-scala version needs only 74 lines of code while the vanilla JavaScript version needs 116 lines of code (57% bigger) and the HaXe version needs 148 lines of code (100% bigger). The jQuery JavaScript version, which code is high-level (54 lines of code, 27% less than js-scala) runs 10 times slower than the js-scala version.

The last part of the figure compares the runtime performance to lines of code ratio. Js-scala shows the best score, being 1.48 times better than the vanilla JavaScript version, 1.88 times better than the HaXe version, 3.45 times better than the GWT version and 7.82 times better than the jQuery JavaScript version.

Code Reuse We were able to share some DOM fragment definitions between server-side and client-side only in the js-scala version. In the GWT version we don't have a choice: dynamic DOM fragments are always built only on client-side (a practice that makes it more difficult to make the pages content crawlable by search engines and may increase the initial display time). In the other versions the code for building the DOM fragment is duplicated between client and server sides, representing 20 lines of JavaScript code (17% of the total) and 15 lines of HTML (5% of the total) in the JavaScript version, and 19 lines of HaXe code (13% of the total) and 15 lines of HTML (5% of the total) in the HaXe version. In the js-scala version the DOM fragment definitions shared between clients and servers represent 22 lines of Scala code (30% of the total) and save 15 lines of HTML (5% of the total).

Threats to Validity Our goal was to put the runtime performance in perspective with the level of abstraction. We are aware that the indicator we chose as an inverse approximation of the abstraction level, the number of lines of code, is not scientifically established and may be subject to discussion. However, we think it is a reasonable approximation in our case because all the candidate languages we use have a similar syntax, inherited from the C programming language.

Another weakness of our validation may come from the fact that our application does not make a heavy use of client-side code and thus may not be representative of the way large Web applications are written. However, we think

that a richer application would have more parts of code susceptible to be shared between client and server sides, thus giving even better results on the code reuse statistics.

Finally, the GWT version may not have been written in an as idiomatic as possible way. Indeed, we mainly catch the events directly on the HTML DOM, as we do in JavaScript, without reusing all the GWT widgets. We do not build the application as a blank page with a set of widgets. However, this way of developing using GWT has no impact on the performance and a minor impact on the number of lines of code.

6 Discussion

We implemented our solution as compiled embedded DSLs in Scala. Generating code from our DSLs is a two step process: an initial Scala program first evaluates to an intermediate representation of the final program that is traversed by code generators to produce the final JavaScript code. Domain-specific optimizations can happen during the IR construction (as shown in section 4.1) or during the code generation (as shown in section 4.2).

An important consequence of the implementation as compiled embedded DSLs is that defining a DSL that can be shared between server and client sides requires a low effort: compiled embedded DSLs are simply defined as libraries but let developers specialize the generated code according to each target platform (as shown in section 4.3).

In other words, the compiled embedded DSL approach gives us a way to exploit the Scala host language to define high-level language units that integrate seamlessly together and bring domain-specific knowledge to the code generation scheme to produce efficient code for client and server sides.

These characteristics allowed us to capture some Web programming patterns as high-level abstractions, making the code of our application simpler to reason about and making some parts of the code reusable between client and server sides, while keeping execution performance on client-side as high as if we used hand-tuned low-level JavaScript code.

7 Conclusion

High-level abstractions for Web programming, which are useful to decrease the complexity of the code and to abstract over the differences between the client and server environments, must be implemented in a way to efficiently run on hardware with limited capabilities.

In this paper we showed how to leverage staging to implement high-level abstractions for Web programming that are efficiently compiled for heterogeneous platforms such as Web clients and servers that differ in their technical API. We also showed how these abstractions can be shared between client and server sides.

Our two kinds of benchmarks, (i) micro-benchmarks to evaluate one abstraction and (ii) a benchmark on a real application using these abstractions, show performance similar to hand-optimized low-level code.

In a future work we may investigate more coarse-grained optimizations like smart DOM updates minimizing the number of browser reflows.

References

1. Mikkonen, T., Taivalsaari, A.: Web applications - spaghetti code for the 21st century. In: Proceedings of the 2008 Sixth International Conference on Software Engineering Research, Management and Applications, Washington, DC, USA, IEEE Computer Society (2008) 319–328
2. Preciado, J.C., Trigueros, M.L., Sánchez-Figueroa, F., Comai, S.: Necessity of methodologies to model rich internet applications. In: WSE, IEEE Computer Society (2005) 7–13
3. Rodríguez-Echeverría, R.: Ria: more than a nice face. In: Proceedings of the Doctolral Consortium of the International Conference on Web Engineering. Volume 484., CEUR-WS.org (2009)
4. Kuuskeri, J., Mikkonen, T.: Partitioning web applications between the server and the client. In: Proceedings of the 2009 ACM symposium on Applied Computing. SAC '09, New York, NY, USA, ACM (2009) 647–652
5. Souders, S.: High-performance web sites. Communications of the ACM **51**(12) (2008) 36–41
6. Huang, J., Xu, Q., Tiwana, B., Mao, Z.M., Zhang, M., Bahl, P.: Anatomizing application performance differences on smartphones. In: Proceedings of the 8th international conference on Mobile systems, applications, and services, ACM (2010) 165–178
7. Chaganti, P.: Google Web Toolkit: GWT Java Ajax Programming. Packt Pub Limited (2007)
8. Griffith, R.: The dart programming language for non-programmers-overview. (2011)
9. McGranaghan, M.: Clojurescript: Functional programming for javascript platforms. Internet Computing, IEEE **15**(6) (2011) 97–102
10. Cannasse, N.: Using haxe. The Essential Guide to Open Source Flash Development (2008) 227–244
11. Hudak, P.: Building domain-specific embedded languages. ACM Computing Surveys **28** (1996)
12. Elliott, C., Finne, S., De Moor, O.: Compiling embedded languages. Journal of Functional Programming **13**(3) (2003) 455–481
13. Kossakowski, G., Amin, N., Rompf, T., Odersky, M.: JavaScript as an Embedded DSL. In Noble, J., ed.: ECOOP 2012 – Object-Oriented Programming. Volume 7313 of Lecture Notes in Computer Science., Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 409–434
14. Cooper, E., Lindley, S., Wadler, P., Yallop, J.: Links: Web programming without tiers. In: Formal Methods for Components and Objects, Springer (2007) 266–296
15. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: an annotated bibliography. ACM Sigplan Notices **35**(6) (2000) 26–36

16. Visser, E.: WebDSL: A case study in domain-specific language engineering. In Lämmel, R., Visser, J., Saraiva, J., eds.: Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007. Volume 5235 of Lecture Notes in Computer Science., Braga, Portugal, Springer (2007) 291–373
17. Bibeault, B., Kats, Y.: jQuery in Action. Dreamtech Press (2008)
18. Rompf, T.: Lightweight Modular Staging and Embedded Compilers: Abstraction without Regret for High-Level High-Performance Programming. PhD thesis, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE (2012)
19. Rompf, T., Sujeeth, A., Amin, N., Brown, K., Jovanovic, V., Lee, H., Jonnalagedda, M., Olukotun, K., Odersky, M.: Optimizing Data Structures in High-Level Programs: New Directions for Extensible Compilers based on Staging. Technical report (2012)
20. Brown, K.J., Sujeeth, A.K., Lee, H.J., Rompf, T., Chafi, H., Odersky, M., Olukotun, K.: A heterogeneous parallel framework for domain-specific languages. In: Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on, IEEE (2011) 89–100
21. Jørring, U., Scherlis, W.L.: Compilers and staging transformations. In: Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, ACM (1986) 86–96
22. Hoare, T.: Null references: The billion dollar mistake. Presentation at QCon London (2009)
23. Nanda, M., Sinha, S.: Accurate interprocedural null-dereference analysis for java. In: Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, IEEE (2009) 133–143
24. Ashkenas, J.: Coffeescript (2011)

ACTRESS: Domain-Specific Modeling of Self-Adaptive Software Architectures

Filip Křikava
University Lille 1 / LIFL
Inria Lille - Nord Europe, France
filip.krikava@inria.fr

Philippe Collet
Université Nice
Sophia Antipolis, France
I3S - CNRS UMR 7271
philippe.collet@unice.fr

Robert B. France
Colorado State University
Fort Collins, USA
CS Department
france@cs.colostate.edu

ABSTRACT

A common approach for engineering self-adaptive software systems is to use Feedback Control Loops (FCLs). Advances have led to more explicit and safer design of some control architectures, however, there is a need for more integrated and systematic approaches that support end-to-end integration of FCLs into software systems.

In this paper, we propose a tooled approach that enables researchers and engineers to design and integrate adaptation mechanisms into software systems through FCLs. It consists of a domain-specific modeling language that raises the level of abstraction on which FCLs are defined, making them amenable to automated analysis and implementation code synthesis. The language supports composition, distribution and reflection, thereby enabling coordination and composition of multiple distributed FCLs. Its use is facilitated by a modeling environment, ACTRESS, that provides support for modeling, verification and complete code generation. We report on its application to a concrete adaptation case study and also discuss resulting properties.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

Keywords

self-adaptive software systems; model-driven engineering; domain-specific modeling; domain-specific languages

1. INTRODUCTION

The growing complexity and operational costs of contemporary software systems points to an inevitable need for making them autonomously adaptable at runtime [9]. A common approach for engineering such *self-adaptive software systems* is to use *Feedback Control Loops* (FCLs) [7]. Using measurements of a system outputs (*e.g.*, response times, utilizations), a FCL adjusts the system control inputs (*e.g.*, scheduling, concurrency policies) to achieve some externally specified goals [19]. Realizing FCLs in software systems is challenging [7, 9]. It requires addressing issues related to enabling adaptation in target systems, *i.e.* providing all necessary interfaces that expose the target system state and management operations (touchpoints), designing an adaptation engine, *i.e.* a control model that drives the adaptation itself, and finally forming the architecture integrating the two together [33].

There are a number of approaches that address some of these challenges. They aim at reducing the implementation effort and provide a solid foundation for engineering of self-adaptive software systems (*cf.* surveys in Salehie and Tahvildari [33] or Villegas *et al.* [35]). However, they often target specific types of adaptation problems and require the use of certain adaptation mechanism (*e.g.* utility theory in Rainbow [16]) or are applicable to a single domain (*e.g.* mobile applications in MUSIC [30]) or technology (*e.g.* Java-based systems in StarMX [3]), thereby limiting their applicability with respect to the problem being addressed [29]. Furthermore, while there have been advances in mechanisms enabling self-adaptation and control, less effort has been put into providing a systematic approach facilitating the integration of these mechanisms from an end-to-end system perspective. Often, the integration is done manually requiring extensive handcrafting of non-trivial code, which gives rise to significant accidental complexities, particularly in the case of distributed systems or complex control schemes.

In this paper, we propose a tooled approach that provides researchers and engineers with flexible abstractions of FCLs allowing them to more easily integrate self-adaptation mechanisms into software systems. It promotes separation of concerns whereby the development of system touchpoints, adaptation engine and the overall architecture can be decomposed and implemented by experts in the respective domains at different levels of abstraction. It is based on a technologically agnostic domain-specific modeling language called *Feedback Control Definition Language* (FCDL). It defines feedback architectures as hierarchically organized networks of adaptive elements, representing FCL processes such as monitoring, decision-making and reconfiguration. The language is statically typed, handles composition and supports element distribution via location transparency. Moreover it is reflective thereby enabling to coordinate and organize multiple control loops using different control schemes. The use of the language is facilitated by an Eclipse-based modeling environment called ACTRESS. It includes support for automated architecture consistency checking, and for code generation in which FCDL architectures are transformed into executable applications. This provides a strong mapping between the control system design and its runtime implementation.

The remainder of this paper is organized as follows. Section 2 presents a survey of related work. Section 3 introduces the adaptation scenario we use to illustrate our approach. Section 4 presents the domain-specific modeling language and is followed by an overview of the supporting tools in Section 5. Section 6 presents the evaluation and includes a discussion of the self-adaptive capabilities and quality attributes of the approach. Finally, Section 7 concludes the paper.

SAC'14 March 24–28, 2014, Gyeongju, Korea

2. RELATED WORK

A number of approaches have been proposed to facilitate engineering of self-adaptive software systems. In this section we focus on the techniques that are the most relevant with respect to our approach and on the work that has influenced our design decisions.

Frameworks. IBM proposed what has become a widely referenced model for autonomic systems, referred to as the MAPE-K decomposition [22]. A number of MAPE-K framework-based approaches have been developed focusing on different aspects of self-adaptation in software systems. Rainbow [16] consists of a two-layer framework with an external fixed control loop for architecture-based adaptation using utility theory. While the loop is made explicit, the framework was designed for scenarios that can be solved by centralized control loop and does not support hierarchical and distributed control schemes. StarMX [3] and ASF [17] are frameworks designed for building self-managing Java-based applications using closed FCLs. They use Java management extension for target system touchpoints and a policy-rule language for adaption engine implementation. Similarly to Rainbow, they do not support runtime modification of the management logic. Other approaches focus on component adaptations, *e.g.*, K-components [10] and CASA [25]. The former introduces a component model enabling individual components self-adaption using machine learning techniques. The latter supports dynamic application adaptation by recomposing Java components.

The advantage of a framework is that it provides an architectural basis of an application, defining its structure and control, and therefore it can simplify its development [15]. On the other hand, frameworks operate within boundaries of some programming language and therefore they are limited in the level of abstraction they can provide. The possibility of a formal reasoning and verification is also limited since the structure and behavior is an integral part of the implementation. Furthermore, they always impose the use of a certain technological stack.

Middlewares. Next to frameworks, an effort has been put into extending middlewares with self-adaptation capabilities. Adaptive CORBA Template [31] focuses on CORBA applications transparently weaving adaptive behavior into object request brokers at runtime. MADAM [12] and MUSIC [30] are examples of middleware infrastructures supporting development of self-adaptive mobile applications. The former exploits architecture models to enable runtime adaptation with utility functions to compare adaptation variability. The latter provides QoS-driven adaptation including dynamic service discovery, binding, negotiation and provisioning.

These approaches aim at shielding developers from complex tasks such as resource distribution, component probing, network communication or application reconfiguration [29]. However, middleware poses highly-specific execution environments which might not be directly applicable for some systems.

Model-based Approaches. Software models have been extensively used for various parts of self-adaptive software system development. Zhang and Cheng [38] introduced an approach to create formal models of adaptive programs behavior for analysis and implementation synthesis. Their approach separates specifications of adaptive and non-adaptive behavior thereby simplifying their use.

Using models as formal specifications of self-adaptive software systems has been also proposed, *e.g.*, FORMS [37] and DYNAMICO [34]. The former supports composition of adaptation mechanisms capturing their key characteristics to allow one to compare alternative solutions. The latter is based on a three-layer architecture defining three types of FCL, each managing different parts of context dynamics (control objectives, target system adaptation and dynamic monitoring).

There is also a large body of work that concerns designing feedback control for embedded computing, for example Ptolemy II [11]. Ptolemy II is an extensive framework for simulation of concurrent actor-oriented systems with the major emphasis on the ability to combine heterogeneous models of computation. We follow a similar actor-oriented approach and our execution semantics is comparable with Ptolemy *Push-Pull* model of computation (*cf.* Section 4.6). However, Ptolemy focus rather on simulation of the executable models and their transformations to the embedded systems.

Several approaches are exploiting the use of Model-Driven Engineering (MDE) techniques to develop particular classes of self-adaptive software. Genie [5] uses architectural models to support generation and execution of adaptive systems for component-based middlewares. The adaptive logic is specified as state machines, with each state being a system configuration and transitions being reconfiguration scripts. Diasuite [6] is a tool suite based on generative programming techniques for engineering *Sense-Compute-Control* (SCC) applications. An interesting feature of Diasuite is that the SCC architecture is enriched with a notion of interaction contract expressing the allowed interaction between its components, constraining the data and control flow. We use and extend this notion for our execution semantics (*cf.* Section 4.6).

A different model-based approach is based on the idea of using MDE techniques at runtime. The *model@run.time* represents an abstraction of a running system or its part and can be used to support dynamic adaptation of structure, behavior or goals of the underlying software systems [14]. For example, Vogel *et al.* [36] promotes the use of runtime executable megamodels. They present a modeling language for adaptation logic modeling together with a runtime interpreter that executes the megamodels. This is similar to what we develop in our approach, as they can also represent loop coordination and hierarchically organize them into layers. However, this solution is only a high-level overview of how the actual adaptations look like. They rely on an implicit synchronization between the megamodels and running system. Finally, their metamodel is based on EMF that has some limitations for the use at runtime such as higher memory footprint and lack of thread-safe access [13].

3. ADAPTATION SCENARIO

The adaptation scenario used throughout this paper is based on the work of Abdelzaher *et al.* [1] on QoS management control of web servers by content delivery adaptation. This work notably provides (1) a control theory-based solution to a well-known and well-scoped problem, and (2) enough details for its re-engineering. For our illustration, we only consider a single server case with all requests having the same priority.

The aim of the adaptation is to maintain web server load at a certain pre-set value preventing both its underutilization and its overload. The content of the web server is pre-processed and stored in M content trees where each one offers the same content but of a different quality and size (*e.g.* different image quality). For example let us take two trees `/full_content` and `/degraded_content`. At runtime, a given URL request, *e.g.* `photo.jpg`, is served from either `/full_content/photo.jpg` or `/degraded_content/photo.jpg` depending on the current load of the server. Since the resource utilization is proportional to the size of the content delivered, offering the content from the degraded tree helps reducing the server load when the server is under heavy load.

Figure 1 shows the block diagram of the proposed control. The *Load Monitor* is responsible for quantifying server utilization U. It periodically measures request rate R and delivered bandwidth W. These measurements are then translated into a single value, U. Since service time of a request constitutes of a fixed overhead and a data-size dependent overhead, using some algebraic manipulations, the utilization from the request rate and delivered bandwidth

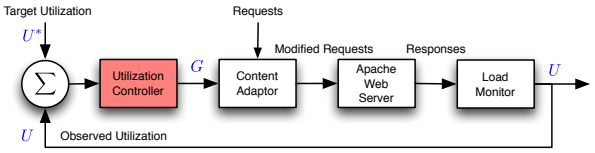


Figure 1: Block diagram of the adaptation scenario [1]

is derived as

$$U = aR + bW = a\frac{\sum r}{t} + b\frac{\sum w}{t} \quad (1)$$

where a and b are some platform constants derived by server profiling (details in Abdelzaher *et al.* [1]), $\sum r$ and $\sum w$ are respectively the number of request and the amount of bytes sent over some period of time t. The *Utilization Controller* is a *Proportional Integral* (PI) controller, which, based on the difference between the target utilization U^* (set by a system administrator) and the observed utilization U, computes an abstract parameter G representing the severity of the adaptation action. This value is used by the *Content Adaptor* to choose which content tree should be used for the URL rewriting. The achieved degradation spectrum ranges from $G = M$, servicing all requests using the highest quality content tree to $G = 0$ in which case all requests are rejected. It is computed as

$$G = G + kE = G + k(U^* - U) \quad (2)$$

where k is the controller tuning parameter that is determined *a priori* using some control analytic techniques (details in Abdelzaher *et al.* [1]). Shall $G < 0$ then $G = 0$ and similarly shall $G > M$ then $G = M$. If the server is overloaded $(U > U^*)$ the negative error will result in decrease of G which in turn changes the content tree decreasing the server utilization and vice versa.

4. MODELING FCL ARCHITECTURES

In this section, we present our approach for integrating the self-adaptive mechanisms into software systems through external FCLs.

4.1 Principles

Extracting from challenges identified in recent studies [9, 7, 33], we identify the following desirable properties for our solution:

- *Generality.* The approach should be both domain-agnostic and technology-agnostic, being applicable to a wide range of software systems and adaptation properties.
- *Visibility.* The FCLs, their processes and interactions should be made explicit at design time as well as at runtime, facilitating coordination of multiple control loops using different control schemes.
- *Tooling.* Provide tool support allowing developers to automate some recurring development tasks involving design, implementation and analysis of FCL. It should support traceability from the control design to the runtime implementation and should ensure a strong mapping between design and runtime control concepts. Together, these properties aim at increasing the overall understanding of the self-adaptive capabilities.

Furthermore, feedback control might cross boundaries of single system and thus the approach should support remote distribution of FCL. It should also follow good software engineering practices allowing modular specification, as well as composition and reuse of existing (parts of) FCLs across multiple scenarios. Finally, the approach should be efficient in terms of performance, having small execution overhead.

To meet these requirements we propose a domain-specific modeling language that is based on an actor-oriented design. The key advantage of using domain-specific modeling is in the possibility to raise the level of abstraction on which the FCLs are described, making them amenable to automated analysis and implementation code synthesis. Indeed it allows FCLs structure and behavior to be separated from its implementation since it is captured at a conceptual level using the problem domain concepts, rather than the implementation concepts as is the case in framework-based solutions. Since FCLs are inherently concurrent and concurrent programming is known to be difficult [24], we choose to use an actor-oriented design [20] for our model. The FCL processes are represented as message-passing actors that encapsulate their state and behavior. It allows one to implement these processes without worrying about thread safety, which greatly simplifies code [24]. The actor model is also scalable [18], supports distribution computation, and is easily applicable as there exist several high-performing actor libraries[1].

4.2 Feedback Control Definition Language

Our approach is based on a domain-specific modeling language called *Feedback Control Definition Language* (FCDL). It is grounded on an actor-oriented component meta-model representing abstractions of FCL architectures. The components are actor-like entities called *Adaptive Elements* (AE). An architecture is created by assembling and connecting AEs into hierarchically composed networks that form closed feedback control loops.

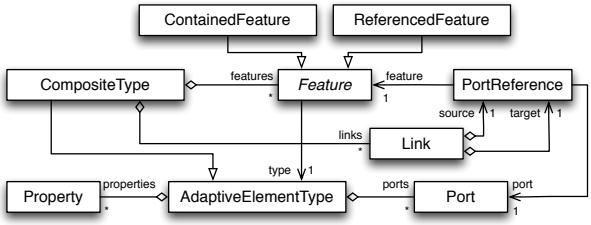


Figure 2: Excerpt of the FCDL abstract syntax

Figure 2 shown an excerpt[2] of the FCDL abstract syntax. AEs (`AdaptiveElementType`) have a well-defined interface that abstracts their internal state and behavior and restricts how they interact with their environments. It defines properties (`Property`) together with input and output ports (`Port`) that are the points of communications through which elements can exchange messages. The model supports both data-driven (push) and demand-driven (pull) communication. Once an AE receives a message, it activates and executes its associated behavior. The result of the execution may or may not be sent further to the connected downstream elements that in turn cause them to active and so forth. An AE can be passive or active. The former is activated by receiving a message while the latter attaches an appropriate event listener to activate itself when an event of interest occurs. Each AE represents a process of a FCL, which may either be: *a sensor* (collecting raw information about the state of the target system and its environment), *an effector* (carrying out changes on the target system using provided management operations), *a processor* (processing and analyzing incoming data both in the monitoring and reconfiguration parts), and *a controller* (special case of a passive processor that is directly responsible for the decision making). FCDL also allows to construct *composite* components (`CompositeType`) from both basic adaptive elements and from other composites. A composite is also the primary unit

[1] `http://bit.ly/1f41vHw`

[2] The complete abstract syntax is available at the companion website `http://fikovnik.github.io/Actress/DADS14.html`

of deployment. It defines both the instances of other components (`Feature`) they contain and the connections between the instances ports (`Link`). It can also define ports which are used to promote ports of the contained features.

To enforce data type compatibility, the FCDL modeling language uses static typing. For each port and property one has to explicitly declare the data type that restricts the data values it accepts. To improve reusability, the meta-model also supports parametric polymorphism, making adaptive elements work uniformly on a range of data types.

4.3 Illustration

Figure 3 shows one possible FCDL implementation of our adaptation scenario. It is derived from the block diagram depicted in Figure 1. The figure uses an informal FCDL graphical notation. Its purpose is to provide an intuitive and expressive visual representation of the model that can be easily sketched by hand. A formal textual syntax that supports all necessary properties is presented in Section 5.1.

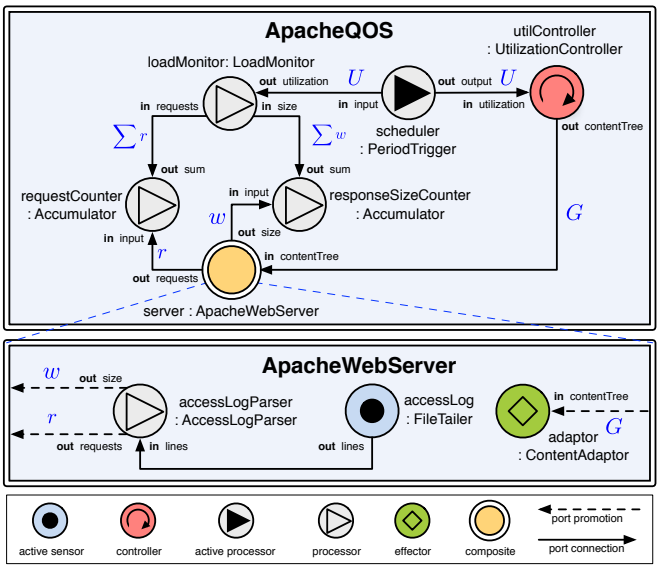


Figure 3: FCDL schema of the adaptation scenario

Decision-making. The PI controller (*Utilization Controller* from the block diagram) maps the current system utilization characteristics U into the abstract parameter G controlling which content tree should be used by the web server. In FCDL it is represented by the `UtilizationController` controller that has one push input port, `utilization`, for U and one push output port, `contentTree`, for G. Once a new utilization value is pushed to its input port, it computes G using (2) and pushes the result to the output port

Monitoring. The system utilization U depends on request rate R and bandwidth W. Both information can be obtained from Apache access log file. We create an active sensor, `FileTailer`, that activates every time a content of a file changes and sends the new lines over its push output port. It is connected to `AccessLogParser` that parses the incoming lines and computes the number of requests r and the size of the responses w, pushing the values to the corresponding `requests` and `size` ports. Consequently this increments the values of two connected counters `requestCounter` and `responseSizeCounter`, implemented as simple passive processors that accumulate the sum of all received values.

To compute utilization U, the sum of requests $\sum r$ and response size $\sum w$ has to be converted to request rate R and bandwidth W, *i.e.*, the number of request and sent bytes over certain time period t. One way of doing this is by adding a `PeriodicTrigger`, an active processor that every t milliseconds pulls data from its pull input port and in turn pushes the received value to its output port. Essentially, it is a scheduler that acts as a mediator between the two connected AEs. In this scenario, it is responsible for the timing of the FCL execution. By pulling data from its input port, it activates the `LoadMonitor` processor that (1) fetches the corresponding sums of requests $\sum r$ and response sizes $\sum w$ using the two pull input ports; (2) converts them to request rate R and bandwidth W; and (3) finally computes U using (1). The resulting utilization is then forwarded by the scheduler into the `UtilizationController`

Reconfiguration. Upon receiving the extent of adaptation G, the `ContentAdaptor` reconfigures the web server URL rewrite rules so that the newly computed content tree is used to serve the upcoming requests.

To demonstrate composition, the presented elements are assembled into two composites `ApacheQOS` and `ApacheWebServer`, representing respectively the control part and the target system touchpoints.

4.4 Reflection

Conceptually, each AE can be seen as a target system itself, and as such it can provide sensors and effectors enabling the AE to be introspected and modified. The *provided sensors* and *provided effectors* are essentially AEs touchpoints making them reflective and thereby enabling them to be adaptable. This is a crucial feature that permits one to hierarchically organize multiple feedback control loop in an uniform way and therefore realize complex control schemes from simple building blocks.

Figure 4 shows an example of an adaptive monitoring added into the adaptation scenario. Based on a periodically observed current system load using the `SystemLoad` sensor, the `PeriodController` modifies the execution timing of the `QOSControl` using the `setPeriod` effector. The `setPeriod` is a provided effector that adjusts the trigger rate t of the `PeriodicTrigger` inside the `QOSControl` composite.

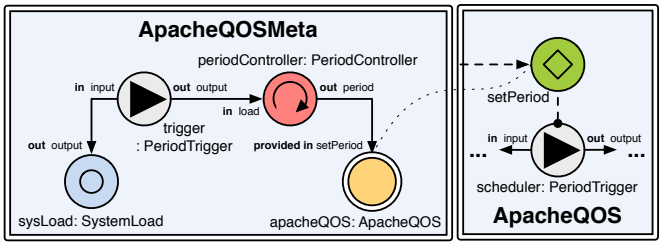


Figure 4: FCDL schema of the adaptation scenario with adaptive monitoring. *A provided sensor is visualized as an active sensor with one push output port, while provided effector is shown as a passive effector with one push input port. An additional dotted line indicates to which element do they belong.*

Technically, provided sensors and effectors are realized as AEs push output and push input ports respectively. However a crucial difference is that the messages sent from or to provided ports have a higher priority and thus will be processed before the regular messages. This is also reflected in the graphical notation (*cf.* Figure 4).

A structural adaptation, *i.e.*, changing loop composition and bindings is realized by sensors and effectors that operate on the actor model itself. These touchpoints include sensors observing adaptive

elements life-cycles (*e.g.* notifying when a new adaptive element is deployed), effectors deploying new elements or removing the existing ones and changing connections between them. By implementing the model reflection this way, we do not need any particular language support since these touchpoints are just regular AEs implemented using the underlying API. On the other hand, they have to be reimplemented for each targeted actor runtime support.

4.5 Distribution

Being based on the actor model, FCDL supports remoting using location transparency [2]. Remote elements are represented as first class entities using references. At the composite level, instead of declaring a new contained feature (`ContainedFeature`), one can declare a referenced feature (`ReferencedFeature`), formed by a reference to an existing feature in some composite, and a destination *endpoint* which is a URI of the remotely running AE. At runtime, during composite instantiation, for each referenced feature the system skips creating new AE and instead it only creates a reference that points to the given location. For example, in our adaptation scenario, we can deploy the `ApacheWebServer` composite on a different host than the `QOSControl`.

4.6 Execution Semantics

The execution semantics is based on the Ptolemy push-pull model of computation [39] coupled with an extended version of *Interaction Contracts* (IC) introduced by Cassou *et al.* [8]. The notion of IC is extended to support multiple-input, multiple-output elements, composites, optional contracts and architecture completion verification checking whether all required ports are connected [23].

The message communication originates in ports. A port can be configured in one of the three modes: *push*, *pull* or *agnostic*, in which case the exact mode is resolved during element instantiation according to the connected ports. The model is restricted to allow only same port-mode combination. Connecting a push output port to a pull input port indirectly implies using a queue and analogically connecting a pull output port to a push input requires to use a scheduler. In FCDL, this is intended to be explicitly modeled in the architecture in order to properly define the storage and the trigger mechanisms.

An active AE can cause its own activation from its associated event handler by sending a message to itself through an implicit self port. In FCDL, a message can only be sent by an AE. Therefore there always has to be at least one active element and for the model to be well-formed, each element has to become eventually active. An actor is eventually active if it is an active actor, or it has a pull output or push output port connected to an eventually activated element. The ordering of the activations is determined by the actor framework dispatcher.

An AE can execute different behaviors depending on what port or combination of ports caused its activation. For example, the `Accumulator` from the adaptation scenario either adds the pushed value from the `input` port or returns the accumulated sum when pulled over the `sum` port. To precise this, each non-composite AE specifies one or more basic IC that defines the element allowed interactions. It is a tuple $\langle A; R; E \rangle$ that indicates what interactions activates the AE (A), what additional data it might need to request through its pull input ports (R), and over which output ports it will push the results of its computation (E). For example the IC associated with `PeriodicTrigger` is $\langle self; \Downarrow(\texttt{input}); \Uparrow(\texttt{output?}) \rangle$. It denotes an interaction caused by *self* activation where `input` port might be pulled and conditionally data pushed to the `output` port. The IC for `Accumulator` is a composition of two basic interaction contracts $\langle \Uparrow(\texttt{input}); \emptyset; \emptyset \rangle \parallel \langle \Downarrow(\texttt{sum}); \emptyset; \emptyset \rangle$. Interaction contracts for composites are automatically inferred based on the IC of the contained AEs, *e.g.* `ApacheWebServer` has IC $\langle self; \emptyset; \Uparrow(\texttt{requests}, \texttt{size}) \rangle \parallel \langle \Uparrow(\texttt{contentTree}); \emptyset; \emptyset \rangle$.

The use of ICs brings following advantages. By using ICs we can assert certain architectural properties such as consistency, determinacy, and completeness. Different AE activations are clearly visible in its interface and therefore amenable to automatized analysis and verification. Furthermore, an IC denotes the type of the associate activation function. Therefore, it allows the generated code to be both *prescriptive* (guiding the developer) and *restrictive* (limiting the developer to what the architecture allows). For example, following is a Java code generated for the `PeriodicTrigger`:

```
public class PeriodicTrigger<T> extends AdaptiveElement {
  public void init();
  public void destroy();
  protected void activate(long self, Pull<T> input, Push<T> output);
  protected void onSetPeriod(Duration setPeriod);
}
```

Listing 1: Example of an AE class

The `Pull` and `Pull` interfaces denote the optional interaction for data requirements and data emission. The use of the generic parameter T is because `PeriodTrigger` is a polymorphic adaptive element capable of pulling and pushing any data type. The `init` and `destroy` methods are the AE life-cycle methods executed respectively during its initialization and termination.

5. ACTRESS

The aim of the ACTRESS modeling environment is to provide support for an integrated development of external self-adaptive software systems using FCDL. We do not focus on the control mechanisms themselves, since for this, there already exist sophisticated tools such as MATLAB [19].

In its core, ACTRESS consists of a series of model transformation and verification processes automatizing various aspects of FCDL development. This section gives a high-level overview of the main ACTRESS components. Additional details are available in a technical report [23].

5.1 Modeling Support

The ACTRESS modeling support provides a reference implementation of the FCDL meta-model and tools facilitating FCDL models authoring. The implementation is based on the EMF metamodeling technology. The heart of the modeling support is a domain-specific language called *Extended Feedback Control Definition Language* (XFCDL) for creating FCDL models. It is a textual DSL for creating FCDL models that further supports modularization and AE implementation using a Java-like expression language. XFCDL is built using Xtext[3], a software language engineering framework that covers many aspects of a language infrastructure including sophisticated Eclipse IDE integration. The language is close to Java and it uses some of its concepts such as modularization (packages and imports), type system and naming conventions.

The architecture consists in defining AE types that participate in the FCLs. The following code shows an example of how to create the `PeriodicTrigger` from the running scenario[4]:

```
1  active processor PeriodicTrigger<T> {
2    push in port output: T
3    pull in port input: T
4    self port selfport: long // self port for self-activation
5
6    provided effector setPeriod: Duration
7    property initialPeriod: Duration = 10.seconds
8
9    act activate(selfport; input; output?)
10   act onSetPeriod(setPeriod; ;)
11 }
```

[3] http://www.eclipse.org/Xtext/

[4] The complete code is available at the companion website http://fikovnik.github.io/Actress/DAIS14.html

Line 1 defines a new active polymorphic processor type with data type parameter T. Lines 2-4 declare ports including the implicit self port in order to specify its data type. The provided effector is defined on line 6, followed by a property definition on line 7. Finally, lines 9-10 defines ICs.

Next, in order to form a FCL, we need to connect the AEs together. This is done by creating a composite in which we define all the elements of the loop and specify the data-flow by connecting their ports. For example, following is an excerpt of the `ApacheQOS` definition from Figure 4:

```
1  composite ApacheQOS {
2    property targetUtilization: double // U*
3
4    feature scheduler = new PeriodicTrigger<Double> {
5      initialPeriod = 30.seconds
6    }
7    feature utilController = new UtilizationController {
8      targetUtilization = this.targetUtilization // ref composite property
9    }
10   // ...
11   connect scheduler.output to utilController.utilization
12   promote scheduler.setPeriod
13 }
```

It is similar to an AE definition, but further it includes definitions of contained AEs (lines 4 and 7) port connections (line 11) and promotions (line 12). On line 4 a concrete data type is specified for the data type parameter T. Lines 5 and 8 specify values for the AEs properties including property reference.

Instead of creating a new adaptive element, it is possible to reference a remotely running one. For example, the following code creates an AE reference of the `ApacheWebServer` composite that runs at `remote-host`[5]:

```
feature server = ref ApacheWebServer @
  "akka://actress@remote-host/user/ApacheWebServer"
```

Finally, xFCDL also allows to specify the implementation of AEs (their ICs) directly using Xbase[6], a statically typed Java-like expression language that supports lambda expressions, type inference and Java interoperability. For example, the `UtilizationController` implementation using the equation (2) can be expressed as:

```
1  controller UtilizationController {
2    in push port utilization: double // U
3    out push port contentTree: double // G
4    property targetUtilization: double // U*
5
6    act activate(utilization; ; contentTree)
7
8    implementation xbase {
9      var G = M // new variable
10     // implementation of the 'act activate(utilization; ; contentTree)'
11     act activate {
12        val E = targetUtilization - utilization // computes the error
13        G = G + k * E // computes new extend of adaptation
14        if (G < 0) G = 0; if (G > M) G = M // correct bounds
15        G // returns the result
16     }
17 }
```

Next to ICs implementation, the Xbase block can contain variable declarations, life-cycle method implementations and auxiliary methods. While Xbase provides a convenient way of specifying adaptive elements implementation directly in xFCDL, it might not always be the most suitable option and a developer can use Java instead. Moreover, Xbase support for lambda expressions allows to use functions types as properties, which results in higher-order AEs definitions.

[5]The URIs are implementation dependent. Currently, ACTRESS uses Akka as the underlying actor runtime (*cf.* Section 5.2).

[6]`http://www.eclipse.org/Xtext/documentation.html\#xbaseLanguageRef_Introduction`

5.2 Code Generation and Runtime Support

Through text-to-model and model-to-model transformations the code in xFCDL is translated into FCDL. From the FCDL model, the code generator synthesizes an executable application for a concrete runtime platform. Currently, ACTRESS supports Akka[7], a scalable and lightweight framework and a runtime for actor-based applications on the *Java Virtual Machine* (JVM). Because the FCDL model is already an actor-oriented model, the source code transformation is rather straightforward as it does not need to build any other intermediate representation. Essentially, each AE type is turned into a Java class like the one shown in Listing 1. Any Xbase implementation is compiled into corresponding Java methods in the generated class. These classes are used as delegates by underlying actor classes that translate the lower level actor interactions into life-cycle and interaction contracts method calls. Using this pattern, developers never have to deal with any lower-level actor API and only use the higher-level API provided by ACTRESS. This also simplifies AE testing which can be done in isolation without any actor runtime. Additionally, the code generator outputs application launchers for top level composites providing a convenient way to execute them.

5.3 Verification Support

The verification support automates consistency checking of FCDL structural invariants including user-defined ones, as well as connectivity and data reachability properties through the means of external verification. Invariants are used in the FCDL meta-model for asserting the model well-formedness. Additionally, developers can define their own set of invariants for FCDL model instances using either OCL [27] or Xbase. Usually, they are used to identify architecture bad smells such as adaptive element overlaps (*e.g.* an effector being orchestrated by multiple controllers).

Furthermore, the use of models and MDE techniques brings the possibility of external model verification. Concretely, ACTRESS provides a FCDL transformation into Promela model in order to verify connectivity and reachability properties using linear temporal logic and the SPIN model checker [21].

6. ASSESSMENT AND DISCUSSION

In this section we discuss the application, quality attributes and limitations of both FCDL and the ACTRESS modeling environment.

Adaptation Scenario. The adaptation scenario illustrates the systematic integration of real-world control mechanisms into a real-world software system. The implementation consists of 169 xFCDL, 67 Xbase and 97 Java source lines of code (SLOC). Java was used to implement the Apache touchpoints while Xbase was used for all the other AEs. Interpreting SLOC is always problematic, however we advocate that (1) the 97 SLOC of the touchpoints code would have to be implemented in one way or another; (2) the 169 of xFCDL and 67 Xbase SLOC integrates the adaptation engine with the target system, creating an executable system; Moreover, the implementation already includes AEs that could be likely used in other adaptation scenarios since they provide some rather generic functionality (*e.g.*, `PeriodicTrigger`, `Accumulator`, `FileTailer`). Additionally two complete adaptation case studies from high throughput computing domain are available in a companion report [23].

Properties. Following is a qualitative summary of FCDL and ACTRESS support of the desirable properties identified in Section 4.1.

