Actes des Huitièmes journées nationales du

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

FEMTO-ST - Université de Bourgogne Franche-Comté

8 au 10 juin 2016

Editeurs : Frédéric DADEAU
Pierre-Etienne MOREAU

Impression : service de reprographie, FEMTO-ST - Université de Bourgogne Franche-Comté

Table des matières

Groupe de travail IDM **159**

Groupe de travail IE **173**

Groupe de travail LaMHA **187**

Groupe de travail MFDL **217**

Préface

C'est avec grand plaisir que je vous accueille pour les Huitièmes Journées Nationales du GDR Génie de la Programmation et du Logiciel (GPL) à l'Université de Bourgogne Franche-Comté. Succéder à Laurence Duchien pour continuer à rassembler et animer la communauté du GDR GPL est un réel honeur, mais aussi un grand défi. Je remercie très chaleureusement Yves Ledru et Laurence Duchien pour avoir animé, dynamisé et créé les conditions si particulières de cette belle communauté.

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, mais également en direction des mondes académique et socio-économique. 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 huitième 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 5ème édition de la Conférence en IngénieriE du Logiciel (CIEL 2016), la 10ème édition de la Conférence francophone sur les Architectures Logicielles (CAL 2016), ainsi que la 15ème édition d l'atelier francophone sur les Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL 2016).

Ces journées sont une vitrine où chaque groupe de travail 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.

Deux conférenciers invités nous ont fait l'honneur d'accepter notre invitation. Il s'agit de Pascal Cuoq (Trust in Soft) et de Jean-Marc Jézéquel (IRISA - Université de Rennes 1), lauréat de la médaille d'argent du CNRS en 2016. Une table ronde, animée par Martin Quinson, abordera le thème de l'enseignement de l'informatique dans le primaire et le secondaire.

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 quatrième année consécutive. Nous aurons le plaisir de remettre le premier prix de thèse GPL à Mounir Assaf pour sa thèse intitulée *From qualitative to quantitative program analysis : permissive enforcement of secure information flow*, ainsi qu'un accessit à Thibaud Antignac pour sa thèse intitulée *Méthodes formelles pour le respect de la vie privée par construction.* Le jury chargé de sélectionner le lauréat a été présidé par Catherine Dubois, que je remercie tout particulièrement, ainsi que l'ensemble des membres du jury.

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, les membres du comité de direction du GDR GPL et tout particulièrement le comité d'organisation de ces journées nationales présidé par Frédéric Dadeau. Je remercie chaleureusement l'ensemble des collègues bisontins qui n'ont pas ménagé leurs efforts pour nous accueillir dans les meilleures conditions.

Pierre-Etienne Moreau
Directeur du GDR Génie de la Programmation et du Logiciel

Comités

Comité de programme des journées nationales

Le comité de programme des journées nationales 2016 est composé par les membres du comité de direction du GDR GPL et les responsables de groupes de travail.

Pierre-Etienne Moreau (président), LORIA, Université de Lorraine

Yamine Ait Ameur, IRIT, ENSEEIHT
Nicolas Anquetil, CRIStAL, Université de Lille
Xavier Blanc, LaBRI, Université de Bordeaux, IUF
Mireille Blay-Fornarino, I3S, Université Nice-Sophia-Antipolis
Sandrine Blazy, IRISA, Université de Rennes
Florian Brandner, ENSTA
Eric Cariou, LIUPPA, Université de Pau et des pays de l'Adour
Khalil Drira, LAAS, CNRS
Catherine Dubois, Samovar, ENSIIE
Jean-Rémy Falleri, LABRI, ENSEIRB-MATMECA
Jean-Christophe Filliatre, LRI, CNRS
Aurélie Hurault, IRIT, ENSEEIHT
Laure Gonnord, LIP (ENS Lyon), Université Lyon 1
Akram Idani, LIG, Université Joseph Fourier
Claude Jard, AtlanSTIC, LINA, Université de Nantes
Nikolai Kosmatov, CEA-LIST
Régine Laleau, LACL, Université de Paris-Est Créteil
Yves Ledru, LIG, Université Joseph Fourier
Axel Legay, IRISA, Inria
Pascale Le Gall, MAS, Centrale Paris
Martin Monperrus, CRIStAL, Université de Lille
Sébastien Mosser, I3S, Université de Nice
Clémentine Nébut, LIRMM, Université de Montpellier
Flavio Oquendo, IRISA, Université de Rennes
Marc Pouzet, LIENS, IUF, ENS, Université Pierre et Marie Curie
Fabrice Rastello, INRIA, ENS Lyon
Olivier H. Roux, IRCCyN, Université de Nantes
Romain Rouvoy, CRIStAL, Université Lille 1
Camille Salinesi, CRI, Université Paris 1 Panthéon-Sorbone
Christelle Urtado, Mines d'Alès
Virginie Wiels, ONERA

Comité scientifique du GDR GPL

Franck Barbier (LIUPPA, Pau)
Pierre Casteran (LABRI, Bordeaux)
Pierre Cointe (LINA, Nantes)
Roberto Di Cosmo (PPS, Paris VII)
Christophe Dony (LIRMM, Montpellier)
Laurence Duchien (CRIStAL, Lille)
Stéphane Ducasse (INRIA, Lille)
Marie-Claude Gaudel (LRI, Orsay)
Jean-Louis Giavitto (IRCAMS, Paris)
Yann-Gaël Guéhéneuc (Polytech, Montréal)
Gaétan Hains (LACL, Créteil)
Nicolas Halbwachs (Verimag, Grenoble)
Olivier Hermant (Mines Paris)
Valérie Issarny (INRIA, Rocquencourt)
Jean-Marc Jézéquel (IRISA, Rennes)
Dominique Méry (LORIA, Nancy)
Christel Seguin (ONERA, Toulouse)

Comité d'organisation

Frédéric Dadeau, (Président), FEMTO-ST - Université de Bourgogne Franche-Comté

Conférenciers invités

SQLite au peigne fin

Auteur : Pascal Cuoq (Trust in Soft)

Résumé :

SQLite est une bibliothèque extrêmement utilisée, inclue pour prendre deux exemples sur chaque téléphone Android et sur chaque téléphone iOS. Au cours du développement d'un nouvel outil de détection dynamique, tis-interpreter, nous nous sommes fixé pour objectif de faire passer dans tis-interpreter l'imposante suite de tests avec couverture MC/DC existant pour SQLite. Cette présentation résume le travail qui a été nécessaire pour passer du SQLite d'origine, déjà soumis à et amélioré sur la base des diagnostiques de tous les outils disponible, en un SQLite dans lequel tis-interpreter ne détecte pas de comportement non défini. Il sera aussi question du travail nécessaire pour passer du tis-interpreter d'origine, basé sur la technologie Frama-C utilisée opérationnellement dans les domaines aéronautique, nucléaire et spatial, et soumis à l'évaluation du NIST sur la suite de tests Juliet, en un outil capable d'analyser le code source de SQLite.