– *Generality.* FCDL is a domain-agnostic model language for modeling architectures of FCLs. It uses concepts from control theory and its syntax is close to the block diagram one. Unlike

[7]`http://akka.io`

most frameworks [29], it does not dictate any particular system architecture. Since an FCL is decomposed into a number of explicit and interconnected adaptive elements, a number of *self-** adaptation properties are likely to be expressed. Furthermore, The reflection and distribution capabilities support the organization of FCLs into complex and distributed control schemes, such as hierarchical or decentralized controls [28].
FCDL is also a technologically-agnostic model. It focuses only on the FCL architectures, hiding the details not relevant to the design. ACTRESS is based on Java technologies, however, by no means it is limited to only adapt Java systems as shown in Section 3. FCDL can also target other runtime platforms. For example, the CORONA project [26] uses FCDL for *Service Component Architecture* systems adaptation, transforming AE elements into components for the FraSCAti runtime [32].

– *Visibility.* The FCDL language syntax is using concepts from control theory. It is based on an actor-oriented model with known concepts such as ports and composites. The FCL processes are represented as first-class reusable entities with explicit interactions that are precisely guided by interaction contracts. Relying on the actor model, the system is highly concurrent while allowing for simple AE implementation without the need to protect mutable state. Moreover, interaction contracts make the architecture both prescriptive and restrictive, and thus guide AE implementations.
Using FCDL developers work on a higher-level of abstraction using concepts from the self-adaptive system domain. Without a domain-specific modeling language like FCDL, developers are likely to use GPLs that do not convey domain-specific concerns and semantics [14]. It is important to note here that the abstraction we have chosen is not the only one and it is possible to have even higher-level models. FCDL matches block diagrams providing an established abstraction of FCLs which is flexible, yet rigid enough for automated code synthesis.
– *Tooling.* The usage of FCDL is facilitated by the ACTRESS modeling environment. Integrated in the Eclipse IDE, it provides modeling, code generation and verification support. The modeling support uses a textual DSL, XFCDL, that enables modularization and optional AE implementations using Xbase expressions. The code generator transforms FCDL architectures into executable Java applications, providing a strong mapping between the control system design and its runtime implementation. The verifier can automatically check assumptions about modeled architectures using structural and temporal constraints. Using Xbase for implementation, the code generator emits a complete executable applications, yet with customization and configuration opportunities. During the implementation of the case studies, we observed, that the automation of the development process helps developing the solution incrementally. It allowed to start with a basic control scheme and to refine it step-by-step into a more advanced one. At the end of each step ACTRESS generates the complete code that can be directly tested and executed. Finally, our approach supports separation of concerns in the sense that the system architecture and control mechanisms can be defined by control engineers while the implementation of the technical/system-level processors or touchpoints can be carried out by software engineers. Thanks to the Eclipse integration both tasks can be realized within the ACTRESS modeling environment, which should simplify and promote collaboration.

Performance. We consider the overhead caused by the execution of the self-adaptive layer. A single instance of the ACTRESS runtime with no composites deployed accounts for 1.5MB[8]. The ACTRESS domain framework is based on Akka. In Akka 2.0 version, the memory overhead is about 400 bytes per actor instance (2.7 million actors per GB of heap) with a possible throughput of 50 million messages per sec on a single machine[9]. The size of an adaptive element is mostly affected by the amount of state it keeps. The same applies for the execution time whose majority is spent in running the user-code of adaptive element activation methods (*e.g.* a sample push/pull communication with a throughput of 5000 messages per second amounts for 5% of CPU time). The main potential performance issues is in the indirect load caused by the sensors and effectors, which might become significant and as such it must be taken into account while designing any self-adaptive software system.

Limitations. While FCDL is technologically agnostic, XFCDL is tightly coupled with Java. This currently limits the implementation of AE to Java-based languages. This might pose a problem for scenarios where the touchpoints need to interact with an API that is not accessible from Java nor JNI. With the MDE approach, however, it is possible to target different runtime platforms that are themselves based on the actor model. The increasing popularity of the actor model gives us a variety of different frameworks available in various programming languages.

Besides the FCDL uses static typing, but does not support physical units and therefore there is nothing to prevent typing errors such as `speed = time / distance`.

Xbase provides a convenient way for expressing mathematical equations, but it might be too low level for control based on concepts such as decision tables, rule-based policies or state transition diagrams. Declarative policy-rule languages can be used through their respective API, however, as in the case of StarMX or ASF, they are not directly embedded in the adaptive element definition (*i.e.* in XFCDL). Furthermore, the external adaptation relies on the fact that the target system is able to provide, or be instrumented to provide, all the required touchpoints.

7. CONCLUSIONS

In this paper, we have proposed a domain-specific modeling language, FCDL, for integrating adaptation mechanisms into software systems through external FCL. It is centered around an actor-oriented model for defining FCL architectures using a hierarchically organized networks of AEs that explicitly represent different parts of the adaptation process as first-class entities. To facilitate the development using FCDL, a modeling environment called ACTRESS has been implemented. Integrated in the Eclipse IDE, it provides a reference implementation of FCDL together with dedicated support for modeling, verification and complete code generation. The approach has been illustrated on a real-world adaptation scenario on web server QoS management control.

Current work in progress mainly concerns carrying more case studies targeting different self-adaptive properties in order to identify the strengths as well as limitations of the approach. Several improvements are planned for the future such as support for deployment in distributed environments, dealing with issues related to loop coordination, failure propagation and extending data type system with physical units. Future work also include providing a native implementation of the ACTRESS runtime and experiment with DSL embedding to allow to specify AE implementations in a variety of languages.

Acknowledgments. The work reported in this paper is partly funded by the ANR SALTY project under contract ANR-09-SEGI-012.

8. REFERENCES

[1] T. Abdelzaher and N. Bhatti. Web Server QoS Management by Adaptive Content Delivery. In *International Workshop Quality of Service*, IWQoS, London, 1999.

[8] All further measurements were conducted on MacBook Pro 2.53 Ghz Intel i5, 8GB RAM, Java 1.70_17, Akka 2.2.0

[9] `http://bit.ly/1gHM975`

[2] J. L. Armstrong, B. O. Dacker, S. R. Virding, and M. C. Williams. Implementing a functional language for highly parallel real time applications. In *Software Engineering for Telecommunication Systems and Services*, 1992.

[3] R. Asadollahi, M. Salehie, and L. Tahvildari. StarMX: A framework for developing self-managing Java-based systems. In *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, 2009.

[4] O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, A. van Moorsel, and M. van Steen. The Self-Star Vision, volume 3460 of LNCS, 2005.

[5] N. Bencomo, P. Grace, C. Flores, D. Hughes, and G. Blair. Genie: supporting the model driven development of reflective, component-based adaptive systems. In *International conference on Software engineering*, 2008.

[6] B. Bertran, J. Bruneau, D. Cassou, N. Loriant, E. Balland, and C. Consel. DiaSuite: A tool suite to develop Sense/Compute/Control applications. *Science of Computer Programming*, pages 1–28, Apr. 2012.

[7] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Muller, M. Pezzè, and M. Shaw. Engineering Self-Adaptive Systems Through Feedback Loops. *Software Engineering for Self-Adaptive Systems*, pages 48–70, 2009.

[8] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of sense/compute/control applications. *Proceeding of the 33rd international conference on Software engineering*, 2011.

[9] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and Others. Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Software Engineering for Self-Adaptive Systems*, pages 1–26, 2009.

[10] J. Dowling and V. Cahill. Self-managed decentralised systems using K-components and collaborative reinforcement learning. *ACM SIGSOFT workshop on Self-managed systems*, 2004.

[11] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs. Taming heterogeneity - the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan. 2003.

[12] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, and E. Gjorven. Using architecture models for runtime adaptability. *IEEE Software*, 23(2):62–70, Mar. 2006.

[13] F. Fouquet, G. Nain, B. Morin, E. Daubert, O. Barais, N. Plouzeau, and J.-M. Jézéquel. An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements. In *MoDELS*, 2012.

[14] R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Future of Software Engineering*, 2007.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[16] D. Garlan, B. Schmerl, and P. Steenkiste. Rainbow: architecture-based self-adaptation with reusable infrastructure. In *International Conference on Autonomic Computing*, 2004.

[17] I. Gorton, Y. Liu, and N. Trivedi. An extensible, lightweight architecture for adaptive J2EE applications. *International Workshop on Software engineering and middleware*, 2006.

[18] P. Haller and M. Odersky. Scala Actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, Feb. 2009.

[19] J. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury. *Feedback control of computing systems*. Wiley Online Library, 2004.

[20] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323–364, June 1977.

[21] G. J. Holzmann. *Spin Model Checker*. Addison-Wesley Professional, 1. edition edition, 2003.

[22] J. Kephart and D. Chess. The Vision of Autonomic Computing. *Computer*, 36(1):41–50, Jan. 2003.

[23] F. Křikava and P. Collet. Feedback Control Definition Language. Technical report, I3S CNRS - UMR 7271, 2013, `https://fikovnik.github.io/Actress/FCDL.pdf`

[24] E. A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

[25] A. Mukhija and M. Glinz. Runtime adaptation of applications through dynamic recomposition of components. In *Int. Conf. on Architecture of Computing Systems*, 2005.

[26] R. Nzekwa. *Building Manageable Autonomic Control Loops for Large Scale Systems*. PhD thesis, Université des Sciences et Technologie de Lille - Lille I, 2013.

[27] Object Management Group. OMG Object Constraint Language (OCL). Technical report, 2012.

[28] T. Patikirikorala, A. Colman, J. Han, and L. Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Software Engineering for Adaptive and Self-Managing Systems*, 2012.

[29] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *Software Engineering for Adaptive and Self-Managing Systems*, 2010.

[30] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments. In *1st workshop on Mobile middleware*, 2008.

[31] S. Sadjadi and P. McKinley. ACT: an adaptive CORBA template to support unanticipated adaptation. In *Int. Conf. on Distributed Computing Systems*, 2004.

[32] L. Seinturier, P. Merle, D. Furnier, N. Dolet, V. Schiavoni, and J.-B. Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *International Conference on Service Computing*, 2009.

[33] M. Salehie and L. Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 4(2):1–42, 2009.

[34] N. Villegas, G. Tamura, H. Müller, L. Duchien, and R. Casallas. DYNAMICO : A Reference Model for Governing Control Objectives and Context Relevance in Self-Adaptive Software Systems. *Software Engineering for Self-adaptive Systems 2*, pages 265–293, 2013.

[35] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas. A framework for evaluating quality-driven self-adaptive software systems. In *Software Engineering for Adaptive and Self-Managing Systems*, 2011

[36] T. Vogel and H. Giese. A Language for Feedback Loops in Self-Adaptive Systems: Executable Runtime Megamodels. In *7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, 2012.

[37] D. Weyns and S. Malek. FORMS: a formal reference model for self-adaptation. In *Proceedings of the 2010 International Conference on Autonomic Computing*, 2010.

[38] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceeding of the 28th international conference on Software engineering*, 2006.

[39] Y. Zhao. A Model of Computation with Push and Pull Processing. Technical report, Technical Memorandum UCB/ERL M03/51, University of California, Berkeley, 2003.

Session du groupe de travail Compilation

Un langage synchrone fonctionnel avec des horloges entiéres

Auteur : A. Guatto Doctorant (ENS Ulm)

Résumé :
Dans cet exposé, je présenterai un nouveau langage synchrone fonctionnel dans la lignée de Lustre et un prototype de compilateur pour ce langage.

Comme en Lustre, les programmes sont des ensembles d'équations sur des flots infinis de valeurs ; comme en Lustre, ces flots sont indicés par une échelle de temps partagée entre plusieurs équations. En revanche, contrairement à Lustre, une échelle de temps peut être locale, et un flot peut contenir plusieurs valeurs par instants.

Je montrerai comment ces deux principes se combinent pour permettre de compiler, de maniére automatique ou semi-automatique, un même programme source vers des implémentations représentant des compromis temps-espace différents. En particulier, le code généré peut contenir des boucles imbriquées et des tableaux absents du source.

Les seuls problèmes intéressants sont les problèmes indécidables - application à la synthèse de preuve de terminaison de programmes

Auteur : L. Gonnord (LIP, U. Lyon1)

Résumé :
Le problème de la preuve de terminaison de programme est bien connu pour être, dans le cas général, un problème indécidable. Ce constat étant fait, on peut quand m ême essayer de ne pas jeter complétement le bébé avec l'eau du bain. Le choix qui est fait ici est d'essayer de traiter le cas général par des algorithmes conservatifs : si l'algorithme répond oui, alors le programme termine, dans le cas contraire, on ne peut conclure.

Pour le cas qui nous intéresse, nous commençons par "prouver" que le programme termine en exhibant une fonction qui décroît ? chaque transition et qui reste positive (fonction "de rang"). Pour ce faire, nous utilisons des algorithmes classiques en compilation et en analyse statique : compilation d'un programme vers un automate affine, calcul d'invariants polyédriques, et enfin nous adaptons un algorithme glouton de calcul d'un ordonnancement multidimensionnel pour calculer une fonction de ranking affine multidimensionnelle. J'exposerai ces techniques ainsi que l'algorithme final.

Une fois la fonction de rang exhibée, nous sommes en mesure de calculer un effet de bord sympathique : une borne supérieure de la complexité pire cas du programme.

J'exposerai ensuite nos résultats expérimentaux et les pistes que nous explorons actuellement pour améliorer le passage à l'échelle, ainsi que quelques applications à d'autres domaines que la "simple" preuve de terminaison.

Session du groupe de travail FORWAL

Formalismes et Outils pour la Vérification et la Validation

Synthèse d'Arbres d'Attaques pour une Analyse de Risques assistée par Ordinateur

Stéphanie Georges et Sophie Pinchinat

INRIA, IRISA, University of Rennes 1
35042 Rennes
France

1 Introduction

Assurer la sécurité d'un système d'information consiste à garantir la disponibilité, l'intégrité et la confidentialité de ses données. Pour atteindre cet objectif, une étude préliminaire, appelée *analyse de risques*, du système et de son environnement est nécessaire [2] pour identifier et évaluer les risques qui pèsent sur un système donné. Beaucoup de méthodes se limitent à la recherche de dangers liés à l'informatique pure et négligent les menaces qui pèsent sur les locaux qui abritent les systèmes à protéger. Il est évident que les meilleures mesures de protection logique seront inefficaces contre une destruction physique de matériel [8]. Pour cette raison ainsi que pour une compréhension plus aisée des points théoriques abordés, nous nous sommes concentrés, à travers un exemple simple que nous présenterons, sur cet aspect (physique) de la sécurité de l'information.

2 Motivation et méthode proposée

Un rapport OTAN [6] montre que les méthodes actuelles ne sont pas adaptées aux systèmes complexes (gros systèmes constitués d'un grand nombre de matériels en interaction et de types très variés).

Les méthodes formelles et les outils d'analyse peuvent être la solution à ce problème. Les arbres d'attaques sont, par exemple, très utilisés pour les analyses de risques de systèmes électroniques, de systèmes de contrôle informatique et autres systèmes physiques [1, 4, 5, 7, 9–11]. Sur la base de l'analyse de ces arbres d'attaques, les analystes peuvent alors définir des actions à mener pour réduire ou supprimer les risques.

Jusqu'à présent, la construction de ces arbres d'attaques était faite à la main, et sur la base des connaissances des experts en le système considéré. Ce type de travail est chronophage et sujet aux erreurs, particulièrement lorsque la taille des arbres dépassent une taille intelligible. Notre but est donc d'assister les analystes en rendant cette construction automatique.

Nous avons développé une méthodologie permettant la synthèse d'arbres d'attaques à partir de la modélisation du système étudié. Celle-ci consiste à (1) décrire le système à protéger sous la forme d'un *graphe d'attaque (AG)*, puis à (2) extraire les *attaques* (chemins-solution permettant de relier un état initial à

un état final), pour finir par (3) rassembler toutes ces attaques dans un *arbre d'attaques*.

Cette procédure outillée peut synthétiser un arbre d'attaques à partir de la description "haut-niveau" d'un bâtiment militaire et d'un ensemble d'attaques. Cette spécification de haut niveau permet à l'utilisateur d'exprimer plus facilement ses objectifs de défense.

3 Contributions

Une synthèse complètement automatisée peut rapidement dévier et produire des arbres inexploitables. Mauw and Oostdijk [5] ont montré que de nombreux arbres structurellement différents peuvent contenir la même information, tandis que peu d'entre eux sont facilement lisibles et éloquents pour un expert.

L'une des caractéristiques originales de notre méthodologie est l'usage d'*actions de haut-niveau* permettant d'abstraire et de structurer des séquences d'actions (on peut voir ces actions de haut-niveau comme des sous-buts). Ainsi, l'utilisateur peut contrôler le processus de synthèse et obtenir des arbres d'attaques proches de ceux qu'il aurait construit lui-même à la main.

Nous formalisons également des *hiérarchies d'actions* et utilisons des techniques standards de pattern-matching pour calculer des *stratégies* d'attaques, abstractions des chemins extraits du graphe. Pour finir, ces stratégies sont "factorisées" sous la forme d'un arbre d'attaque lisible et compréhensible.

Nos arbres d'attaques suivent la définition classique [3, 5]. Les feuilles correspondent aux actions primitives et les noeuds internes sont de deux types : *ou* et *et (séquentiel)*. Les noeuds "ou" portent des actions de haut-niveau/(sous-)buts, dont les fils décrivent des alternatives pour atteindre le but. Les séquences de fils sont typées "et" et portent des actions de plus bas niveau.

Ainsi, notre méthodologie passe par la description exhaustive d'un environnement et d'un ensemble de langages pour générer les attaques et par la spécification d'actions de haut-niveau pour synthétiser des arbres d'attaques.

Références

1. AttackTree+. http ://www.isograph.com/software/attacktree/.
2. ISO, Geneva, Switzerland. *Norm ISO/IEC 27002 - Information Technology - Security Techniques - Code of Practice for Information Security Management*, ISO/IEC 27002 :2005 edition, 2005. Section 9.
3. B. Kordy, S. Mauw, S. Radomirović, and P. Schweitzer. Foundations of attack–defense trees. In *Formal Aspects of Security and Trust*, pages 80–95. Springer, 2011.
4. B. Kordy, L. Piètre-Cambacédès, and P. Schweitzer. Dag-based attack and defense modeling : Don't miss the forest for the attack trees. *arXiv preprint arXiv :1303.7397*, 2013.

5. S. Mauw and M. Oostdijk. Foundations of Attack Trees. In *International Conference on Information Security and Cryptology - ICISC 2005. LNCS 3935*, pages 186–198. Springer, 2005.
6. N. Research and T. O. (RTO). Improving Common Security Risk Analysis. Technical Report AC/323(ISP-049)TP/193, North Atlantic Treaty Organisation, University of California, Berkeley, 2008.
7. B. Schneier. Attack Trees : Modeling Security Threats. *Dr. Dobb's Journal*, 1999.
8. E. E. Schultz. Risks due to the Convergence of Physical Security and Information Technology Environments. *Inf. Secur. Tech. Rep.*, 12 :80–84, 2007.
9. Seamonster. http ://sourceforge.net/apps/mediawiki/seamonster/.
10. SecurITree. http ://www.amenaza.com/.
11. O. M. Sheyner. *Scenario Graphs and Attack Graphs*. PhD thesis, 2004.

3

Analyse d'atteignabilité par réécriture sous la stratégie "innermost"

Yann Salmon, Thomas Genet

I.R.I.S.A. / I.N.R.I.A. Rennes, Université Rennes 1

Les méthodes d'approximation de l'ensemble des termes accessibles par réécriture connaissent de plus en plus d'applications, qu'il s'agisse des preuves de terminaison des systèmes de réécriture, de la vérification de protocoles cryptographiques ou de l'analyse statique de programmes.

Nous présentons la complétion d'automates d'arbres [1] ainsi que l'adaptation que nous y apportons pour tenir compte de la stratégie de réécriture "innermost" [2, 3], c'est-à-dire calculer une sur-approximation de l'ensemble des seuls termes accessibles en respectant cette stratégie, laquelle correspond par exemple à l'appel par valeur en usage dans OCaml.

Tenir compte de la stratégie d'évaluation par valeur permet ainsi d'obtenir des approximations plus fines et rend possibles davantage de preuves sur les systèmes correspondants.

[1] Equational Approximations for Tree Automata Completion
Thomas Genet, Vlad Rusu, 2010
http://hal.inria.fr/docs/00/49/54/05/PDF/genet-rusu-JSC-SCSS.pdf

[2] Tree Automata Completion for Static Analysis of Functional Programs
Thomas Genet, Yann Salmon, 2013
http://hal.archives-ouvertes.fr/docs/00/82/64/87/PDF/main.pdf

[3] Reachability Analysis of Innermost Rewriting
Thomas Genet, Yann Salmon, 2014
http://hal.inria.fr/docs/00/94/46/63/PDF/main.pdf

Exploration Aléatoire d'Automates à Pile pour le Test

A. Dreyfus, P.-C. Héam et O. Kouchnarenko

FEMTO-ST - CNRS UMR 6174 - Inria CASSIS – Université de Franche-Comté

Les travaux présentés ici ont fait l'objet de la publication [5], étendue dans [3].

1 Introduction

Le test aléatoire a montré son efficacité en pratique, permettant en général de détecter de nombreuses erreurs. Son principal inconvénient est qu'il capture les comportements arrivant avec une très faible probabilité, et ne garantit pas une couverture du système comme les autres techniques.

Par exemple, lorsque l'on souhaite explorer aléatoirement un modèle fini, la topologie de ce modèle influe directement sur le taux de couverture (pour des critères comme *tous les sommets*). Si l'on considère par exemple le cas simple d'un modèle sous forme de graphe (ou d'automate fini), deux grandes approches possibles coexistent : celle des marches aléatoires où l'on avance récursivement dans le graphe en utilisant équiprobablement chaque arête sortante. L'autre approche consiste à tirer aléatoirement un chemin de taille fixée en utilisant des techniques combinatoires. Selon l'approche utilisée et la topologie du graphe, la couverture du graphe que l'on obtient peut être très variable. Par exemple, sur le graphe de la figure 1 (à $2n$ sommets), pour un chemin de taille $N \geq 2n$, la probabilité de visiter l'état $2n$ sera de l'ordre de $\frac{1}{2^n}$, alors qu'avec un tirage uniforme sur les chemins cette probabilité sera de l'ordre de $\frac{1}{n}$. En revanche sur le graphe de la figure 2, la probabilité de couvrir l'état 1 par une marche aléatoire est de $\frac{1}{2}$, alors qu'avec la génération uniforme de chemins elle est de $\frac{1}{2^N+1}$.

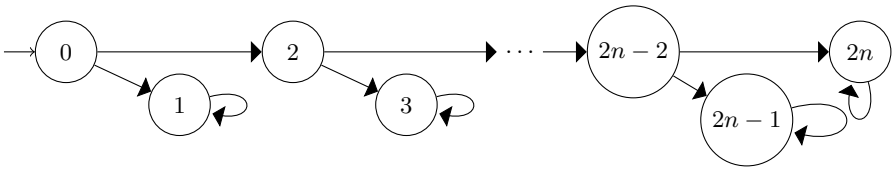


Figure 1.

Approches aléatoires et couverture ne se conjuguent pas forcément bien. Dans un article récent [2], il est montré comment utiliser la technique de génération uniforme de chemins en biaisant la distribution (qui n'est donc plus uniforme), afin d'optimiser la probabilité de couverture – des sommets ou des transitions –

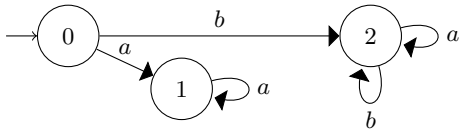


Figure 2.

du graphe. Ces approches sont utilisées dans le cadre de la fiabilité logicielle afin de tester des programmes à partir de graphes de flots de données notamment.

Il existe de très nombreux travaux, livres et ouvrages sur les marches aléatoires, dans le graphe ou dans des diverses structures algébriques. On peut à titre d'exemple citer [6]. La génération aléatoire uniforme de chemins de taille fixe se fait par application directe de techniques combinatoires (voir par exemple [4]), ou par des algorithmes spécifiques [7,1].

2 Contributions

La technique biaisant l'uniformité des tirages n'est pas facile à mettre en œuvre sur des graphes de grande taille. L'utilisation de graphes plus petits introduit généralement une abstraction qui rend les tests difficiles, voire impossibles, à jouer sur le système. Nous montrerons comment étendre les travaux sur des modèles à pile (codant par exemple les appels récursifs de fonctions), qui permettent une abstraction plus fine des systèmes. Nous donnerons des résultats expérimentaux obtenus sur divers exemples.

Références

1. O. Bernardi and O. Giménez. A linear algorithm for the random sampling from regular languages. *Algorithmica*, 62(1-2) :130–145, 2012.
2. A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet. Coverage-biased random exploration of large models and application to testing. *STTT*, 14(1) :73–93, 2012.
3. A. Dreyfus, P.-C. Héam, O. Kouchnarenko, and C. Masson. A random testing approach using pushdown automata. *Softw. Test., Verif. Reliab.*, 2014. Accepted.
4. P. Flajolet and R. Sedgwick. *Analiytic Combinatorics*. Cambridge University Press, 2009.
5. P.-C. Héam and C. Masson. A random testing approach using pushdown automata. In M. Gogolla and B. Wolff, editors, *TAP*, volume 6706 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2011.
6. D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Society, 2009.
7. J. Oudinet, A. Denise, and M.-C. Gaudel. A new dichotomic algorithm for the uniform random generation of words in regular languages. *Theor. Comput. Sci.*, 502 :165–176, 2013.

Session de l'action IDM

Ingénierie dirigée par les modèles

Composite UML à l'OMG

Auteur : Arnaud Cuccuru (CEA)

Résumé :
The purpose of this presentation is to give an overview of a new OMG standard called Precise Semantics of UML Composite Structures (PSCS). This specification includes semantic definitions for all the UML metaclasses supporting the ability of classfiers to have both an internal structure (comprising a network of linked parts) and an external structure (consisting of one or more ports). It covers both structural semantics (e.g. the runtime manifestations of connectors, ports, and parts) and behavioral semantics (e.g. life-cycles of composite objects and their constituents, the nature and characteristics of flows through ports and connectors). It builds on the precise semantics of fUML, which specifies execution semantics of a computationally complete and compact subset of UML 2 to support execution of activities.

Community development in MDE/ Community development by MDE

Auteur : Jordi Cabot (EMN, Inria, Lina)

Résumé :
Model-driven engineering is collaborative in nature. The best example is the creation of Domain-Specific Modeling Languages (DSMLs), which are (modeling) languages specifically designed to carry out the tasks of a particular domain. In this talk we will describe Collaboro, a community-aware language development process by enabling the active participation of all community members (both developers and end-users of the DSML) from the very beginning. But collaboration is a key aspect of all software engineering tasks. MDE techniques can help to improve collaboration beyond MDE itself. In particular, we will see how MDE techniques can also help to improve the collaboration and governance in (open source) software development projects.

Partager la connaissance sans contraintes : surmonter les limitations des référentiels de modèles

Auteur : Philippe Desfray, (Softeam)

Résumé :
Dans notre ère de l'ubiquité du partage de données, de la communication immédiate et de la répartition mondiale des participants à des projets, à une époque où on demande aux équipes d'être de plus en plus agiles, l'approche traditionnelle des référentiels de modèles ne correspond plus aux attentes. Les organisations centralisées deviennent incohérentes avec le mode de fonctionnement des sociétés et du monde. Dans le monde actuel, il est en pratique impossible de mettre en place un référentiel de modèles pour différentes entités de l'entreprise, ou pour des systèmes ou projets à grande échelle, qui puissent être accessibles par tous les participants (lecteurs, contributeurs, partenaires). Les techniques usuelles, basées sur un référentiel centralisé avec un gestionnaire dédié, sont en contradiction avec un grand nombre de situations où les participants ne veulent ni n'ont la possibilité de se conformer à des règles uniformisées d'accès.

Cette situation inhibe la possibilité de gérer les connaissances via les modèles à un niveau global (entreprise, inter-entreprises). Elle inhibe également l'agilité et la coopération ouverte des équipes. Nous sommes convaincus que ceci constitue un obstacle majeur à la dissémination des approches guidées par la modélisation ; la lourdeur des référentiels de modèles masque les avantages réels de l'approche guidée par les modèles.

En s'appuyant sur les dernières technologies et recherches pour les référentiels de modèles, cette présentation dé taille en quoi les technologies actuelles de référentiel sont un obstacle majeur, et présentera un moyen de supporter les organisations fortement distribuées ainsi que les coopérations agiles et ouvertes. Monter en échelle et élargir la portée des référentiels de modèles permettront aux modèles de s'appliquer à l'entreprise étendue, incluant son écosystéme, ainsi que toute organisation distribuée et coopérative.

Session du groupe de travail LaMHA

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

Programmation avec les homomorphismes quasi synchrones

Auteur : Julien Tesson (LACL, Paris-Est Créteil)

Résumé :
Les squelettes algorithmiques, avec les homomorphismes de listes, jouent un rôle important dans le éveloppement formelle d'algorithmes parallèles. Nous avons conçu une notion proche des homomorphismes dédiée au parallélisme quasi-synchrone. Nous présenterons les homomorphismes quasi-synchrones ; puis l'intégration d'un squelette algorithmique BH dans la bibliothèque Orléans Skeleton Library (OSL), une bibliothèque C++ de programmation par squelettes algorithmiques. Enfin une implantation à l'aide du squelette BH du problème des plus proches valeurs inférieures qui est un sous problème connu de nombreux algorithmes.

Automatic Task-based Code Generation for High Performance Domain Specific Embedded Language

Auteur : Antoine Tran Tan, Joel Falcou, Daniel Etiemble (LRI, U. Paris XI)

Résumé :
Providing high level tools for parallel programming while sustaining a high level of performance has been a challenge that tech- niques like Domain Specific Embedded Languages try to solve. In previous works, we investigated the design of such a DSEL - NT2 - providing a Matlab -like syntax for parallel numerical computations inside a C++ library. In this paper, we show how NT2 has been redesigned for shared memory systems in an extensible and portable way. The new NT2 design relies on a tiered Parallel Skeleton system built using asynchronous task management and automatic compile-time taskification of user level code. We describe how this system can operate various shared memory runtimes and evaluate the design by using several benchmarks implementing linear algebra algorithms.

Handling Data-skew Effects in Join Operations using MapReduce

Auteur : Mohamad Al Hajj Hassan, Mostafa Bamha, Frédéric Loulergue (LIFO, U. Orléans)

Résumé :
For over a decade, MapReduce has become a prominent programming model to handle vast amounts of raw data in large scale systems. This model ensures scalability, reliability and availability aspects with reasonable query processing time. However these large scale systems still face some challenges : data skew, task imbalance, high disk I/O and redistribution costs can have disastrous effects on performance. In this talk, we introduce MRFA-Join algorithm : a new frequency adaptive algorithm based on MapReduce programming model and a randomised key redistribution approach for join processing of large-scale datasets. A cost analysis of this algorithm shows that our approach is insensitive to data skew and ensures perfect balancing properties during all stages of join computation. These performances have been confirmed by a series of experimentations.

Session du groupe de travail LTP

Langages, Types et Preuves

Using high-level program properties to enhance WCET estimation

Auteur : Pascal Raymond (Verimag U. Grenoble)

Résumé :
Real-time critical systems can be considered as correct if they compute both "right" and "fast enough". Functionality aspects (computing right) can be addressed using high level design methods, such as the synchronous approach that provides languages, compilers and verification tools. Real-time aspects (computing fast enough) can be addressed with static timing analysis, that aims at discovering safe bounds on the Worst-Case Execution Time (WCET) of the binary code. In this work, we aim at improving the estimated WCET in the case where the binary code comes from a high-level synchronous design. The key idea is that some high-level functional properties may imply that some execution paths of the binary code are actually infeasible, and thus, can be removed from the worst-case candidates. In order to automatize the method, we show (1) how to trace semantic information between the high-level design and the executable code, (2) how to use a model-checker to prove infeasibility of some execution paths, and (3) how to integrate such infeasibility information into an existing timing analysis framework. Based on a realistic example, we show that there is a large possible improvement for a reasonable computation time overhead.

Formal Verification of a C Value Analysis Based on Abstract Interpretation *†

Sandrine Blazy[1], Vincent Laporte[1], Andre Maroneze[1], and David Pichardie[2]

[1] Université Rennes 1 - IRISA
[2] ENS Rennes - IRISA

Over the last decade, significant progress has been made in developing tools to support mathematical and program-analytic reasoning. Proof assistants are now successfully applied both in mathematics and in formal verification of critical software systems. Over the same time, automatic verification tools have become widely used by the critical software industry. The main reason for their success is that they strengthen the confidence we can have in critical software by providing evidence of software correctness. The next step is to strengthen the confidence in the results of these verification tools, and proof assistants seem to be mature and adequate for this task. This paper presents a foundational step towards the formal verification of a static analysis based on abstract interpretation [3]: the formal verification using the Coq proof assistant of a value-range analysis operating over a real-world language.

Static analyzers based on abstract interpretation are complex pieces of software that implement delicate symbolic algorithms and numerical computations. Their design requires a deep understanding of the targeted programming language. Misinterpretations of the programming language informal semantics may lead to subtle soundness bugs that may be hard to detect by using only testing techniques. Implementing a value analysis raises specific issues related to low-level numeric computations. First, the analysis must handle the machine arithmetic that is (more or less) defined in the programming language. Second, some computations done by the analyzer rely on this machine arithmetic.

Thus, a prerequisite for implementing a static analyzer operating over a C-like language is to rely on a formal semantics of the programming language defining precisely the expected behaviors of any program execution (and including low-level features such as machine arithmetic). Such formal semantics are defined in the CompCert compiler (and it is unusual for a compiler). More precisely, each language of the compiler is defined by a formal semantics (in Coq) associating observable behaviors to any program. Observable behaviors consist in normal termination, divergence, and going wrong behaviors. We have chosen one language of the compiler having the same expressiveness as C and we have formalized a static analyzer operating over this language. Thus, our analyzer as well as the formal semantics operate exactly over the same language.

The different languages of CompCert feature both low-level aspects such as pointers, pointer arithmetic and nested objects, and high-level aspects such as separation and freshness guarantees. A memory model [5] is shared by the semantics of all these languages. Memory states are collections of blocks, each block being an array of abstract bytes. A block represents a C variable or an invocation of `malloc`. Pointers are represented by pairs `(b,i)` of a block identifier and a byte offset `i` within this block. Values stored in memory are the disjoint union of 32-bit integers `vint(i)`, 64-bit floating-point numbers, locations `vptr(b,i)`, and a special value representing the contents of uninitialized memory.

With the help of J.H. Jourdan and X. Leroy, we have designed a new intermediate language called CFG that is adapted to static analysis: its expressions are side-effect free C expressions, its programs are represented by their control flow graphs with explicit program points and the control flow is restricted to simple unconditional and conditional jumps. The CFG semantics is defined as a transition relation between execution states (tuples called σ). Among the components of σ are the current program point, the memory state and the environment mapping program variables to values. We use $\sigma.E$ to denote the environment of σ, and $\texttt{dom}(\sigma.E)$ to denote its domain. We use $\texttt{reach}(P)$ to denote the set of states belonging to the execution trace of P.

Our value analysis `value_analysis` computes for each program point the estimated values of the program variables. When the value of a variable is an integer `i` or a pointer value of offset `i`, the estimate provides 2 numerical ranges `signed_range` and `unsigned_range`. The first one over-approximates the signed interpretation of `i` and the other range over-approximates its unsigned interpretation. We note `ints_in_range (signed_range, unsigned_range) i` this fact. Thus, given a program P, $\texttt{value\_analysis}(P)$ yields a map such that for each node l in its control flow graph and each variable v, $\texttt{value\_analysis}(P)[l, v]$ is a pair of sound ranges for v. Theorem 1 states the soundness of the value analysis: for every program state that may be reached during a program execution, any program point and variable, every variable valuation computed by the analysis is a correct estimation of the exact value given by the concrete semantics.

Theorem 1. *Let P be a program, $\sigma \in \texttt{reach}(P)$ and $res = \texttt{value\_analysis}(P)$ be the result of the value analysis. Then, for each program point l, for each local variable $v \in \texttt{dom}(\sigma.E)$ that contains an integer i (i.e., $\sigma.E(v) = \texttt{vint}(i) \vee \exists b, \sigma.E(v) = \texttt{vptr}(b, i)$), the property* (`ints_in_range` $res[l, v]$ i) *holds.*

* Supported by Agence Nationale de la Recherche, grant ANR-11-INSE-003 Verasco.

† A long version of this paper [1] has been published at SAS 2013.

Our value analysis is designed in a modular way: a generic fixpoint iterator operates over generic abstract domains. The iterator is based on the Bourdoncle [2] algorithm that provides both efficiency and precision. The modular design of the abstract domains is inspired from the design of the Astrée analyzer. It consists in three layers that are showed in Figure 1. The simplest domains are numerical abstract domains made of intervals of machine integers; they are not aware of the C memory model. In C, a same piece of data can be used both in signed and unsigned operations, and the results of these operations differ from one interpretation to the other. Thus, we have 2 numerical abstract domains. Our analysis computes the reduced product of the 2 domains in order to make a continuous fruitful information exchange between these 2 domains.

Then, we build abstract domains representing numerical environments. We provide a non-relational abstraction that is parameterized by a numerical abstract domain. The last layer is the abstract domain representing memory. It is parameterized by the previous layer and links the abstract interpreter with the numerical abstract domains. This modular design is targeted to connect at each layer other abstract domains, represented in dotted lines in Figure 1. For example, several abstract memory models can be used instead of the current one while maintaining the same interfaces with the rest of the formal development. The ultimate goal is to enhance our current abstract interpreter in order to connect it to a memory domain *à la* Miné [6]. The current interfaces are also compatible with any relational numerical abstract domain. At the top, more basic numerical abstractions as congruence could be added and plugged into our reduced product.

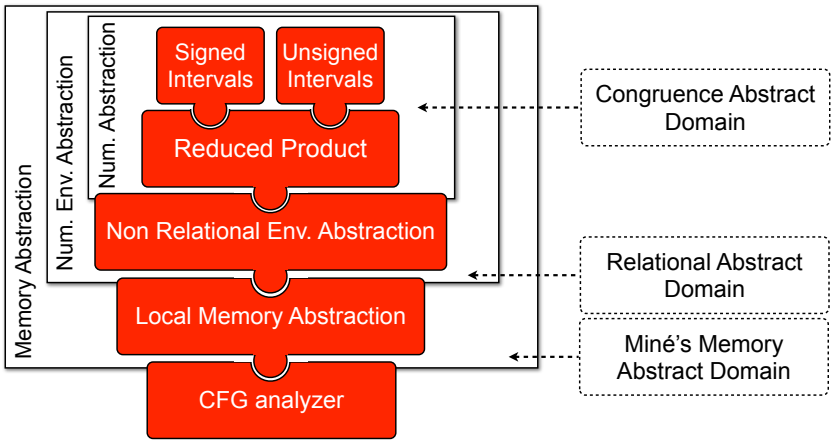


Fig. 1. Design of abstract domains: a three-layer view

This work provides the first verified value analysis for a realistic language as C. Implementing a precise value analysis for C is highly error-prone. We hope that our work shows the feasibility of developing such a tool together with a machine-checked proof. The precision of the analysis has been experimentally evaluated and compared on several benchmarks. The paper's technology performs comparably to existing off-the-shelf (unverified!) tools, Frama-C [4] and Wrapped [7]. Our contribution is also methodological. Our formalization, its lightweight interfaces and its proofs can be easily reused to develop different formally verified analyses. One of our several challenging directions is to replace the current memory abstraction with a domain similar to Miné's memory model [6]. This domain tracks finely the content of (statically allocated) memory cells. Verifying such a domain raises specific challenges not only in terms of semantic proofs but also in terms of efficient implementation of the transfer functions.

References

1. S. Blazy, V. Laporte, A. Maroneze, and D. Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *Proc. of SAS 2013*, volume 7935 of *LNCS*, pages 324–344. Springer, 2013.
2. F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of FMPA*, volume 735 of *LNCS*, pages 128–141, 1993.
3. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
4. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C - a software analysis perspective. In *Proc. of SEFM 2012*, volume 7504 of *LNCS*, pages 233–247. Springer, 2012.
5. X. Leroy, A.W. Appel, S. Blazy, and G. Stewart. *The CompCert Memory Model*. Cambridge University Press, 2014. In Program Logics for Certified Compilers.
6. A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of LCTES'06*, pages 54–63. ACM, Jun. 2006.
7. J. Navas, P. Schachte, H. Søndergaard, and P. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In *Proc. of APLAS 2012*, volume 7705 of *LNCS*. Springer, 2012.