Biographie :

Après une thèse avec Marc Pouzet et un post-doctorat avec Kwangkeun Yi, Pascal Cuoq est entré au CEA, où il a travaillé dans les domaines de l'analyse statique et de la vérification formelle de logiciel pendant dix ans. Au CEA, il a travaillé sur le logiciel de vérification Caveat, et a, avec Benjamin Monate, créé la plate-forme de vérification de programmes Frama-C. Pascal Cuoq est co-fondateur et directeur scientifique de la société TrustInSoft, qui fournit produits et services basés sur Frama-C.

Families of DSLs

Auteur : Jean-Marc Jézéquel (IRISA - Université de Rennes 1)

Résumé :
The engineering of complex systems involves many different stakeholders, each with their own domain of expertise. Hence more and more organizations are adopting Domain Specific Languages (DSLs) to allow domain experts to express solutions directly in terms of relevant domain concepts. This new trend raises new challenges about designing DSLs, handling variation points among DSLs, evolving a set of DSLs and coordinating the use of multiple DSLs. In this talk we explore various dimensions of these challenges, and outline a possible research roadmap for addressing them. We detail one of these challenges, which is the safe reuse of model transformations across variants of DSLs.

Biographie :
Jean-Marc Jézéquel is a Professor at the University of Rennes and Director of IRISA, one of the largest public research lab in Informatics in France. His interests include model driven software engineering for software product lines, and specifically component based, dynamically adaptable systems with quality of service constraints, including reliability, performance, timeliness etc. He is the author of several books published by Addison-Wesley and of more than 200 publications in international journals and conferences. He was a member of the steering committees of the AOSD and MODELS conference series. He also served on the editorial boards of IEEE Computer, IEEE Transactions on Software Engineering, the Journal on Software and Systems, on the Journal on Software and System Modeling and the Journal of Object Technology. He received an engineering degree from Telecom Bretagne in 1986, and a Ph.D. degree in Computer Science from the University of Rennes, France, in 1989.

Sessions des groupes de travail

Session commune aux groupes de travail Compilation et LTP

Compilation — Langages, Types et Preuves

Verifying clock-directed modular code generation for Lustre

Timothy Bourke[1,2], Pierre-Évariste Dagand[4,3,1], Marc Pouzet[4,2,1], and Lionel Rieg[5]

[1] Inria Paris
[2] École normale supérieure, PSL Research University
[3] CNRS, LIP6 UMR 7606
[4] Sorbonne Universités, UPMC Univ Paris 06
[5] Collège de France

Lustre was presented in 1987 as a programming language for control and signal processing systems [Caspi et al., 1987]. Several properties made it suitable for safety-critical applications: constructs for programming reactive controllers, execution in statically-bounded time and memory, and traceable compilation schemes [Biernacki et al., 2008]. In particular, compilation consists in transforming a set of equations, which define streams of values, into a sequence of imperative instructions, which manipulate the memory of a machine. Repeatedly executing the instructions is supposed to generate successive values of the original streams: but how can this be ensured?

Our response consists in formally specifying the source and target languages, implementing the compiler in an Interactive Theorem Prover (ITP) and proving a correctness relation between source and target programs. Building on prior work [Auger, 2013] that treats scheduling and normalization of dataflow programs in the Coq theorem prover, this paper focuses on bridging the gap between the dataflow world and the imperative one.

1 Source and Target Languages: CoreDF & Minimp

Compared to Lustre, CoreDF eschew separate initialization (`->`) and delay operators (`pre`) in favor of initialized registers (`fby`). This choice obviates the need for an analysis pass to determine whether delays are initialized before use. Additionally, the inputs of a node application must all be on the same clock, unlike in Lustre where they may be on subclocks of the clock of the first input. Prior formalizations [Auger, 2013] make the same two assumptions, but, unlike us, they treat generalized merges and modular resets. While generalized merges introduce only technical issues, modular resets pose important semantic issues even if their compilation is uncomplicated; we leave them for future work. The semantics $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)$ of a node f in a program G relates a list of input streams xs to an output stream ys, where we model streams as functions from the natural numbers to a domain of values.

Minimp is a fairly conventional imperative language whose expressions and commands read and manipulate a pair of memory environments. A *local memory* (*env*) models a stack frame, mapping variable names to boolean or integer values. A *global memory* (*mem*) models a static memory containing two mappings, variable names to values and variable names to instances (sub-memories). The memory instances of programs compiled from CoreDF reflect the tree of nodes in the original sources: there is an entry in values for each **fby** and one in instances for each node application. A class groups together a class name, a `step` method, and a `reset` method. A program is a list of classes. A step invocation looks up the given class name and executes the associated `step` method in a global (sub-)memory retrieved from instances. Reset invocations initialize an instance memory.

2 Code Generation

The translation maps a list of dataflow nodes into a list of imperative classes. Basic equations become assignments to local memory, node applications become step invocations that update local memory with a result and global memory with an updated instance, and **fby**s become assignments to global memory. Our definitions encode the standard technique [Biernacki et al., 2008].

3 Relating dataflow and imperative programs

In the context of a dataflow program, the semantics of a node relates input streams to an output stream. The imperative code produced by translating the program must satisfy an essential property: repeated execution against successive values of the input streams generates the successive values of the output stream. The correctness of the translation is thus captured by

Proposition 1. *Let G be a well-formed CoreDF program containing a node called f with semantics $G \vdash_{\mathsf{node}} f(\overrightarrow{xs}, ys)$. Translating G into an imperative program and iterating f's step statement n times against successive values of $\overrightarrow{xs}$ gives an environment containing the nth value of ys iff the latter is present:*

$$\begin{aligned}\exists env\ mem,\quad & \mathrm{step}(n+1, r, f, \overrightarrow{xs}, env, mem)\\ & \wedge\ \forall o,\ ys_{(n)} = \boldsymbol{present}\, o \iff env(r) = o.\end{aligned}$$

where the step predicate executes f's `step` method n times from an environment created by its `reset` method to give the environments env' and mem'; passing the appropriate input value from $\overrightarrow{xs}$ at each instant.

Intermediate dataflow semantics with exposed memory: The proposition above is too weak to prove directly because it says nothing about the global memory. Indeed, the generated program manipulates a tree of *mem* elements that mirrors the structure of node instantiations in the original program. For the correctness proof to go through, we must state an invariant that relates the sequences of values taken by the **fby**-streams to the values successively read from and written to the corresponding registers. To do so, we have introduced a new semantic judgment $G \vdash_{\mathsf{mnode}} f(\overrightarrow{xs}, M, ys)$, which exposes a memory tree M isomorphic to that of the translated code but in which instance variables are streams of constant values. The behavior of this model is intentionally very close to that of the translated code. From the fact that a node has a semantics, we can prove that it also has a semantics with exposed memories.

Proving translation correctness: Correctness is shown via three nested inductions: over instants, node instantiations, and the equations within a node; and two nested case distinctions: on the three classes of equations, and whether or not each is executed at a given instant. It relies on a few dozen auxiliary lemmas that include the correctness of expression translation and nested conditional generation.

4 Future Work

We have yet to verify the typing and clocking systems; in other words, that well-typed and well-clocked programs have a semantics, which should allow us to derive rather than decree the timing properties required for the correctness proof. Developing the link with CompCert is another objective, which will involve adapting our treatment of types and operators, and compiling our memory trees into nested records.

Bibliography

C. Auger. *Compilation certifiée de SCADE/LUSTRE.* PhD thesis, Univ. Paris Sud 11, Orsay, France, Apr. 2013.

D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet. Clock-directed modular code generation for synchronous data-flow languages. In *Proc. SIGPLAN Conf. on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 121–130, Tucson, AZ, USA, June 2008. ACM.

P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. LUSTRE: A declarative language for programming synchronous systems. In *Proc. 14th Symp. Principles Of Programming Languages (POPL)*, pages 178–188, Munich, Germany, Jan. 1987. ACM.

Test et preuve pour des structures combinatoires : Coq et Prolog

Catherine Dubois[1], Alain Giorgetti[2], and Richard Genestier[2]

[1] Samovar (UMR CNRS 5157), ENSIIE, Évry, France
catherine.dubois@ensiie.fr
[2] FEMTO-ST institute (UMR CNRS 6174 - UBFC/UFC/ENSMM/UTBM)
Université de Franche-Comté, Besançon, France
alain.giorgetti@femto-st.fr, richard.genestier@femto-st.fr

Faire une preuve interactivement à l'aide d'un assistant à la preuve - quiconque s'y est essayé le dira - n'est pas chose facile. Mais le plus frustrant est sans doute d'essayer de faire une preuve d'un lemme incorrect. Il est donc intéressant de pouvoir *tester* ces conjectures avant de les prouver. Des outils plus ou moins aboutis traitent cette question pour la plupart des assistants à la preuve (Isabelle [1], Agda [4], PVS [5], FoCaLiZe [2] et plus récemment Coq [6]). Ces outils sont très souvent inspirés de QuickCheck [3]. Ils permettent d'acquérir une certaine confiance dans le lemme testé et dans les définitions utilisées dans son énoncé ou, en cas d'échec, d'obtenir un ou plusieurs contre-exemples. Dans le cadre de l'étude de structures combinatoires comme les cartes combinatoires [9] ou les λ-termes [7], les objectifs principaux concernent le comptage de ces structures et la mise en place de bijections entre différentes familles de structures. Dans ce cadre, il est souvent utile d'énumérer les structures jusqu'à une certaine taille et d'utiliser ces éléments pour tester une certaine propriété.

Nous proposons une méthodologie alliant test aléatoire et test exhaustif borné pour tester des propriétés écrites en Coq et portant sur des structures combinatoires. Plus précisément, nous utilisons conjointement le plugin QuickChick de Coq et Prolog (ainsi que la bibliothèque Prolog de validation développée par V. Senni) pour réaliser cette combinaison. La méthodologie est exposée sur deux exemples : les permutations et les cartes combinatoires enracinées.

Méthodologie et outils utilisés

Test aléatoire. QuickChick[3] est un plugin de test développé pour Coq [6]. Il permet de tester la validité de propriétés exécutables avec des données générées aléatoirement. QuickChick fournit différents combinateurs pour écrire des générateurs aléatoires et le code dédié au test. La propriété sous test doit être exécutable, ce qui demande en général de transformer un prédicat en une fonction booléenne équivalente. Dans certains cas, il est possible, pour ce faire, d'utiliser le plugin Relation Extraction [8].

Test exhaustif borné. Nous proposons d'utiliser Prolog pour énumérer les structures combinatoires jusqu'à une certaine taille, ce qui est en général très

[3] https://github.com/QuickChick

facile à obtenir grâce au mécanisme de *backtracking* de Prolog. Chaque solution proposée par Prolog est traduite en un objet Coq avec lequel des lemmes Coq ou leur version exécutable sont instanciés. Dans le cas non exécutable, des tactiques appropriées peuvent être appliquées pour démontrer chaque instance des lemmes.

Cas d'étude

Nous illustrons ces différents modes de validation avec la mise au point de spécifications Coq pour les structures combinatoires des permutations et des cartes enracinées. Une permutation est définie comme une fonction injective sur un intervalle d'entiers naturels. Une telle permutation est isomorphe à une liste sans doublons contenant les éléments de l'intervalle de définition. Nous définissons ensuite la somme directe de deux permutations ainsi qu'une opération d'insertion. Dans un premier temps, l'objectif est de tester (puis démontrer) que ces deux opérations construisent bien des permutations lorsqu'elles sont appliquées à des permutations. Nous nous intéressons ensuite au cas des cartes combinatoires enracinées définies comme des paires transitives de permutations. Deux opérations spécifiques d'ajout d'une arête sont ensuite définies. Elles permettent de construire des cartes à partir de cartes plus petites. Ici la propriété que nous cherchons à valider concerne la préservation de la transitivité.