Hunting Concurrency Compiler Bugs

R. Morisset[1,2], P. Pawan[1], and F. Zappa Nardelli[1,2]

[1] INRIA
[2] ENS

Concurrency compiler bugs Compilers sometimes generate correct sequential code but break the concurrency memory model of the programming language: these subtle compiler bugs are observable only when the miscompiled functions interact with concurrent contexts, making them particularly hard to detect.

This is a critical time to be able to detect concurrency compiler bugs. The C and C++ languages were originally designed without concurrency support: threads were available via external libraries, yielding unexpected behaviours and misunderstandings between programmers and compiler writers. The recent revision of the C and C++ standards[3] does provide a precise semantics for threads: well-synchronised programs must exhibit only sequentially consistent behaviours, racy programs can have any behaviour, and an escape mechanism with a complex semantics, called low-level atomics, enables programmers to write high-performance but portable concurrent code. The resulting model is intricate and the interactions with compiler optimisations are not entirely understood. Todays C and C++ compilers, whose optimisers were initially developed in absence of any well-defined memory model, are being extended to support the new concurrency standard.

How to search for concurrency compiler bugs? Differential random testing proved successful at hunting compiler bugs. The idea is simple: a test harness generates random, well-defined, source programs, compiles them using several compilers, runs the executables, and compares the outputs. However this approach is unlikely to scale to concurrency compiler bugs because concurrent programs are inherently non-deterministic and optimisers can compile away non-determism: comparing the outputs is not enough to reliably detect miscompilations.

Despite this, in this work we show that differential random testing can be used successfully for hunting concurrency compiler bugs. Our first contribution is a theory of sound optimisations in the C11/C++11 memory model, covering most of the optimisations we have observed in real compilers and validating the claim that common compiler optisations are sound in the C11/C++11 memory model. Our second contribution is to show how, building on this theory, concurrency compiler bugs can be identified by comparing the memory trace of compiled code against a reference memory trace for the source code. We put this idea at work

[3] P. Becker. Standard for Programming Language C++ - ISO/IEC 14882, 2011.

2

and build a tool, `cmmtest`, that identified several mistaken write introductions and other unexpected behaviours in the latest release of the `gcc` compiler. These have been promptly fixed by the `gcc` developers.

Dissemination This work is presented in a paper that appeared in Proc. PLDI 2013[4], while the `cmmtest` tool is available from

`http://www.di.ens.fr/ zappa/projects/cmmtest/`

together with extensive documentation about the project.

[4] R. Morisset, P. Pawan, F. Zappa Nardelli, Compiler Testing via a Theory of Sound Optimisations in the C11/C++11 Memory Model, ACM SIGPLAN Notices - PLDI 2013, Volume 47 Issue 6, Pages 187-196, June 2013. Copyright ACM. `http://doi.acm.org/10.1145/2491956.2491967`

Session du groupe de travail MTV^2

Méthodes de test pour la validation et la vérification

An Optimized Memory Monitoring Library for Runtime Assertion Checking with Frama-C

Nikolai Kosmatov, Guillaume Petiot, and Julien Signoles

CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
firstname.lastname@cea.fr

Extended Abstract

Memory related errors, including invalid pointers, out-of-bounds memory accesses, uninitialized variables and memory leaks, are very frequent. They are particularly an issue for a programming language like C that is paradoxically both the most commonly used for development of system software with various critical components, and one of the most poorly equipped with adequate protection mechanisms. The C developer remains responsible for correct allocation and deallocation of memory, pointer dereferencing and manipulation (like casts, offsets, etc.), as well as for the validity of indices in array accesses.

Among the most practical techniques for detecting and locating software errors, *runtime assertion checking* is now a widely used programming practice [1]. Many researchers have worked on efficient techniques and tools for runtime assertion checking. Leucker and Schallhart provide a survey on *runtime verification* and conclude that "one of its main technical challenges is the synthesis of efficient monitors from logical specifications" [2]. An efficient memory monitoring for C programs is the purpose of the work [3] presented in this extended abstract.

This work relies on E-ACSL [4, 5], the specification language designed to support runtime assertion checking in Frama-C. Frama-C [6] is a platform dedicated to analysis of C programs that includes various analyzers, such as abstract interpretation based value analysis (Value plug-in), dependency analysis, program slicing, Jessie and WP plug-ins for proof of programs, etc. ACSL [7] is a behavioral specification language shared by different Frama-C analyzers inspired by JML. ACSL is expressive enough to express most functional properties of C programs and has already been used in many projects. It is based on a typed first-order logic in which terms may contain *pure* (*i.e.* side-effect free) C expressions and special keywords. A contract may be associated to each function in order to specify its pre- and postconditions. The contract can be split into several named guarded behaviors. Contracts may also be associated to statements, as well as assertions, loop invariants and loop variants. ACSL annotations also include definitions of (inductive) predicates, axiomatics, lemmas, logic functions, data invariants and ghost code.

Designed as a rich subset of ACSL, E-ACSL preserves ACSL semantics. Moreover, the E-ACSL language is *executable*: its annotations can be translated into C monitors and executed at runtime. This makes it suitable for runtime assertion checking.

We present a solution for memory monitoring of C programs we have developed for runtime assertion checking in Frama-C. It includes a translator, called EACSL2C in this paper, that automatically translates an E-ACSL specification into C code [4, 8]. In order to support memory-related annotations for pointers and memory locations (such as being valid, initialized, in a particular block, with a particular offset, etc.), we need to keep track of relevant memory operations previously executed by the program. Hence, we have developed a monitoring library for recording and retrieving validity and initialization information for the program's memory locations, as well as an automatic instrumentation of source code inserting necessary calls to the library during the translation of an E-ACSL specification into C.

In order to evaluate memory-related E-ACSL annotations, we record information on validity and initialization of memory locations during program execution in a dedicated data store, that we call below *the store.* The memory monitoring library provides primitives for both evaluating memory-related E-ACSL annotations (by making queries to the store) and recording in the store

all necessary data on allocation, deallocation and initialization of memory blocks. Thus EACSL2C inserts calls to library primitives for two purposes: *1)* to translate into C and evaluate memory-related E-ACSL annotations; and *2)* to record memory-related program operations in the store. The library has been optimized in several ways to accelerate runtime verification.

The code instrumented by EACSL2C reports an E-ACSL annotation failure at runtime if and only if this E-ACSL annotation is indeed violated. However it has the major drawback of being hugely verbose and time-consuming: for each variable, each (de)allocation and each assignment, one or even several new statements are generated. It is however sufficient to monitor the memory locations involved in memory-related constructs in the provided E-ACSL annotations. To solve this drawback, we have designed an interprocedural backward dataflow analysis which computes an over-approximated set of memory locations that it is sufficient to monitor in order to preserve soundness and completeness of the instrumentation.

Our solution implements a *non-invasive* source code instrumentation, that is, monitoring routines do not change the observed behavior of the program. In particular, it does not modify the memory layout and size of variables and memory blocks already present in the original program, and may only record additional monitoring data in a separate memory store.

To evaluate our solution, we performed in total more than 300 executions for more than 30 programs obtained from about 10 examples. These initial experiments were conducted on small-size examples because they were mostly manually specified in E-ACSL. We measured the execution time of the original code and of the code instrumented by EACSL2C with various options in order to evaluate their performances (with and without our optimizations, with four different implementations of the store, etc.). Such indicators as the number of monitored variables, memory allocations, records and queries in the store were recorded as well. The results confirm the benefits of our optimizations. In addition to performance evaluation, we used mutational testing to evaluate the capacity of error detection using runtime assertion checking with Frama-C. Each mutant was instrumented by EACSL2C and executed on a test suite (generated by another Frama-C plug-in) in order to check at runtime if the specification was satisfied. All erroneous mutants were killed by runtime assertion checking with Frama-C.

References

1. Clarke, L.A., Rosenblum, D.S.: A historical perspective on runtime assertion checking in software development. ACM SIGSOFT Software Engineering Notes **31**(3) (2006) 25–37
2. Leucker, M., Schallhart, C.: A brief account of runtime verification. J. Log. Algebr. Program. **78**(5) (2009) 293–303
3. Kosmatov, N., Petiot, G., Signoles, J.: An optimized memory monitoring for runtime assertion checking of c programs. In: the 4th International Conference on Runtime Verification (RV 2013). Volume 8174 of LNCS., Springer (2013) 167–182
4. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: the 28th Annual ACM Symposium on Applied Computing (SAC 2013), ACM (2013) 1230–1235
5. Signoles, J.: E-ACSL: Executable ANSI/ISO C Specification Language. URL: http://frama-c.com/download/e-acsl/e-acsl.pdf.
6. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C, a program analysis perspective. In: the 10th International Conference on Software Engineering and Formal Methods (SEFM 2012). Volume 7504 of LNCS., Springer (2012) 233–247
7. Baudin, P., Filliâtre, J.C., Hubert, T., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language. URL: http://frama-c.com/acsl.html.
8. Kosmatov, N., Signoles, J.: A lesson on runtime assertion checking with Frama-C. In: the 4th International Conference on Runtime Verification (RV 2013). Volume 8174 of LNCS., Springer (2013) 386–399

Unfolding-based Test Selection for Concurrent Conformance

Hernán Ponce de León[1], Stefan Haar[1], and Delphine Longuet[2]

[1] INRIA and LSV, École Normale Supérieure de Cachan and CNRS, France
ponce@lsv.ens-cachan.fr , stefan.haar@inria.fr
[2] Univ Paris-Sud, LRI UMR8623, Orsay, F-91405
longuet@lri.fr

The **ioco** conformance relation has become a standard for testing reactive systems from behavioural models like input output labeled transition systems (IOLTS). This kind of models is well-suited to specify sequential systems, however this is not the case for systems composed of concurrent components. Such systems are naturally modeled as a network of finite automata, a formal class of models that can be captured equivalently by safe Petri nets.

Concurrency in a specification can be interpreted in different ways. We consider that two events specified as concurrent are physically localized on different components, and thus are "naturally" independent of one another. Therefore, we want concurrency of the specification (i.e. independancy of concurrent events) to be preserved in the implementation.

Model-based testing of concurrent systems has been studied for a long time, however it is most of the time studied in the context of interleaving semantics, which is known to suffer the state space explosion problem. Work by Ulrich and König [1] and by Haar et al [2] for example define conformance relations based on labeled partial orders in order to keep concurrency explicit. In order to enlarge the application domain, and at stronger benefits from concurrency modeling, we have introduced a concurrent conformance relation named **co-ioco**, as a generalization of **ioco** [3, 4].

We extended this work [5] with a conformance relation where actions specified as concurrent must occur independently, on different processes, in any conformant implementation. We gave an algorithm based on Petri net unfolding to construct a complete test suite, as well as a selection criterion based on the notions of complete prefix and cut-off event to build a practical test suite.[3]

Testing Framework for IOPNs. We choose to use Petri nets as specifications to have explicit concurrency. The semantics associated to a Petri net is given by its unfolding to an occurrence net, which can also be seen as an event structure. The execution traces for this semantics are not sequences but partial orders, which keep concurrency explicit.

Our conformance relation compares partially ordered traces of the implementation to those of its Petri net specification. In particular, we compare the outputs via partially ordered sets so we need any set of outputs to be entirely produced by the system under test before we send a new input, in order to detect outputs depending on extra inputs. Moreover, we do not assume the input-enabledness of the system under test, but only that we can observe refusals.

Our **co-ioco** conformance relation for input-output labeled event structures (IOLES) can be informally described as follows. The behavior of a correct **co-ioco** implementation after some observations (obtained from the specification) should respect the following restrictions: (1) the outputs produced by the implementation should be specified; (2) if a quiescent configuration is reached, this should also be the case in the specification; (3) any time an input is possible in the specification, this should also be the case in the implementation. When several outputs in conflict are possible, our conformance relation allows implementations where at least one of them is implemented. Extra inputs are allowed in any configuration, but extra outputs, extra quiescence and extra causality between events specified as concurrent are forbidden.

A test case is a finite deterministic IOLES where there are no immediate conflicts between inputs. As IOLES can be seen as occurrence nets, we can model the test execution as the parallel composition of labeled nets. For obtaining sound and exhaustive test suites, we give the following sufficient conditions [4]. First, for a test suite to be sound, each test must produce only traces of the specification, and preserve all possible outputs for each such trace. Second, a test suite is exhaustive if each trace of the specification appears in at least one test and if tests preserve quiescence.

[3] See [5] for a full paper version of this abstract.

Test case generation and selection. Our test case generation algorithm builds a test case from an IOLES by resolving immediate conflicts between inputs, while accepting several branches in case of conflict between outputs. At the end of the algorithm, all such conflicts have been resolved in one way, following one fixed strategy of resolution of immediate input conflicts; the resulting object, the test case, is thus one branching prefix of the IOLES. In order to cover the other branches, the algorithm must be run several times with different conflict resolution schemes, to obtain a test suite that represents every possible event in at least one test case. Each such scheme can be represented as a linearization of the causality relation that specifies in which order the events are selected by the algorithm. We prove that the collection of linearizations that we use considers all resolutions of immediate input conflicts, meaning that our test case generation algorithm is general enough to produce a complete test suite from the set of all the prefixes of the specification.

The behavior of the system described by the specification usually consists of infinite traces. However, in practice, these long traces can be considered as a sequence of (finite) "basic" behaviors. We choose an inclusion selection criterion, fulfilled if each basic behavior described by the specification is covered once. To build a test suite for this criterion, we define a proper notion of complete prefix of a Petri net unfolding. The complete finite prefix algorithm [6] depends on the notion of *cut-off event* which determines how long the net is unfolded. We define a notion of cut-off event corresponding to our inclusion criterion, "every cycle is unfolded once": an event is cut-off iff the prefix already contains an event with the same marking.

Nevertheless completeness does not imply that the information about outputs and quiescence is preserved, which we need to prove the soundness of the test suite generated from such a prefix. In order to preserve this information, we follow Gaston et al [7] and modify the complete finite prefix algorithm adding all the outputs from the unfolding that the complete prefix enables. We give an algorithm to compute the quiescent closure of the complete finite prefix and prove that the test suite build based on the resulting prefix is sound.

Conclusion. The present testing approach is global, meaning that a global control and observation of the distributed system is assumed, and tests are performed in a centralized way. The next step of our work is to distribute control and observation over several concurrent components. This will necessarily weaken the conformance relation, since dependencies between events occurring on different components cannot be observed anymore. The local test cases should, roughly speaking, be projections of the global test cases onto the different components, since concurrency of the specification was preserved in the test cases. We still have to investigate how distribution affects the power of testing, and how the resulting methods compares to others, such as the **dioco** framework of Hierons et al. [8] for multi-port IOTS.

Acknowledgment: This work was funded by the DIGITEO/DIM-LSC project TECSTES, convention DIGITEO Number 2011-052D - TECSTES.

References

1. Ulrich, A., König, H.: Specification-based testing of concurrent systems. In: Formal Description Techniques for Distributed Systems and Communication Protocols. Volume 107 of IFIP Conference Proceedings. (1998) 7–22
2. von Bochmann, G., Haar, S., Jard, C., Jourdan, G.V.: Testing systems specified as partial order input/output automata. In: Testing of Software and Communicating Systems. Volume 5047 of LNCS., Springer (2008) 169–183
3. Ponce de León, H., Haar, S., Longuet, D.: Conformance relations for labeled event structures. In: Tests and Proofs. Volume 7305 of LNCS., Springer (2012) 83–98
4. Ponce de León, H., Haar, S., Longuet, D.: Model-based testing for concurrent systems with labeled event structures. http://hal.inria.fr/hal-00796006 (2012)
5. Ponce de León, H., Haar, S., Longuet, D.: Unfolding-based test selection for concurrent conformance. In: International Conference on Testing Software and Systems. Volume 8254 of Lecture Notes in Computer Science., Springer (2013) 98–113
6. Esparza, J., Römer, S., Vogler, W.: An improvement of McMillan's unfolding algorithm. In: Tools and Algorithms for Construction and Analysis of Systems. Volume 1055 of LNCS., Springer (1996) 87–106
7. Gaston, C., Le Gall, P., Rapin, N., Touil, A.: Symbolic execution techniques for test purpose definition. In: Testing of Software and Communicating Systems. Volume 3964 of LNCS., Springer (2006) 1–18
8. Hierons, R.M., Merayo, M.G., Núñez, M.: Implementation relations for the distributed test architecture. In: Testing of Software and Communicating Systems. Volume 5047 of LNCS., Springer (2008) 200–215

Test de vulnérabilité Web à base de patterns et de modèles

Alexandre Vernotte[1], Bruno Legeard[1,2], and Fabien Peureux[1,2]

[1] Institut FEMTO-ST, UMR CNRS 6174 - Route de Gray, 25030 Besançon, France
{avernott, blegeard, fpeureux}@femto-st.fr
[2] Smartesting R&D Center - 2G, Avenue des Montboucons, 25000 Besançon, France,
{legeard, peureux}@smartesting.com

Résumé Cette présentation introduit une approche de génération de tests de vulnérabilité pour les applications Web à partir de modèles et guidée par des patterns de test. Cette approche mixte, baptisée PMVT (Pattern-driven and Model-based Vulnerability Testing), vise à tirer parti des bénéfices et diminuer les faiblesses des techniques de test à base de modèles, généralement utilisées pour le test fonctionnel, et des techniques de test de vulnérabilité, principalement manuelles ou assistées par des scanners de vulnérabilité. Finalement, une synthèse des résultats expérimentaux obtenus et les travaux futurs sont présentés. Ces travaux sont supportés par le projet FSN DAST (http://dast.deptinfo-st.univ-fcomte.fr).

Mots-clés: Test de vulnérabilité, pattern de test de sécurité, test à partir de modèles (MBT), vulnérabilité d'applications Web, Cross-Site Scripting (XSS), injection SQL (SQLI).

1 Contexte et motivations

L'état de l'art actuel sur la sécurité et les différents rapports de sécurité, comme l'OWASP Top Ten 2013 [1], montrent que les applications Web sont aujourd'hui la cible principale des cyber-attaques. Le test de vulnérabilité est une activité de plus en plus pratiquée pour répondre au besoin croissant de sécurité des applications Web. On distingue deux techniques principales : le test de pénétration et les scanners de vulnérabilité. La première approche, généralement manuelle, nécessite un coût humain important et n'est pas exhaustive. La seconde approche est certes peu coûteuse car automatisée mais reste imprécise en détectant un taux important de faux positifs et de faux négatifs [2]. Ce papier propose une approche alternative, baptisée PMVT pour Pattern-driven and Model-based Vulnerability Testing, qui vise à agréger les bénéfices des deux approches historiques tout en maîtrisant leurs faiblesses. Les contributions au test de sécurité se situent :

- dans la capture des comportements applicatifs par un modèle abstrait UML/OCL permettant une couverture de test plus complète de l'application sous test ;
- dans la création d'un langage de script dédié pour minimiser l'effort de modélisation ;
- dans l'extension d'un langage de pattern de test qui permet de guider le générateur de tests à travers le modèle pour couvrir les vulnérabilités à cibler ;
- dans l'automatisation de la génération des tests à partir du modèle et des patterns de test, de l'exécution des test générés, et finalement de la définition du verdict.

2 Principes de l'approche PMVT

L'approche PMVT est une dérivation du Model-Based Testing (MBT) classique pour permettre la génération de tests de vulnérabilité. Cette approche, illustrées en figure 1, est composée des quatre activités principales (une description plus détaillée de chacune des activités est disponible dans [3]). La première activité concerne la définition de *Test Purposes* (①) qui formalisent les objectifs de test de vulnérabilité. Une *Modélisation* (②) de l'application Web en UML/OCL permet ensuite de capturer ses aspects structurels et comportementaux. A partir des artefacts définis lors des deux précédentes activités, on réalise alors une *Génération* automatique de cas de test abstraits (③). Finalement, la phase de *Concrétisation* et d'*Exécution* (④) permet de traduire les cas de test abstraits générés en scripts exécutables, d'exécuter ces scripts sur l'application Web, et d'observer les réponses et de les comparer aux résultats attendus pour assigner un verdict.

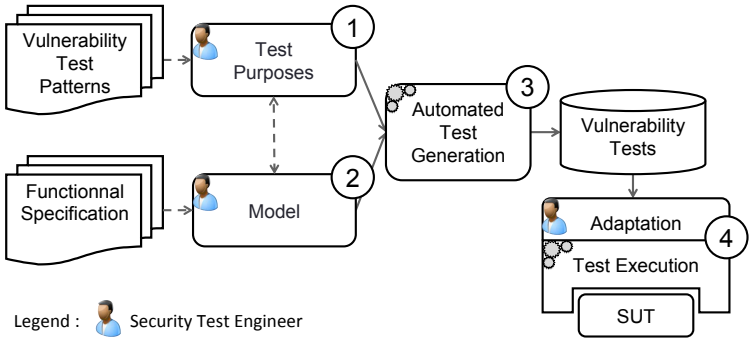


Figure 1. Description de l'approche PMVT

3 Expérimentations et résultats

L'approche PMVT a notamment été expérimentée sur une application de e-learning *Stud-E* (15000 utilisateurs en France) dans l'objectif spécifique de découvrir des vulnérabilités de type multi-step XSS (dans ce type d'injection, les pages d'observation ne se situent pas immédiatement après la page dans laquelle l'injection XSS est réalisée, ce qui nécessite de naviguer dans l'application entre les phases d'injection et d'observation). La mise en œuvre complète de l'approche PMVT, par un ingénieur expert de cette approche, a duré environ douze heures et a donné lieu à la production d'environ 1500 tests et la découverte de 2 failles. Comparativement, nous avons mené une campagne de test de pénétration (avec proxy intrusif) et une campagne d'exécution de scanner de vulnérabilité sur cette même application. D'une part, dix-neuf heures de test de pénétration ont ainsi été nécessaires pour atteindre une couverture de test comparable et détecter les vulnérabilités de type multi-step XSS. D'autre part, l'utilisation de différents scanners (IBM AppScan, NTOSpider, w3af, skipfish, et arachni) n'a pas permis de détecter les 2 failles. Ceci s'explique par la nécessité de connaître non seulement le point d'injection mais aussi le point d'observation.

4 Conclusion et travaux futurs

Cet article a introduit PMVT, une approche pour l'automatisation du test de vulnérabilité Web à partir de modèles et guidée par des patterns de test de vulnérabilité. Les résultats expérimentaux et les études comparatives avec des techniques existantes (test de pénétration et exécution de scanner) ont montré la pertinence de cette approche originale. Ces expérimentations ont également mis en évidence les faiblesses de l'approche PMVT. Elle requiert effectivement un effort encore important de modélisation et de développement (concrétisation des tests) en dépit du langage de description qui permet d'atteindre un premier niveau d'accélération de la phase de modélisation. Pour réduire ces efforts, nous poursuivons en outre plusieurs directions de recherche : d'une part, inférer le modèle UML/OCL (ou une partie du modèle) par utilisation de techniques de Web crawling, et d'autre part de compléter le résultat du crawler avec des traces utilisateurs afin d'obtenir une description suffisante de l'application et conserver un lien entre le modèle UML/OCL et l'application réelle en vue de simplifier la phase de concrétisation. Finalement, nous projetons d'augmenter la couverture des vulnérabilités (notamment les attaques Cross-Site Request Forgery).

Références

1. Wichers, D. : Owasp top 10. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project` (October 2013) Last visited : February 2014.
2. Doupé, A., Cova, M., Vigna, G. : Why Johnny can't pentest : an analysis of black-box web vulnerability scanners. In : Proc. of the 7^{th} Int. Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10), Bonn, Germany, Springer (July 2010) 111–131
3. Lebeau, F., Legeard, B., Peureux, F., Vernotte, A. : Model-Based Vulnerability Testing for Web Applications. In : Proc. of the 4^{th} Int. Workshop on Security Testing (SECTEST'13), Luxembourg, IEEE CS Press (March 2013) 445–452

Session du groupe de travail RIMEL

Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Automatic Discovery of Function Mappings between Similar Libraries

Cédric Teyton[1], Jean-Rémy Falleri[1], Xavier Blanc[1]

Univ. Bordeaux, LaBRI, UMR 5800
F-33400 Talence, France
cteyton,falleri,xblanc@labri.fr

Abstract. *Library migration* is the process of replacing a third-party library in favor of a competing one during software maintenance. The process of transforming a software source code to become compliant with a new library is cumbersome and error-prone. Indeed, developers have to understand a new Application Programming Interface (API) and search for the right replacements for the functions they use from the old library. As the two libraries are independent, the functions may have totally different structures and names, making the search of mappings very difficult. To assist the developers in this difficult task, we introduce an approach that analyzes source code changes from software projects that already underwent a given library migration to extract *mappings* between functions. We demonstrate the applicability of our approach on several library migrations performed on the Java open source software projects.

Software systems depend more and more on third party libraries that provide robust and efficient functionalities. Using libraries saves development time as it prevents developers to redevelop existing features. However, as both software systems and libraries irremediably evolve in their own direction, developers sometime need to replace some used libraries by another ones, for maintenance reasons. This phenomenon is known as *library migration* and has been studied and observed on Open Source Software (OSS) projects [3, 4].

When a developer has to perform a migration between two libraries L_A and L_B, she has to translate all the dependencies of L_A into dependencies of L_B. In other words, if we consider that a library provides a set of functions, she has to find for each function she use in L_A the corresponding function(s) in L_B. Such a task is known to be tedious and error prone, in particular if she does not know any of the function of L_B [1]. The main challenge is then to identify the mappings between the functions provided by the libraries.

Our proposal consists in extracting function mappings by mining existing software projects that have already performed library migrations. The main idea is to identify commits where the migrations have been done and then to analyze the changes that have been performed to extract function mappings. For instance, Listing 1.1 presents a commit where a migration has been performed between the libraries *commons.lang* and *guava*. An analysis of the commit can infer a mapping between the function *Validate.notNull(int)* and *Preconditions.checkArgument(boolean)*.

Listing 1.1: An example of migration commons.lang → guava.lang.

```
-import org.apache.commons.lang.Validate;
+import com.google.common.base.Preconditions;

public long getProblemVersion(String id) {
-    Validate.notNull(id);
+    Preconditions.checkArgument(id != null);
}
```

Several issues have to be faced to achieve our proposal. First, software projects that replaced their libraries have to be identified as well as their commits where migrations have been performed. Second, an analysis of the commits have to be performed to detect library migrations. Third, a process of knowledge extraction has to be deployed to reveal the existence of mappings.

We introduce a two-step approach to address the previous issues:

– We apply an efficient algorithm that finds the migration segments contained in a software repository.
– We use a fine-grained algorithm, based on text differencing and static code analysis (called *hunk grain*) to extract function mappings from the migration segments. It differs from previous articles that only use static code analysis (called *method grain*).

We applied our approach on a huge corpus of Java OSS projects, looking for migrations between five pairs of commonly used Java libraries. The first two ones are included in the well-known projects Apache Commons and Google Guava. Apache Commons "*is an Apache project focused on all aspects of reusable Java components*", while Google Guava "*contains several of Google's core libraries that we rely on in Java-based projects*". In other words, Commons and Guava extend or re-implement functionalities provided by the Java standard library. We focus on two migration rules within these projects: guava.io ↔ commons.io (called *I/O*) and guava.lang ↔ commons.lang (called *Lang*). The third migration rule is between two libraries that manipulate JSON documents: the standard *org.json* library, and the Google *gson* library. This rule, org.json ↔ gson is called *JSON*. Finally, the last migration rule is called *Mock* and is between two testing libraries that support the writing of tests containing *mock* objects: *jmock* and *mockito* (jmock ↔ mockito).

Regarding our fast algorithm for migration segment extraction gives excellent results. It is significantly faster than the exact approach while having the very same results. Regarding the function mappings, we obtain the results shown in Table 1. The hunk-grain function mappings extraction works well in the context of library migration. It generates mappings with a very good precision in two cases out of four.

References

1. Duala-Ekoko, E., Robillard, M.P.: Asking and answering questions about unfamiliar APIs: an exploratory study. In: Proceedings of the 2012 International Conference on Software Engineering. p. 266–276. ICSE 2012, IEEE Press, Piscataway, NJ, USA (2012), http://dl.acm.org/citation.cfm?id=2337223.2337255
2. Schäfer, T., Jonas, J., Mezini, M.: Mining framework usage changes from instantiation code. In: Proceedings of the 13th international conference on Software engineering - ICSE '08. p. 471 (2008), http://portal.acm.org/citation.cfm?doid=1368088.1368153
3. Teyton, C., Falleri, J.R., Blanc, X.: Mining library migration graphs. In: IEEE (ed.) 19th Working Conference on Reverse Engineering 2012, 15th-18th October 2012, Kingston, Ontario, Canada. pp. 289–298. Kingston, Ontario, Canada (Oct 2012), http://hal.archives-ouvertes.fr/hal-00761204
4. Teyton, C., Falleri, J.R., Palyart, M., Blanc, X.: A study of library migration in java software. Tech. rep., Univ. Bordeaux, LaBRI, UMR 5800 (2013), http://arxiv.org/abs/1306.6262

Table 1: Precision and recall of the function mappings extracted by our approach (*hunk grain*) and the *method grain* approach of Schäfer et al. [2]. The recall is computed using the union of the correct mappings found by the two approaches.

	hunk grain		**method grain**	
Rule	#Correct	#Wrong	#Correct	#Wrong
I/O	21	1	18	10
Lang	40	7	38	30
JSON	29	64	25	163
Mock	25	41	11	42
Total	**115**	**113**	**92**	**245**
Precision	**0.50 %**		**0.27 %**	
Recall	**0.85 %**		**0.68 %**	

Broken Sets in Software Repository Evolution

Jérôme Vouillon[1] and Roberto Di Cosmo[2]

[1] CNRS, PPS UMR 7126, Univ Paris Diderot, Sorbonne Paris Cité, F-75205 Paris, France
[2] Univ Paris Diderot, Sorbonne Paris Cité, PPS UMR 7126, CNRS, INRIA, F-75205 Paris, France

Component-based software architectures, maintained in a distributed fashion and evolving at a very quick pace are nowadays commonplace, in particular in the world of free and open source software (FOSS). Components are usually made available via a *repository*, and are equipped with metadata, such as *dependencies* and *conflicts*, that specify concisely the contexts in which a component can or cannot be installed.

A typical example taken from the Debian GNU/Linux distribution, is shown in Figure 1, where we can see that the logical language used for expressing dependencies and conflicts is quite powerful, as it allows conjunctions (symbol '`,`'), disjunctions (symbol '`|`') and version constraints.

```
1  Package: tesseract-ocr
2  Source: tesseract (2.04-2.1)
3  Version: 2.04-2.1+b1
4  Depends: libc6 (>= 2.2.5), libgcc1 (>= 1:4.1.1),
5   libjpeg8 (>= 8c), libstdc++6 (>= 4.1.1),
6   libtiff4, zlib1g (>= 1:1.1.4),
7   tesseract-ocr-eng | tesseract-ocr-language
8
9  Package: tesseract-ocr-eng
10 Source: tesseract-eng
11 Version: 3.02-2
12 Conflicts: tesseract-ocr (<< 3.02-2)
```

Fig. 1. Inter-package relationships of `tesseract-ocr`, an optical character recognition engine, and `tesseract-ocr-eng`, the english language pack, as found on 20 February 2012, in the testing suite of the Debian GNU/Linux distribution.

Maintaining and evolving component repositories is an important task and requires an extensive quality assurance process: besides traditional issues concerning the bugs in the code inside each component, the quality of a large component repository rests also on how well components can be combined with each other, a property known as co-installability [2].

Similarly to what happens with the source code of a single component, which is passed through regression tests to ensure that one did not re-introduce issues that were fixed before [1], we are naturally led to check whether there is any set of components which were co-installable in the old repository, but become non co-installable in the new release of the repository. Indeed, new incompatibilities are much more likely to be bugs than long standing incompatibilities. Besides, by comparing two versions of a repository, it becomes possible to provide a detailed explanation of which changes made the set of components non co-installable.

Sets of components with this property, which we call *broken sets*, are particularly damaging in the evolution of a repository: their existence means that there may be perfectly functional deployments based on the old repository that will be disrupted as soon as one tries to upgrade their components to the version in the new repository. The configuration in Example 1 is a real world example of such a broken set: the `tesseract` optical character recognition program, which is split in several related packages[3], was perfectly functional before February 20^{th} 2012, but on that day, the introduction of the updated english language pack `tesseract-ocr-eng` made the installation of this program temporarily impossible for the English language.

[3] Essentially, `tesseract-ocr` for the core engine, and `tesseract-ocr-`*lang* for all supported language *lang*

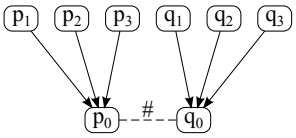


Fig. 2. Many minimal broken sets, one minimal explanation

Broken sets need to be identified early in the evolution process, and fixed well before the release of the new repository, but finding broken sets in repositories whose size is in the tens of thousands of components is a daunting task. Besides, there may be exponentially many broken sets, and listing them all would be both computationally unfeasible and of no use for a quality assurance team, which would be flooded under the error reports. Indeed, consider the configuration in Figure 2. We have three packages depending on $\mathtt{p}_0$ and three packages depending on $\mathtt{q}_0$ (arrows). Adding a conflict between $\mathtt{p}_0$ and $\mathtt{q}_0$ in the new repository (dashed line) makes it impossible to install any longer any of the packages $\mathtt{p}_i$ together with any of the packages $\mathtt{q}_i$. Hence, adding a single conflict easily results in a quadratic number of broken sets.

We have been able to find a highly efficient algorithm that solves the problem by finding a very small subset of the broken sets, which subsume all the others, and is close to *minimal*. Concretely, in the example above, only the pair $\mathtt{p}_0$ and $\mathtt{q}_0$ is reported. This work has been presented in detail in [3]. It builds on top of the theoretical framework developed in [2], which is based on formally certified semantic preserving graph-theoretic transformations. We have developed a tool that implements this algorithm and finds such minimal problematic configurations on real component repositories in a few seconds; it also provides very concise *explanations* that allow to identify the origin of the problem. We found several such issues in the evolution of the Debian distribution using the tool.

References

1. Orso, A., Harrold, M.J., Rosenblum, D., Rothermel, G., Do, H., Soffa, M.L.: Using component metacontent to support the regression testing of component-based software. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01). pp. 716–. ICSM '01, IEEE Computer Society, Washington, DC, USA (2001), http://dx.doi.org/10.1109/ICSM.2001.972790
2. Vouillon, J., Di Cosmo, R.: On software component co-installability. In: SIGSOFT FSE. pp. 256–266 (2011)
3. Vouillon, J., Di Cosmo, R.: Broken sets in software repository evolution. In: International Conference on Software Engineering (ICSE) (2013), http://www.pps.univ-paris-diderot.fr/ vouillon/publi/upgrades.pdf

Generic Name Resolution for Specific Language Models

Yuriy Tymchuk*, Benjamin Arezki†, Gustavo Santos‡, Rafael Durelli§,
Anne Etien†, Nicolas Anquetil† and Stéphane Ducasse†
* REVEAL - Faculty of Informatics, University of Lugano (USI)
Via G. Buffi 13, CH-6904 Lugano, Switzerland
Email: yuriy.tymchuk@usi.ch
† Inria Lille-Nord Europe - University Lille 1, LIFL CNRS UMR 8022,
40 Avenue Halley, Park Plaza, 59650 Villeneuve d'Ascq, France
Email: *firstname.lastname*@inria.fr
‡ Department of Computer Science UFMG, Belo Horizonte, Brazil
Email: gustavojss@dcc.ufmg.br
§ Software Engineering Laboratory - LabES
University of São Paulo - ICMC/USP, São Paulo, Brazil
Email: rdurelli@icmc.usp.br

***Abstract*—Analysing a software system supposes two preliminary tasks: parsing the source code and resolving the names (identifiers) it contains. The parsing results in an Abstract Syntax Tree (AST) representing the source code. Name resolution maps all the identifiers found in the code to the software entities they refer to (variables, functions, classes, ...). If there are open-source solutions for some popular programming languages (*e.g.,* JDT for the Java language), these two tasks can impose a significant burden on multi-language platforms (*e.g.,* Eclipse, Spoofax, Rascal, Cast, Synectique) where a parser with name resolution must be implemented for each language. For the parser, one may use a grammar of the language and a parser generator tool. For name resolution, solutions are *ad-hoc* and one must develop them by hand. We encountered this problem while working on our Moose analysis platform. As a solution, we propose in this paper, a generic name resolution algorithm that is based on an AST metamodel (similar to ASTM, proposed by the OMG). One part of the solution comes from decomposing name resolution into two phases: First, looking-up for candidate entities that could map to a name use; second selecting among these candidates the entity that actually maps to the name. Given some precautions, the first part can be made generic to most programming languages we know of. The second part is more language dependant, but we identified generic access right patterns that can be re-used in different languages. We discuss implementation of our solution for three languages: Cobol, Pharo (a Smalltalk dialect), and Java.**

I. INTRODUCTION

Nowadays almost all modern IDE provides editor services such as reference resolving while writing code to identify where a given symbol has been defined, constraints checking to highlight duplicate definitions, use before definition, unresolved reference... or code completion by proposing valid identifier in a given context. In parallel, in maintenance, lot of metrics or program analysis approaches rely on the identification of dependencies between concepts. All of these services rely on name resolution also known as symbol resolution.

Providing such services for programs written in a language not currently supported (old languages like Cobol or new ones as any Domain Specific Language (DSL)) requires as a first essential step to develop a name resolution algorithm for this language. Dealing with all the specific cases of the language may put a damper on the developer's enthusiast that will choose tools already providing name resolution algorithm for the given language and plug the new services to this tool (*e.g.,* [Dol05]). Such a solution has the major advantage to avoid writing the name resolution algorithm for the new language. Nevertheless, it has the major drawback for the new service to be dependent to another tool. Moreover, if such tools exist for new languages such as Java, there is not any for old languages like Cobol or for DSL.

Traditionally, name resolution algorithms rely on abstract syntax tree (AST) as representation of the program. However the types of AST nodes depend on the language parsed. This harms reuse of tools and analysis. In this paper, we propose a generic name resolution algorithm that can be easily adapted to a new language. For this purpose, we adopt a model-based approach and provide an AST metamodel. The algorithm relies on the scope associated to each element of the AST and generic scoping rules. We proceed in two steps, lookup and candidate selection. When a symbol is found, the software entity to which it refers is looked for in its scope and recursively in the scope of its scope. This basic algorithm is mostly independent of the language. In a second step, we select the appropriate entity in the list of entities returned by the lookup. This part is more specific to the language, but generic patterns can be found.

This model-based generic name resolution algorithm has been experimented on Cobol, Pharo [BCDL13] (a Smalltalk inspired language) and Java.

The paper is organized as follows. In section II, we describe the existing issues in name resolution. In Section III we describe the abstract syntax tree metamodel. In Section IV we detail the generic name resolution algorithm. In Section V, we adapt this algorithm for Cobol, Pharo and Java. In Sec-

tion VI we discuss related work and finally, in Section VII we conclude the paper.

II. Issues in Name Resolution

Name resolution amounts to linking a name (an identifier) in the source code to an entity of the program: in the expression i++, the symbol i refers to a variable of the program that must be incremented by one. In some cases, it might be possible to infer the kind of entity one is considering. For example, in the above expression, i should be a variable.

Finding to what entity the name refers, first depends on the scoping policy of the language: lexical or dynamic scoping. In *dynamic scoping*, the entity to which an identifier refers is looked for in the execution stack. This is used by languages as Dynamic Lisp or Perl. This can only be resolved at execution time and we will therefore not consider dynamic scoping in this paper.

In *lexical scoping* the name refers to the closest entity in the current lexical environment. Lexical environments are created by some language constructs like packages, classes, or functions. They are nested: a method scope is nested within its class scope, which is nested within its package scope. Most current languages use lexical scoping: C, Pascal, Java, etc.

The basic rule for name resolution in lexical scoping is to look for the entity in the current scope, e.g. a variable name will be first searched in the scope of the function within which it appears. Different conditions must be checked. First the entity must match the identifier, that is to say have the right name. Second it must match the kind of entity (variable, function, method, class) if it is known. Third in some languages the type of the name must also matched (for example in Java methods are matched on their signature). If a matching entity is not found in the current scope, one searches recursively in the containing scope.