Références

1. Berghofer, S., Nipkow, T. : Random testing in Isabelle/HOL. In : Cuellar, J., Liu, Z. (eds.) Software Engineering and Formal Methods (SEFM 2004). pp. 230–239. IEEE Computer Society (2004)
2. Carlier, M., Dubois, C., Gotlieb, A. : Constraint Reasoning in FOCALTEST. In : Int. Conf. on Soft. and Data Tech. (ICSOFT'10). Athens (Jul 2010)
3. Claessen, K., Hughes, J. : QuickCheck : a lightweight tool for random testing of Haskell programs. In : Proceedings of Int. Conf. on Functional Programming (ICFP 2000). SIGPLAN Not., vol. 35, pp. 268–279. ACM, New York, NY, USA (2000)
4. Dybjer, P., Haiyan, Q., Takeyama, M. : Combining testing and proving in dependent type theory. In : Basin, D., Wolff, B. (eds.) Proceedings of Theorem Proving in Higher Order Logics. LNCS, vol. 2758, pp. 188–203. Springer (2003)
5. Owre, S. : Random testing in PVS. In : Workshop on Automated Formal Methods (AFM) (2006)
6. Paraskevopoulou, Z., Hritcu, C., Dénès, M., Lampropoulos, L., Pierce, B.C. : Foundational property-based testing. In : Interactive Theorem Proving - ITP 2015, Nanjing, China. LNCS, vol. 9236, pp. 325–343. Springer (2015)
7. Tarau, P. : Ranking/unranking of lambda terms with compressed de bruijn indices. In : Intelligent Computer Mathematics - International Conference, CICM 2015, Washington, DC, USA. LNCS, vol. 9150, pp. 118–133. Springer (2015)
8. Tollitte, P., Delahaye, D., Dubois, C. : Producing certified functional code from inductive specifications. In : Certified Programs and Proofs, CPP 2012, Kyoto, Japan, 2012. LNCS, vol. 7679, pp. 76–91. Springer (2012)
9. Tutte, W.T. : On the enumeration of planar maps. Bull. Amer. Math. Soc. 74, 64–74 (1968)

Call-By-Push-Value du point de vue de la logique linéaire

Thomas Ehrhard
IRIF — UMR 8243

Avril 2016

A l'origine, la correspondance de Curry-Howard établit un isomorphisme entre preuves de la logique intuitionniste (exprimées en déduction naturelle) et lambda-calcul typé, c'est-à-dire programmes purement fonctionnels typés. Elle propose une façon de concevoir le lien entre un programme (terme du lambda-calcul) et sa spécification (formule logique prouvée, ou, plus généralement, réalisée) qui est à la base de du système d'extraction de `Coq` ou de la réalisabilité classique de Jean-Louis Krivine.

Cette correspondance présente également un versant catégorique de *sémantique dénotationnelle* dans lequel les formules (ou types) sont interprétées comme des objets d'une catégorie et les preuves (ou programmes) sont interprétés comme des morphismes de cette catégorie. Étendre la correspondance de Curry-Howard à des langages qui ne sont plus purement fonctionnels passe par la compréhension catégorique de ces extensions de la pure fonctionnalité. Au milieu des années 1980, Eugnenio Moggi propose d'utiliser les *monades* pour "encapsuler" ces effets dans un cadre de sémantique dénotationnelle : il parvient ainsi à capturer l'usage de variables affectables (comme celles des langages non fonctionnels usuels, variables dont la valeur peut être modifiée par le programme), la manipulation des continuations (comme permet de le faire le `call/cc` de `scheme`), le choix non déterministe etc. Le choix probabiliste est plus difficile à prendre en compte dans ce cadre monadique.

À la même époque, Jean-Yves Girard découvre la logique linéaire qui se présente comme un raffinement de la logique intuitionniste, complètement compatible avec la correspondance de Curry-Howard, et dans lequel les règles structurelles (affaiblissement, contraction) prennent un statut logique grâce à l'introduction des connecteurs exponentiels. L'effet majeur de ce raffinement est la réintroduction d'une *négation involutive*, comme celle de la logique classique qui était réputée non susceptible d'une interprétation opérationnelle comme celle de la logique intuitionniste. Ce "miracle" est rendu possible par l'introduction du concept fondamental de linéarité : d'un point de vue catégorique, les preuves de la logique linéaire sont interprétées typiquement comme des morphismes linéaire (en un sens similaire à celui de l'algèbre linéaire) et la négation linéaire représente tout simplement la notion familière de *dualité*. Opérationnellement, cette dualité correspond à une symétrie parfaite entre le programme et son environnement.

Un peu plus tard, Timothy Griffin fait une observation fondamentale : la construction `call/cc` peut être typée au moyen de la *loi de Peirce*

$$((A \to B) \to A) \to A$$

qui est une tautologie classique non prouvable en logique intuitionniste. Autrement dit : la construction `call/cc` permet d'étendre la correspondance de Curry-Howard à la logique classique. En introduisant la *polarisation*, la logique linéaire a immédiatement fourni une explication de ce phénomène. On peut isoler deux classes duales de formules de logique linéaire, les positives et les négatives, qui préservent en un certain sens les règles structurelles. Les formules de la logique classique s'interpètent par des formules négatives, et les règles structurelles qui leur sont associées (plus les propriétés de la négation linéaire) permettent de rendre compte en logique linéaire des règles de la logique classique.

À la fin des années 1990, Paul Blain Levy introduit *Call-By-Push-Value* (CBPV) pour étendre l'approche monadique de Moggi à un langage qui n'est plus strictement en appel par valeur. Je proposerai dans mon exposé une interpétation *a priori* purement fonctionnelle de CBPV du point de vue de la logique linéaire, et plus précisément, de la polarisation. Alors que l'interpétation de la logique classique repose sur la stricte dualité entre formules positives et négatives, CBPV repose sur l'identification de formules positives au sein de d'un univers plus vaste de formules non nécessairement polarisées. Dans cette approche, les valeurs sont des termes particuliers *de type positif* qui sont interprétés en sémantique dénotationnelle par des morphismes respectant la "structure structurelle" des objets interprétant ces types. Syntaxiquement, cela signifie que ces termes sont librement duplicables et effaçables par les programmes qui les prennent en argument.