But this algorithm has many variations according to the programming language. Here are the specificities of some languages:

- In Cobol, there is one lexical scope, the program, for variables and labels (GOTOs), and a dynamic scope for some instructions like COPY <a-copybook>[1] or CALL <a-program>.
  Incidentally, Cobol is not case sensitive, the name MyParagraph matches the paragraph label MYPARAGRAPH (typically Cobol programs are written all uppercase).
- In C, there is a global scope for the entire program and a local scope for each function. Functions can only be defined at the global scope, variables and types can be defined globally or locally.
  The preprocessing instruction include (actually not part of the compiled language) depends on the environment to find the appropriate included file. Again this is akin to dynamic scoping as one cannot decide from the source code what the included file will be.
  C is case sensitive, variables dog, Dog, and DOG[2] are all different.
- Pascal has the same global and local scopes as C, but all scopes can include functions (or procedures), types, and variables. This means, unlike C, a function can define a function.
  Pascal also has the with instruction that creates a temporary scope for a given structured type. Assuming there is a structured type rec containing an attribute a (*e.g.* "type rec = record a:int; ... end;") and a variable v with this type ("var v:rec;"). One normally accesses the attribute by writing v.a, but the with instruction creates a temporary scope within the variable v such that "var a:char; v:rec; with v do writeln(a);" will print the content of attribute v.a, instead of local variable a.
  Like Cobol and differently from C, Pascal is not case sensitive.
- In OO languages, on top of lexical inclusion of scopes (method scope included in the scope of its class), inheritance also defines an inclusion of scopes: the scope of a subclass is included in the scope of its superclass. Thus, if the subclass does not define a method, it must be looked for in the superclass definition.
  OO languages also assume two implicit variables, this (or self) and super, that are never defined but accessible within the scope of a class.
- Java has a global scope for packages, and packages are not included one in the other (although their names suggest so). Java introduces local scopes with packages, classes (interfaces, annotations, enums), methods, and statement blocks. Packages can only be defined in the global scope, classes can be defined at all scopes, methods can only be defined in a class scope, variables can be defined at class scope (attributes), and method scope. A subclass can redefine (overload) an inner-class defined in the superclass.
  Access modifiers (public, protected, private, and default package) affect the scope of a definition.
  Classes and variables must be uniquely named within their enclosing scope, methods must have a unique signature (including return type and parameters' types) within their enclosing scope (class), but a subclass can define a method with the same signature as a method in a superclass (overloading) However, it is an error for a subclass to overload an attribute (define an attribute with the same name as an inherited attribute).
- C++ has rules very similar to Java. However, C++ accepts functions which are defined out of a class scope. There is no default package access modifier. There is a friend access modifier that bypasses the private and protected modifiers (everything becomes public for a friend).
- Pharo (a Smalltalk dialect) has a global scope for packages and classes, and local scopes introduced by classes, methods and blocks (closures, similar to lambda functions). Only variables and blocks can be defined in a local scope. Packages can be defined only at the global scope; classes can be defined only within packages; variables can be defined at all scopes (global, class, method, block); and blocks can be defined only in methods and blocks. Class attributes are always protected; methods are always public. Pharo is dynamically typed, which means that variables have no declared type. As a consequence name resolution

[1] Similar to an include in C, it inserts the content of the copybook, a source file, into the code.

[2] See: http://rosettacode.org/wiki/Case-sensitivity_of_identifiers, last consulted on 10/10/2013

cannot be restricted on the type of variables, or parameters.
- Ruby, Objective-C, Pharo, C# offer package extensions. Packages can add methods to classes of other packages (*extension*). For example package p1 introduces class C, and package p2 extends C with a new method m that was not originally defined. An extended method (m) is not available as long as the package (p2) that introduces it is not loaded, and this even by the class where it is inserted. In practice, such an extended method can only be safely called from the package (p2) that introduces it.

One can assume that every individual programming language will have a set of specific constraints or rules that affects how name resolution works. It is clear from this list that a truly generic name resolution mechanism cannot be defined. There are too many rules depending on the language. Nevertheless, some characteristics are common to every languages. Our generic resolution algorithm exploits these common points while letting some parts to adapt. Before detailing the algorithm, we present the AST metamodel that enables a unified representation of programs written in various languages and serve as base to our approach. Such AST Metamodel is an effort similar to Famix [DAB+11], the Dagsthul Middle Metamodel [LTP04] or ASTM [AST11] for structural source code representation but at the AST level.

III. THE FAST ABSTRACT SYNTAX TREE METAMODEL

As it is common in name resolution, our algorithm relies on a representation of the program to analyse as an AST. This AST is a model of the program that follows the specifications of a metamodel (called FAST). This metamodel is important because it also constrains what the name resolution algorithm can do in a generic manner. For this reason, it is important to discuss our AST metamodel and its genericity.

Our metamodel has the same goals as ASTM [AST11], the standard defined by the OMG. However, we adopted a radically different approach in our definition. Whereas ASTM is defined with a focus on completeness, FAST focuses on genericity.

A. ASTM drawbacks

ASTM is composed of two parts: a core specification, the Generic Abstract Syntax Tree Metamodel (GASTM) and a set of complementary specifications that extend the core, called the Specialized Abstract Syntax Tree Metamodels (SASTM). GASTM defines a core set of modeling elements that are common to many programming languages. In fact, GASTM is the union of concepts from almost all the languages. It considers object-oriented programming languages with concepts such as ClassType, ExceptionType or AccessKind. It has also concepts specific to procedural programming languages such as JumpStatement or Pointer. In total GASTM of the OMG defines 188 concepts. With so many concepts, the metamodel is hard to understand. Moreover, it is difficult to develop an algorithm adapted to all languages because of the conceptual difficulty of dealing with so many different concepts with definitions sometimes unclear. For example, we saw that although Java and Pharo, both have packages, classes and methods, the rules for scoping are different. This implies that having all the concepts in the metamodel does not prevent us from having to specialize them for each language.

B. FAST metamodel

Our metamodel, FAST, is defined as the intersection of all programming languages. By doing this, we have a metamodel with less than 20 concepts that can still accommodate the same large spectrum of programming languages while being much easier to apprehend and extend.

The result is presented in Figure 1 (white boxes). It starts with an abstract concept FASTEntity serving as the root class of the FAST metamodel. A FASTEntity may have a scope FASTScope or not. Four types of entities are distinguished:

- A FASTBehaviouralEntity is an abstract concept for all entities having a behavior like methods or functions. Such entities may be named (in most cases) or not (*e.g.,* lambda-functions).
- A FASTStatement is also an abstract concept. *IfStatement* or *LoopStatement* does not exist in all languages. For example, they do not appear in Pharo or Cobol. These two languages do offer the possibility of branching and iterating, but not in the form of the "traditional" statements we know in Java, or C.
  Some languages may also offer specific loop statements (as the "extended for" in Java).
  A *ReturnStatement* is probably universal in languages that accept subprograms, but it will not always return a value (*e.g.,* Cobol). Similarly, an assignment may be considered as an expression-statements in many languages (meaning it returns a value), whereas it is a simple statement in other (*e.g.,* Pascal or Cobol). In the end we chose to be conservative and did not include any specific statement in the core FAST.
  Statements can be FASTStatementBlock, for example to represent the body of a function.
- A FASTExpression is an abstract concept that has a value. Again it would be difficult to try to be too specific here as even arithmetic expressions can be treated in different ways by different languages (*e.g.,* Pharo, Lisp). We believe some literals (FASTLiteral) are truly generic and included them.
- A FASTNamedEntity represents an identifier and is probably the most truly generic concepts of all as it is hard to imagine a programming language not using any identifier.

C. FAST and Famix Interconnection

FAST is an extension of the Famix metamodel [DTD01], [DAB+11]. Indeed, the FAST metamodel (that stands for Famix AST metamodel) relies on the Famix concepts to design the high level code structure.

Famix offers a structural representation of the source code. It has packages, classes, methods, functions, variables (parameters, attributes, local variables, etc.). Typically in Famix, one also stores relationship between entities (invocations between functions, accesses to variables, etc.) but we are not using this part in this work as establishing these relationships already requires name resolution. So we work with a simplified Famix model. The only relationship we require between the Famix entities are the structural ones (parent or container), which are easily built by traversing the AST top-down.

A FAMIXSourcedEntity models any construct in a source program. Moreover, it is the superclass (root class) of all

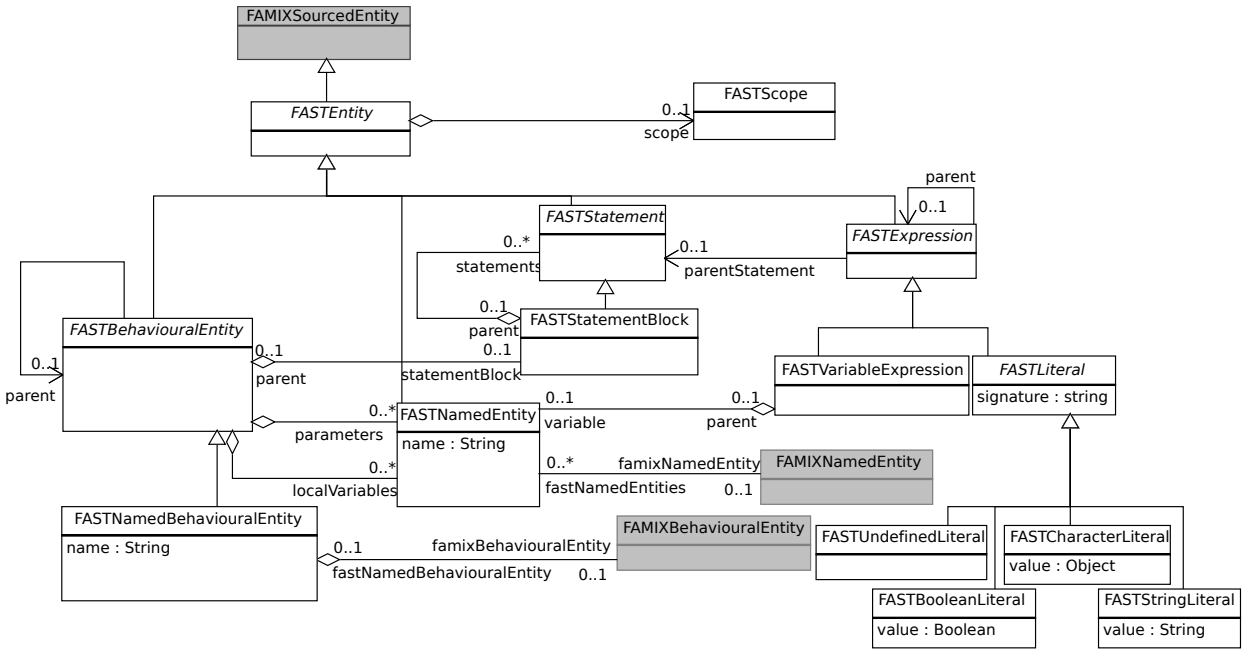


Figure 1. The FAST metamodel (in white boxes), dark boxes represent concept from the Famix metamodel, used as symbol table.

source code entities and their relationships. The FAMIXNamedEntity, FAMIXBehaviouralEntity and FAMIXSourceEntity concepts (in light grey in the figure) enable to establish links with the Famix metamodel.

All these concepts are very generic and exist in any language. They do not capture the specificity of any language or even any paradigm (procedural, object, list ...). Our name resolution algorithm as shown in the next section relies only on these concepts making it generic.

Typically, an AST represents the entire source code, including import of packages, definition of classes, etc. In this paper, we simplify the presentation by discussing only the content of methods or programs, and leave the structural part to Famix as discussed above. This does not limit the generality of the discussion, as resolution of names appearing in the definition of the class (inherited superclasses) or the attributes is not different than those appearing within a method.

IV. Generic Name Resolution

We use Famix as the "symbol table" of our algorithm. This means that names found in the AST (in a FAST model) are linked to entities defined in the Famix model. As explained above, the Famix model contains the entities of the program and their structural relationships (which entity defines which other).

We do name resolution for program analysis purposes. In consequence, we assume the analysed program is valid. This means, we assume a name does resolve to some entity, we do not check for errors. It is not clear to us at this moment whether this represent a simplification of the problem or not (it does not seem so).

A. Generic Name Resolution Algorithm

The generic algorithm is decomposed in two parts: lookup and selection as illustrated in Figure 2. The *lookup* is the most generic part. From a name in a the AST, it generates an ordered list of candidate entities that it could resolve to. Given this list of candidates, and the referring entity, *selection* finds the first candidate that answers all the criteria of the programming language. This part is language dependent as explained in the Section II, however, generic patterns can be found.

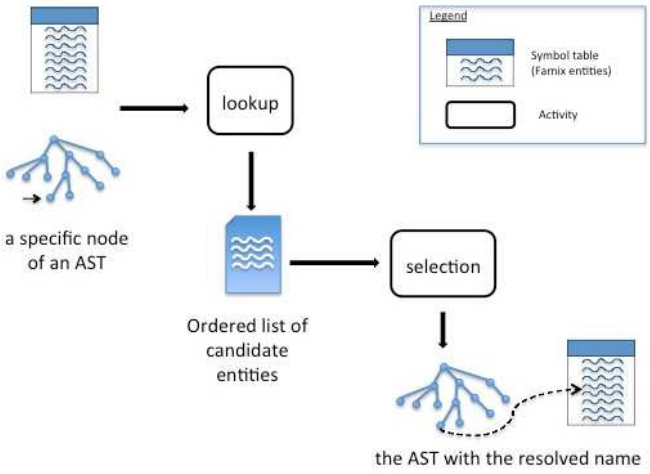


Figure 2. Sketch of the algorithm

B. Generic Lookup Algorithm

The lookup algorithm is based on the notion of scope. A scope is a partition of the system in subsets. In a scope, a name (sometimes associated with the kind of entity) always refers to the same entity whereas the same name may refer to different entities if it is encountered in different scopes. Sometimes, the kind of entity must also be taken into account, in [KKWV13]

this is treated by creating different namespace for each entity kind.

Scopes form a containment chain. For example, the scope of a FASTStatementBlock may have for parent the scope of a method definition, FASTBehaviouralEntity, that has in turn for parent the scope of a class definition or one of its extension. Not all AST nodes have a scope (statement or expression typically don't), so the lookup goes up the AST to find the first node with a scope (like a statement-block (*e.g.,* statements between curly brackets in Java), a function, a class). Knowing which AST node has a scope or not is language dependent and decided by the parser of that language.

Scopes for classes present a specificity. They are different from other scopes in that they may have several parents: the scope of their containing package plus the scopes of each superclass (and/or interface, traits, etc). We defined a generic class scope for which candidate entities are looked first in the scopes of the superclasses (and/or interface, traits, etc), in the order of their declaration and then in the scope of the containing entity. The rational is that, for example, if *aVariableName* is an attribute of a class, even if it is inherited, it must have priority over a global variable with the same name defined in the package of the class. We believe this is a generic rule although we did not check all OO languages[3].

The input of the name resolution algorithm is the name of a given AST node. In details, the *lookup* searches entities with this name and specified in the current scope of this AST node. Then it recursively searches more candidate entities in the containing scopes. Candidates are returned in order of proximity; candidates in the immediate scope appear before candidates from a parent scope. This rule is one of the foundations of lexical scoping and is therefore generic. The output of this first part is an ordered list of candidate entities.

Candidates are matched on their name, and the kind of the entity (a function, a variable, etc.) Mapping of the name and kind of the entity is delegated to the AST node that contains the name. For the name this is because some languages are not case sensitive (*e.g.,* Pascal) and thus given the opportunity to define nodes with an appropriate name matching test. The default behaviour (*e.g.,* in FASTNamedEntity) is to match on equal name. For the entity kind, each node must know what it can accept. For example, a FASTMessageSend contains the name of the method called (in selector, see Figure 4). This node will only accept as candidate method entities. On the other hand, a FASTNamedEntity will typically accept variable entities or type entities.

The nodes of core FAST, are provided with default behavior that is as generic as possible (see above). Languages requiring specific behavior will subclass these core nodes. In this sense, this part of the algorithm is as generic as it can be, by delegating a part of the work to the AST nodes which are created by the parser for the language.

Again for OO languages, the lookup for attribute and method names has a different rule. Namely, when one accesses a member of an instance (attribute or method), the name of this attribute must be looked-up in the scope of the class of the instance and not in the enclosing scope of the sender. We give an example of this in Section IV-D

[3]It does matches Java, C++ and C# behaviors

Note that this implies one also needs a type inferencer. We will not discuss this issue in this paper.

C. Generic Selection Algorithm

Given an ordered list of candidate entities, the *selection* algorithm will return the first candidate that matches the name resolution rules of the language. This part is dependent of the language but may be done more generic by the use of different selectors, for example to deal with the cases of public, protected, or private entities.

Each element of the input ordered list of candidate entities is successively studied. By construction of the lookup algorithm (see previous section), it matches the searched name and the expected kind. We now check if the (Famix) entity that uses the name may access to each candidate entity or not. Each non accessible entity is eliminated from the list of candidates. The first candidate satisfying this accessibility condition is the searched entity; it corresponds to the output of the name resolution algorithm.

To check the accessibility of the entities, several rules ("selectors") depending of the used language have been implemented. Mostly, these selectors correspond to the visibility rules associated to the access modifiers (public, protected, private, default-package). Thus, the access modifier of a candidate is analyzed and the corresponding accessibility rules are checked through the call to the appropriate selector.

- For public entities, the FASTPublicSelector is used. It accepts all candidates. As a candidate returned by the lookup, the entity matches name and kind; being public, it also is accessible by all entities and thus the one associated to the input AST node.
- For private entity, the FASTPrivateSelector is used. It refuses all candidates except if the input AST node is in the same class as the candidate.
- The FASTProtectedSelector checks the accessibility rule when the access modifier is protected. It verifies that the input AST node is in a subclass of the class owning the candidate, if not it eliminates the candidate.
- The FASTDefaultPackageSelector implements the default package accessibility rule. It verifies that the input AST node is in the same package as the candidate.

These visibility rules and thus their implementation are shared by several languages and not only Java. The main example is the FASTPublicSelector which is appropriate for all languages that do not have access restrictions (*e.g.,* Cobol, pascal, C, Smalltalk, etc.)

For other languages their could be a need to define a new specialized selector, for example in C++, one would need to implement the accessibility rules relative to friend classes. We do not expect that many new selectors need to be defined.

D. Illustrating Example

We now explain on a Java example, how the generic algorithm works (see Figure 3). We consider a superclass ClassA that defines an inner class InA, itself with an attribute att. We also have SubA, a subclass of ClassA, with a method m(). This method defines a local variable v of type InA and accesses the attribute.

We will explain how our algorithms resolves the name v.att found at the end of the right listing. First, some name

```
class ClassA {
  class InA {
    int att;
  }
}
```

```
class SubA
  extends ClassA {
  void m() {
    InA v = new InA();
    v.att = 0;
  }
}
```

Figure 3. Code sample for Java name resolution

resolution occurs at the beginning of the method for the name "InA". The lookup algorithm searches for it in the enclosing scope, that of the method body. It finds nothing. Enclosing scopes are then looked-up: the method itself, the class SubA, and the superclass ClassA. In A, a candidate entity is found, matching both name and kind of "InA". There may be another "InA" defined in the scope of ClassA, but the selection algorithm will take the first that is acceptable in this context. If InA is either public, protected of default-package, it can be accessed by a method in SubA. We will assume it is the case here.

Then on the next line, "v" must be resolved. The lookup algorithm searches for it in the scope of the method body. It finds a candidate. Enclosing scopes (method itself, class SubA, superclass ClassA and package) are also looked-up but no other candidate is found. The selection algorithm considers the candidate entity and finds that it is suitable as we are in the same statement block.

Finally, from the variable entity we know the type of v. The name "att" is resolved in the scope of this type which means the lookup algorithm looks in the scope of InA (and finds a candidate), then in the scope of the superclasses of InA and the packages. The selection algorithm will first look at the InA.att candidate entity and will find it acceptable, therefore it stops at this point.

V. Practical Application

We now explain how FAST and the generic algorithm for name resolution can be applied to three different languages: Cobol, Pharo, and Java. Applying to a new language actually means:

- Specializing the FAST core metamodel by adding elements necessary to represent the concepts of this language;
- Writing a parser for the language and generating the AST;
- Deciding which FAST elements specific to the language have a scope, and;
- Choosing (eventually by implementing it) the selection strategy *i.e.,* the accessibility rules.

For each of the three languages above, we will now discuss these points, except for the parsing step which falls outside the scope of this paper.

A. Cobol

Cobol is a simple language when it comes to name resolution but a difficult one for parsing. Because it pre-dates the theory on language parsing, Cobol does not fit well in the general understanding of programming language we are now accustomed to.

Specializing the AST. We actually did not implement a complete Cobol parser and AST[4]. As many, we use an island grammar, we only extract conditional and looping statement, calls to sub-programs (PERFORM <paragraph>), and GOTOs. All these elements are specializations of FASTStatement whereas a Cobol program is a specialization of FASTBehaviouralEntity. Paragraphs are very crude sub-programs as they do not accept any parameter and do not return any value. All communication is done via global variables.

Determining elements with scope. As explained in Section II, data definitions or labels (to mark paragraphs) are defined globally for a program. Therefore the only element that needs to be associated with a scope is the program itself. Moreover, programs define several parallel namespaces. Thus, data and paragraphs can have the same name. This issue is already handled by our lookup algorithm that also considers the kind of the entity when searching for candidates to match a name. Instructions manipulating paragraphs (GOTO and PERFORM) are different from those manipulating data and this is enough to distinguish the kind of the entity a name must be referring to.

Accessibility definition. For the selection strategy, only the generic FASTPublicSelector is used. It imposes no restriction on an entity accessing another one. This corresponds precisely to Cobol semantics where any paragraph can use any data defined in the program or call any other paragraph in the same program.

In addition, Cobol program may call another Cobol program (outside its own lexical scope therefore), but this better fit dynamic scoping as the actual program that will be called depends on settings of the environment. We do not consider this part of our resolver, although it could be implemented by creating an additional scope representing the environment within which all the available programs would be "declared". Note that this would only work if each program is guaranteed to have a unique name which is not an actual constraint in reality.

B. Pharo (Smalltalk)

Another language that we deal with is Pharo, a Smalltalk inspired language [BCDL13]. Here, it presents the advantage of being an object-oriented language (as opposed to a language like Cobol), and much simpler than Java.

The specificities of Pharo that we identified are:

- Only methods are part of the compiler AST. Classes are created by sending a message to their superclass, indicating the name of the subclass and its attributes (instance variables). Still methods conceptually belong to their classes.
- A "block" definition is a closure (similar to lambda-functions). Pharo blocks may have parameters, define local variables, and contain statements.
- Everything is an object, and for example, the conditional and loops are implemented as messages sent to the condition (a boolean object). Thus there is no *IfStatement* or *LoopStatement*.
- Pharo has an extension mechanism, whereby a package can add a method to a class defined by another package.

[4]Cobol has close to 500 reserved words! [Cob94, Appendix E, pp.471–74]

For us this is completely transparent, if the extending package is loaded in the environment, the method will simply appear as belonging to the extended class; if the extending package it is not loaded, we will not even know that this extension exists.
- Pharo is dynamically typed, which means the name resolution algorithm does not use type information.

Specializing the AST. Smalltalk has a very simple grammar and AST, with only 13 concepts to add to our generic AST. These concepts enrich the notions of statement, expression and literal that where restrict to the minimum in the core FAST metamodel. They respectively extend FASTStatement, FASTExpression and FASTLiteral. For example, FASTReturnStatement, FASTExpressionStatement, several types of expression and literals have been added. FASTMessageSend is an important concept since, as previously explained, it enables to express *IfStatement* or *LoopStatement*. Moreover, the notion of block definition has been introduced with FASTBlockDefinition. Figure 4 presents the resulting metamodel where the dark grey nodes are the core FAST one (generic, language independent, AST), and white nodes for the Pharo extensions.

Determining elements with scope. In Pharo, classes may be defined in packages, but these ones are not namespaces, but only organizational units. This means that even in different packages, two classes cannot have the same name. In other words, packages do not have FASTScope and classes are defined globally. Methods are defined locally in classes. Because block are closures, and not lambda-functions, their scope is included within the lexical scope of their parent (*e.g.,* a method) and they can, thus, access its variables. Variables may be defined in the global scope (Pharo environment), implicitly in a class scope (self, super), in methods or in a Pharo block. Following these rules, the nodes that have scopes are: a global scope, classes, methods (FASTNamedBehaviouralEntity), and blocks.

Accessibility definition. As for Cobol, the selection algorithm uses the generic FASTPublicSelector: Classes are all defined in the global scope and accessible from everywhere; methods are defined in a class scope, but there is no access restriction; and variables are defined within local scopes, that can be found by going up the chain of nested scopes from an instruction to the global scope.

C. Java

The last language we experimented with is Java. Like Pharo, it is an object-oriented language but it has more complex access rights and concepts such as innerclasses.

The specificities of Java that we identified are:

- Packages are namespaces. Two different classes may have the same name but not in the same package.
- Classes may define other inner classes. The scope of a class may be another class and not only a package. We illustrated in section IV-D that our generic algorithm can handle such a situation.
- Statement blocks (between "{}") can introduce variables and contain statements.
- Arbitrary statements can define anonymous classes. The scope of a class may thus be any arbitrary statement.
- Four different access modifiers (public, private, protected and default package modify the visibility of elements the one for the others and thus affect the scope definition.
- Java has an import of class mechanism. It enables the direct access of this added class from the entity defined in file where it is imported.

Specializing the AST. Java has many different statements and more complex grammar rules for expressions to take into account for operator precedence. This results in a much larger extension of core FAST than for Pharo with about 50 additional elements, mostly as subclasses of FASTStatement and FASTExpression. In contrast, in Pharo, arithmetic or logical operators are methods and therefore appear in the AST as FASTMessageSend. They do not result in additional nodes in the FAST definition. Figure 5 presents an extract, relative to the statements and their relationships with expression, of the resulting AST. As expected, Pharo and Java share some constructs.

Determining elements with scope. In comparison to the other two languages, more nodes can have a scope: Class, and method nodes have scope, like in Pharo, but also the package node, BlockStatement, the ForStatement (the for can define a variable in the initialization part), the "EnhancedForStatement" (use of an implicit Iterator over a Collection), and the import statement.

For-loops can even introduce two scopes, one for the statement itself and a second for the block statement in the body of the loop.

Imports also create scopes. The statement import some.package.AClass; in a Java file implies that the name AClass becomes accessible to the entities defined in this file. We propose to deal with this by considering that this instruction creates a nested scope, containing the definition of AClass, and within which the rest of the file will be inserted. The parent scope of the import is the package within which that statement appears. Therefore this falls down to the Java parser and knowing that import is a statement introducing a new scope.

The fact that scopes can or not contain class declaration is transparent for us as we assume the program is correct and we only want to understand it.

Accessibility definition. For the selection phase of the algorithm, we define a special JavaAccessChecker. Given a candidate entity returned by the lookup, and the entity where the name is used, the JavaAccessChecker delegates access checking to the four checkers previously defined according to the access modifier for the candidate entity, either public, protected, private, or default-package.

VI. RELATED WORK

In [KKWV13], the authors identify recurring patterns for name binding and introduce a metalanguage to specify name binding in terms of namespaces, scopes. They provide a language parametric algorithm for static name resolution during compile time. Their approach differs from ours in a number of ways.

First they aim at making the name resolution rules of a language explicit through a DSL they describe. Our goal was primarily to have a generic name resolution solution and we opted for a more programmatic "description" of the rules. However, by decomposing the overall name resolution task in two subtasks and identifying further subparts of these tasks

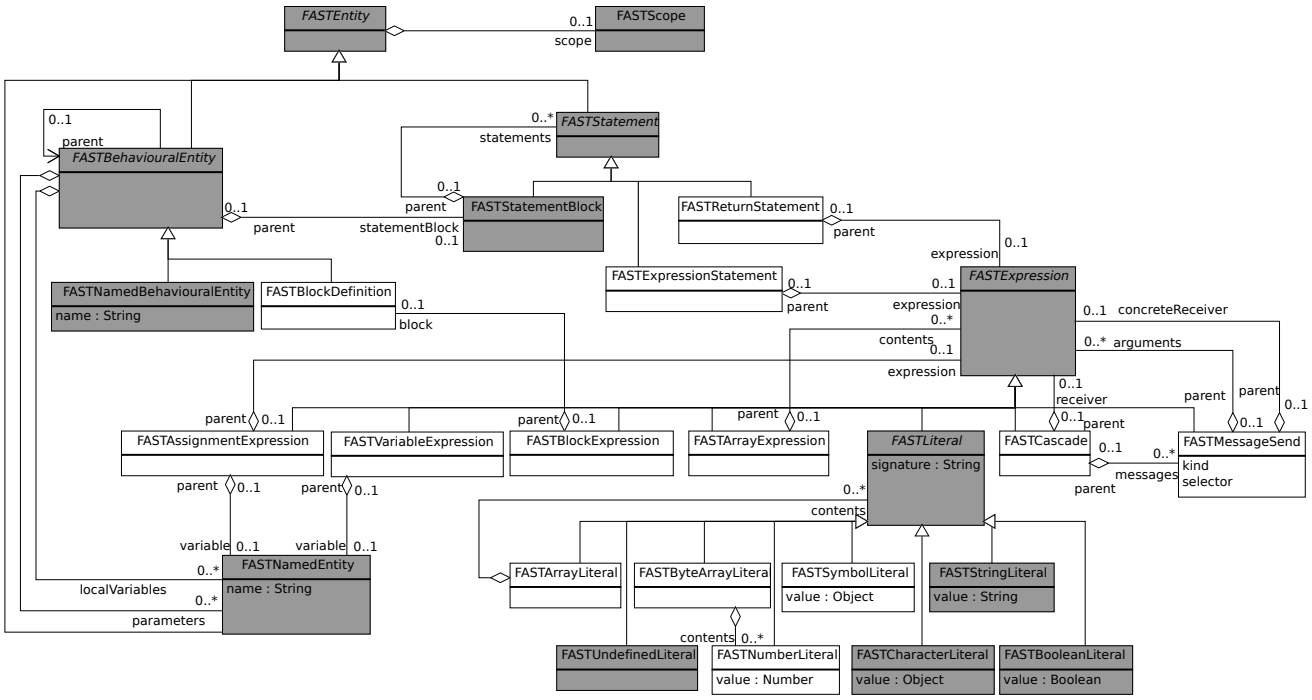


Figure 4. Pharo AST metamodel in FAST (white boxes), the dark boxes are concepts from the core FAST metamodel (see Figure 1)

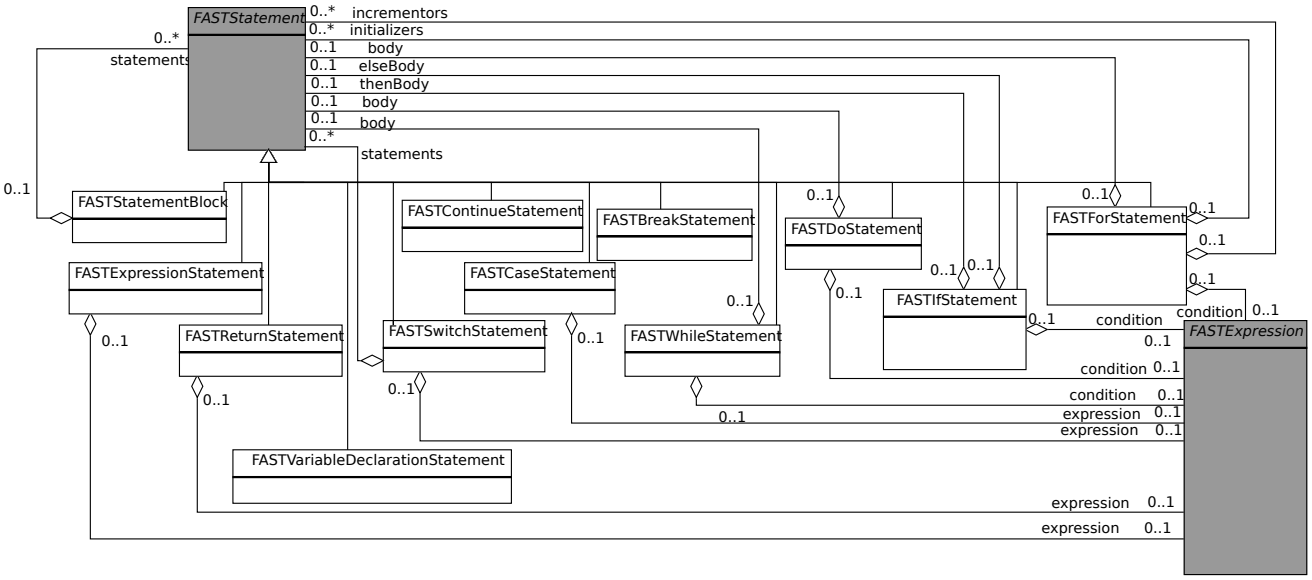


Figure 5. Extract of the Java AST metamodel in FAST (white boxes), the dark boxes are concepts from the core FAST metamodel (see Figure 1)

(*e.g.,* the various selectors described in Section IV-C) we made some steps in the direction of having the name resolution rules of a languages being described at a higher level of abstraction than just raw source code.

Second, interestingly Konat *et al.* also decompose their solution into three subtasks that are slightly different than ours. During the first phase all definitions, are mapped to an entity. We do not consider this phase, as it is very straightforward and independent of the language. For us definitions are mapped to entities defined in our Famix metamodel. In the second phase, they do type inference, something that we explicitly left outside of the scope of this paper (see Section IV-D). What we describe in this paper seems, therefore, to correspond to their third phase. We have decomposed this into two more detailed steps.

Third, they have different usages than ours. Their solution is integrated into an IDE, with code edition, code highlighting, compilation error checking, refactoring, etc. Among other things, this implies that they must deal with incomplete and erroneous programs. As mentioned in Section IV, for now, we are assuming a complete and valid program (that compiles). It is not clear to us at this point whether this is a

significant restriction (whether it makes any difference). This is something we did not test.

Finally, although they claim language independence, all the examples given in the paper are focusing C++. Actually dealing with different languages does imply some amount of tweaking.

In [BPM04], the authors propose the DMS Software Reengineering Toolkit, a generalized compiler technology. Their approach and ours share the same purpose: providing a generic name resolution system. However, DMS relies on a representation of the AST as a hyper graph and not as a model as in our approach. Furthermore, as far as we understood from the paper, the look up function is a parameter of their algorithm that the developer must provide. By decomposing the algorithm into lookup + selection, we can reuse more parts and we expected that our look up algorithm should be already generic enough for new languages, and our selection algorithm would need to be extended in very few language instances.

In [KRV10], the authors adopt a textual modeling approach and propose a framework named MontiCore for the compositional development of textual DSL and their supporting rules. Concrete and abstract syntaxes are defined using the MontiCore grammar. They provide default implementations for simple resolving problems like file-wide flat or simple hierarchical namespaces.

Similarly to [BPM04], more complex resolution algorithms are let to the responsibility of the programmers. In [JBK06], the authors propose a similar approach based on a DSL named Textual Concrete Syntax to provide a concrete syntax for an abstract syntax given as a metamodel. So contrarily to [KRV10], abstract and concrete syntaxes are defined in two different languages. Concerning the name resolution algorithm, only simple cases are tackled: unique symbol tables or nested ones. The way the tables are nested is not described in the paper.

The OMG has defined two metamodels to specify concrete and abstract syntax, KDM (Knowledge Discovery Metamodel) [PCdGP11] and ASTM [AST11] respectively. KDM specifies a set of common concepts required for understanding existing software systems, whatever the used language, in preparation for software assurance and modernization. ASTM has been previously introduced in section III-A. It is divided into two parts GASTM that involves syntactical concepts that are common in different programming languages and SASTM that extend the first to represent specificities of languages. The combination of those standards provides a modelling framework for designing and analysing software syntax and semantics. The purpose of these two metamodels and FAST are the same: providing a core metamodel to represent concepts common to different programming languages. Nevertheless, FAST core is reduce to the strict minimal set of concepts and gives more place to specific extensions. Indeed some concepts (like method or function) may have the same name in several languages but be a little bit different and thus appear only in the extensions. Moreover, no name resolution algorithm is provided by the OMG or other authors on these metamodels.

VII. CONCLUSION

Name resolution is a fundamental part of most language parsing activities, whether it is for compiling a program, or analyzing it. It is needed if one wants to refactor, build a call graph, analyze module dependences, etc. When defining parsers for various programming languages in our Moose platform [DTD01] we often had to face the task.

In this paper, we propose a generic name resolution algorithm based on FAST, an AST metamodel. The algorithm is composed of two parts, the *lookup* that searches for candidates entities matching a name in a chain of parent scopes from the point where this name is used; and the *selection* that chooses the entity responding to access rule specific to each language. By the definition of several selectors, depending on the visibility modifier of each candidate entity, we are able to answer the need of many different languages. We discussed the implementation of this solution in three languages: Cobol, Pharo (a Smalltalk implementation), and Java.

Future works include checking the genericity of this solution in more languages (*e.g.,* Ada or Python already started) and more paradigms (*e.g.,* Lisp). We also expect to extend a bit some core functionalities. For example we already discussed the probable need for new selectors to take care of things such as friend classes in C++, or may be partial classes in C#.

REFERENCES

[AST11] *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0*. Object Management Group, January 2011.

[BCDL13] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013.

[BPM04] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms®: Program transformations for practical scalable software evolution. In Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum, editors, *ICSE*, pages 625–634. IEEE Computer Society, 2004.

[Cob94] IBM. *COBOL/400 Reference — IBM*, 1st edition, June 1994. Available at http://publib.boulder.ibm.com/iseries/v5r2/ic2924/books/c0918130.pdf (last accessed on 10/01/2013).

[DAB+11] Stéphane Ducasse, Nicolas Anquetil, Usman Bhatti, Andre Cavalcante Hora, Jannik Laval, and Tudor Girba. MSE and FAMIX 3.0: an interexchange format and source code model family. Technical report, RMod – INRIA Lille-Nord Europe, 2011.

[Dol05] Julian Dolby. Using static analysis for ide's for dynamic languages. In *The Eclipse Languages Symposium*, 2005.

[DTD01] Serge Demeyer, Sander Tichelaar, and Stéphane Ducasse. FAMIX 2.1 — The FAMOOS Information Exchange Model. Technical report, University of Bern, 2001.

[JBK06] F. Jouault, J. Bézivin, and I. Kurtev. Tcs:a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 249–254, New York, 2006. ACM Press.

[KKWV13] Gabriël Konat, Lennart Kats, Guido Wachsmuth, and Eelco Visser. Declarative name binding and scope rules. In *Software Language Engineering*, pages 311–331. Springer, 2013.

[KRV10] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, 2010.

[LTP04] Timothy Lethbridge, Sander Tichelaar, and Erhard Plödereder. The dagstuhl middle metamodel: A schema for reverse engineering. In *Electronic Notes in Theoretical Computer Science*, volume 94, pages 7–18, 2004.

[PCdGP11] Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, and Mario Piattini. Knowledge discovery metamodel-iso/iec 19506: A standard to modernize legacy systems. *Comput. Stand. Interfaces*, 33(6):519–532, November 2011.

Session Retour sur les actions spécifiques 2013

IOP : Intégration d'Outils à la Plate-forme *CosyVerif*

Étienne André, Laure Petrucci
Université Paris 13, Sorbonne Paris Cité, LIPN, CNRS, UMR 7030, F-93430, Villetaneuse, France
Fabrice Kordon
LIP6, CNRS UMR 7606, Université P. & M. Curie and Université Paris Ouest, France
Alban Linard
LSV, CNRS, INRIA & ENS Cachan, France

Abstract

CosyVerif aims at gathering within a common framework various existing tools for specification and verification. It has been designed in order to 1) support different formalisms with the ability to easily create new ones, 2) provide a graphical user interface for every formalism, 3) include verification tools called via the graphical interface or via an API as a Web service, and 4) offer the possibility for a developer to integrate his/her own tool without much effort, also allowing it to interact with the other tools. We present here a project that aims at integrating more tools into the *CosyVerif* platform.

1 Context

Formal verification of complex concurrent and heterogeneous systems often requires their model checking on complementary facets (such as discrete, timed, stochastic, etc.) of their behaviour. No single formalism being complete enough to encompass all these facets, such systems can consequently be modelled using different formalisms such as (different types of) Petri nets and timed automata. Various tools support these formalisms, each having different input and output syntaxes for models and analysis results. This often impedes integrated and comprehensive verification campaigns on complex concurrent and heterogeneous systems.

The IOP project aimed at integrating tools within the *CosyVerif* platform, a verification environment providing several formalisms and tools, and allowing for transparent tool invocations through Web services.

1.1 The *CosyVerif* Platform

CosyVerif [AHHH+13] is a distributed and open verification environment that currently handles two families of formalisms: Petri nets and timed automata. So far, 12 declared concrete formalisms from these 2 families are available, interrelated through a modular architecture of definitions, reusing common concepts, and enabling easy addition of new notations. They are syntactically supported by a two-layered XML-based language: the Formalism Markup Language (FML, the superstructure) and the Graph Markup Language (GrML, the infrastructure) [ABD+13].

Tools developers can declare a new formalism in the platform using FML, by reusing portions of existing formalisms (when they share common concepts). GrML is the internal representation of specifications in *CosyVerif*. FML and GrML ensure syntactic interoperability among tools that may only manipulate abstract syntax trees. These XML-based technologies enable rapid development and reuse of parsers and syntactic validation. Thanks to such facilities,

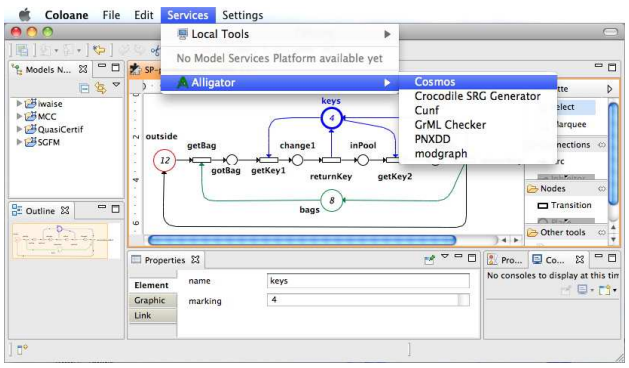


Figure 1 – Screenshot of the Coloane interface

the typical integration effort for tools developers is a day once they have been prepared (e.g. adaptation of inputs and outputs). More work is required for pre-existing tools that are ported to populate the new verification environment (e.g. several tools from the CPN-AMI Petri net verification environment that is being replaced by *CosyVerif*).

CosyVerif is an open distributed environment that can be enriched by any researcher willing to contribute. A registration mechanism allows for the diffusion of any services over a federation of *CosyVerif* nodes, which greatly improves the time-to-availability for new tools.

Tools are invoked through Web services transparently to end users, thanks to Coloane, an open source extensible graphical editor based on Eclipse (see Fig. 1). It offers modelling facilities and a way to apply tools services on models. Since *CosyVerif* relies on Web services, the use of Coloane is not mandatory and verification services can be accessed directly via the underlying XML-based protocol.

The *CosyVerif* project also provides a repository of models, that may be used for benchmark purposes. These models mostly come from industrial real-time case studies and the Model Checking Contest in 2011 [KLB+12a] and 2012 [KLB+12b].

1.2 Verification Tools in *CosyVerif*

Before this project, the following tools were integrated into *CosyVerif*:

- Cosmos [BDD+11]: a statistical model checker;
- Crocodile [CBKTM11]: tool for the so-called symbolic/symbolic approach dealing with Symmetric Nets with Bags [HKP+09];

  item CUNF [BBC+12]: toolset for carrying out *unfolding-based* verification of Petri nets extended with read arcs, also called *contextual nets* (c-nets);
- IMITATOR [AFKS12]: tool for the parameter synthesis for parametric timed automata augmented with variables and stopwatches;
- LoLA [Wol07]: explicit Petri Net state space verification tool;
- PNXDD [HKPAE12]: tool generating the state space and evaluating CTL formulæ on P/T nets.

2 Objectives of the Project and Results

The main objective of this action is to integrate more tools into the *CosyVerif* platform. In particular, some of these tools were extracted from predecessors of this environment (in particular, CPN-AMI).

A second objective was to establish an integration procedure that would benefit for other tools to be integrated in *CosyVerif*.

This has been done thanks to two interns hired at LIPN and LIP6, viz., Henoc Khouilla and Idrissa Sokhona.

In both cases, the integration procedure was carefully thought. The already available procedure has been documented by the interns, and adjusted to meet their specific requirements.

We nosw list the tool that were integrated in *CosyVerif*.

GreatSPN invariant computation for Petri nets GreatSPN is a well know Petri net tool that offers numerous services (model checking, stochastic analysis and invariant computation). CPN-AMI integrated the invariant computation modules from its user interface. This module computes:

- Place invariants,
- Transition invariants,
- Minimal syphon,
- Minimal traps.

We hired a student that prepared the integration of these functions in *CosyVerif* by translating the internal *CosyVerif* format into the one of CPN-AMI, thus enabling the reuse of the previous translators. Then, testing and benchmarking was done to access the new integration's results compared to ones provided by CPN-AMI.

Petri net structural bound computation CPN-AMI offered a tool to compute structural bounds of a tool based on the net's structure (i.e. with a lower complexity than the precise bounds to be computed by model checking). Based on the previous experience of the greatSPN integration, a similar work was done for this service, that could benefit from part of the previous work.

ModGraph [LP04] This tool performs construction and analysis of modular state spaces. Instead of actually synchronising a set of automata sharing some common transitions, it builds a synchronisation structure and keeps only the reachable parts of the automata. Thus, interleaving is avoided as much as possible. The tool also provides some analysis features, in particular reachability and deadlock-checking, that can be specified only on a subset of the interacting modules.

This tool had previously been integrated in *CosyVerif*, but it provided only a poor user interface. For instance, all the results were in a big text field, whereas *CosyVerif* services should show them as several typed fields.

The internship was in two parts:

- upgrade the ModGraph service to the latest version of the tool;
- enhance the user interface provided by the service.

ObsGraph [KO12] This BDD-based tool implements a verification approach for workflows using Symbolic Observation Graphs [HIK04]. This approach abstracts the given workflows, described as Petri net models, allowing for confidentiality (e.g. to preserve companies internal processes), and showing only the actions meant to be composed with other actions. Deadlock verification is therefore reduced to verifying only the synchronised product of the abstractions corresponding to the components.

As for ModGraph, this tool had previously been integrated in *CosyVerif*, but it provided only a poor user interface. Again, all the results were in a big text field, whereas *CosyVerif* services should show them as several typed fields.

The internship was in three parts:

- upgrade the ModGraph service to the latest version of the tool;
- enhance the user interface provided by the service;
- upgrade the service by interaction with the tool developer, for instance the addition of new services above the ObsGraph tool.

Helena [hel] Helena is an explicit state model checker. A High-level Petri net is used for models. Helena features an efficient firing rule mechanism and code generation to speed up the analysis. It provides an interface with C code. It also implements different techniques for efficient state space analysis: optimised state space storage, partial order methods and allows for LTL model checking.

The internship was a first attempt to the integration of Helena in *CosyVerif*. A prototype was obtained, but not polished enough to be released yet. Integrating Helena is difficult because a translation from the *CosyVerif* model format to Helena's one must be defined.

3 Perspectives

Perspectives for the *CosyVerif* platform include the integration of new tools, but also several improvements on the server side.

Asynchronous Tool Invocation The end user will be able to launch a verification process, and get the result later, even if the connection between the graphical client (e.g. Coloane) and the server is broken. In future releases, the result could also be for instance sent by email when the verification is finished.

Command-Line Client Tools in *CosyVerif* are not intended to be accessed only via the provided user interface. If this can be useful for demonstration or educational purposes, direct access via web services is also of interest. For instance, *CosyVerif* could be used as a back-end verification platform for other tools dedicated to higher order languages like AADL or VHDL via a transformation into one of the available formalisms. To ease the integration of such tools, a basic command-line library is being developed.

Federation of Servers In order to ease deployment and perform load balancing over a set of servers, *CosyVerif* will integrate the transparent construction of a federation of servers. The user still connects to his/her usual server that also acts as a proxy for the whole federation. Then, services are executed on the less loaded machine among those that provide it.

Acknowledgements

The development of the *CosyVerif* integration platform was supported by: LIP6, the FEDER Île-de-France/System@tic–free software thanks to the NEOPPOD project (support of two engineers), LIPN (support of one engineer), as well as LSV and Inria (support of several engineers). We finally would like to warmly thank GDR GPL and in particular Laurence Duchien for the support to the *CosyVerif* platform.

References

[ABD+13] Étienne André, Benoît Barbot, Clément Démoulins, Lom Messan Hillah, Francis Hulin-Hubard, Fabrice Kordon, Alban Linard, and Laure Petrucci. A modular approach for reusing formalisms in verification tools of concurrent systems. In Lindsay Groves and Jing Sun, editors, *15th International Conference on Formal Engineering Methods (ICFEM'13)*, volume 8144 of *Lecture Notes in Computer Science*, pages 199–214. Springer, October 2013.

[AFKS12] Étienne André, Laurent Fribourg, Ulrich Kühne, and Romain Soulat. IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In *Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 33–36. Springer, 2012.

[AHHH+13] Étienne André, Lom-Messan Hillah, Francis Hulin-Hubard, Fabrice Kordon, Yousra Lembachar, Alban Linard, and Laure Petrucci. CosyVerif: An open source extensible verification environment. In Yang Liu and Andrew Martin, editors, *18th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'13)*, pages 33–36. IEEE Computer Society, July 2013.

[BBC+12] P. Baldan, A. Bruni, A. Corradini, B. König, C. Rodríguez, and S. Schwoon. Efficient unfolding of contextual Petri nets. *Theoretical Computer Science*, 449:2–22, 2012.

[BDD+11] Paolo Ballarini, Hilal Djafri, Marie Duflot, Serge Haddad, and Nihal Pekergin. HASL: An expressive language for statistical verification of stochastic models. In *VALUETOOLS*, pages 306–315, 2011.

[CBKTM11] M. Colange, S. Baarir, F. Kordon, and Y. Thierry-Mieg. Crocodile: A symbolic/symbolic tool for the analysis of symmetric nets with bags. In *ICATPN*, volume 6709 of *Lecture Notes in Computer Science*, pages 338–347. Springer, 2011.

[hel] http://lipn.univ-paris13.fr/~evangelista/helena/.

[HIK04] Serge Haddad, Jean-Michel Ilié, and Kais Klai. Design and evaluation of a symbolic and abstraction-based model checker. In *ATVA*, pages 196–210, 2004.

[HKP+09] S. Haddad, F. Kordon, L. Petrucci, J-F. Pradat-Peyre, and N. Trèves. Efficient state-based analysis by introducing bags in Petri net color domains. In *ACC*, pages 5018–5025. Omnipress IEEE, 2009.

[HKPAE12] S. Hong, F. Kordon, E. Paviot-Adet, and S. Evangelista. Computing a hierarchical static order for decision diagram-based representation from P/T nets. *Transactions on Petri Nets and Other Models of Concurrency*, V:121–140, 2012.

[KLB+12a] F. Kordon, A. Linard, D. Buchs, M. Colange, S. Evangelista, K. Lampka, N. Lohmann, E. Paviot-Adet, Y. Thierry-Mieg, and H. Wimmel. Report on the model checking contest at Petri nets 2011. *Transactions on Petri Nets and Other Models of Concurrency*, VI:169–196, 2012.

[KLB+12b] Fabrice Kordon, Alban Linard, Didier Buchs, Maximilien Colange, Sami Evangelista, Lukasz Fronc, Lom-Messan Hillah, N. Lohmann, Emmanuel Paviot-Adet, Franck Pommereau, C. Rohr, Yann Thierry-Mieg, Harro Wimmel, and Karsten Wolf. Raw report on the model checking contest at Petri nets 2012. Technical report, 2012. CoRR.

[KO12] Kais Klai and Hanen Ochi. Modular verification of inter-enterprise business processes. In *eKNOW*, pages 155–161, 2012.

[LP04] Charles Lakos and Laure Petrucci. Modular analysis of systems composed of semiautonomous subsystems. In *ACSD*, pages 185–196. IEEE Computer Society, 2004.

[Wol07] Karsten Wolf. Generating Petri net state spaces. In *ICATPN*, volume 4546 of *Lecture Notes in Computer Science*, pages 29–42. Springer, 2007.

Plateforme d'enseIgNement du Génie logiciel

Action Émergente du GDR GPL, Année 2013.

Sébastien Mosser, pour les membres[1] de l'action
Univ. Nice Sophia Antipolis, I3S, UMR 7271, 06900 Sophia Antipolis, France
CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

1. Introduction

Ce rapport d'activité décrit le travail mené dans le cadre de l'Action Émergente PING, financée par le GDR *Génie de la Programmation et du Logiciel* pour l'année 2013. A terme, l'idée sous-jacente à cette action est la mise en place d'une plate-forme nationale support à l'enseignement du génie logiciel, en relation forte avec les activités de recherche menées dans le domaine par les équipes du GDR. A plus court terme, l'objectif de cette action est d'amorcer ce mouvement, en organisant une discussion nationale sur le sujet : tour d'horizon des enseignements innovants, de leurs contraintes, détermination des éléments pertinents d'un cours pour le rendre visible à la communauté. L'action est composée de 22 enseignants-chercheurs, représentant 13 laboratoires du CNRS. Une liste de diffusion "action-ping@polytech.unice.fr" ainsi qu'un site web http://ping.i3s.unice.fr servent de média de communication en support à l'action.

2. Contexte de travail

Le génie logiciel est de plus en plus enseigné de manière spécifique dans de nombreuses universités. Cependant, ces initiatives demeurent locales à chaque université, demandant une grande énergie pour accompagner leur mise en place. Le génie logiciel étant une discipline en constante évolution, la mise en place de supports d'enseignements en accord avec l'état de l'art théorique et technologique demande un travail de veille continu et important. De plus, une attention particulière doit généralement être portée sur le retour d'expérience de ces enseignements. Par exemple, un enseignement sur les métriques logicielles peut être aisément mal interprété par le public étudiant, et conduire à des dérives "systématiques" (typiquement une baisse artificielle des métriques de complexité cyclomatique traitant les symptômes sans la moindre analyse des causes réelles expliquant cette métrique) qui doivent être anticipées et surtout désamorcées par l'équipe enseignante afin de former les étudiants aux bonnes pratiques en matière de développement de logiciel.

Nous défendons l'idée qu'il est de la responsabilité des équipes membres du GDR GPL de faciliter l'enseignement de notre discipline, tout en favorisant ainsi une meilleure diffusion des résultats de nos recherches aux travers de nos enseignements, et ce de manière pragmatique. Notre champ disciplinaire couvre des domaines aussi variés que les méthodes de développement (*e.g.*, agilité), la vérification et la validation formelle de logiciel, ou la maintenance

[1] Contributions de Philippe Collet, Hervé Verjus et Mireille-Blay-Fornarino.

(cf. Computer Science Curricula[2]). Nous soutenons donc que l'unification des efforts consentis de manière individuelle permettra de mettre en place un support de référence pour mieux enseigner le génie logiciel auprès du plus grand nombre d'étudiants. Un effort particulier doit être réalisé sur le pragmatisme des supports proposés et leur outillage pratique, afin de proposer aux étudiants une formation de pointe sur des outils de l'état de l'art, tout en renforçant le lien entre tissu industriel et recherche académique. Le partage de ces informations doit nous permettre d'améliorer aussi notre compréhension des pratiques du GL, étayées par de très larges expérimentations avec des publics variés.

4. Motivations : Exemple d'utilisation de la plate-forme

Au sein de l'Université Nice-Sophia Antipolis, les enseignements supports au génie logiciel sont mis en œuvre en 3ème année de formation, en cycle ingénieur et en licences MIAGE et Informatique. Pour l'Université Lille 1, ces enseignements ont lieu au niveau Master 2 au sein des filières MIAGE et Informatique. Si le contenu des cours est identique en terme de notions abordées, le niveau de détails et d'expertise attendu par le public étudiant est différent. Il est donc nécessaire de trouver un agencement des notions à mettre en œuvre basé sur deux dimensions: (*i)* le niveau de détails et *(ii)* l'autonomie (ou expérience) du public. Ces deux axes dimensionnement aussi bien la profondeur de l'enseignement (de la présentation "pratique" d'outils à la dimension recherche en Génie Logiciel associée) que les capacités de mise en œuvre associées (synthèse bibliographique, TP encadrés, ou projets en autonomie). Un enseignant-chercheur qui doit élaborer un module d'enseignement en Génie Logiciel se retrouve donc confronté à des difficultés de choix, difficultés renforcées par l'évolution constante des outils supports (nouvelles versions de logiciels, nouveaux logiciels, nouveaux paradigmes ou méthodes issue de la recherche). La mise en place d'une telle plate-forme collaborative permettra donc de capitaliser sur les supports d'enseignement tout en favorisant l'amélioration de leur qualité.

3. Spécifications de la plate-forme

L'action PING s'inscrit dans la lignée de l'initiative COS-Tools (groupe de travail COSMAL du GDR), et a pour objectif de mettre en place une plate-forme participative d'enseignement du génie logiciel, en s'inspirant notamment du modèle *Wikipedia*. Son alimentation sera assurée dans un premier temps par l'expérience forte dans ce domaine de chacun des membres impliqués dans l'action. La concrétisation des résultats de l'action prendra ainsi la forme d'un ouvrage collectif dynamique, répertoriant des "fiches" de support aux différentes notions et outils nécessaires à l'enseignement du génie logiciel. Ces fiches (organisées de manière consistante d'une notion à l'autre) présenteront :

- les concepts sous-jacents à la notion (*e.g.,* fondement théoriques du test et de la spécification logicielle),
- les outils support à l'enseignement de la notion (*e.g.,* canevas logiciel JUnit, langage de spécification formel),

[2] http://ai.stanford.edu/users/sahami/CS2013

- sur la base d'expériences d'enseignement mises en œuvre par les enseignants, différentes "recettes" seront publiées, décrivant ainsi des agencements possibles de ces concepts et outils, en fonction du public visé (*e.g.,* étudiants de L3 orienté développement, étudiants de Master 2 orienté recherche),
- les compétences requises et acquises par les étudiants sur ces notions,
- le matériel d'enseignement (présentations, fiches de synthèses, lectures, vidéos, exercices, projets, références, *etc.*) réutilisables librement sous licence Creative Commons (http://creativecommons.fr).

4. Enseignement du Génie Logiciel (GL) en France

Le soutien du GDR à permis l'organisation d'une journée de présentation hébergée par le LIP6[3] le 27 novembre 2013. L'objectif de cette journée était de servir de forum d'échange afin de décrire le panorama national relatif à l'enseignement du génie logiciel par les membres de l'action. La suite de cette section fait la synthèse des différentes approches d'enseignement décrites. Les présentations plus complètes de chacune de ces approches sont disponibles sur le site web de l'action. D'un point de vue global, le dénominateur commun de toutes ces formations est l'utilisation active d'une pédagogie basée sur les projets.

IAGL (*Romain Rouvoy*, Lille 1). La spécialité de master IAGL a été fondée en 1988, et offre 24 places chaque année. son organisation repose sur de nombreux projets, dont un projet "start-up" fait en groupes de 7 à 8 étudiants permettant d'expérimenter les méthodologie de dévelopement agile avec un encadrement mixte enseignants / industriels. Le master intègre un module "*Recherche Innovation Créativité*" piloté par Laurence Duchien qui est une initiation à l'univers de la recherche. Le module "outils" de ce master présente aussi bien des outils dits "techniques" tel que de la gestion de versions et des outils dits "formels" comme Alloy en support à une activité de modélisation semi-formelle.

Génie Logiciel dans un IAE (*Hervé Verjus*, IAE Savoie Mont-Blanc). La structure d'un IAE est par essence orientée sur les sciences de gestion. Les enseignements commencent en licence 3 pour une spécialité puis dans un master SI. Dans ce contexte, l'enseignement du GL se tourne naturellement plus sur les Systèmes d'Information et l'utilisation du GL en support à la définition d'un système d'information. Comme pour IAGL, on retrouve des projets dits "innovants" qui mettent les étudiants en situation entrepreunariale et des parties pratiques avec une vocation parfois ludique (utilisation des LEGO MindStorms). Les enseignements de GL sont orientés sur la programmation (fonctionnelle, objet, Web), les méthodologies de développement et l'introduction de notions d'architecture logicielle et SOA.

La plateforme ATLAS (*Sébastien Mosser*, Univ. Nice-Sophia Antipolis). Au sein de l'Université Nice-sophia Antipolis (UNS), une plate-forme de gestion de projet logiciel appelée ATLAS est déployée. Basée sur la suite logicielle fournie par la société Atlassian, elle fournit un système de suivi de tâches, un système de gestion de version, suivi des feuilles de temps et un système de revue de code par les pairs. Ces outils sont utilisés dans les différentes formations de l'UNS (école d'ingénieur, UFR Sciences et IUT) en support à une pédagogie par projets. La bonne utilisation des outils fait partie des critères d'évaluation des projets. Selon les formations,

[3] http://gdr-gpl.cnrs.fr/node/113

les enseignants jouent aussi bien sur les concepts présentés que sur la profondeur de l'enseignement (tests unitaire uniquement, tests d'intégration, …).

Génie Logiciel à Rennes (*Olivier Barais*, Rennes 1). L'enseignement du GL à Rennes 1 repose principalement sur 2 équipes de recherche, avec une forte implication des enseignants et chercheurs dans l'offre de formation. L'enseignement repose majoritairement sur une pédagogie par projet, la plupart du temps en utilisant du matériel spécifique (tablette, robot) pour mettre les étudiants en situation. L'enseignement du GL commence dès la seconde année et se poursuit tout au long du cursus. Une des spécificités de cet environnement est le couplage des approches formelles et techniques, sous la forme de projets conjoints. Par exemple, le projet de développement Web est couplé à l'utilisation d'un assistant de preuve utilisé pour prouver le noyau du site développé.

Le master TAL (*Reda Bendraou*, UPMC). Le master *Technique Applicatives* à un mode de fonctionnement complètement différent des autres approches présentées. En effet, il place immédiatement les étudiants en situation de création d'entreprise. Les étudiants sont mobilisable du lundi au vendredi en journée (et exceptionnellement le samedi), et prennent part à la charge collective de gestion de la promotion (gestion du site Web, commande auprès du secrétariat). L'enseignement est organisé par projet, avec une forte implication de partenaires industriels externes.

Projets RCIM (*Didier Donsez*, Université Joseph Fourier). Les projets développés dans ce département sont définies autour d'un thème commun (e.g., SmartHome). Les étudiants sont placée en situation d'innovation, et proposent des solutions allant jusqu'au prototype physique (maquette d'une SmartHome) mettant en oeuvre leur solution. L'approche "pratique" et "créatrice" fait partie intégrante des projets, permettant aux étudiants de mieux comprendre ce qui à été enseigné.

Point de vue industriel (*Guilhem Molines*, IBM). Intervenant à titre personnel pour nous faire part de son expérience en tant que recruteur, Guilhem Molines a insisté sur le besoin d'assimilation des concepts par les étudiants. En effet, les recruteurs "sérieux" ne sont pas intéressés par une liste de compétences de type "soupe de lettres" (e.g., XML, XSLT, UML, JAVA, C++, WSDL, SOAP), mais plutôt par la maîtrise des concepts sous-jacents (e.g., programmation fonctionnelle, conception objet). Les étudiants doivent comprendre que peu des technologies présentés aujourd'hui en support aux enseignements existeront encore dans 10 ans, mais que les concepts de GL sous-jacents leur permettront d'appréhender ces ruptures technologiques.

5. Conclusion et Perspectives

Le support financier du GDR à cette action à permis la mise en place d'une réflexion nationale sur l'enseignement de notre discipline, impliquant de nombreux laboratoires du CNRS. Sur la base des discussions amorcées grâce à cette action, la plate-forme collaborative visée par cette action est en train de se mettre en place sous la forme d'un site participatif de type *"wiki"*. Le travail qui émerge de cette action à pour vocation d'être continué via le site participatif, et l'organisation de nouveaux évènements (e.g., participation à l'EJCP). Une des perspectives de ces travaux est l'identification du "kit du doctorant du GDR GPL", i.e., la définition du nécessaire à tout doctorant intégré dans une équipe du GDR.

Rapport de l'Action Émergente "Empirical Software Engineering"

9 mars 2014

But de l'action

L'action émergente "Empirical Software Engineering" (Génie Logiciel Empirique) a été financée en 2014 par le GDR-GPL dans le but de favoriser la dynamique de la recherche en génie logiciel empirique dans le paysage académique français. L'aide du GDR-GPL a été utilisée pour organiser une atelier d'une journée le 23 octobre 2013 à Paris auquel 29 personnes ont participé.

Résumé de l'atelier

L'atelier "Empirical Software Engineering" (Génie Logiciel Empirique) s'est organisé en trois parties. Dans un premier temps, David Lo (Univ. de Singapour) a donné une présentation invité sur le thème de "To what extent could we detect field defects - An empirical study of false negatives in static bug finding tools". Dans un second temps, ont eu lieu 4 présentations par des chercheurs en génie logiciel empirique issus des laboratoires français (programme complet ci-dessous). Enfin, une série de brainstorm a permis d'identifier des axes de collaboration en termes de sujets, d'échange d'étudiants et d'articles collaboratifs.

L'atelier a rassemblé 29 personnes d'horizons divers. Les 5 présentations données ont permis d'apercevoir l'étendue des préoccupations de la recherche française en génie logiciel empirique. Les discussions en petits groupes ont fait émerger plusieurs thèmes d'intérêt, en particulier:
- comment faire de la recherche en génie logiciel empirique ? (méta-questions)
- quelle est la nature des bugs et comment les gérer ?
- quels sont les liens entre qualité du logiciel et génie logiciel empirique ?
- quels sont les liens entre évolution du logiciel et génie logiciel empirique ? (avec un focus sur les clones)

Programme complet de la journée:

9:00 - 9:30 Café de bienvenue
9:30 - 9:45 Introduction et tour de table
9:45 - 10:30 Keynote de David Lo (30 minutes) "To what extent could we detect field defects - An empirical study of false negatives in static bug finding tools"

http://dl.acm.org/citation.cfm?id=2351685

10:30 - 11:00 Pause café matin

11:00 - 11:30 Session Oops:

- Julia Lawall (Oops! What about a Million Kernel Oopses?)
- Lisong Guo, LIP6 (What developers can do with kernel oopses)

11:30 - 12:30 What is empirical software engineering? Open brainstorm, constitution des groupes d'intérêt pour l'après-midi

12:30 - 14:30 Déjeuner au restaurant Le Buisson Ardent

14:30 - 15:00 Session Data et Metadata

- Talk: Olivier Berger (Package metadata traceability in FLOSS development platforms) http://www-public.telecom-sudparis.eu/~berger_o/presentation-ESE-2013.pdf
- Talk: Xavier Blanc, LABRI (Making Survey thanks to repository mining)
- Harmony : https://se.labri.fr/wp/?page_id=1462

15:30 - 16:00 Pause café après midi

16:00 - 16:45 Discussion en groupe d'intérêt

16:45 - 17:30 Restitution des groupes

17:30 - Fin du workshop

Liste des participants à l'atelier

29 personnes ont participé à l'atelier:
Martin Monperrus, Julia Lawall, David Lo, Benoit Baudry, Arnaud Blouin, Clémentine Nebut, Tegawendé F. Bissyandé, Jacques Klein, Anne Etien, JC Bajard, B. Folliot, Nic Volanschi, Olivier Berger, JR Falleri, Basile Starynkevitch, Andre Hora, Lisong Guo, Laurent Réveillère, Xavier Blanc, Yves Le Traon, Luis Rodriguez, Matias Martinez, Benoit Cornu, David Bromberg, Hélène Waeselynck, Xavier Le Pallec, Mike Papadakis, Sophie Dupuy-Chessa, Reda Bendraou, Lydie du Bousquet.

Ils venaient d'horizons différents: le Laboratoire d'Informatique Fondamentale de Lille (LIFL), l'Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA) à Rennes, le Laboratoire Bordelais de Recherche en Informatique (LaBRI), le Laboratoire d'informatique de Paris 6 (LIP6), le Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM).

Nous notons la présence de deux personnes de l'industrie: Nic Volanschi (Metaware) et Luis Rodriguez (Qualcomm).

Utilisation des fonds

Du fait de l'affluence au workshop, les fonds ont été utilisés intégralement par le repas du midi et les 2 pauses cafés (matin et après-midi).

Session industrielle

Session Industrielle

Retours et perspectives chez Siemens

Auteur : François Gerin (Directeur Général adjoint de Siemens France)

Bibliographie :
François Gerin, 65 ans, X - Mines, INSEAD AMP, lauréat de la Fondation Nationale Entreprise et Performance, est Directeur Général adjoint de Siemens SAS depuis le 1995. Il y a dirigé, de 1991 à 1998, les activités de télécommunications. Au sein de la Direction Générale, il développe les synergies de ventes entre entités du groupe. Il coordonne également les activités d'innovation des entités du groupe en France.Il préside également le Conseil d'Administration de l'Ecole des Mines de Saint Etienne depuis septembre 2002, après avoir présidé celui d'Alès depuis 1997.

Par ailleurs François Gerin avait aussi été membre de la Commission Consultative des Radiocommunications, travaillant auprès de l'ARCEP, entre 1994 à 2009. François Gerin avait été précédemment Directeur des activités audiovisuelles puis des services de Télécommunications du groupe Lyonnaise des Eaux. Auparavant, il avait été Directeur de la Délégation aux vidéocommunications auprès du Directeur général des télécommunications, et Directeur de l'opération Biarritz Fibre optique. François Gerin est Chevalier dans l'Ordre National du Mérite. Enfin François Gerin est Président de la SEE (Société de l'Electricité, de l'Electronique et des Technologies de l'information et de la Communication) depuis février 2013.

Retours et perspectives des programmes ANR

Auteur : Eva Crück (ANR)

Bibliographie :
Eva Crück est titulaire d'un diplôme d'ingénieur en système électroniques de l'ENSIETA et d'une thèse en mathématiques appliquées. Elle a repris le programme INS de l'ANR en 2013. Avant de rejoindre l'ANR, elle a travaillé à la DGA comme expert technique en guidage-pilotage pour les missiles et les drones (y-compris aspects embarqués). Toujours à la DGA, elle a ensuite rejoint la direction de la stratégie pour s'occuper de la définition et la mise en œuvre de la politique scientifique en STIC.

Table ronde : Les défis du Génie de la Programmation et du Logiciel 2025

Table ronde : Les défis du Génie de la Programmation et du Logiciel 2025

Animateurs

Philippe Collet (I3S, U. Nice Sophia-Antipolis)
Lydie Du Bousquet (LIG, U. Grenoble)
Laurence Duchien (LIFL, U. Lille)

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@univ-pau.fr

***Abstract*—Systems of systems (SoS) are large-scale systems composed of complex systems with difficult to predict emergent properties. One of the most significant challenges in the engineering of such systems is how to model and 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, because the 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 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 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].

G. Possible Agenda for Tackling the Challenges

Following a Software Engineering approach, a possible agenda to tackle these challenges could be inspired from an iterative, incremental, V-like software development life-cycle. As such, a first step would consist in extracting and formulating requirements from the challenges. As these requirements could be divergent or even conflictual, several partial solutions could be expected to emerge. Therefore, in a second step, one or more architectural frameworks proposing an architecture for one or several software tools and processes to use them could be proposed. To validate and verify the requirements and the architecture(s), several test cases could be proposed. In a third step, the proposed framework(s) would be implemented in one or several programming languages. The fourth step would use the test cases to verify and validate the implementation. These steps would be repeted in an incremental way, until teh requirements are considered addressed.

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, 5th Intl Conf on*, pages 1–8, 2010.

[3] W. Beyeler, R. Glass, and G. Lodi. Modeling and risk analysis of information sharing in the financial infrastructure. In R. Baldoni and G. Chockler, editors, *Collaborative Financial Infrastructure Protection*, pages 41–52. Springer, 2012.

[4] J. Boardman and B. Sauser. System of systems - the meaning of OF. In *System of Systems Eng, 2006 IEEE/SMC Intl Conf*, page 6 pp, 2006.

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

[6] M. Ciampi, G. Pietro, C. Esposito, M. Sicuranza, P. Mori, A. Gebrehiwot, and P. Donzelli. On Securing Communications among Federated Health Information Systems. In F. Ortmeier and P. Daniel, editors, *Computer Safety, Reliability, and Security*, volume 7613 of *LNCS*, pages 235–246. Springer, 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, IEEE Intl*, pages 364–369, 2013.

[9] M. Duren, H. Aldridge, R. K. Abercrombie, and F. T. Sheldon. Designing and Operating Through Compromise: Architectural Analysis of CKMS for the Advanced Metering Infrastructure. In *The 8th Annual Cyber Security and Information Intelligence Research Wksh*, number 48, pages 1–3, 2013.

[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*, pages 182 –188, 2011.

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

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

[13] M. Jamshidi. System of Systems - Innovations for 21st Century. In *Industrial and Information Systems, 3rd Intl Conf on*, pages 6–7, 2008.

[14] Roy S. Kalawsky. The Next Generation of Grand Challenges for Systems Engineering Research. *Procedia C. S.*, 16:834 – 843, 2013.

[15] J. Klein and H. van Vliet. A Systematic Review of System-of-systems Architecture Research. In *The 9th Intl ACM Sigsoft Conf on Quality of Software Architectures*, pages 13–22, 2013.

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

[17] M. 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, Intl Conf*, volume 4, pages 3149–3154, 2005.

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

[20] C. Nichols and R. Dove. Architectural Patterns for Self-Organizing Systems-of-Systems. *Insight*, 4:42–45, 2011.

[21] C. E. Phillips, Jr., T.C. Ting, and S. A. Demurjian. Information sharing and security in dynamic coalitions. In *7th ACM Symposium on Access Control Models and Technologies*, pages 87–96, 2002.

[22] N. Ricci, A. M. Ross, and D. H. Rhodes. A Generalized Options-based Approach to Mitigate Perturbations in a Maritime Security System-of-Systems . *Procedia C. S.*, 16:718 – 727, 2013.

[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] F. Skopik, A. Treytl, A. Geven, B. Hirschler, T. Bleier, A. Eckel, C. El-Salloum, and A. Wasicek. Towards Secure Time-triggered Systems. In *Int Cf on Comp Safety, Reliability, and Security*, pages 365–372, 2012.

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

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

[27] Z. Zhang, C. Wu, and D. W.L. Cheung. A Survey on Cloud Interoperability: Taxonomies, Standards, and Practice. *SIGMETRICS Perform. Eval. Rev.*, 40:13–22, 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

1 Context and Motivation

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 parametrised 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 parametrised 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.

2 Challenges and Agenda

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 for formalisms such as parametric timed automata (PTA) [AHV93] or parametric time Petri nets [TLR09] are known to be undecidable in the general case. However, in the past few years, some problems were shown to be decidable, in particular integer parameter synthesis [JLR13a], or characterization of the system robustness (see, e.g., [Mar11]), which are subproblems of the main parameter synthesis issue.

Decidability problems may appear to be disconnected from applications, but they are not: although undecidable problems may yield useful semi-algorithms that can output interesting results, finding decidable subclasses of models is an incentive for scientists to seek efficient algorithms (that always terminate, by definition).

Studying concurrent systems with both discrete and continuous parameters can lead to several types of parameters (discrete, timed, hybrid, probabilistic), and combining them can lead to many different problems. We believe that the ultimate goal would be to combine all kinds of parameters in a single model. This also implies the definition of adequate formalisms, either decidable, or with efficient semi-decidable algorithms.

Discrete Parameters Regular model checking (RMC) techniques [BJNT00,BHH+08,BHRV12] and cut-off based algorithms [CTTV04,BHV08] apply to the analysis of systems where the number of entities is *a priori* unknown, but not to the analysis of all parametrised systems. Indeed, RMC addresses systems with linear or tree-like topologies, and cut-off techniques particular kinds of systems in an *ad hoc* manner. A first goal will be to develop techniques as general as RMC but that apply to general topologies. One way consists in extending the RMC framework to deal with graphs and to develop techniques based on graph automata for the symbolic representation of (infinite) sets of graphs.

A second goal is to deal with timed models with discrete parameters, where some discrete components such as the number of processes are *a priori* unknown. The discrete parametric model checking problem for timed models is likely to be undecidable. A first direction will be to consider subclasses of timed automata (or time Petri Nets) with discrete parameters where the abstract state space of the timed part of the model (zone graph, state class graph) could be handled with an extension of the RMC framework based on their topology properties. Another direction will be consider decidable subclasses of this parameter synthesis problem (e.g. bounded parameters) and propose efficient symbolic synthesis algorithms based on symbolic state space abstractions.

Timing Parameters The parameter synthesis problem is known to be undecidable for PTA [AHV93,BLT09], but decidable for subclasses such as L/U automata [HRSV02], although this model has a strong syntactical restriction for practical purposes. In [JLR13a] an approach based on restricting to integers the possible values of the parameters, leads to decidability. Although extending to rationals this results appears to be possible for non-reachability properties, it remains to be done for more elaborated properties (such as unavoidability, equality of trace sets [ACEF09], games [JLR13b], etc.). These results should be extended so as to exhibit subclasses of PTA for which parameter synthesis (possibly under- or over-approximated) is guaranteed to terminate.

From these results and those related to discrete parameter synthesis, a further goal will be to synthesise constraints of good behaviour based on both these timing parameters and the discrete parameters.

Cost Parameters A challenge is to investigate the use of richer dynamics in models to make them suitable for the modelling of a wider range of applications, such as energy consumption. This leads to the so-called generic *hybrid* setting, in which the continuous variables may have dynamics defined by arbitrary differential equations. The study of this class of models is notoriously difficult and the decidability results are scarce in this area. In terms of parametrisation, two decidable subclasses of hybrid automata seem promising: O-minimal automata [LPS00,BMRT04] and interrupt timed automata (ITA) [BHS12]. First, O-minimal automata feature extremely rich dynamics but each discrete transition must reset all continuous variables. Interrupt timed automata (ITA) have been introduced with the aim to describe timed multitask systems with interruptions in a single processor environment. The accepted language of ITA is incomparable to the one of TA, and reachability is decidable.

Furthermore, weighted or priced models [ALP04,BFH+01], restrict the richer dynamics to *cost* variables that are never tested, only updated, and therefore do not participate in the actual trajectory of the system. Previous results show that the question of knowing if there exists some parameter values such that so location is reachable within T time units, for some given T, is undecidable for PTA.

In both cases, it will then be challenging to extend the obtained decidable parametrised subclasses with parametrised discrete behaviours, much as for continuous parameters.

Probabilistic Parameters In real-life applications, probabilities are often used as a building block that allows abstracting from physical constraints or unknown environments. Hence, a challenge is to extend the models considered above with probabilities, and study parametric probabilistic timed systems where parameters can range over time constraints, cost variables *and* transition probabilities. Obtaining fundamental results in this domain would carry much weight as they would impact many applicative fields.

In another setting, probabilities can also be seen as a tool for synthesising optimal values of parameters: probabilities can be artificially injected on the parameter space of *non-probabilistic* parametrised systems, and Statistical Model Checking (SMC) [LDB10] can then be used in order to identify regions in the parameter space that optimise given properties. Since SMC is still at its early stages, it suffers from many limitations that will have to be overcome in order to produce significant results.

Applications Beyond classical applications (hardware verification, process management, embedded and cyber-physical systems), typical applications in the near future are *smart homes*, in particular catering for elderly or disabled people in a safe manner. Parametrisation 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. More generally, distributed applications (with a variable number of processes, of local environment) will be a natural application of both discrete and continuous parameter synthesis.

Acknowledgements We would like to thank the anonymous reviewers for their useful comments.

References

ACEF09. É. André, T. Chatain, E. Encrenaz, and L. Fribourg. An inverse method for parametric timed automata. *IJFCS*, 20(5):819–836, 2009.

AHV93. R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *STOC*, 1993.

ALP04. R. Alur, S. La Torre, and G. J. Pappas. Optimal paths in weighted timed automata. *Theoretical Computer Science*, 318(3):297–322, 2004.

BFH+01. G. Behrmann, A. Fehnker, T. Hune, K. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum-cost reachability for priced timed automata. In *HSCC*, 2001.

BHH+08. A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, 2008.

BHRV12. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular (tree) model checking. *STTT*, 14(2):167–191, 2012.

BHS12. B. Bérard, S. Haddad, and M. Sassolas. Interrupt timed automata: verification and expressiveness. *Formal Methods in System Design*, 40(1):41–87, 2012.

BHV08. A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of parametric concurrent systems with prioritized FIFO resource management. *FMSD*, 32(2):129–172, 2008.

BJNT00. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *CAV*, 2000.