J'illustrerai ces propriétés structurelles des valeurs dans le cadre d'une extension probabiliste de CBPV qui admet une interprétation dénotationnelle naturelle dans le modèle des *espaces cohérents probabilistes* de la logique linéaire. Ainsi, le terme dice de type int qui réduit en 0 ou 1 avec probabilité 1/2 n'est pas une valeur de type int et n'est donc pas duplicable en CBPV. Par contre, le résultat de son évaluation (soit 0, soit 1) est une valeur de type int, et est donc duplicable. Il reste possible de dupliquer dice à condition de le mettre dans une boîte (au sens figuré, ou au sens de la logique linéaire, il se trouve par chance que les deux coïncident), mais cette mise en boîte apparaît dans les types. Ces caractéristiques permettent, sans s'imposer une stricte stratégie d'appel par valeur, d'écrire des programmes fonctionnels probabilistes qui ne sont pas représentables en pur appel par nom. Et en effet CBPV "contient" à la fois l'appel par nom et l'appel par valeur.

PharOS is Deterministic, Provably

Selma Azaiez[1], Damien Doligez[2], Matthieu Lemerre[1],
Tomer Libal[3], and Stephan Merz[4,5]

[1] CEA, Saclay, France
[2] Inria, Paris, France
[3] Inria, Saclay, France
[4] Inria, Villers-lès-Nancy, France
[5] CNRS, Université de Lorraine, LORIA, UMR 7503, Vandoeuvre-lès-Nancy, France

The observable behavior of a multi-process system depends not only on the inputs received from the system's environment, but is also influenced by the relative order in which processes are scheduled for execution. Because programmers usually have very little influence on the way processes are scheduled, the overall behavior can be non-deterministic even when every process operates deterministically. This leads to so-called "Heisenbugs" that make testing and debugging concurrent systems very challenging.

Designers of (embedded) real-time systems, such as controllers of safety-critical components in cars or airplanes, have devised principles for avoiding non-deterministic behavior. In particular, they can rely on the access of system components to a common time base for enforcing stricter scheduling disciplines. For example, the original idea of time-triggered architecture [3] was to assign fixed slots of execution to each process and to use a deterministic communication layer. In the PharOS real-time system [5, 6], commercialized[6] under the name Asterios®, every instruction that a process wishes to execute is associated with a temporal window of execution. Moreover, a message can only be received by a process if the execution window of the receiving instruction is strictly later than the execution window of the sending instruction. In this way, a message that can be received in some execution must be received in all executions that respect the timing constraints, independently of the order in which processes are scheduled. This argument is at the core of the pencil-and-paper proof establishing determinacy for PharOS [5].

In the work reported here [1], we represented the execution model of PharOS in the specification language TLA$^+$ [4] and used TLAPS, the TLA$^+$ Proof System [2] to formally prove determinacy for that model. Our proof is based on the paper-and-pencil proof of [5] but is written in assertional style, i.e., based on explicit inductive invariants. In order to express the property of determinacy in a linear-time temporal logic, we statically define "witness" executions where each process executes infinitely often and then show that at any point of an actual execution, the sequence of local states of each process is a prefix of the sequence of the states of the same process in an (arbitrary) fixed witness execution. We also prove the existence of witness executions by exhibiting a specific scheduling strategy. The proof represents a non-trivial case study for TLAPS, with approx-

[6] `http://www.krono-safe.com`

imately 2,000 lines of proof, not counting some general-purpose lemmas that are now included in the TLAPS standard library.

The work reinforces our confidence in the result that PharOS executions are indeed deterministic. The formal proof did not find any actual error in the original proof, but we found it useful to introduce some suitable intermediate abstractions, and we also sharpened some of the assumptions. The main assumption is that deadlines are never missed, a hypothesis that is validated by schedulability analysis for actual systems, but we did not formalize the extensions discussed in [5] to cases where some deadlines are missed or abrupt termination occurs. An interesting direction of future work would be to formally prove that an actual implementation, such as the Asterios® system, is a refinement of our high-level model of execution.

References

1. Selma Azaiez, Damien Doligez, Matthieu Lemerre, Tomer Libal, and Stephan Merz. Proving determinacy of the PharOS real-time operating system. In Michael Butler and Klaus-Dieter Schewe, editors, *5th Intl. Conf. Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2016)*, LNCS, Linz, Austria, 2016. Springer. To appear.
2. Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts, and Hernán Vanzetto. TLA$^+$ proofs. In Dimitra Giannakopoulou and Dominique Méry, editors, *18th Intl. Symp. Formal Methods (FM 2012)*, volume 7436 of *LNCS*, pages 147–154, Paris, France, 2012. Springer.
3. Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proc. of the IEEE*, 91(1):112–126, 2003.
4. Leslie Lamport. *Specifying Systems.* Addison-Wesley, Boston, Mass., 2002.
5. Matthieu Lemerre and Emmanuel Ohayon. A model of parallel deterministic real-time computation. In *Proc. 33rd IEEE Real-Time Systems Symp. (RTSS 2012)*, pages 273–282, San Juan, PR, U.S.A., 2012. IEEE Comp. Soc.
6. Matthieu Lemerre, Emmanuel Ohayon, Damien Chabrol, Mathieu Jan, and Marie-Bénédicte Jacques. Method and tools for mixed-criticality real-time applications within PharOS. In *14th IEEE Intl. Symp. Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, pages 41–48, Newport Beach, CA, U.S.A., 2011. IEEE Comp. Soc.

Amélioration à la Compilation de la Précision de Programmes Numériques

Nasrine Damouche[1], **Matthieu Martel**[1], **Alexandre Chapoutot**[2]

[1] Laboratoire de Mathématiques et de Physiques, LAMPS,
Université de Perpignan Via Domitia, France.
e-mail: nasrine.damouche@univ-perp.fr
e-mail: matthieu.martel@univ-perp.fr
[2] U2IS, ENSTA ParisTech, Université de Paris-Saclay,
828 bd des Maréchaux, 91762 Palaiseau cedex France.
e-mail: alexandre.chapoutot@ensta-paristech.fr