BLT09. L. Bozzelli and S. La Torre. Decision problems for lower/upper bound parametric timed automata. *Formal Methods in System Design*, 35(2):121–151, 2009.

BMRT04. T. Brihaye, C. Michaux, C. Rivière, and C. Troestler. On O-minimal hybrid systems. In *HSCC*, 2004.

CTTV04. E. M. Clarke, M. Talupur, T. Touili, and H. Veith. Verification by network decomposition. In *CONCUR*, 2004.

HRSV02. T. Hune, J. Romijn, M. Stoelinga, and F. W. Vaandrager. Linear parametric model checking of timed automata. *JLAP*, 52-53, 2002.

JLR13a. A. Jovanović, D. Lime, and O. H. Roux. Integer parameter synthesis for timed automata. In *TACAS*, 2013.

JLR13b. A. Jovanović, D. Lime, and O. H. Roux. Synthesis of bounded integer parameters for parametric timed reachability games. In *ATVA*, volume 8172 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 2013.

LDB10. A. Legay, B. Delahaye, and S. Bensalem. Statistical model checking: An overview. In *RV*, 2010.

LPS00. G. Lafferriere, G. Pappas, and S. Sastry. O-minimal hybrid systems. *MCSS*, 13:1–21, 2000.

Mar11. N. Markey. Robustness in real-time systems. In *SIES*, pages 28–34. IEEE Computer Society Press, 2011.

TLR09. L.-M. Traonouez, D. Lime, and O. H. Roux. Parametric model-checking of stopwatch Petri nets. *Journal of Universal Computer Science*, 15(17):3273–3304, 2009.

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 défis, 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 mis 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 [9]. 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 [1]. 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 [8, 15]. 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 [11]. 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, de par la taille des systèmes à considérer qui ne pourra se combattre que par la définition de critères de sélection de tests pertinents qui limitent le nombre de cas de tests à générer, et les besoins d'explorer l'espace d'états des programmes ou des modèles [14].

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 personnelles 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) [24, 16, 7].

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 [2, 3]. Pour compléter ces vérifications macroscopiques relatives aux interactions entre noeuds, on peut également s'intéresser à l ?observation fine de l ?exécution au niveau d ?un dispositif (ex : tablette ou téléphone mobile). Selon les propriétés à vérifier, un défi serait d'exploiter au mieux 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 [18] et des technologies basées sur des composants et des services [12]. 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 [20, 17].

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'exemples 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 [4]. 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 [5].

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) [10]. 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 [6].

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 [19, 13, 21, 23]. 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 [22, 25]. 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) [1] 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) [2] offrent une formation ad hoc et permettent de valider un certain niveau de connaissance des pratiques du test.

1. `www.reseau-figure.fr`
2. `www.istqb.org`

Références

[1] BernhardK. Aichernig. Contract-based testing. In BernhardK. Aichernig and Tom Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *Lecture Notes in Computer Science*, pages 34–48. Springer Berlin Heidelberg, 2003.

[2] P. André, H. Waeselynck, and N. Rivière. A uml-based environment for test scenarios in mobile settings. In *Computer, Information and Telecommunication Systems (CITS), 2013 International Conference on*, pages 1–5, May 2013.

[3] Pierre André, Nicolas Rivière, and Hélène Waeselynck. Graphseq revisited : More efficient search for patterns in mobility traces. In Marco Vieira and Joao Carlos Cunha, editors, *Dependable Computing*, volume 7869 of *Lecture Notes in Computer Science*, pages 88–95. Springer Berlin Heidelberg, 2013.

[4] James Arnold and Rob Alexander. Testing autonomous robot control software using procedural content generation. In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaaniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 33–44. Springer Berlin Heidelberg, 2013.

[5] SuriyaPriyaR. Asaithambi and Stan Jarzabek. Towards test case reuse : A study of redundancies in android platform test libraries. In John Favaro and Maurizio Morisio, editors, *Safe and Secure Software Reuse*, volume 7925 of *Lecture Notes in Computer Science*, pages 49–64. Springer Berlin Heidelberg, 2013.

[6] Julien Botella, Fabrice Bouquet, Jean-François Capuron, Franck Lebeau, Bruno Legeard, and Florence Schadle. Model-based testing of cryptographic components – lessons learned from experience. In *ICST'13, 6th IEEE Int. Conf. on Software Testing, Verification and Validation*, pages 192–201, March 2013.

[7] Matthias Büchler, Karim Hossen, Petru Florin Mihancea, Marius Minea, Roland Groz, and Catherine Oriat. Model inference and security testing in the spacios project. In Serge Demeyer, Dave Binkley, and Filippo Ricca, editors, *CSMR-WCRE*, pages 411–414. IEEE, 2014.

[8] Cristian Cadar and Koushik Sen. Symbolic execution for software testing : three decades later. *Commun. ACM*, 56(2) :82–90, 2013.

[9] Omar Chebaro, Nikolaï Kosmatov, Alain Giorgetti, and Jacques Julliand. Combining static analysis and test generation for C program debugging. In G. Fraser and A. Gargantini, editors, *TAP'10, 4th Int. Conf. on Tests and Proofs*, volume 6143 of *LNCS*, pages 94–100, Malaga, Spain, July 2010.

[10] Frédéric Dadeau, Kalou Cabrera Castillos, Yves Ledru, Taha Triki, German Vega, Julien Botella, and Safouan Taha. Test generation and evaluation from high-level properties for common criteria evaluations - the TASCCC testing tool. In B. Baudry and A. Orso, editors, *ICST 2013, 6th Int. Conf. on Software Testing, Verification and Validation, Testing Tool track*, pages 431–438, Luxemburg, Luxemburg, March 2013. IEEE Computer Society Press.

[11] Alain Denise, Marie-Claude Gaudel, Sandrine-Dominique Gouraud, Richard Lassaigne, Johan Oudinet, and Sylvain Peyronnet. Coverage-biased random exploration of large models and application to testing. *STTT*, 14(1) :73–93, 2012.

[12] Quentin Enard, Miruna Stoicescu, Emilie Balland, Charles Consel, Laurence Duchien, Jean-Charles Fabre, and Matthieu Roy. Design-Driven Development Methodology for Resilient Computing. In *CBSE'13 : Proceedings of the 16th International ACM Sigsoft Symposium on Component-Based Software Engineering*, Vancouver, Canada, June 2013.

[13] C. Ghezzi, A. Mocci, and M. Sangiorgio. Runtime monitoring of component changes with spy@runtime. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 1403–1406, June 2012.

[14] Hadi Hemmati, Andrea Arcuri, and Lionel C. Briand. Achieving scalable model-based testing through test case diversity. *ACM Trans. Softw. Eng. Methodol.*, 22(1) :6, 2013.

[15] Thierry Jéron, Margus Veanes, and Burkhart Wolff. Symbolic Methods in Testing (Dagstuhl Seminar 13021). *Dagstuhl Reports*, 3(1) :1–29, 2013.

[16] Ralph LaBarge and Thomas McGuire. Cloud penetration testing. *CoRR*, abs/1301.1912, 2013.

[17] Beatriz Pérez Lamancha, Macario Polo, and Mario Piattini. Systematic review on software product line testing. In José Cordeiro, Maria Virvou, and Boris Shishkov, editors, *Software and Data Technologies*, volume 170 of *Communications in Computer and Information Science*, pages 58–71. Springer Berlin Heidelberg, 2013.

[18] Jimmy Lauret, Jean-Charles Fabre, and Hélène Waeselynck. Fine-grained implementation of fault tolerance mechanisms with aop : To what extent ? In Friedemann Bitsch, Jérémie Guiochet, and Mohamed Kaaniche, editors, *Computer Safety, Reliability, and Security*, volume 8153 of *Lecture Notes in Computer Science*, pages 45–56. Springer Berlin Heidelberg, 2013.

[19] Choonghwan Lee, Feng Chen, and Grigore Roşu. Mining parametric specifications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 591–600, New York, NY, USA, 2011. ACM.

[20] Jihyun Lee, Sungwon Kang, and Danhyung Lee. A survey on software product line testing. In *Proceedings of the 16th International Software Product Line Conference - Volume 1*, SPLC'12, pages 31–40, New York, NY, USA, 2012. ACM.

[21] David Lo, Leonardo Mariani, and Mauro Santoro. Learning extended {FSA} from software : An empirical assessment. *Journal of Systems and Software*, 85(9) :2063 – 2076, 2012. Selected papers from the 2011 Joint Working IEEE/IFIP Conference on Software Architecture (WICSA 2011).

[22] Harald Raffelt, Bernhard Steffen, Therese Berg, and Tiziana Margaria. Learnlib : a framework for extrapolating behavioral models. *STTT*, 11(5) :393–407, 2009.

[23] G. Reger, H. Barringer, and D. Rydeheard. A pattern-based approach to parametric specification mining. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 658–663, Nov 2013.

[24] M. I. P. Salas and E. Martins. Security testing methodology for vulnerabilities detection of xss in web services and ws-security. *Electr. Notes Theor. Comput. Sci.*, 302 :133–154, 2014.

[25] Muzammil Shahbaz and Roland Groz. Analysis and testing of black-box component based systems by inferring partial models. *Software Testing, Verification and Reliability*, feb 2013.

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 à 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 [3] car certains aspects de l'approche furent totalement occultés [21] ? 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. De plus en plus souvent, nous sommes face à ce que [2, 3] qualifient de modèles complexes : *des modèles hétérogènes, structurés suivant plusieurs dimensions métiers et possiblement de grande*

1. Model Driven Architecture
2. Computer Aided Software Engineering
3. Dans le sens de [21] : outils privilégiant la programmation visuelle

taille. Les langages et/ou les environnements logiciels associéés se révèlent inopérants renvoyant notamment à divers problèmes de formalisation, d'abstraction, représentation et de navigation. C'est précisément au carrefour de ces difficultés de plus en plus fréqueutes que se situe 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 tels modèles.

Face à ce défi, les verrous identifiés se situent au niveau :

— de la définition de "modèle complexe". Différentes dimensions liées à la complexité d'un modèle apparaissent dans la littérature : l'abstraction [20], la viscosité [9], la discriminabilité perceptuelle [17] ... ? Mais quel est leur poids dans la complexité globale ? L'efficacité d'un diagramme étant liée à la tâche à réaliser [9], peut-on spécialiser la définition de la complexité à l'IDM ?

— des critères d'efficacité. À partir de la précédente définition, il sera possible de donner les critères qui définissent une "bonne" modélisation dans le sens d'une modélisation cognitivement appréhendable par l'utilisateur ;

— des solutions. Trouver de nouveaux paradigmes de visualisation et d'interaction pour la modélisation, qui permettent à l'utlisateur de dépasser les problèmes inhérant à la complexité.

L'objectif 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. En fait, 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. Plus largement, nous souhaitons ici établir un dialogue 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, 23] 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. La SG définit 6 variables visuelles en lien avec notre capacité à percevoir la profondeur. 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 celles de Bertin) [22] : 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 [24]. 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]).

3.3 Les jalons

Les pistes à étudier sont jalonnées par les étapes suivantes de compréhension et de maîtrise des modèles complexes.

Dans un premier temps, il est nécessaire de définir les caractéristiques d'un modèle complexe. Ce travail devra s'appuyer sur des évaluations auprès d'utilisateurs de modèles et des études de cas réelles. Il constitue le socle sur lequel sont fondés les deux jalons suivants.

Les caractéristiques de la complexité d'un modèle doivent permettre d'aborder la mesure de la complexité d'un modèle. Cette mesure pourra s'appuyer sur la qualité des langages de modélisation, en particulier la qualité de la syntaxe concrète. Un jalon sera franchi lorsqu'il sera possible d'évaluer automatiquement ou semi-automatiquement la complexité d'un modèle.

En parallèle de cette deuxième étape sur les modèles eux-même, leur manipulation doit être explorée. En effet, l'utilisation de techniques d'interaction et de design adaptées à la complexité des modèles pourrait permettre de modifier la perception de la complexité par les utilisateurs. Le jalon est de savoir proposer des techniques de manipulation, de visulation et d'exploration

de modèles adaptées aux caractéristiques de complexité d'un modèle.

L'objectif ultime est de proposer des environnements complets capables de mesurer la complexité des modèles, mais aussi de proposer aux utilisateurs des techniques d'interaction permettant de mieux la gérer.

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] Catherine Recanati. Characteristics of diagrammatic reasoning. In Daniel Kayser edited by Stella Vosniadou and Athanassios Protopapas, editors, *Proceedings of EuroCogSci07, the european cognitive science conference*, the second european cognitive science conference, pages pp 510–515, Delphi, Grèce, May 2007. Lawrence Erlbaum Associates.

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

[22] 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.

[23] 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.

[24] 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).

Prix de thèse du GDR Génie de la Programmation et du Logiciel

Abstractions performantes pour cartes graphiques

Auteur : Mathias Bourgoin (Université Pierre et Marie Curie)

Résumé :

Résumé : Les cartes graphiques (GPU) sont des dispositifs performants et spécialisés dotés de nombreuses unités de calcul, dédiés à l'affichage et au traitement 3D. Les systèmes Cuda et OpenCL permettent d'en détourner l'usage pour réaliser des calculs généralistes, normalement effectués par le CPU (Central Processing Unit) : la programmation GPGPU (General Purpose GPU). De très bas niveau d'abstraction, ils demandent de manipuler explicitement de nombreux paramètres matériels comme la mémoire ou le placement des calculs sur les différentes unités. Le but de cette thèse est l'étude de solutions de plus haut niveau d'abstraction pour la programmation GPGPU, afin de la rendre à la fois plus accessible et plus sûre. Nous introduisons deux langages de programmation dédiés à la programmation GPGPU, SPML et Sarek ainsi que leur sémantique opérationnelle, et les garanties qu'ils apportent. Nous présentons ensuite une implantation de ces langages, en OCaml, à travers la bibliothèque SPOC et le langage dédié intégré, Sarek. Des tests montrent que notre solution permet d'atteindre un haut niveau de performance, pour des exemples simples, comme pour le portage d'une application numérique réaliste depuis Fortran et Cuda, vers OCaml. Nous montrons alors comment notre solution permet de définir des squelettes de programmation offrant davantage d'abstractions. A travers un exemple, nous présentons comment ils simplifient la programmation GPGPU et autorisent le développement d'optimisations supplémentaires. Enfin, nous discutons les possibilités offertes par l'évolution des systèmes matériels et logiciels pour offrir une solution unifiée pour la programmation GPGPU.

Biographie :
Mathias Bourgoin a réalisé son doctorat d'informatique au Laboratoire d'Informatique de Paris 6 (LIP6), au sein des équipes APR (Algorithmes, Programmes et Résolution) et PEQUAN (PErformance et QUalité des Algorithmes Numériques), sous la direction du Pr. Emmanuel Chailloux et du Pr. Jean-Luc Lamotte. Sa thèse s'est attachée à la formalisation et au développement d'abstractions de haut niveau pour la programmation des cartes graphiques dans le cadre de la programmation haute performance des systèmes hétérogènes. Il est actuellement Attaché Temporaire d'Enseignement et de Recherche à l'Université Pierre et Marie Curie (Paris 6) où il étudie la composition et la prédictibilité des performances des calculs sur cartes graphiques ainsi que le développement d'applications web performantes.

Démonstrations et Posters

StaDy: a Frama-C Plugin to Combine Static and Dynamic Software Analyses

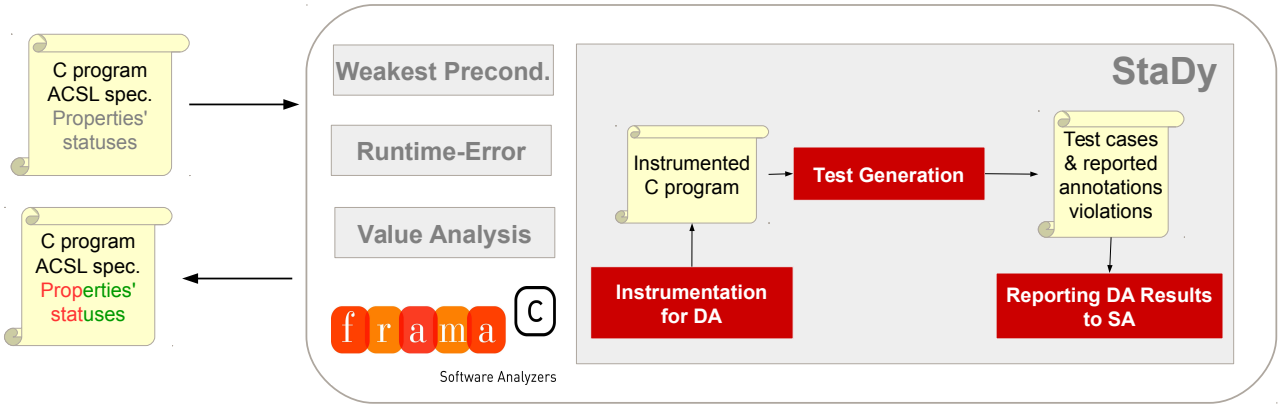
G. Petiot, N. Kosmatov, J. Julliand, A. Giorgetti
CEA LIST – FEMTO-ST/DISC

Context

ACSL Specification Language for C (First Order Logic)
Frama-C framework for the analysis of C programs
Static Analysis (SA) analysis without any execution
Dynamic Analysis (DA) analysis during the execution

Motivations

- Combining **Static and Dynamic** Analysis
- **Complete the results** of Automatic Provers with Testing
- Handle **complex ACSL specifications**



Overview of StaDy

input	concrete output	symbolic output
strlen = 3 sublen = 1 dest[0] = 106 dest[1] = -43 dest[2] = -37 str[0] = 117 str[1] = 117 str[2] = 79 substr[0] = 117	return value = 0 dest[0] = 117 dest[1] = 79	return value = 0 dest[0] = str[0] dest[1] = str[2]

Invalidated properties:
- Counter-examples generated
- Inputs, concrete/symbolic outputs displayed

Overview

- **Instrumentation**: translating ACSL specification to executable C for Dynamic Analysis
- **Test generation**: white-box testing aiming at covering all feasible program paths
- **Reporting**: transmitting the results of DA to Frama-C for re-use by other analyzers

```
    typically \old(strlen) <= 5;
    assigns *(dest+(0 .. strlen-1));

    behavior not_present:
      assumes
        !(\exists integer i;
            (0 <= i && i < strlen-sublen) &&
            (\forall integer j;
               0 <= j && j < sublen ==> *(str+(i+j)) != *(substr+j)));
      ensures
        \forall integer k;
          0 <= k && k < \old(strlen) ==> \old(*(str+k)) == *(\old(dest)+k);
      ensures \result == 0;
 */
int delete_substr(char *str, int strlen, char *substr, int sublen, char *dest)
{
```

Finding counter-examples for ACSL properties

Property invalidated by StaDy

Strengthened for testing

```
    typically \old(strlen) <= 5;
    assigns *(dest+(0 .. strlen-1));

    behavior not_present:
      assumes
        !(\exists integer i;
            (0 <= i && i < strlen-sublen) &&
            (\forall integer j;
               0 <= j && j < sublen ==> *(str+(i+j)) == *(substr+j)));
      ensures
        \forall integer k;
          0 <= k && k < \old(strlen) ==> \old(*(str+k)) == *(\old(dest)+k);
      ensures \result == 0;
 */
int delete_substr(char *str, int strlen, char *substr, int sublen, char *dest)
{
```

Validating ACSL properties with a precondition strengthened for testing

Property validated (under hypotheses) by StaDy

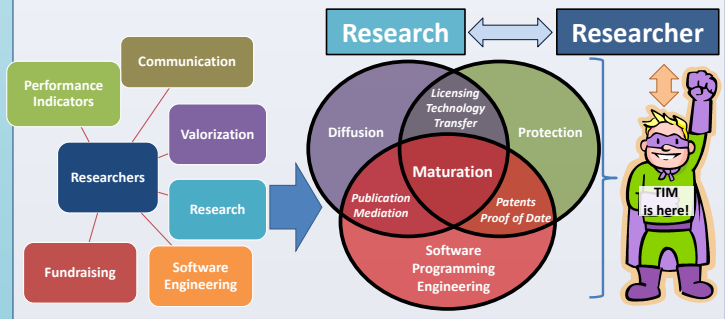
References
[CKGJ12] Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC (2012)
[CKKPSY12] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C - a software analysis perspective. In: SEFM (2012)
[PKGJ14] Petiot, G., Kosmatov, N., Giorgetti, A., Julliand, J.: How Test Generation Helps Software Specification and Deductive Verification. Submitted In: TAP (2014)

Software Assets Management in Public Research Institutes: a Technological Infrastructure for Maturation

*D. Bursztyn, A. Constantin, C. Dumont, M. Mangili, J.C. Souplet**
Pôle Developpement Logiciel - Laboratoire de Recherche en Informatique (LRI)
UMR8623 - Université Paris-Sud / CNRS, *contact : *jcsouplet@lri.fr*

Introduction

- **Researchers (in ICT or not) produce scientific software**
- **Managing these software life cycle include several aspects that have to be done on research time**
- **Help researchers stay focused on research is the aim of our *Technological Infrastructure for Maturation* (TIM)**

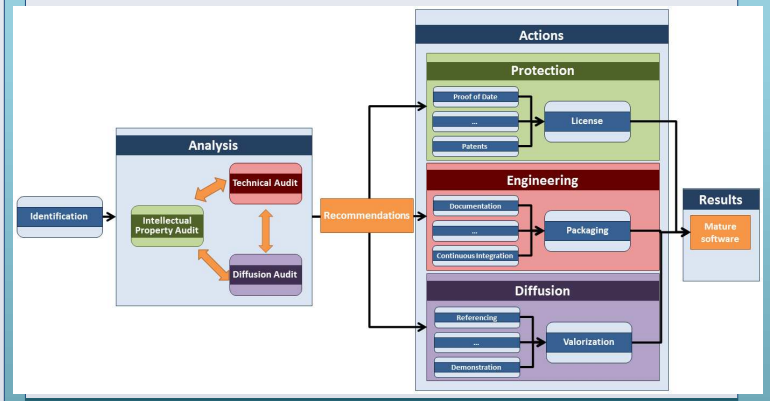


Description of TIM

- **TIM starts its work from the identification of a software**
  - Name, Utility, Researcher leading the project …
  - Can TIM have access to the source code ?
- **If the leading researcher wants to collaborate with TIM, an analysis (including simultaneously protection, diffusion and technological aspects) is performed using two approaches:**
  - Answers to a set of predefined questions
  - Source code audit using specialized software (e.g. Sonar, Antepedia Reporter, …)
- **From the analysis results, recommendations are formulated leading to**
  - Use new programming tools (e.g. Source code versioning tools, building tools, …)
  - Improve the test policy (e.g. continuous integration)
  - Improve documentation (for users, developers, …)
  - Develop new functionalities or interface (e.g. add a GUI)
  - …
- **The researcher prioritizes these recommendations. A collaboration between him and the Development Team of LRI can be fostered.**
  - The Development Team can realize the "service" defined by the recommendation

Focus on TIM analysis

- **According to TIM, only one analysis is performed : Diffusion, Protection and technological aspects have to be studied at the same time. For example,**
  - To define the license of your software, you need to be aware of both : what is the software target and what are the used external components and their corresponding licenses
    - ➔ Protection needs Engineering and Diffusion
  - A software addresses one or more requirements. These needs include which kind of protection do you want and how you want to diffuse it
    - ➔ Engineering needs Protection and Diffusion
  - To diffuse a software, you need to define a license and to give a minimum of documentation
    - ➔ Diffusion needs Protection and Engineering
- **Consequently, a TIM's recommendation is produced taking into account these three aspects but the corresponding action is performed on a single one**



Discussion

- **Tim was born in may 2011. In 3 years, it grew from a "spirit of continual improvement process" (support by a person) to a formalization of steps (realized by 6 persons < 4 full-time equivalents) required to analyze and improve the "level of maturity" of a software.**
- **Tim's originality is to mix technical, intellectual property and diffusion aspects.**
- **According to us, it is too early to evaluate Tim results. However, an indicator is: the number of APP deposit of the laboratory doubled in 3 years (from 11 to 22).**
- **Tim is still growing and we would like to add new tools (software, methodologies, …) to each of these steps:**

Can your tools be included in TIM ?

"TIM gives your research wings"

Measuring the robustness of source program obfuscation

A study of the impact of compiler optimizations on the obfuscation of C programs

Main Objective :

Protect software from reverse engineering :
defend from **static analysis**
(disassemblers, ...)
or from **dynamic analysis**
(debuggers, ...)

Initial Program P

```
struct test{int a;int b;};

int main (void){
struct test t;
int c;
t.a =10;
t.b =20;
c = t.a;
return c + t.b;
}
```

Obfuscation

Automatic

Obfuscated Program P^{obf}

Accessed fields

Obfuscated main function

```
struct test {int a;int b;};

int main(void){
struct test t;
int c;
access_counter = 0;

*(address_array + 1) = &t.b;
*(address_array + 0) = &t.a;

*access()= 10;
*access()= 20;
c = *access();
return (c + *access());
}
```

Each field access is hidden in a function

Auxiliary functions and variables

```
int access_counter;
int access_array[4] = {0, 1, 0, 1} ;
int *address_array[2];

int *access(){
int nb_elem, id1, id2, t, *ptr;
nb_elem = 2;
id2 = rand_a_b(0, nb_elem);
id1 = rand_a_b(0, nb_elem);

t = **(address_array + id1);
**(address_array + id1) = **(address_array + id2);
**(address_array + id2) = t;

ptr = *(address_array + id1);
*(address_array + id1) = *(address_array + id2);
*(address_array + id2) = ptr;

access_counter = access_counter + 1;
return *(address_array [*(access_array [access_counter - 1]]);
}
//Returns an int in [a;b]
int rand_a_b(int a, int b)...
```

4 = Nb of field accesses

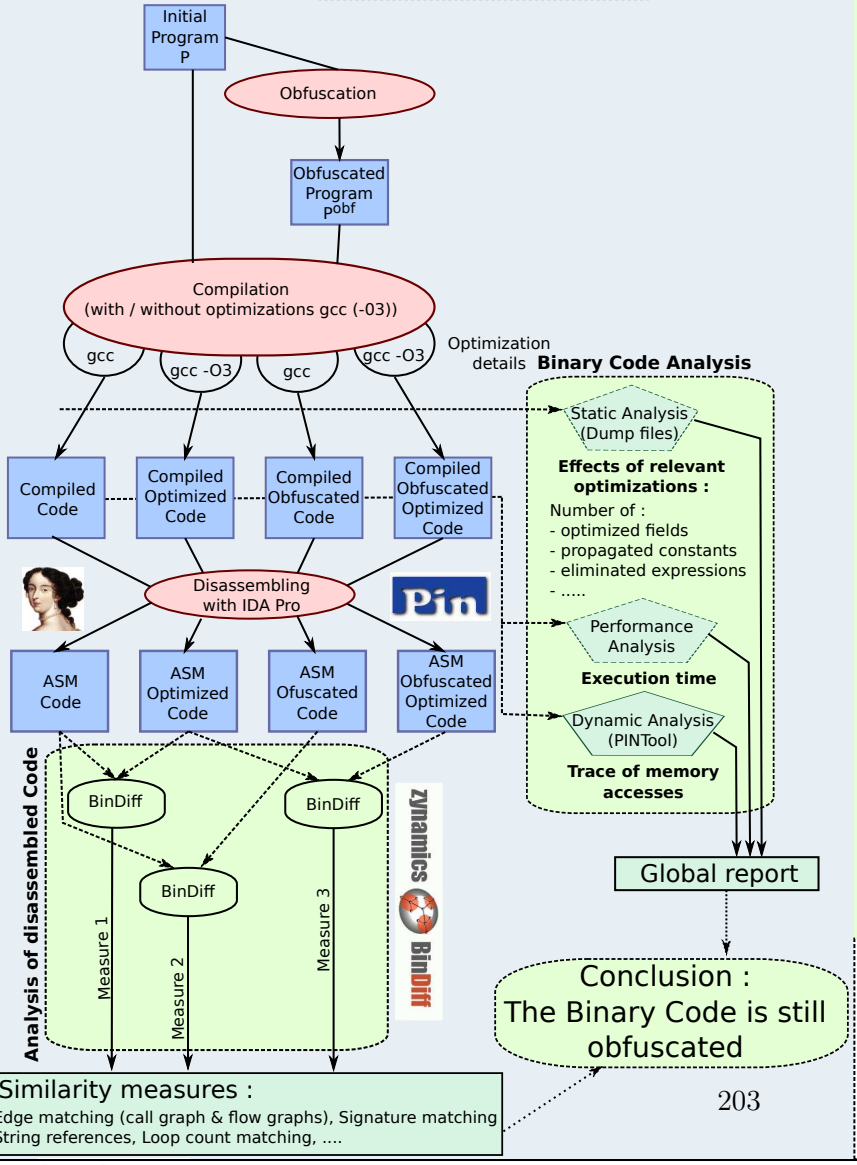
Field addresses are stored in address_array

Field values are exchanged randomly

Matching between field values and addresses

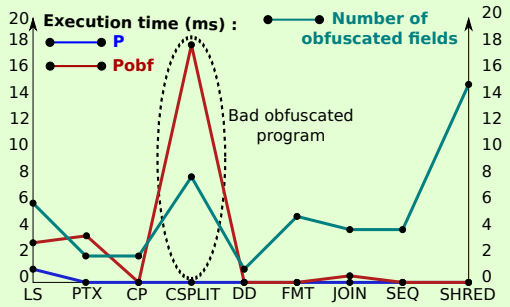
Address of the next accessed element is returned

Evaluation Process



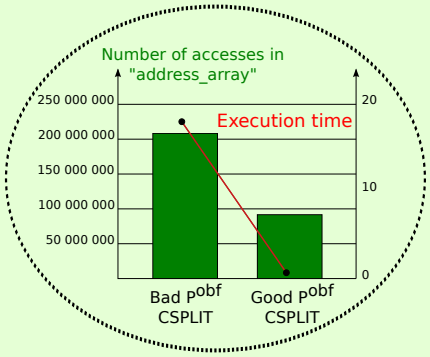
Performance of binary code

Execution time does not depend on the number of obfuscated fields



New obfuscation improving array accesses and execution time

Execution times depend on the number of accesses to the array : "address_array".



Measuring the robustness of source program obfuscation: studying the impact of compiler optimizations on the obfuscation of C programs.
In Proceedings of the 4th ACM conference on Data and application security and privacy (CODASPY '14).
ACM, New York, NY, USA, 123-126

Sandrine Blazy and Stephanie Riaud

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

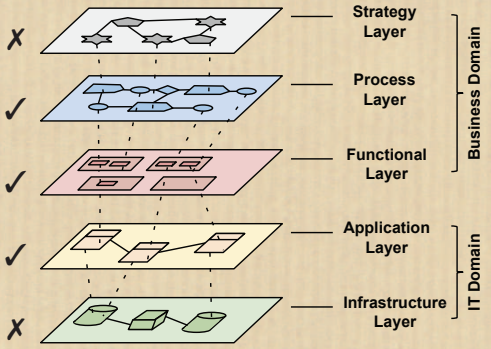
[2] Mia-Software
Nantes
ebreton@sodifrance.fr

Enterprise Architecture: Can Business Models be Aligned with IT ?

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

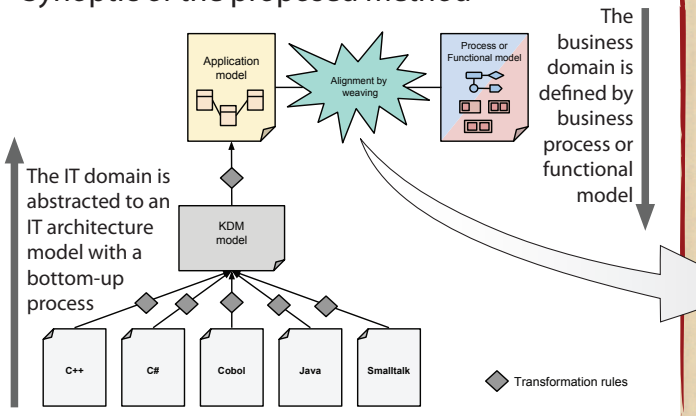
Challenges

- Reconcile IT and Business viewpoints
- Two-way navigation through stacked layers
- Assisted alignment of legacy IT and Business model
- Using metrics to drive Information System (IS)

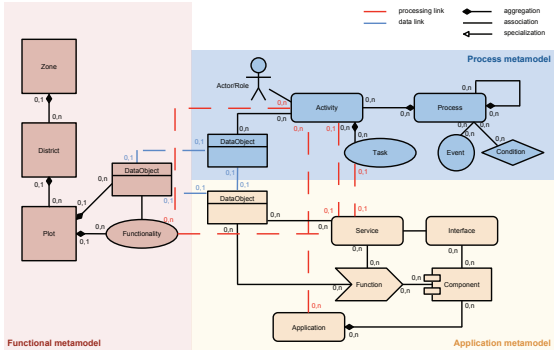


Information System Layer Stack

Synoptic of the proposed method



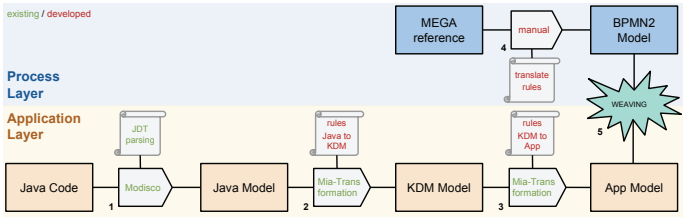
Enterprise Architecture metamodels and links



Data and process links are defined between business domain (functional and process metamodel) and IT domain (application metamodel)

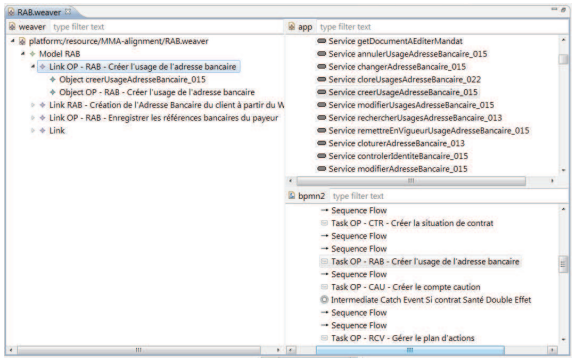
Experimentation and tools

A case study from a French mutual insurance company:
- core business IS: customers and contracts management software
- huge Java source code (33,400 classes)
- business model diagrams from MEGA Enterprise Architecture software



1. Reverse engineering of the Java code to obtain a Java model
2. Java model transformation into KDM intermediary model
3. Transformation and abstraction from KDM into Application model
4. Manual translation from MEGA model to BPMN2 standard
5. Application and business models alignment by weaving

Our developed Eclipse Plugin weaver assistant:
- based on a specific weaving metamodel
- links created by drag & drop
- tree-like browser and quick search of concepts
- creation of specific link using constraints



Results

- Feasability of the alignment tool chain supported by model transformation
- Full coverage of concepts contained in Enterprise Architecture models

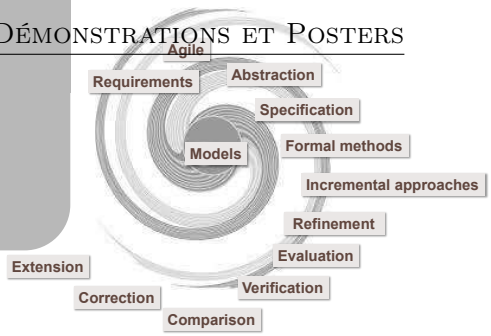
Perspectives

- Alignment analyser to compute Dependency Structure Matrix (DSM)
- Navigation through layers by drill down

lina

Mia-Software
IT Agility Solutions

Incremental development and behavioural verification of reactive systems



Institution

INSTITUT Mines-Télécom

MINES Alès

Authors

Anne-Lise Courbis
Thomas Lambolais
Hong-Viet Luong
Thanh-Liem Phan

Partners

IRIT

Christian Percebois

UM2

Thérèse Libourel

Problem

How to assist model development of critical reactive systems?

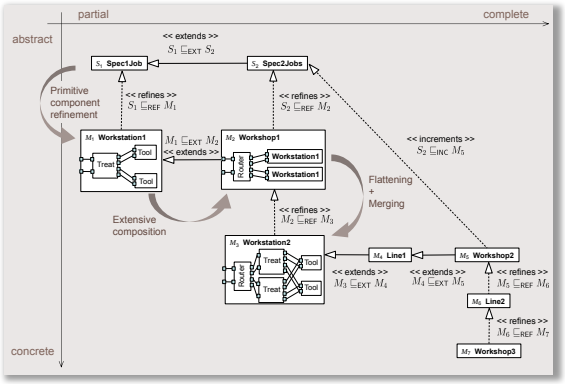
Model engineering is not mature.

- "Engineers don't know why their system works. [...] They can not be sure a critical system is free of critical errors." J. Sifakis
- "Today for most software systems, the analogy of building something like a cathedral is no longer a good choice. [...] Requirements change all the time, we need a short time-to-market, we need feed back all the time..." M. Lippert
- "If you want to get it right, be ready to start over at least once." E.S. Raymond

Need to develop and **verify several** model versions:
from abstract and partial ones,
to detailed and completed ones.

IDF – Incremental Development Framework

Combining model refinements and extensions



Two sets of techniques … to support:

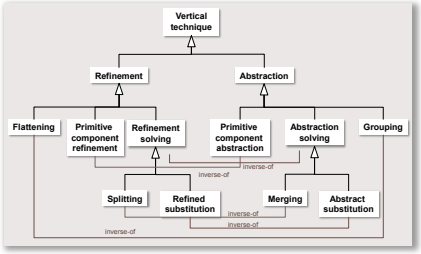
- Construction techniques
- Evaluation techniques

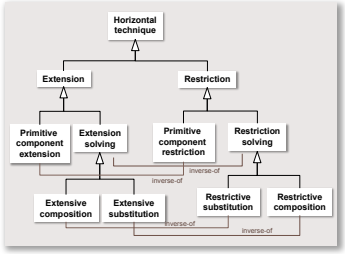
⇒ Incremental development processes

Two axis:

- abstraction level (vertically)
- completion level (horizontally)

Construction techniques





Evaluation techniques

M_2 conf M_1 — M_2 is a correct implementation of M_1: M_2 preserves liveness properties of M_1.

$M_1 \sqsubseteq_{INC} M_2$ — M_2 increments M_1: any implementation of M_2 is an implementation of M_1.

$M_1 \sqsubseteq_{EXT} M_2$ — M_2 extends M_1: M_2 preserves liveness properties of M_1 and has more behaviours.

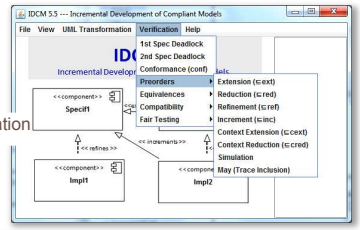
$M_1 \sqsubseteq_{REF} M_2$ — M_2 refines M_1: M_2 preserves liveness and safety properties of M_1.

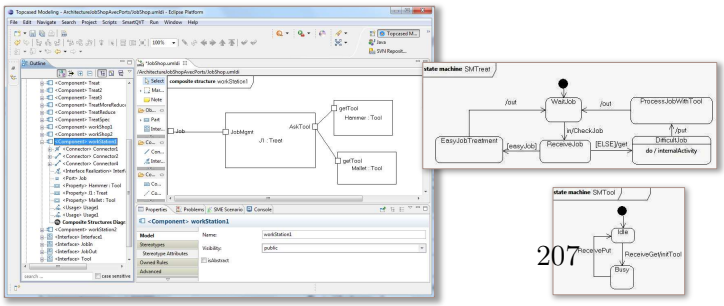
$M_1 \sqsubseteq_{SUB} M_2$ — M_2 can substitute M_1: M_2 refines M_1 and can safely replace M_1.

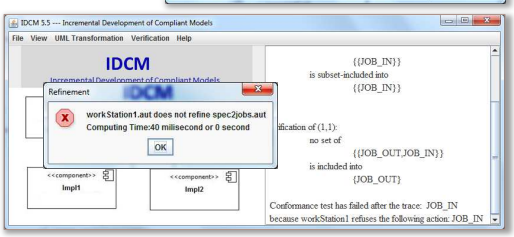
IDCM – Incremental Development of Compliant Models

A tool to support IDF

- Transformation of UML models into LTS (Labelled Transition Systems):
  - UML primary components (state machines) and
  - architectures (composite structures).
- Use of CADP (Construction and Analysis of Distributed Processes) features for LTS composition and minimisation
- Implementation of conformance, increment, extension, refinement and substitution relations
- Analysis of models pointing out traces of failure and denied actions whenever relations are not satisfied







Contact thomas.lambolais@mines-ales.fr Web site http://lgi2p.mines-ales.fr

Trace-based test suite reduction

G. Vega[2,1], T. Triki[1,2], Y. Ledru[1,2], L. du Bousquet[1,2]
[1]Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2]CNRS, LIG, F-38000 Grenoble, France
VASCO team, http://vasco.imag.fr
German.Vega@imag.fr, Yves.Ledru@imag.fr, Lydie.du-Bousquet@imag.fr

Motivations

- Continuous testing and continuous integration need to select a subset of a test suite that requires limited time to play
- Current techniques minimize test suite size at the cost of lower fault detection capabilities

Our goal

- Provide a family of test reduction criteria which **improve fault detection capabilities**

An (erroneous) program instrumented by Jacoco :
findMax returns the maximum of the list, and 0 if the list is empty

```
public class ListMax {
  ArrayList<Integer> l ;

  public ListMax() {
    this.l = new ArrayList<Integer>();// Probe 0
  }
  public void addPos(int i){
    l.add(Math.abs(i));              // Probe 1
  }
  public void addNeg(int i){
    l.add(-1*Math.abs(i));           // Probe 2
  }
  public int findMax(){
    int current = 0;
    int max = 0;                     // Probe 3
    while (current < l.size()){
      if (l.get(current)>max){
        max = l.get(current);        // Probe 5
      }              // else branch : Probe 4
      current++;                     // Probe 6
    }                3
    return max;                      // Probe 7
  }
}
```

Bug : 0 is returned for a list of negative integers

In order to detect the bug, the testcase must avoid Probe 5 while running findMax().

Covers all branches does not find the bug

Testsuite

generated by Randoop (19 tests)

```
public void test3() throws Throwable {
    ListMax var0 = new ListMax();
    int var1 = var0.findMax();
    var0.addNeg(-100);
    int var4 = var0.findMax();
    assertTrue(var1 == 0);
    assertTrue(var4 == -100);
  }

public void test8() throws Throwable {
    ListMax var0 = new ListMax();
    var0.addNeg(-1);
    var0.addPos(1);
    var0.addPos(0);
    int var7 = var0.findMax();

    assertTrue(var7 == 1);
  }

public void test10() throws Throwable {
    ListMax var0 = new ListMax();
    int var1 = var0.findMax();
    var0.addNeg(-100);
    var0.addPos(10);
    int var6 = var0.findMax();

    var0.addNeg(0);
    assertTrue(var1 == 0);
    assertTrue(var6 == 10);
  }

public void test13() throws Throwable {
    ListMax var0 = new ListMax();
    var0.addNeg((-1));
    int var3 = var0.findMax();
    var0.addPos(1);
    var0.addNeg(-10);
    assertTrue(var3 == -1);
  }

public void test18() throws Throwable {
    ListMax var0 = new ListMax();
    int var1 = var0.findMax();
    var0.addNeg(100);
    var0.addPos(10);
    var0.addNeg(100);
    var0.addPos(1);
    int var10 = var0.findMax();


    var0.addPos(10);
    var0.addNeg(10);
    assertTrue(var1 == 0);
    assertTrue(var10 == 10);
  }
```

Traces for these tests

(using a modified version of Jacoco)

```
TestListMax0 test3 {
        [Probe_0]
        [Probe_3,Probe_7]
        [Probe_2]
        [Probe_3,Probe_4,Probe_6,Probe_7]
}

TestListMax0 test8 {
        [Probe_0]
        [Probe_2]
        [Probe_1]
        [Probe_1]
        [Probe_3,Probe_4,Probe_6,Probe_5,
         Probe_6,Probe_4,Probe_6,Probe_7]
}

TestListMax0 test10 {
        [Probe_0]
        [Probe_3,Probe_7]
        [Probe_2]
        [Probe_1]
        [Probe_3,Probe_4,Probe_6,Probe_5,
          Probe_6,Probe_7]
        [Probe_2]
}

TestListMax0 test13 {
        [Probe_0]
        [Probe_2]
        [Probe_3,Probe_4,Probe_6,Probe_7]
        [Probe_1]
        [Probe_2]
}

TestListMax0 test18 {
        [Probe_0]
        [Probe_3,Probe_7]
        [Probe_2]
        [Probe_1]
        [Probe_2]
        [Probe_1]
        [Probe_3,Probe_4,Probe_6,Probe_5,
        Probe_6,Probe_4,Probe_6,
        Probe_4,Probe_6,Probe_7]
        [Probe_1]
        [Probe_2]
}
```

Reduction algorithm

- Collect the traces of each test case
- Build equivalence classes for the tests
- (Option) Suppress subsumed equivalence classes
- Choose one test per equivalence class

Equivalence Criteria	Reduced test suite using the criterion
Same set of probes (e.g. test8, test10 and test18)	10 tests
Same set of sets of probes (e.g. test10 and test 18)	12 tests
Same set of sequences of probes	13 tests
Same sequence of sets of probes	17 tests
Same sequence of sequences of probes	17 tests
Subsumption Criteria	
Subset of probes (e.g. test3 subsumed by test8, test10 and test18)	1 test: test8
Subset of sets of probes (e.g. test8 subsumed by test 10 and test18)	3 tests: test18, test13, test3
Subset of sequences of probes	5 tests: test18, test8, test17, test13, test3
Prefix of sequence of sets of probes	15 tests
Prefix of sequence of sequences of probes	15 tests

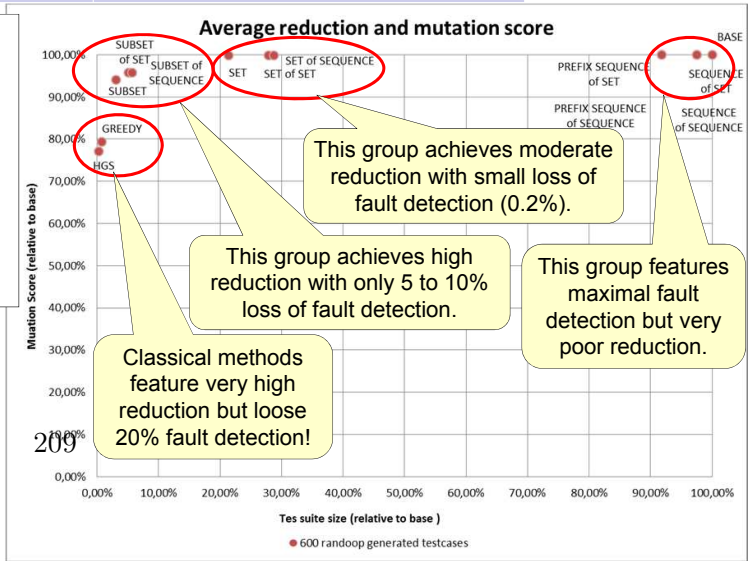
Same result as classical methods based on branch coverage: does not find the bug … All other reduced suites find the bug!

Experimentation

- 11 programs (most from Software Artifact Repository) : 44 to 434 lines
- Test suite generated using Randoop (counting 600 test cases)
- Mutants created for each subject using MuJava

Preliminary results

- Classical methods based on branch coverage lead to better reduction
- Our method produces reductions of various sizes, with better fault detection capabilities than the classical methods.



References

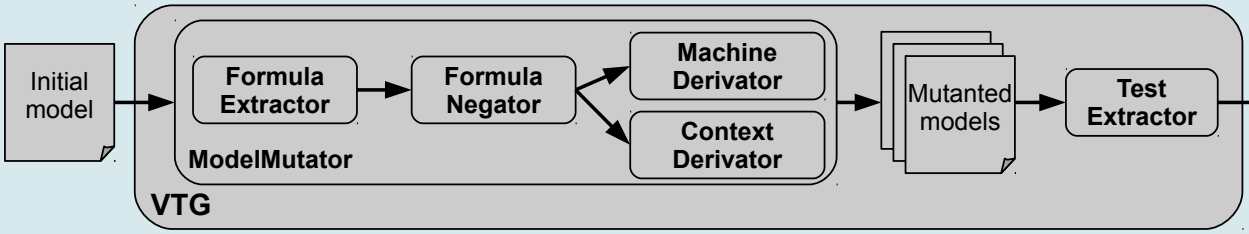
- T. Triki, L. du Bousquet, Y. Ledru. *Réduction de suites de tests avec des critères d'équivalence basés sur la couverture structurelle*, AFADL'12:120-134, Grenoble, 2012.
- T. Triki, *Réduction et filtrage de tests combinatoires*, PhD Thesis, Université de Grenoble, France, oct 2013.

This work was supported partially by the ANR TASCCC project (2009-12) #ANR-09-SEGI-014, and by the ARC6 network of the Rhône-Alpes region.

VTG 2.0: *Vulnerability Tests Generator*

Aymerick Savary, Jean-Louis Lanet, Marc Frappier
Xlim labs. (France) and GRIL labs. (Canada)
aymerick.savary@{xlim.fr, usherbrooke.ca}

Vulnerability Tests Generator : tool based on mutation testing and model-based testing, used for software penetration testing.



Methodology

Original model

```
AXIOMS
    axm1 : i1 ∈ ℕ
    axm2 : i2 ∈ ℕ
    axm3_t : i1 ≥ i2 ∧ i2 < 4
END
```

```
Event push ≙
    where
        grd1 : pc < maxpc
        grd2_t : ss ≤ Mss − 1
    then
        act1 : pc := pc + 1
        act2 : ss := ss + 1
    end
```

Negation rules

1. $neg(p_1 \wedge p_2) \rightsquigarrow \{ neg(p_1) \wedge p_2,\ p_1 \wedge neg(p_2),\ neg(p_1) \wedge neg(p_2)\}$
2. $neg(p_1 \vee p_2) \rightsquigarrow \{neg(p_1) \wedge neg(p_2)\}$
3. $neg(i_1 \geq i_2) \rightsquigarrow \{i_1 < i_2\}$
4. $neg(i_1 \leq i_2) \rightsquigarrow \{i_1 > i_2\}$
5. $neg(t_1 = t_2) \rightsquigarrow \{\neg(t_1 = t_2)\}$

Mutated models

```
AXIOMS
    axm1 : i1 ∈ ℕ
    axm2 : i2 ∈ ℕ
    axm3_t : i1 < i2 ∧ i2 = 4
END
```

```
Event push ≙
    where
        grd : pc < maxpc
        grd_t : ss > Mss − 1
        grd_EUT : eut = FALSE
    then
        act_EUT : eut := TRUE
    end
```

Intrusion tests

```
axm1 : i1 = 1        axm1 : i1 = 2
axm2 : i2 = 4        axm2 : i2 = 4

axm1 : i1 = 3
axm2 : i2 = 4
```

Event	ss
push	1
push	2
push	3 (> Mss)
return	??

Based on :

- **Tom** : Tom is a language extension designed to manipulate tree structures, used in Rodin to analyse models.
- **Rodin** : The Rodin Platform is an Eclipse-based IDE for Event-B that provides effective support for refinement and mathematical proof.
- **ProB** : ProB is an animator and model checker for the B-Method (also supports Event-B, CSP-M, TLA+, and Z.).

Bibliography :

- *Formula Negator, Outil de négation de formule, AFADL 2014, A. Savary, M. Lassale, M. Frappier, J.-L. Lanet, Juin 2014*
- *Detecting Vulnerabilities in Java-Card Bytecode Verifiers using Model-Based Testing, iFM 2013, A. Savary, M. Frappier, J.-L. Lanet, Juin 2013*
- *VTG - Vulnerability Test Generator, a Plug-in for Rodin, Workshop Deploy 2012, A. Savary, J.-L. Lanet, M. Frappier, T. Razafindralambo, J. Dolhen, February 2012*

Préservation de preuve lors de la compilation sur microcontrôleur

Vivien Maisonneuve, thèse encadrée par François Irigoin et Olivier Hermant
Centre de recherche en informatique, Mathématiques et systèmes, MINES ParisTech, Fontainebleau

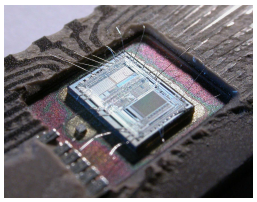
Démonstrations et Posters

Conception d'un système embarqué

Un **système embarqué** est un système informatique autonome spécialisé.
Contraintes d'autonomie, de temps d'exécution, de sécurité & sûreté.

Utilise un microprocesseur basse consommation ou un microcontrôleur.

Systèmes embarqués dans les moteurs, les télécommandes, les appareils de bureau, l'électroménager, les jouets, les téléphones, etc.



Travail de conception sur deux niveaux :

Formalisation :
- Conception du système ;
- Modélisation physique de l'environnement ;
- **Preuve** mathématique que le système se comporte correctement.

MATLAB, Simulink

Réalisation : **programme** C de très bas niveau
- Plusieurs milliers de LOC ;
- Calculs décomposés en opérations élémentaires ;
- Gestion des moteurs et des capteurs.

GCC, Clang

Transformations graduelles

Objectifs

Comment être sûr que le programme exécuté est correct ?

Preuves de **stabilité numérique** : montrer que les paramètres d'un système restent dans une certaine enveloppe durant son exécution (**stabilité de Lyapunov**).
Primordiales pour la sécurité du système.

Modélisation en **boucle ouverte** ou en **boucle fermée** (en tenant compte de la **rétroaction**).

Comment adapter la preuve sur le modèle physique en une preuve équivalente sur le programme ?

Nombres à virgule flottante (norme IEEE 754)

Dans le programme, les valeurs numériques du modèle sont **approximées** par des valeurs binaires de précision limitée, p. ex. **nombres flottants**.

$(-1)^s \times 2^{e-127} \times m$

Les preuves de stabilité ne s'appliquent plus :
- Altération des constantes numériques du système ;
- **Erreurs d'arrondi** lors des calculs, qui se cumulent.

Approche pour prouver la stabilité en flottants :
- **Transposition** des arguments de preuve en tenant compte de ces effets ;
- Vérification de la validité de la preuve : si oui, le programme est stable.

Bibliographie

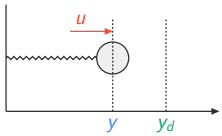
[1] E. Feron. From Control Systems to Control Software. *IEEE Control Systems Magazine* 30(6):50–71, Dec. 2010.
[2] J. Feret. Static Analysis of Digital Filters. ESOP 2004, LNCS 2986, Springer (2004) 33–48.

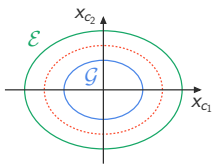
Pour en savoir plus :
`http://www.cri.ensmp.fr/classement/doc/A-556.pdf`

Exemple : système masse-ressort [1]

Système mécanique à 1 degré de liberté : **masse** accrochée à un **ressort**, contrainte de se déplacer dans une seule direction.

Le contrôleur exerce une action u afin que la position y de la masse atteigne la consigne y_d.






```
Ac = [0.4990, -0.0500; 0.0100, 1.00];
Bc = [1; 0];
Cc = [564.48, 0];
Dc = -1280;
xc = zeros(2, 1);
receive(y, 2); receive(yd, 3);
while (1)
  % xc ∈ E
  yc = max(min(y - yd, 1), -1);
  u = Cc*xc + Dc*yc;
  xc = Ac*xc + Bc*yc;
  send(u, 1);
  receive(y, 2);
  receive(yd, 3);
  % xc ∈ G ⊂ E
end
```

Condition de stabilité (Lyapunov) : la variable d'état x_c reste dans une certaine ellipse $\mathcal{E}$ durant l'exécution.

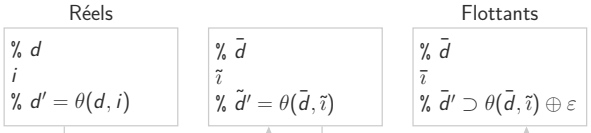
Preuve de stabilité dans R : fournie sous forme d'**invariants**
- À l'entrée de la boucle, x_c appartient à $\mathcal{E}$;
- En sortie, x_c appartient à une ellipse $\mathcal{G} \subset \mathcal{E}$;

En flottants ?
- Altération des constantes numériques A_c, B_c, C_c, D_c ;
- Erreurs d'arrondi lors du calcul de x_c.

$\Longrightarrow$ **Preuve inutilisable en flottants.**

Passage aux flottants — cadre théorique

Transposition code + invariants en 2 étapes :



Code : **constantes** converties en flottants

Invariants recalculés en utilisant les mêmes théorèmes θ, appliqués aux nouvelles constantes

Code : **fonctions** (+, ∗, ...) remplacées par leurs équivalents flottants

Invariants « élargis » pour inclure l'erreur d'arrondi
Conservation de la forme ellipsoïdale pour propagation

Cadre prouvé en Coq.

Passage aux flottants — automatisation

LyaFloat : implémentation pour systèmes linéaires à invariants de Lyapunov
- Traduction du programme en flottants ;
- Transposition **automatique** des invariants ;
- Vérification de la condition de stabilité.

Possibilité de jouer sur la **précision**.

Programmé en Python (~ 500 LOC) avec la bibliothèque SymPy.

Application au système masse-ressort

Calcul d'une ellipse $\mathcal{F}$ (en pointillés rouges ci-dessus) déduite de $\mathcal{G}$ telle qu'en sortie de boucle $x_c \in \mathcal{F}$, puis vérification de l'inclusion $\mathcal{F} \subset \mathcal{E}$.

Résultats :
- **Boucle ouverte** : système stable avec des flottants 32 bits.
- **Boucle fermée** : échec de l'analyse avec 32 bits, fonctionne avec 128 bits.

An efficient off-line configuration of an electric vehicle energy management software

Borjan Tchakaloff[1,2], Sébastien Saudrais[1], and Jean-Philippe Babau[2]

[1] CERIE, ESTACA, F-53000 Laval, France
[2] Univ. Bretagne Occidentale, UMR 6285, Lab-STICC, F-29200 Brest, France
borjan.tchakaloff@estaca.fr

1 Introduction

Current electric vehicles can handle a few hundred kilometres. One way to deal with the embedded energy issue of the electric vehicles is through software. An energy management software (EMS) is a high level software monitoring and managing an environment through specific-purposed components. EMS are commonly used in (full- and hybrid-) electrical vehicles, though they mainly manage only the engine and ignore the end-user provided Quality of Service (QoS). In order to offer an efficient energy management and to take into account the user-related QoS, an electric vehicle EMS has to consider every embedded devices and the user expectations.

In [1], we present the Orqa framework to tackle the global energy management while providing a QoS as good as possible. In Orqa, each embedded device is characterised at design phase by its energy consumption(s) and its quality(ies), if applicable. The framework offers to realise a component architecture which will elaborate on-line a solution to achieve the driver request (reach a destination) while providing the best possible vehicle QoS. The main idea is to limit by software the engine and the other devices usage to match the driver policy. Also, the driver preferences are introduced to have a QoS matching his expectations. But the solution space in which operates Orqa exponentially increases with the routes amount and the number of devices. To reduce the solution space, the challenge is to propose efficient models and an associate configuration for the framework. So the solution domain is tuned according to the target vehicle characteristics and abilities, while still providing various viable solutions. This paper is a summary of [2] and presents two approaches to effectively reduce the on-line complexity of the search process.

2 Off-line configuration

The solution space explored by the EMS is composed of the routes and their variations (different driving conditions). A specific coefficient (the *velocity coefficient*) is applied to the nominal route velocities to generate a route variation. So the amount of velocity coefficients directly impacts the on-line search process as it determines the number of possibilities. We define two approaches to accelerate the search process: a reduction of the search input dimension and an approximation of the route variations.

The first approach is based on data clustering, it lies on grouping the routes to limit the on-line exploration. The idea is to define a limited number of representative groups of velocity coefficients, instead of using the whole domain. The partitioning is based on the *k-medoids* partitioning algorithm initialised with the *k-means++* algorithm. The input of the algorithm is a set of vectors. Each vector represents a velocity coefficient, it is characterised by the duration and consumption ratios of every routes for that coefficient. Duration and consumption ratios evolve independently so they both have to be considered for partitioning. The partitioning algorithm yields a set of k (from 1 to the number of coefficients) groups of vectors. A set of routes are in the same group if they minimise the relative distance-based error to one particular vector of the group (the representative vector). As a vector represents a velocity coefficient, the k groups of vectors returned by the partitioning algorithm lead to k groups of velocity coefficients. And the representative vector of a group leads to the representative velocity coefficient of this group. If k is not fixed, a full

range of clustering has to be realised to find the "best" number of groups. The *elbow criterion* is a visual method to determine an adequate configuration based on the evolution of the global error metric. In this method, the solution is found when the results plot forms an important angle (the elbow). So the best coefficients group is selected based on the global error metric evolution of all the groups.

In the second approach, we consider the fact that ratios, for different routes, evolve in a same way. We propose to approximate a set of ratios evolution with a representative evolution (called the *approximation function*) based on a regression analysis. At each velocity coefficient (the evolution step), the ratios are represented by one ratio called the *approximated ratio*. The approximation functions resulting of this approach are given to the designers as hints to optimise the variation computation. Indeed, the complete evaluation of the variations is replaced by approximating the nominal route results, so the on-line complexity is greatly reduced. The duration and the consumption results evolve differently, their ratios are thus approximated independently. So it is possible that there is one approximation for the duration ratios and many for the consumption ratios. As we deal with approximations, we have to evaluate the reliability of such solutions. We define the *reliability* of an approximating ratio by its relative deviation range. For instance, we choose three reliability levels: 1) high (a relative deviation less than 5%), 2) mild (a relative deviation in-between 5% and 10%), and 3) weak (a relative deviation more than 10%). A highly reliable approximation means that the approximated ratio does not vary more than 5% from the exact ratio. And as the ratios are used to obtain the route variations results, the reliability is transitive to the result values. Also, the *reliability coverage* of an approximation function is the number of items for which the approximation belongs to a certain reliability level. So it is a more global metric than the reliability that applies to a whole group of velocity coefficients.

3 Conclusion

The two reduction approaches are experimented on three hundred generated routes (100 per route environment: urban, rural and motorway) for the specific vehicle presented in [1]. We then evaluate the two approaches against three different routes (one per environment). We see that the new solution spaces are effectively reduced (24 evaluations in the first approach) or less complex to compute (6 evaluations and 360 approximations in the second approach) than the complete solution space (366 evaluations). On the three example routes, the obtained driving strategies are quite close to the optimal ones for the two proposed approaches. For the urban and the rural routes, we see that the error is within the relative deviation range when relying on the ratios approximation approach but not for the motorway route. The first approach, grouping the velocity coefficients, produces less accurate solutions with results variating more than 10% from the optimal ones. On the other hand, a composition of the two approaches (6 evaluations and 24 approximations) gives mixed results. They are coherent with the two approaches but the results are sub-optimal solutions. This composition is not adequate in this use-case.

We present two different approaches to effectively configure the decision models of an Energy Management System off-line. They are based on extensive results matching the targeted vehicle capacities. The approaches can be used exclusively or in combination. The application of these approaches leads to a suitable configuration optimised for the vehicle EMS.

References

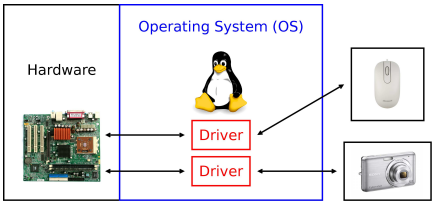
1. Tchakaloff, B., Saudrais, S., Babau, J.P.: ORQA: Modeling Energy and Quality of Service within AUTOSAR Models. In: Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures - QoSA'13. p. 3. ACM Press, Vancouver, British Columbia, Canada (2013), `http://dl.acm.org/citation.cfm?doid=2465478.2465488`
2. Tchakaloff, B., Saudrais, S., Babau, J.P.: Efficient models configuration for an electric vehicle energy management software. In: Proceedings of the 40th Euromicro Conference on Software Engineering and Advanced Applications - SEAA'14. IEEE, Verona, Italy (2014)

Understanding the Genetic Makeup of Linux Device Drivers

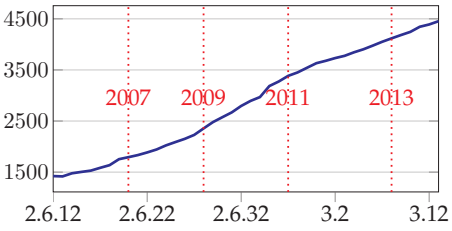
Peter Senna Tschudin, Laurent Réveillère, Lingxiao Jiang, David Lo, Julia Lawall, Gilles Muller

LIP6 Inria & UPMC, LaBRI, Singapore Management University

What is a device driver?



Number of drivers



Continual linear increase over ~10 years

Problem

Device driver development is:

- Critical to OS reliability
- Error-Prone
- Complex
- Expensive
- Slow

Our vision

Drivers are made of genes!

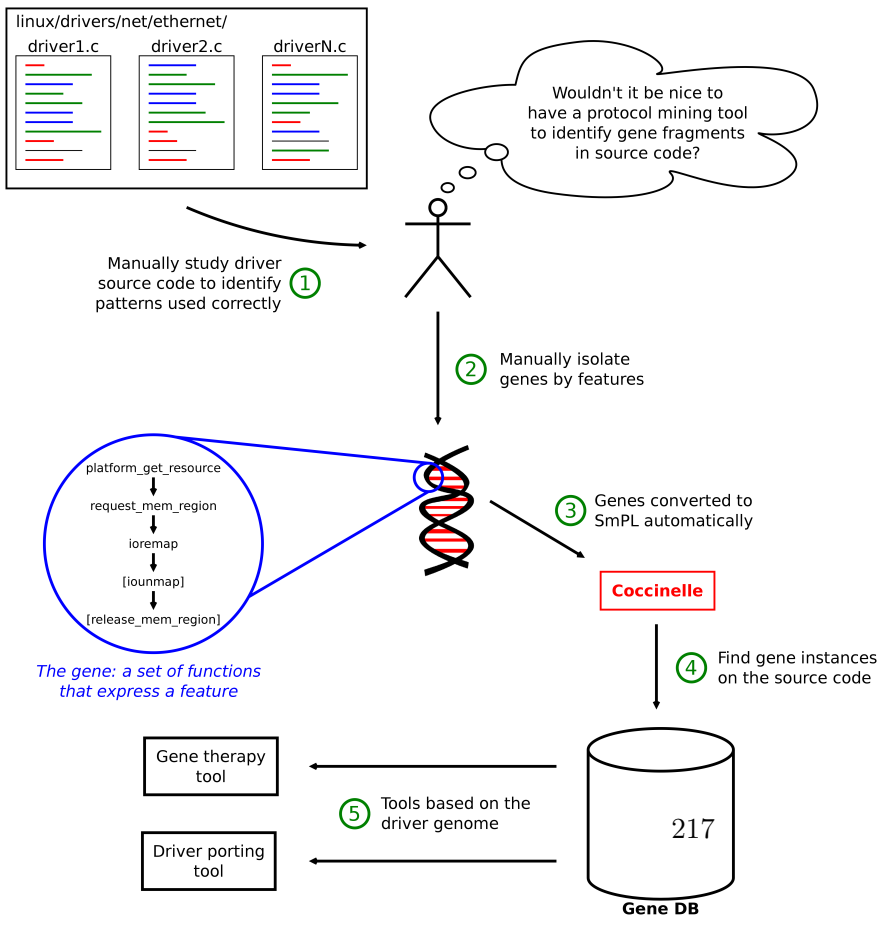
A gene:

- Motivated by device features and OS API
- Set of possibly non-contiguous code fragments
- Express the behavior of a feature

Applications

- Evolution (Gene therapy)
- Porting
- Ease development

Our proposal



Coccinelle

- Static analysis tool
- For program matching and transformation
- Provides the SmPL Semantic Patch Language

Protocol Mining

- Identifies commonly used function sequences
- Better results than clone detection (common blocks)
- Work in progress

Software Evolution Multi-Views From the Smart Home to the Cloud

Amal Tahri

Context

Smart Home characteristics

- Heterogeneity: different *elements* e.g., devices, services and technologies.
- Variability: different characteristics e.g., the amount of offered resources for a given device.
- Volatility: changing set of applications deployed on a changing set of elements.

➢ Where to deploy new applications? In the Smart Home? On the Cloud? Or distributed on both?

➢ How to deploy them? Only in one place? By distributing the applications between the Cloud and the Smart Home?

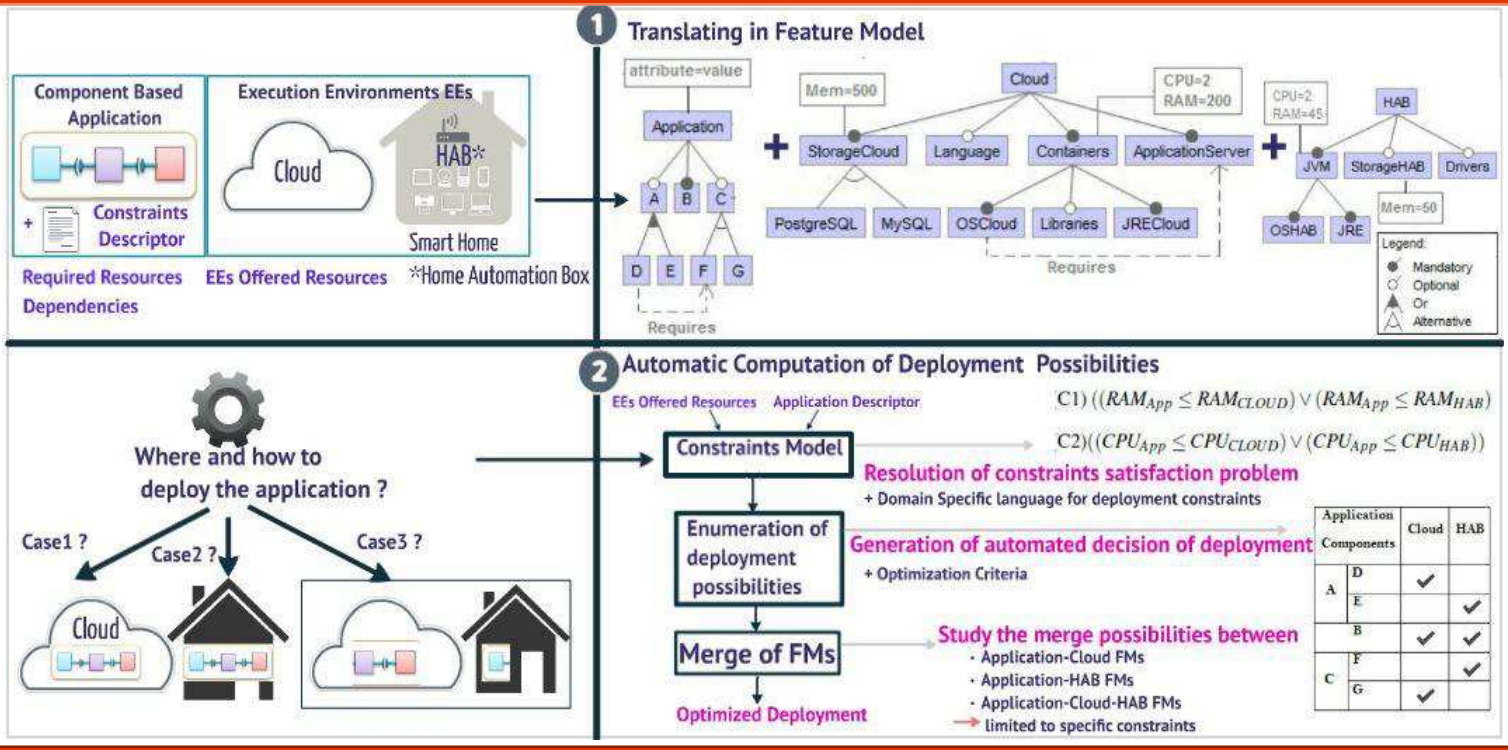
Motivations

1. Balance the computational load between the Smart Home and the Cloud.
2. Reduce the acquisition cost for users by limiting the embedded systems resources.
3. Reduce the scalability cost when application set grows.

Challenges

1. Optimization of the deployment of a new component-based application onto a distributed environment (Smart Home + Cloud) despite the heterogeneity and the variability.
2. Run Time self-adaptation of the application deployment to volatility.

Approach



Ongoing Work

1. Choosing FM formalism to manage variability and heterogeneity [1].
2. Extending FM by addressing *Localization Constraints* e.g., Colocation, Dislocation of software components.
3. Proposing a DSL to express the deployment constraints.

Future Work

1. React to the volatility by run time self-adaption of the application deployment using code offloading and migration between the Smart Home and the Cloud
2. Build a self-* architecture supporting the self-configuration, self-optimization and self-adaptation of the application deployment[2].

References

[1] Benavides, David, Sergio Segura, and Antonio Ruiz-Cortés, « Automated Analysis of feature models 20 years later: A Literature view » Information System 35.6 (2010):615-636.

[2]Movahedi, Zeinab et al. « A Survey of Autonomic Network Architectures and Evaluation Criteria » Communications Surveys & Tutorials, IEEE 14-2(2012):464-490.

Evolution « agile » de lignes de systèmes d'information

Eddy Ghabach[1, 2], Mireille Blay-Fornarino[1], Franjieh El Khoury[2,3] et Badih Baz[2]

[1]Univ. Nice Sophia Antipolis, CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
ghabach.eddy@etu.unice.fr, blay@unice.fr
[2]Université Saint Esprit de Kaslik, 446 Jounieh – Liban
franjiehelkhoury@usek.edu.lb, badihbaz@usek.edu.lb
[3]Université Lyon1, Laboratoire ERIC, Equipe de recherche DMD, 5, avenue Pierre Mendès-France, 69676 Bron Cedex, France
franjiehkhoury@hotmail.com

Mots-clés: Lignes de produits logiciels, Systèmes d'information, Reverse Engineering.

1 Introduction

En janvier 2014, le *Gartner group* annonce que les dépenses en matière de "logiciels d'entreprise" connaitront le taux de croissance annuel le plus important dans le domaine de l'IT, avec des prévisions de 6,8 pour cent[1]. En effet, utilisés au quotidien, ces systèmes doivent évoluer en particulier pour améliorer les relations entre l'homme (client, commercial, gestionnaire de logistiques, ..) et la machine, prendre en compte les nouveaux modes de communication et ainsi accroître l'efficacité même des entreprises. La complexité intrinsèque de ces systèmes a conduit à la réalisation de systèmes d'information (SI) « sur étagères » qui doivent aujourd'hui s'intégrer dans des démarches de développement agiles [11].

Les lignes de produits logiciels visent à supporter la variabilité des applications par l'augmentation de la productivité et de la qualité, et la diminution des coûts et du temps de mise sur marché en s'appuyant sur la réutilisation intensive [13]. Ainsi la construction d'une ligne de produits repose sur la planification d'une réutilisation systématique. Pour répondre aux besoins d'adaptations, différents travaux ont proposé d'allier agilité et lignes de produits logiciels [4,9]. Dans la lignée de ces recommandations nous nous intéressons plus spécifiquement à l'évolution des lignes de systèmes d'information dirigée par les utilisateurs.

2 Problématique & état de l'art

Une ligne de produits logiciels est un ensemble de systèmes logiciels partageant des propriétés communes, développés sur la base d'un ensemble de composants [13]. En particulier les features models sont une manière de capturer les similitudes et les variations entre les produits de la ligne [1]. Pour faire face à la complexité croissante des systèmes et à la nécessité d'un développement centré sur les parties prenantes, des écosystèmes logiciels se sont développés. Ils se caractérisent par une décentralisation des développements par des "communautés " de développeurs et l'association des utilisateurs dans le processus même du développement [4]. Malheureusement, le relevé des exigences des utilisateurs reste une étape difficile, qui peut conduire à des livraisons insatisfaisantes [4]. De plus l'approche de développement Top-Down est couteuse, en temps, voire même en complexité. Nous constatons ainsi sur le terrain l'introduction de nouvelles fonctionnalités directement au niveau des codes. La capitalisation des informations dans la ligne est alors une tâche additionnelle et complexe, qui, si elle n'est pas réalisée, peut conduire à l'obsolescence de la ligne qui perd en qualité et surtout en intérêt. De fait l'évolution des produits issus d'une ligne de SI met en jeux les différents éléments qui composent un SI : base des données, interactions

[1]http://www.gartner.com/newsroom/id/2643919, 31 mars 2014

Homme-Machine(IHM), Processus métiers[2]. Les travaux menés dans le cadre des lignes de produits multiples [3] et de la séparation de préoccupations [12] sont des pistes pour maitriser les lignes de produits complexes. Notre étude porte actuellement sur leur usage dans le contexte spécifique des SI.

3 L'utilisateur au centre du processus d'évolution

Nous proposons de supporter l'ajout de nouveaux produits dans une ligne de SI par une approche dirigée par les modèles [7] et fondée sur des résultats issus du « Reverse Engineering » [6,10]. Au niveau organisationnel, le processus de développement agile proposé par Bosch *et al.*,nous sert de point de départ [4]. En effet il met en avant l'étape de détermination des « features » non présents dans la ligne, mais identifiés par l'utilisateur. Au niveau opérationnel, nous envisageons de travailler au niveau des formules de logique correspondant aux feature models, et des « modèles paramétrés » au niveau de la description du métier. Pour la partie, mémorisation dans la ligne nous nous appuierons sur des méthodes de « Reverse Engineering » pour extraire les modèles conceptuels à partir du code[10]. Le poster présentera ce processus à travers une étude de cas.

Bibliographie

1. Apel, S., Kästner, C.: An Overview of Feature-Oriented Software Development., (2009).
2. Authoserre, A. et al.: Interopérabilité des Systèmes d'Information : approches dirigées par les modèles. 30ème congrès INFORSID(INFORSID'2012). Montpellier (France) (2012).
3. Bosch, J.: Toward Compositional Software Product Lines. IEEE Softw. 27, 3,(2010).
4. Bosch, J., Bosch-Sijtsema, P.M.: Introducing agile customer-centered development in a legacy software product line. Softw. Pract. Exp. 41, 8, 871–882 (2011).
5. Brummermann, H. et al.: Variability issues in the evolution of information system ecosystems. Proceedings of the 5th Workshop on Variability Modeling of Software-Intensive Systems. pp. 159–164 ACM, New York, NY, USA (2011).
6. Deltombe, G. et al.: Bridging KDM and ASTM for Model-Driven Software Modernization. SEKE. pp. 517–524 KnowledgeSystems Institute GraduateSchool (2012).
7. Godet-Bar, G. et al.: Sonata: Flexible connections between interaction and business spaces. J. Syst. Softw. 85, 5, 1105–1118 (2012).
8. Holl, G. et al.: A systematic review and an expert survey on capabilities supporting multi product lines. Inf. Softw. Technol. 54, 0, 828–852 (2012).
9. Leitner, A., Kreiner, C.: Software Product Lines - An Agile Success Factor? In: O'Connor, R. et al. (eds.) EuroSPI. pp. 203–214 Springer (2011).
10. Mannino, M. V: Database Design, Application Development, and Administration. McGraw-Hill/Irwin (2004).
11. Navarrete, F. et al.: Reconciling Agility and Discipline in COTS Selection Processes. Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. Sixth International IEEE Conference on. pp. 103–113 (2007).
12. Rashid, A. et al.: Aspect-Oriented, Model-Driven Software Product Lines The AMPLE Way. Cambridge UniversityPress (2011).
13. Trigaux, J.C., Heymans, P.: Software Product Lines: State of the art. (2003).
14. Bosch J., Bosch-Sijtsema P., "Introducing agile customer-centered development in a legacy software product line", SOFTWARE PRACTICE AND EXPERIENCE, 2011,
15. Bosch J., "Toward Compositional Software Product Lines", IEEE Software, 2010, p.29-34
16. NAVARRETE, F., BOTELLA, P. AND FRANCH, X., 2007. Reconciling Agility and Discipline in COTS Selection Processes. In *Commercial-off-the-Shelf (COTS)-Based Software Systems, 2007. ICCBSS '07. Sixth International IEEE Conference on*. pp. 103–113.

Modèles algorithmiques pour les sciences de la nature

Konrad Hinsen[1,2]

[1] Équipe "Biophysique théorique, simulation moléculaire, et calcul scientifique", CBM, Orléans, France
[2] Synchrotron SOLEIL, Division Expériences, Saint Aubin, France
konrad.hinsen@cnrs-orleans.fr

Introduction

Le rôle principal des logiciels aujourd'hui est celui d'outils. Un logiciel fait quelque chose, et son utilité réside dans cette action. Comme d'autres outils, les logiciels sont en général produits par des professionnels pour leurs clients, et évalués par des critères de conformité à la spécification, efficacité, facilité d'utilisation, etc. Les outils de développement de logiciel et les langages de programmation ont comme objectif de faciliter le travail de ces professionnels.