Résumé Les calculs en nombres flottants sont intensivement utilisés dans divers domaines, notamment les systèmes embarqués critiques. En général, les résultats de ces calculs sont perturbés par les erreurs d'arrondi. Dans un scenario critique, ces erreurs peuvent être accumulées et propagées, générant ainsi des dommages plus ou moins graves sur le plan humain, matériel, financier, etc. Il est donc souhaitable d'obtenir les résultats les plus précis possible lorsque nous utilisons l'arithmétique flottante. Pour ce faire, nous avons développé un outil qui corrige partiellement ces erreurs d'arrondi, par une transformation automatique et source à source des programmes. Notre transformation repose sur une analyse statique par interprétation abstraite qui fournit des intervalles pour les variables présentées dans les codes sources. Nous transformons non seulement des expressions arithmétiques mais aussi des morceaux de code avec des affectations, des boucles, des conditionnelles, des fonctions, etc. Les résultats obtenus par notre outils sont très prometteurs. Nous avons montré que nous améliorions de manière significative la précision numérique des calculs en minimisant l'erreur par rapport à l'arithmétique exacte des réels. Un autre critère très intéressant est que notre technique permet d'accélérer la vitesse de convergence de méthodes numériques itératives par amélioration de leur précision comme les méthodes de Newton, Jacobi, Gram-Schmidt, etc. Nous avons réussi à réduire le nombre d'itérations nécessaire pour converger de plusieurs dizaines de pourcents. Nous avons aussi étudié l'impact de l'optimisation de la précision sur le format des variables (en simple ou double précision). Pour ce faire, nous avons comparé deux programmes sources écrits en simple et en double précision avec celui transformé en simple précision. Les résultats obtenus montrent que le programme transformé (32 Bits) est très proche du résultat exact de celui de départ (64 Bits). Cela permet à l'utilisateur de dégrader la précision sans perdre beaucoup d'informations. D'un point de vue théorique, nous avons prouvé que les programmes générés n'ont pas forcément la même sémantique que les programmes d'origine, mais que mathématiquement, ils sont équivalents.

Mots Clés : Précision numérique, Arithmétique des nombres flottants, Transformation de programmes, Analyse statique, Preuve de correction.

1 Introduction

Suite à des progrès rapides et incessants, l'informatique a pris une ampleur prépondérante dans divers domaines d'application comme l'industrie spatiale, l'aéronautique, les équipements médicaux, le nucléaire, etc. Nous avons tendance à croire aveuglément aux différents calculs effectués par les ordinateurs mais un problème majeur se pose, lié à la fiabilité des traitements numériques, car les ordinateurs utilisent des nombres à virgule flottante qui n'ont qu'un nombre fini de chiffres. Autrement dit, l'arithmétique des ordinateurs basée sur les nombres flottants fait qu'une valeur ne peut être représentée exactement en mémoire, ce qui oblige à l'arrondir. En général, cette approximation est acceptable car la perte est tellement faible que les résultats obtenus sont très proches des résultats réels. Cependant dans un scénario critique, ces approximations engendrent des dégâts considérables sur le plan industriel, financier, humain et bien d'autres. La complexité des calculs en virgule flottante dans les systèmes embarqués ne cesse d'augmenter, rendant ainsi le sujet de la précision numérique de plus en plus sensible. Vu le rôle qu'elle joue sur la fiabilité des systèmes embarqués, l'industrie encourage les chercheurs pour valider [4,9,10,13,14,23] et améliorer [15, 22] leurs logiciels afin d'éviter des failles et éventuellement des catastrophes comme l'échec du missile Patriote en 1991 et l'explosion de la fusée Ariane 5 en 1996.

Cet article traite de la transformation automatique de programmes dans le but d'améliorer leur précision numérique [6, 7,8]. De nombreuses techniques ont été proposées pour transformer automatiquemnt des expressions arithmétiques. Dans ses travaux de thèse [15], A. Ioualalen a introduit une nouvelle représentation intermédiaire (IR) permettant de représenter dans une structure polynomiale, un nombre exponentiel d'expressions arithmétiques équivalentes. Cette représentation, nommée APEG [15,16] pour Abstract Program Expression Graph, a réussi à réduire la complexité de la transformation en un temps et une taille polynomiaux. Le but de notre travail est d'aller au delà des expressions arithmétiques, en s'intéressant à transformer automatiquement des bouts de code de taille plus ou moins grande. Notre transformation opère sur des séquences de commandes comprenant des affectations, des conditionnelles, des boucles, des fonctions,

Correspondence to: nasrine.damouche@univ-perp.fr etc., pour améliorer leur précision numérique. Nous avons défini un ensemble de règles de transformation pour les commandes [7]. Appliquées dans un ordre déterministe, ces règles permettent d'obtenir un programme plus précis parmi tout ceux considérés. Les résultats obtenus montrent que la précision numérique des programmes est significativement améliorée (en moyenne de 20%). Actuellement, nous nous intéressons à optimiser une seule variable de référence à partir des intervalles donnés aux valeurs d'entrées de programme et des bornes d'erreurs calculées en utilisant les techniques d'interprétation abstraite [5] pour l'arithmétique des nombres flottants [7,17].

Théoriquement, nous avons défini un ensemble de règles de transformation qui ont été implémentées dans un logiciel, Salsa. Cet outil se comporte comme un compilateur à la seule différence qu'il utilise les résultats d'une analyse statique fournissant des intervalles pour chaque variable à chaque point de contrôle. Notons que le programme généré ne possède pas forcément la même sémantique que celui de départ mais que les programmes sources et transformés sont mathématiquement équivalents pour les entrées (intervalles) considérées. De plus, le programme transformé est plus précis. La correction de notre approche repose sur une preuve mathématique par induction comparant le programme transformé avec celui d'origine.

Cet article est organisé comme suit. Nous détaillons à la section 2 les bases de l'arithmétique flottante et nous donnons par la suite un bref aperçu de la transformation des expressions arithmétiques. La section 3 concerne les différentes règles de transformation qui nous permettent d'obtenir les programmes optimisés automatiquement. Nous donnerons en section 4 le théorème de correction de notre transformation. En dernier lieu, dans la section 5, nous décrivons les différents résultats expérimentaux obtenus avec notre outil. Nous concluons à la section 6 qui résume nos travaux et ouvre sur de nombreuses perspectives.

2 Analyse et transformation des expressions

Dans cette section, nous présentons les méthodes utilisées pour borner et réduire les erreurs d'arrondi sur les expressions arithmétiques [15,16]. Dans un premier temps, nous présentons brièvement la norme IEEE754 et les méthodes d'analyse statique permettant de calculer les erreurs de calculs. Par la suite, nous évoquons la transformation automatique des expressions arithmétiques.

2.1 Analyse statique pour la précision numérique

x	s	e	m
+0	0	00000000	00000000000000000000000
-0	1	00000000	00000000000000000000000
+∞	0	11111111	00000000000000000000000
-∞	1	11111111	00000000000000000000000
NaN	0	11111111	00001001110000011000001 (exemple)

FIGURE 1. Valeurs spéciales de la norme IEEE754.

La norme IEEE754 est le standard scientifique permettant de spécifier l'arithmétique à virgule flottante [1,21]. Les nombres réels ne peuvent être représentés exactement en mémoire sur machine. A cause des erreurs d'arrondi apparaissant lors des calculs, la précision des résultats numériques est généralement peu intuitive. La représentation d'un nombre x en virgule flottante, en base b, est défini avec :

$$x = s \cdot (x_0.x_1.x_2 \ldots x_{p-1}) \cdot b^e = s \cdot m \cdot b^{e-p+1} \quad , \qquad (1)$$

avec, $s \in \{0,1\}$ le signe, $m = x_0.x_1.x_2 \ldots x_{p-1}$ la mantisse tel que $0 \leq x_i < b$ et $0 \leq i \leq p-1$, p la précision et enfin l'exposant $e \in [e_{min}, e_{max}]$.

En donnant des valeurs spécifiques pour p, b, e_{min} et e_{max}, le standard IEEE754 définit plusieurs formats pour les nombres flottants. En fonction de la mantisse et de l'exposent, la norme IEEE754 dispose de quatre valeurs spéciales comme le montre la Figure 1. Les NAN (Not a Number) correspondent aux exceptions ou aux résultats d'une opération invalide comme $0 \div 0$, $\sqrt{-1}$ ou $0 \times \pm\infty$.

Par exemple, il est clair que le nombre $1/3$ contient une infinité de chiffres après la virgule en bases 10 et 2, ce qui fait qu'on peut pas le représenter avec une suite finie de chiffres sur ordinateur. Même si l'on prend deux nombres exacts qui sont représentables sur machine, le résultat d'une opération n'est généralement pas représentable. Ceci montre la nécessité d'arrondir. Le standard IEEE754 décrit quatre modes d'arrondi pour un nombre x à virgule flottante :

— L'arrondi vers $+\infty$, renvoyant le plus petit nombre machine supérieur ou égal au résultat exact x. On le note $\uparrow_{+\infty}(x)$.

— L'arrondi vers $-\infty$, renvoyant le plus grand nombre machine inférieur ou égal au résultat exact x. On le note $\uparrow_{-\infty}(x)$

— L'arrondi vers 0, renvoyant $\uparrow_{+\infty}(x)$ si x est négatif ou $\uparrow_{-\infty}(x)$ si x est un nombre positif. On le note $\uparrow_0(x)$.

— L'arrondi au plus près, renvoyant le nombre machine le plus proche du résultat exact x. On le note $\uparrow_\sim(x)$.

La sémantique des opérations élémentaires comme défini par le standard IEEE754 pour les quatre modes d'arrondi $r \in \{-\infty, +\infty, 0, \sim\}$ cités précédemment pour $\uparrow_r : \mathbb{R} \to \mathbb{F}$, est donnée par :

$$x \circledast_r y = \uparrow_r (x * y) \quad , \qquad (2)$$

avec, $\circledast_r \in \{+, -, \times, \div\}$ une des quatre opérations élémentaires utilisées pour le calcul des nombres flottants en utilisant le mode d'arrondi r et $* \in \{+, -, \times, \div\}$ l'opération exacte (opérations sur les réels). Clairement, les résultats des calculs à base des nombres flottants ne sont pas exacts et ceci est dû aux erreurs d'arrondi. Par ailleurs, on utilise la fonction $\downarrow_r : \mathbb{R} \to \mathbb{R}$ permettant de renvoyer l'erreur d'arrondi du nombre en question. Cette fonction est définie par :

$$\downarrow_r (x) = x - \uparrow_r (x) \quad . \tag{3}$$

Il est à noter que nos techniques de transformation présentées dans la Section 3 ne dépendent pas d'un mode d'arrondi précis. Pour simplifier notre analyse, on suppose qu'on utilise le mode d'arrondi au plus près dans le reste de cet article, ce qui revient à écrire $\uparrow$ et $\downarrow$ au lieu de $\uparrow_r$ et $\downarrow_r$.

Pour calculer les erreurs se glissant durant l'évaluation des expressions arithmétiques, nous définissons des valeurs non standard faites d'une paire $(x, \mu) \in \mathbb{F} \times \mathbb{R} = \mathbb{E}$, où la valeur x représente un nombre flottant et μ l'erreur exacte liée à x. Plus précisément, μ est la différence exacte entre la valeur réelle et flottante de x comme définit par l'équation (3). A titre d'exemple, prenons le nombre réel $1/3$ qui sera représenté par la valeur suivante :

$$v = (\uparrow_\sim (1/3), \downarrow_\sim (1/3)) = (0.33333333, (1/3 - 0.33333333)).$$

La sémantique concrète des opérations élémentaires dans $\mathbb{E}$ est détaillée dans [18].

La sémantique abstraite associée à $\mathbb{E}$ utilise une paire d'intervalles $(x^\sharp, \mu^\sharp) \in \mathbb{E}^\sharp$, tel que le premier intervalle $x^\sharp$ contient les nombres flottants du programme, et le deuxième intervalle $\mu^\sharp$ contient les erreurs sur $x^\sharp$ obtenues en soustrayant le nombre flottant de la valeur exacte. Cette valeur abstrait un ensemble de valeurs concrètes $\{(x, \mu) : x \in x^\sharp$ et $\mu \in \mu^\sharp\}$. Revenons maintenant à la sémantique des expressions arithmétiques dont l'ensemble des valeurs abstraites est noté par $\mathbb{E}^\sharp$. Un intervalle $x^\sharp$ est approché avec un intervalle défini par l'équation (4) qu'on note $\uparrow^\sharp (x^\sharp)$.

$$\uparrow^\sharp ([\underline{x}, \overline{x}]) = [\uparrow (\underline{x}), \uparrow (\overline{x})] \quad . \tag{4}$$

La fonction d'abstraction $\downarrow^\sharp$, quant à elle, abstrait la fonction concrète $\downarrow$, autrement dit, elle permet de sur-approcher l'ensemble des valeurs exactes d'erreur, $\downarrow (x) = x - \uparrow (x)$ de sorte que chaque erreur associée à l'intervalle $x \in [\underline{x}, \overline{x}]$ est incluse dans $\downarrow^\sharp ([\underline{x}, \overline{x}])$. Pour un mode d'arrondi au plus proche, la fonction d'abstraction est donnée par l'équation (5).