Un des rares domaines d'application ou les logiciels ne sont pas que des simples outils est la recherche scientifique. Au début de l'ère du calcul scientifique, les logiciels étaient des outils pour explorer les modèles mathématiques développés par les théoriciens. Puis, l'augmentation de la puissance des ordinateurs a permis l'application des méthodes de calcul à des modèles de plus en plus complexes, qui ont intégré des algorithmes à côté des équations mathématiques traditionnelles. Aujourd'hui, les logiciels scientifiques sont souvent la seule expression complète et précise des modèles théoriques.

Pourtant, le code source d'un logiciel est une très mauvaise notation pour un modèle scientifique. Le modèle représente habituellement une partie infime de ce code source, dont la majeure partie est dédiée à la gestion des ressources, aux entrées/sorties, etc. Il est presque impossible d'identifier un modèle scientifique dispersé dans des milliers de lignes de code optimisé et parallélisé. Ceci est un facteur majeur de ce qu'on appelle la "crise du logiciel scientifique" [1,2] : on découvre de plus en plus d'erreurs graves dans les travaux de recherche basés sur le calcul, qui sont dues aux erreurs de programmation mais aussi à l'incompréhension des utilisateurs qui ne connaissant pas vraiment les méthodes qu'ils appliquent.

Modèle algorithmique ≠ logiciel

Un *modèle scientifique* est une représentation d'un aspect de la nature qui permet de faire des prédictions qu'on peut comparer avec des observations. Cette comparaison sert à *valider* et à *améliorer* le modèle. Un modèle *quantitatif* fait des prédictions numériques. Il est donc une fonction calculable. Celle-ci peut être exprimée dans un langage Turing-complet et donc aussi dans n'importe quel langage de programmation.

Pourtant, un modèle n'a pas comme vocation principale d'être évalué numériquement comme partie intégrante d'un logiciel. L'évaluation n'est qu'un des nombreux aspects du travail avec un modèle. Il y a aussi : (1) la dérivation d'un modèle concret en partant d'une théorie plus générale exprimée par des équations mathématiques, (2) la transformation d'un modèle, notamment dans le but d'introduire des approximations, (3) la composition de plusieurs modèles qui décrivent des aspects complémentaires d'un phénomène, (4) le raisonnement mathématique, (5) le raisonnement basé sur la connaissance non formalisée du domaine scientifique.

Je propose une nouvelle approche au calcul scientifique basée sur une séparation nette de deux aspects : d'un côté les modèles et méthodes scientifiques, de l'autre côté les outils de calcul [3]. Les modèles scientifiques sont développées par des chercheurs, et validés par la confrontation avec l'observation. Les outils logiciels sont développées par des ingénieurs, vérifiés par les techniques du génie logiciel, y compris (idéalement, ultérieurement) les preuves formelles, et évalués en tant qu'outils pour la recherche.

La distinction entre modèles et outils que je propose correspond aux principes de la science, mais pas à la pratique du calcul scientifique d'aujourd'hui. Tout ce qui influe sur la valeur d'une quantité potentiellement observable dans la nature fait partie du modèle. Ceci inclue des aspects traditionnellement considérés "techniques" comme la discrétisation d'une équation différentielle ou l'approximation d'un nombre réel par un nombre à virgule flottante. Les outils de calcul dans mon approche ne sont pas censés apporter des modifications quelconque aux résultats, ce qui rend possible leur vérification par des preuves formelles.

Informatique + calcul scientifique = meilleure science

Les développements technologiques qui peuvent rendre une telle approche faisable nécessitent la collaboration entre les praticiens du calcul scientifique (dont je fais partie) et les informaticiens. L'objectif de ce communiqué est de faire un premier pas pour établir un échange sur ce sujet. À ce jour, je n'ai aucun résultat à présenter, seulement une liste des technologies existantes qui peuvent servir comme points de départ, et un résumé de ce qui est à faire.

Les points de départ

- Le calcul formel, déjà très utilisé pour le travail avec les équations mathématiques qui sont les précurseurs des modèles calculables.
- Les langages dédiés pour définir les équations mathématiques dans certains logiciels.
- Les modèles de données formalisés pour des représentations partielles de modèles scientifiques (SBML, NeuroML, MOSAIC, ...).
- Les langages de modélisation définis pour certains domaines d'application (Modelica, ...).
- L'ingénierie dirigée par les modèles (IDM) qui est une approche au développement logiciel proche de ce que je propose ici.
- La réécriture, très utilisée dans le calcul formel et pour la transformation de code.

Les développement qui restent à faire

- Une notation formelle pour définir des modèles scientifiques algorithmiques. Elle doit dépasser les équations gérées en calcul formel en permettant des modèles algorithmiques. Elle se distingue des langages dédiés existants du calcul scientifique par son aptitude aux tâches autres que l'évaluation par un logiciel.
- Des outils de manipulation pour ces modèles.
- Des générateurs de code pour produire des outils logiciels à partir des modèles.
- Des techniques de validation pour les outils logiciels, idéalement basées sur les preuves.

Travaux connexes

Deux informaticiens anglais ont récemment publié un agenda [4] qui vise également une séparation entre les modèles scientifiques et leur implémentation, mais se concentre sur la transition entre les technologies d'aujourd'hui et des technologies futures.

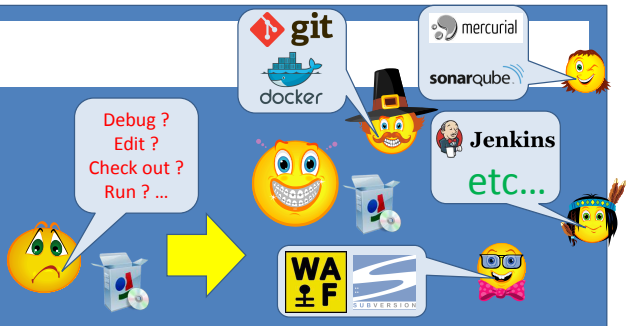
Références

1. Zeeya Merali, "Computational science : ...Error", Nature **467**, 775-777 (2010)
2. A Morin, J Urban, PD Adams, I Foster, A Sali, D Baker, P Sliz, "Research priorities : Shining light into black boxes.", Science **336**, 159–160 (2012)
3. Konrad Hinsen, "Computational science : shifting the focus from tools to models [v1 ; ref status : awaiting peer review, http ://f1000r.es/3af]", F1000Research **3**, 101 (2014)
4. Dominic Orchard, Andrew Rice, "A computational science agenda for programming language research", accepted at International Conference on Computational Science 2014, http ://www.cl.cam.ac.uk/ dao29/publ/iccs14-orchard-rice.pdf

Réseau des acteurs du DÉVeloppement LOGiciel
au sein de l'Enseignement Supérieur et de la Recherche (ESR)

Pour Qui ?

- Toute personne impliquée dans la conception, la programmation, la diffusion, le maintien de logiciels au sein de l'ESR
  - Personnel permanent ou non
  - Chercheurs(euses), Ingénieur(e)s, Techniciens(nes) ou Administratifs(ves)



DevLOG :
Vous n'êtes pas seul avec votre logiciel !

Avec quels objectifs ?

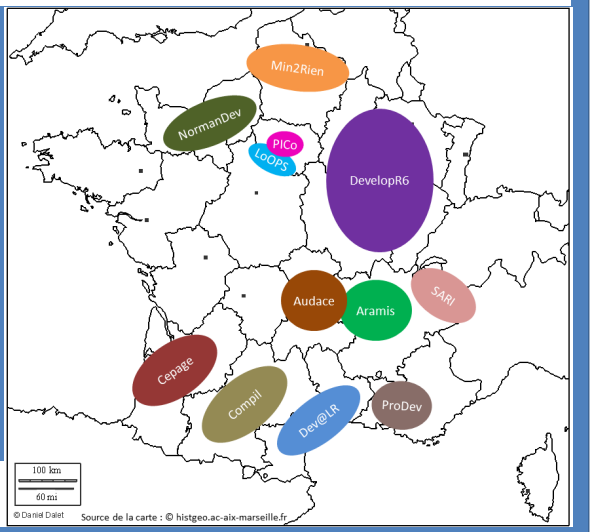
- Favoriser les échanges entre acteurs du développement logiciel au sein de l'ESR
- Soutenir les réseaux régionaux ou thématiques dans leurs actions et formations
- Construire une offre d'actions et de formations pertinente et complémentaire
- Faciliter les échanges entre la communauté et les tutelles (pour remonter les réalités du terrain par exemple)

Comment ?

- Liste de discussion : **devlog@services.cnrs.fr**
- Site Web : **http://devlog.cnrs.fr/**
- Journées Nationales JDEV : **http://devlog.cnrs.fr/jdev2013**
- Groupes de travail : GT NoSQL, GT Développement et Base de données, …
- Organisation de journées d'échanges et de formation, regroupement de demandes de formations « pointues », en lien avec les réseaux régionaux et thématiques
- Mise en commun de ressources, de savoir-faire, de connaissances, …

Rejoignez DevLOG !

- Participez aux Groupes de travail ou proposez en de nouveaux
- Partagez vos besoins et vos connaissances
- Proposez l'organisation de journées d'échanges
- Rapprochez –vous de DevLOG et des réseaux régionaux
- En absence de réseau régional, créez-en un. Si besoin, DevLOG peut vous aider



Soutiens

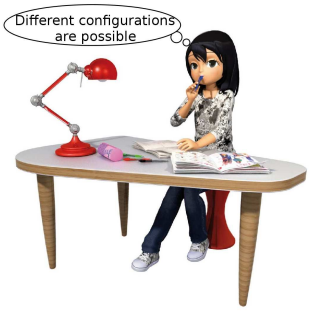
- DevLOG est soutenu par la Mission pour l'interdisciplinarité du
- Un partenariat privilégié existe avec l'

From Natural Language Requirements to Formal Specifications using an Ontology

Driss Sadoun, Catherine Dubois, Yacine Ghamri-Doudane, Brigitte Grau

Project Envie Verte

Allowing a user to configure the behaviour of a system by specifying her requirements using natural language (NL).

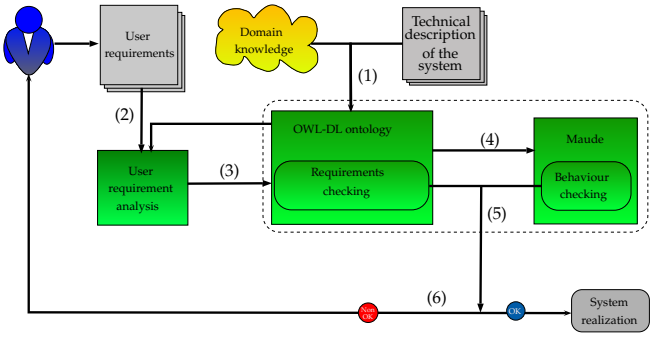


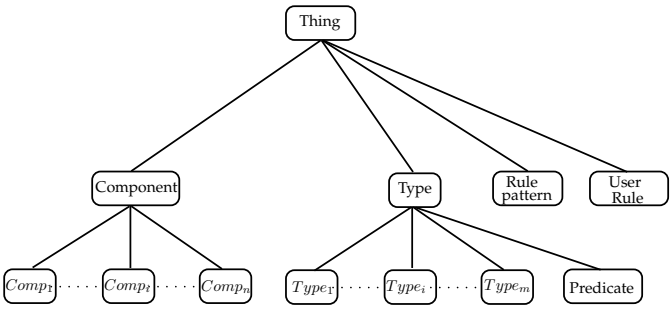
Problematic

Dealing with NL requirements leads to problems related to ambiguous, inconsistent or incomplete textual requirements.
⇒ specifications need to be formalized, checked and corrected.

- How to fill the gap between NL requirements and formal specifications ?
- How to identify user requirements from NL texts ?
- How to produce a formal representation of the system behaviour ?
- How to ensure the consistency of the behaviour described by the requirements ?

Our approach : An ontology as a pivot representation





High level ontology of system behaviour

From NL requirements to ontology representation

(1) Build a high level ontology, modelling the generic behaviour of the system ;

- An ontology models concepts and properties, defining a conceptual vocabulary of a domain.
- An ontology models constraints of a domain defining a formal framework.

(2) Analyse user requirements written in NL ;

- automatic acquisition of the LN vocabulary and syntactico-semantic analysis rules.
- Requirement analysis is guided by the ontology and its semantic constraints.

(3) Specialize the system behaviour through the ontology population from user requirements analysis.

- Populating an ontology consists in adding new instances without changing its conceptual structure.
- Ontology population aims to link instance references in texts to their conceptualizations in the ontology.
- Through its population the ontology models the system behaviour.

From ontology representation to formal specifications

(4) Transform the OWL ontology into a formal specification in Maude ;

- An ontology is sufficiently formal to allow an easy and intuitive translation into a formal specification language.

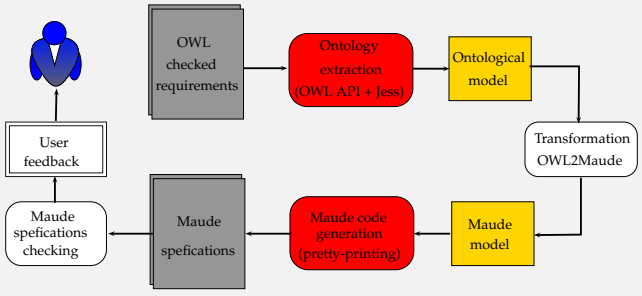
User requirements checking

(5) Check the models issued from NL requirements ;

- An ontology is sufficiently formal to allow different kinds of inferences as completeness and consistency checking ;
- Maude allows different kind of verifications using rewriting logic.

(6) Verification feedbacks to the user until the model is consistent and fits her needs.

(4) From an OWL ontology to Maude specifications



Conclusion

- The ontology provides a formal framework to bridge the gap between NL requirements and formal specifications ;
- Ontology reasoning allows checking requirements consistency and completeness ;
- Maude formal specification enables us to check the system behaviour ;
- Feedback from the checking process helps the user to improve its requirements.

Bibliography

- D. Sadoun, Dubois C., Ghamri-Doudane Y., Grau B. From Natural Language Requirements to Formal Specification using an Ontology. In ICTAI 2013.
- D. Sadoun, Dubois C., Ghamri-Doudane Y., Grau B. Peuplement d'une ontologie guidé par l'identification d'instances de propriété. In TIA 2013.

driss.sadoun@limsi.fr

ourgoin & Emmanuel Chailloux firstname.lastname@lip6.fr
Universités, UPMC Univ Paris 06, UMR 7606, LIP6, APR, F-75005, Paris, France

Performance Web-Client Programming with SPOC

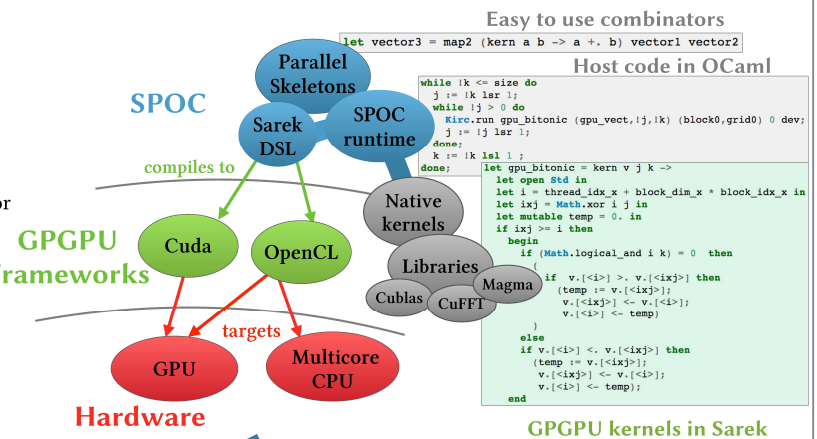
SPOC : Efficient Abstractions for GPGPU Programming

le
es Cuda/OpenCL/**WebCL**
mpilation from Sarek to
/OpenCL

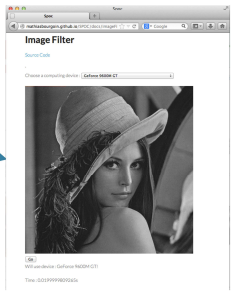
genous
patible with multiple
es
mically adapts its behavior
y target

sible and Safe
matic memory transfers
c type checking
/GPU code in a single file
a coherent language

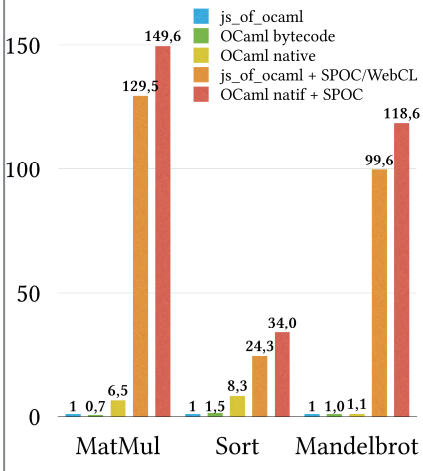
Performance
patible with hand-tuned
e kernels and libraries
lel skeletons offer
matic optimisations



js_of_ocaml : -client programming with OCaml



Benchmarks



Speedups (higher, better) using
- Intel Core-i7 3770 (CPU)
- Geforce GTX 680 (GPU)

SPOC website

http://www.algo-prog.info/spoc

References

J. Vouillon and V. Balat
From bytecode to javascript: the js_of_ocaml compiler.
Software: Practice and Experience, 2013

M. Bourgoin, E. Chailloux and J.-L. Lamotte
Efficient Abstractions for GPGPU Programming
International Journal of Parallel Programming, 2013

M. Bourgoin, E. Chailloux and J.-L. Lamotte
Spoc: GPGPU Programming Through Stream Processing with OCAML
Parallel Processing Letters, 2012

Verifying the Safety of User Pointers Using Static Typing

Etienne Millon[1,2], Emmanuel Chailloux[1], and Sarah Zennou[2]

[1] Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, LIP6, F-75005, Paris, France
firstname.lastname@lip6.fr
[2] EADS Innovation Works firstname.lastname@eads.net

1 Introduction

Modern operating systems such as GNU/Linux are comprised of several independent programs: the kernel, which handles the machine's hardware as well as trusted core features; and user applications. Those cannot run low-level instructions, so in order to perform anything visible, they have to use the kernel's facilities via the narrow interface of *system calls*.

Some system calls pass data through pointers. If the user supplies a pointer whose value is in the kernel's reserved area, this zone will be overwritten, potentially leading to a security breach. In a sense, the kernel has been abused because it accessed the memory with its own privileges instead of the originator's. This is known as the *Confused Deputy Problem* [Hardy(1988)].

When implementing a system call, it is thus necessary to forbid direct dereference of a pointer whose value can be controlled by the user. In particular, one has to dynamically verify that addresses provided by the user lies within the process's memory space. The object of this work is to use a type system to statically detect the places where it is necessary to insert these dynamic checks. We implemented this analysis on top of Newspeak [Hymans and Levillain(2008)], a minimal language for static analysis developed by EADS Innovation Works.

2 Approach

We define Safespeak, an imperative programming language built on top of Newspeak but with a higher-level and safer memory model. The following is the syntax of programs as well as types used in its static semantics.

$e ::=$	**Expressions**	$lv ::=$	**Left-values**
$\mid c$	Constant	$\mid x$	Variable
$\mid lv$	Left-value	$\mid * lv$	Dereference
$\mid \boxminus\ e$	Unary operation	$\mid lv.l_S$	Field access
$\mid e \boxplus e$	Binary operation	$\mid lv[e]$	Indexed access
$\mid \& lv$	Pointer		
$\mid lv \leftarrow e$	Assignment		
$\mid \{l_1 : e_1; \ldots; l_n : e_n\}$	Structure	$t ::=$	**Type**
$\mid [e_1; \ldots; e_n]$	Array	$\mid$ Int, Float, ( )	Ground types
$\mid \text{fun}(x_1, \ldots, x_n)\{i\}$	Function	$\mid t *$	Kernel Pointer
$\mid e(e_1, \ldots, e_n)$	Function call	$\mid t$ @	User Pointer
$\mid$ Taint(e)	Tainted value	$\mid t$ [ ]	Array
$\mid e \Leftarrow_U e$	Load from userspace	$\mid \{l_1 : t_1; \ldots; l_n : t_n\}$	Structure
$\mid e \Rightarrow_U e$	Store to userspace	$\mid (t_1, \ldots, t_n) \rightarrow t$	Function

Two features are particularly important to model our problem. The first one is the explicit left-values which are used to access memory in depth ; and the second one is the distinction between kernel pointers $t *$ and user pointers t @. Both denote a memory address that can be used as a value, but they have different uses.

Kernel pointers are the usual pointers present in C. They correspond to the address of a variable, or of a part of a variable (for example if `a` is an array, then the expression `&a[2]` yields a

valid kernel pointer). It is always possible to dereference a kernel pointer, in the sense that it may produce a runtime error, but it will never create a security hole of the kind described in Section 1.

On the contrary, user pointers are opaque pieces of data that are unsafe to dereference directly, because their value is controlled by userspace. Dereferencing them is not a problem if they point to userspace, but it is a dynamic property.

Before dereferencing a user pointer (created using the TAINT(e) construct, emulating a system call), it is thus necessary to check at runtime that they are indeed who they pretend to be (or signal an error if they point to kernel memory). The two operators $\Leftarrow_U$ and $\Rightarrow_U$ can be used to perform this check and copy data.

3 Implementation and results

This type system has been implemented in a tool of approximately 1600 lines of OCaml code, released as free software under the LGPL and available at `https://bitbucket.org/iwseclabs/c2newspeak` (in directory `src/ptrtype`).

The first step consists in translating C code to NEWSPEAK. Our frontend can translate all of ANSI C and an important share of GNU extensions. Then, all type labels are removed from the program. For example, `int x = 0;` is replaced by `Decl x in x <- 0;`. Type inference is then performed using a variation of Algorithm W. A set of annotations ensures that system call parameters are typed as user pointers. Two outcomes are possible: either the program is printed with full annotations, or a unification error is displayed.

We used our tool to detect several bugs in the Linux kernel. One example is in a video driver where a command manipulates an argument in an unsafe manner (see commit `d8ab3557`).

```
1 int radeon_info_ioctl(struct drm_device *dev, void *data,
2                        struct drm_file *filp) {
3         /*!npk userptr_fieldp data value*/
4         struct drm_radeon_info *info = data;
5         uint32_t *value_ptr = (uint32_t *) ((unsigned long)info->value);
6         uint32_t value = *value_ptr;
7         /* ... */
8 }
```

The `data` parameter is a pointer to a structure whose fields come from a system call to `ioctl()`. It is thus unsafe to do this and line 6 is a security hole. On the type level, the `*` operator forces `value_ptr` to be typed t_1 $*$, but due to the annotation line 3, it is also typed t_2 @; hence, the unification fails. On the other hand, when `copy_from_user` is used, the inference succeeds.

4 Conclusion

We show that type theory can be a useful tool for verifying the absence of certain run-time properties. While adding static labels to variables seems to be a crude approximation of reality, in some cases it has enough power to capture real-world problems.

In this particular example, we work around C's lack of abstract types in order to disallow dereference for a certain class of pointers, distinguished by syntactic rules.

This is similar to the CQual systems where every type is decorated with a qualifier which can encode information such as who controls this value [Johnson and Wagner(2004)]. Our approach more specific because "qualifiers" are added only on pointers, and simpler because it does not introduce subtyping in the system.

References

[Hardy(1988)] N. Hardy. The confused deputy (or why capabilities might have been invented). *ACM Operating Systems Review*, 1988.

[Hymans and Levillain(2008)] C. Hymans and O. Levillain. Newspeak, Doubleplussimple Minilang for Goodthinkful Static Analysis of C. Technical Note 2008-IW-SE-00010-1, EADS IW/SE, 2008.

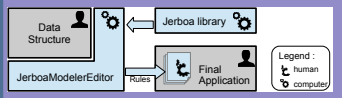
[Johnson and Wagner(2004)] R. Johnson and D. Wagner. Finding user/kernel pointer bugs with type inference. In *USENIX Security Symposium*, 2004.

Génération de modeleurs géométriques à base topologique à l'aide de Jerboa

Hakim Belhaouari et Agnès Arnould
Laboratoire XLIM-SIC – Université de Poitiers

Introduction

Jerboa est une suite d'outils dédiée à la génération de modeleurs géométriques à base topologique, utilisant les transformations de graphes.



Cette suite inclut :
- un éditeur graphique de règles
- une librairie générique d'application des règles
- une interface graphique de base pour les modeleurs générés

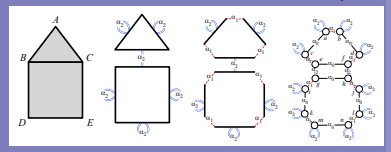
Les règles Jerboa :
- Un langage à base de règles de transformation de graphes qui offre la séparation des préoccupations (topologie, géométrie et propriétés physiques).
- Des conditions syntaxiques sur les règles garantissant la préservation de la cohérence des objets.

Modélisation géométrique à base topologique

Une structure topologique : les cartes généralisées

Les cartes généralisées sont des graphes dont les arcs sont étiquetés par les dimensions topologiques, et qui vérifient les 3 conditions de cohérence topologique : de non orientation, d'arcs incidents, et de cycle.

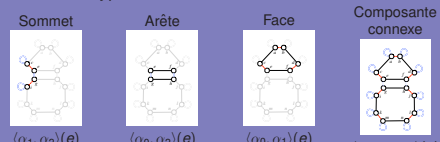
Exemple de décomposition d'un objet 2D en cellules topologique :



Cellules topologiques et orbites

Une orbite $G\langle o\rangle(b)$ est le sous-graphe de G produit à partir du nœud b et des arcs étiquetés sur le type d'orbite o.

Exemple d'orbites pour une carte généralisé 2D :

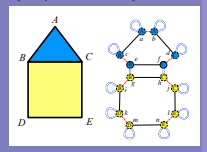


Géométrie et plongements

Les étiquettes des nœuds permettent d'associer aux orbites topologiques, les informations géométriques et physiques de l'objet.

Plongements d'un objet 2D :
- *point* : $\langle\alpha_1,\alpha_2\rangle \rightarrow$ *point_2D*
- *color* : $\langle\alpha_0,\alpha_1\rangle \rightarrow$ *color_RGB*

où *point_2D* et *color_RGB* sont des types de données définies par l'utilisateur (structures de données + opérations de manipulation).

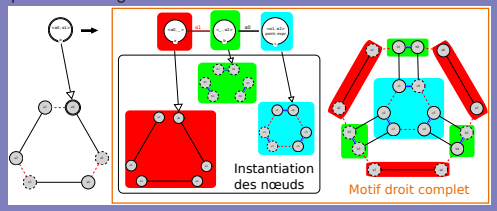


Opérations et règles de transformation

Les opérations topologiques et géométriques sont définies via des règles de transformation de graphes.

Règle Jerboa et instanciation des variables topologiques

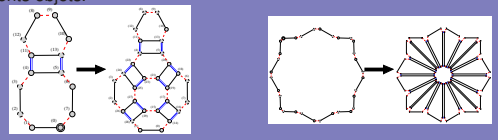
Exemple de la triangulation d'une face.



Le nœud d'accroche à gauche est instancié par l'orbite du nœud filtré (double-cercle). Les instanciations des nœuds à droite sont calculés par renommage et suppression des arcs de l'orbite filtrée (association de couleur). Au final, le motif droit est obtenu en appliquant cette règle au nœud *a*0 sur une face triangulaire.

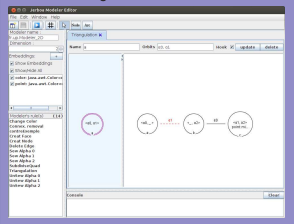
Application d'une règle

Un unique moteur permet l'instanciation et l'application de cette règle sur différents objets.

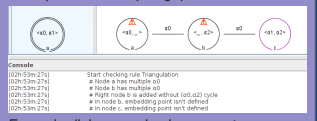


L'éditeur de règles *JerboaModelerEditor*

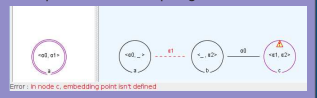
L'éditeur de règles :



Exemple d'erreur topologique :



Exemple d'absence de plongement :

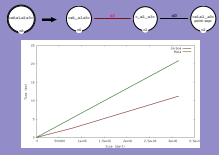


Vérifications :
- Syntaxe des règles.
- Conditions syntaxiques pour garantir la cohérence topologique (non-orientation, arcs incidents et cycle).
- Détection de l'incohérence des plongements (omission ou fusion).

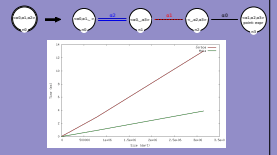
Comparaison au noyau de modeleur *Moka*

Repose sur des exemples d'objets variant jusqu'à 3 millions de nœuds.

Triangulation de toutes les faces :

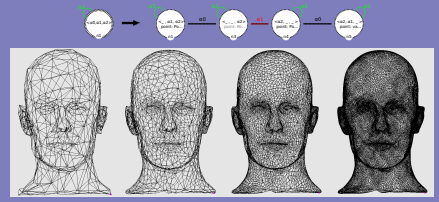


Triangulation volumique :



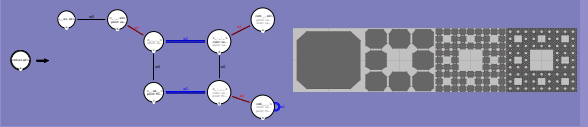
Exemples

Lissage Catmull-Clark



Tapis de Sierpinski

Structure fractale de dimension 2

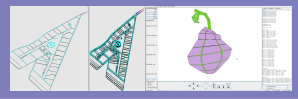


Conclusion et perspective

Conclusion

- Une suite complète de développement de modeleurs spécifiques.
- Des conditions syntaxiques qui garantissent la cohérence des objets.

Modeleur d'architecture avec reconstruction 3D guidée par la sémantique.



Modélisation précédurale pour la croissance de plantes basée sur les G-map L-systems.

Perspectives

- Composer les règles pour développer des opérations complexes.
- Compléter la vérification des expressions des plongements.

Démonstrations et Posters

RAMSES: Refinement of AADL Models for Synthesis of Embedded Systems[1]

Authors

Sébastien Gardoll

Etienne Borde

Fabien Cadoret

Laurent Pautet

Partners

Software Engineering Institute
Carnegie Mellon

Ellidiss Technologies
www.ellidiss.com

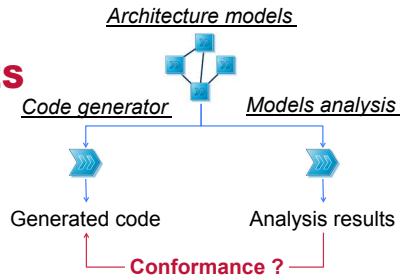
UBO université de bretagne occidentale

Safety-Critical Real-time Embedded Systems (SCRES)

- Complex software architecture
- High verification and validation (V&V) costs
- Shorter time to market

Model Driven Engineering (MDE) for SCRES

- *Architecture models:* **reduce design complexity** by abstracting implementation details
- *Code generator:* **reduce implementation efforts**
- *Models analysis:* **reduce validation costs**



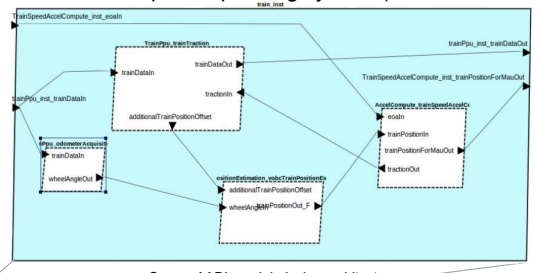
***Problem:* Impact of code generation on the validity of analysis results?**

RAMSES: AADL[1] Models Refinement

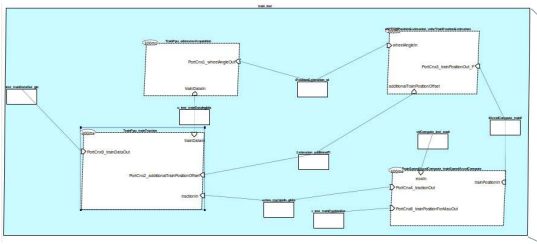
- AADL to AADL model transformation framework (Open-source)
- Integrated with OSATE[2] (Eclipse-based AADL editor)
- Code generator (currently for Ada and C for ARINC653 and OSEK compliant operating systems)
- Intermediate models analysis with AADL Inspector[3]

Results

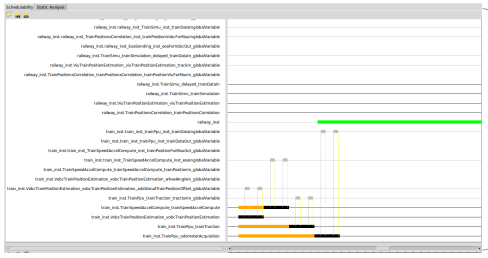
- *Reduced semantic gap* between deployed software and analysis models
- *Simple and reusable code generators*, based on low-level implementation models
- *Selection of model transformations,* for design space exploration
- *Requirements traceability* thanks to transformation links
- *Refinements for different purposes:* safety and security design patterns, etc.



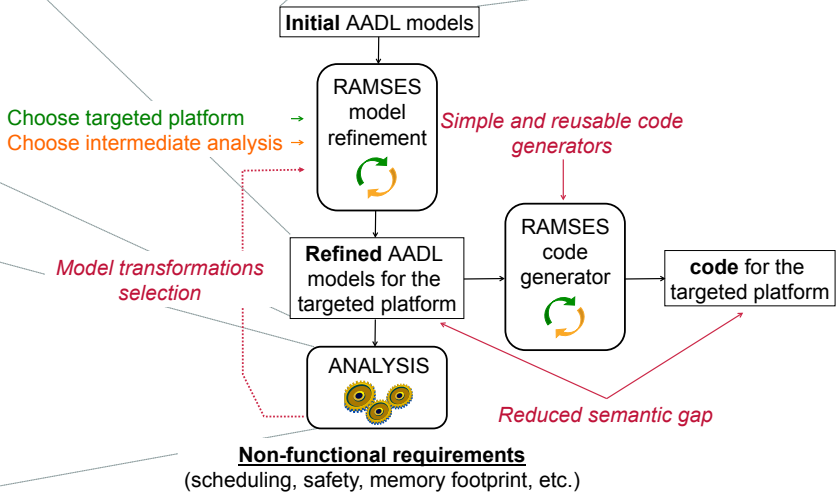
Source AADL model: design architecture



Intermediate AADL model: implementation architecture



AADL Inspector: schedulability analysis with Cheddar[4]



More information

1. RAMSES (TELECOM ParisTech/LTCI): http://penelope.enst.fr/aadl/wiki/Projects#RAMSES
2. Architecture Analysis and Design Language (SAE): http://www.aadl.info
3. OSATE (SEI): https://wiki.sei.cmu.edu/aadl/index.php/Osate_2
4. AADL Inspector (Ellidiss): http://www.ellidiss.com/products/aadl-inspector/
5. Cheddar (Univ Brest): http://beru.univ-brest.fr/~singhoff/cheddar/

Contact etienne.borde@telecom-paristech.fr

Web site http://penelope.enst.fr/aadl

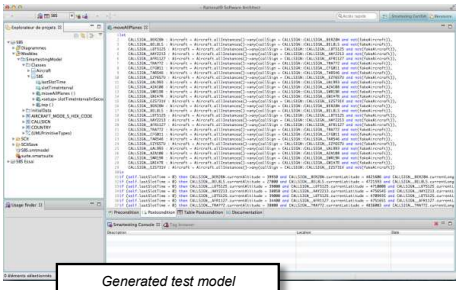
Dynamic Application Security Testing

F. Dadeau, F. Peureux, A. Vernotte, B. Legeard,
J. Botella, C. Civeit, D. Gidoin, P. Cao

Vulnerability testing of Air Traffic Management systems using ADS-B protocol

Motivations

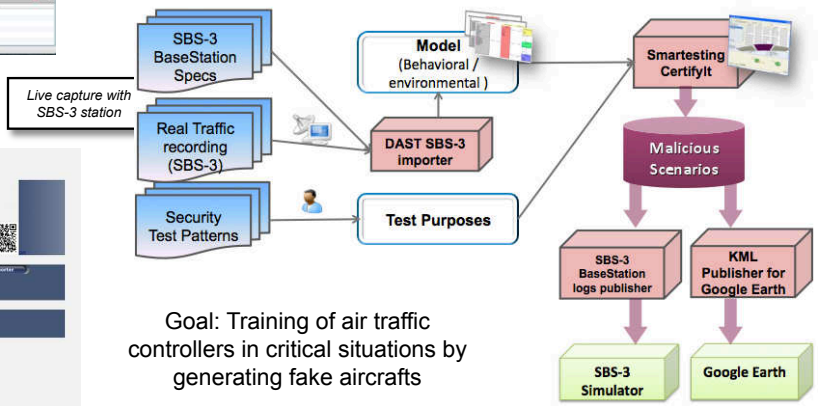
- To address application security vulnerabilities that cannot be detected by the static tests
- To reduce cost of testing and the time taken for industrialization
- To be able to demonstrate the resilience of Air Traffic Management systems
- To absorb the growth in air traffic and improve the security

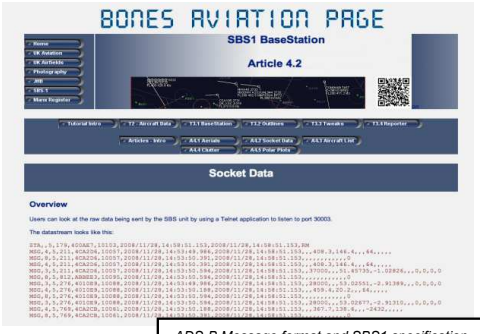


Generated test model

Model and pattern-based vulnerability testing

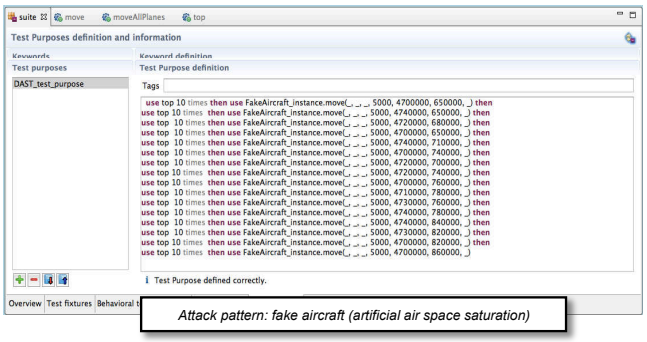
- UML/OCL models represent the test environment
- Test patterns drive the test generation with real-world attack scenarios



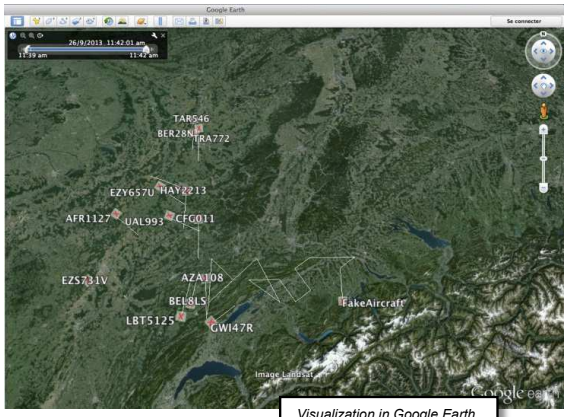


ADS-B Message format and SBS1 specification

Goal: Training of air traffic controllers in critical situations by generating fake aircrafts



Attack pattern: fake aircraft (artificial air space saturation)



Visualization in Google Earth

References

Web: http://dast.univ-fcomte.fr

- J. Botella, P. Cao, C. Civeit, D. Gidoin and F. Peureux. Model-Based generation of aircraft traffic attack scenarios using ADS-B standard signals. UCAAT'2013

Ce document contient les actes des Sixièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées au CNAM à Paris du 11 au 13 juin 2014.

Les contributions présentées dans ce document ont été sélectionnées par les différents groupes de travail du GDR. Il s'agit de résumés, de nouvelles versions, de posters et de démonstrations qui correspondent à des travaux qui ont déjà été validés par les comités de programmes d'autres conférences et revues et dont les droits appartiennent exclusivement à leurs auteurs.