$$\downarrow^\sharp ([\underline{x}, \overline{x}]) = [-y, y] \quad \text{avec} \quad y = \frac{1}{2}\mathrm{ulp}\big(\max(|\underline{x}|, |\overline{x}|)\big) \quad . \tag{5}$$

En pratique, l'ulp(x) qui est une abréviation de *unit in the last place* représente la valeur du dernier chiffre significatif d'un nombre à virgule flottante x. Formellement, la somme de deux nombres à virgule flottante revient à additionner les erreurs générées par l'opérateur avec l'erreur causée par l'arrondi du résultat. Similairement pour la soustraction de deux nombres flottants, on soustrait les erreurs sur les opérateurs et on les ajoute aux erreurs apparues au moment de l'arrondi. Quant à la multiplication de deux nombres à virgule flottante, la nouvelle erreur est obtenue par développement de la formule $(x_1^\sharp + \mu_1^\sharp) \times (x_2^\sharp + \mu_2^\sharp)$. Les équations (6) à (8) donnent la sémantique des opérations élémentaires.

$$(x_1^\sharp, \mu_1^\sharp) + (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp + x_2^\sharp), \mu_1^\sharp + \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp + x_2^\sharp)) \quad , \tag{6}$$

$$(x_1^\sharp, \mu_1^\sharp) - (x_2^\sharp, \mu_2^\sharp) = (\uparrow^\sharp (x_1^\sharp - x_2^\sharp), \mu_1^\sharp - \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp - x_2^\sharp)) \quad , \tag{7}$$

$$\begin{aligned}(x_1^\sharp, \mu_1^\sharp) \times (x_2^\sharp, \mu_2^\sharp) = (&\uparrow^\sharp (x_1^\sharp \times x_2^\sharp), \\ &x_2^\sharp \times \mu_1^\sharp + x_1^\sharp \times \mu_2^\sharp + \mu_1^\sharp \times \mu_2^\sharp + \downarrow^\sharp (x_1^\sharp \times x_2^\sharp)) \quad .\end{aligned} \tag{8}$$

Notons qu'il existe d'autres domaines abstraits plus efficaces, à titre d'exemple [4,13,14], et aussi des techniques complémentaires comme [2,3,9,23]. De plus, on peut faire référence à des méthodes qui transforment, synthétisent ou réparent les expressions arithmétiques basées sur des entiers ou sur la virgule fixe [12]. On citera également [4,11,19,20, 23] qui s'intéressent à améliorer les rangs des variables à virgule flottante.

2.2 *Les expressions arithmétiques*

Nous présentons ici rapidement les travaux de thèse de A. Ioualalen qui portent sur la transformation [6] des expressions arithmétiques en utilisant les APEGs [15,16,24]. Les APEGs, abréviation de *Abstract Program Equivalent Graph*, permettent de représenter en taille polynomiale un nombre exponentiel d'expressions mathématiques équivalentes. Un APEG se compose de classes d'équivalence représentées par des ellipses qui contiennent des opérations, et des boites. Pour former une expression valide, nous construisons l'APEG correspondant à l'expression arithmétique en question d'abord, ensuite, il faut choisir une opération dans chaque classe d'équivalence. Pour éviter le problème lié à l'explosion combinatoire, les APEGs regroupent plusieurs expressions arithmétiques équivalentes à base de commutativité, d'associativité et de distributivité dans des boites. Une boite avec n opérateurs peut représenter un très grand nombre de formules équivalentes allant jusqu'à $1 \times 3 \times 5 ... \times (2n-3)$ expressions. La construction d'un APEG nécessite l'usage de deux algorithmes. Le premier algorithme dit de *propagation* effectue une recherche récursive dans l'APEG afin d'y trouver les opérateurs binaires symétriques qui à leur tour, seront mis dans les boites abstraites. Le deuxième algorithme, d'*expansion*, cherche dans l'APEG une expression plus précise parmi toutes les expressions équivalentes. Enfin, nous recherchons l'expression arithmétique la plus précise selon la sémantique abstraite de la Section 2.1.

La syntaxe des expressions arithmétiques et booléennes est donnée par l'équation (9).

$$\begin{array}{rl}\text{Expr} \ni & e ::= id \mid cst \mid e + e \mid e - e \mid e \times e \mid e \div e \\ \text{BExpr} \ni & b ::= \texttt{true} \mid \texttt{false} \mid b \wedge b \mid b \vee b \mid \neg b \mid \\ & \quad e = e \mid e < e \mid e > e\end{array} \tag{9}$$

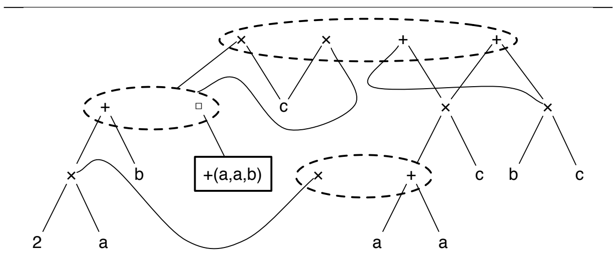


FIGURE 2. *APEG for the expression* $e = \big((a+a)+c\big)\times c$.

Exemple Afin de bien éclairer cette notion, nous donnons ci-dessous quelques expressions arithmétiques correspondant à l'APEG de l'expression $e = ((a+a)+b)\times c)$ de la Figure 2. Sur cette dernière, les ellipses en pointillé correspondent aux classes d'équivalences qui contiennent des APEGs et les rectangles sont les boites constituées d'une opération et n opérandes, autrement dit, c'est les différentes façons de combiner ces opérateurs avec les opérandes en question.

$$\mathscr{A}(p) = \left\{ \begin{array}{c} \big((a+a)+b\big)\times c,\ \big((a+b)+a\big)\times c, \\ \big((b+a)+a\big)\times c,\ \big((2\times a)+b\big)\times c, \\ c\times\big((a+a)+b\big),\ c\times\big((a+b)+a\big), \\ c\times\big((b+a)+a\big),\ c\times\big((2\times a)+b\big), \\ (a+a)\times c+b\times c,\ (2\times a)\times c+b\times c, \\ b\times c+(a+a)\times c,\ b\times c+(2\times a)\times c \end{array} \right\} . \quad (10)$$

■

3 Transformation des commandes

Afin d'améliorer au mieux la précision numérique des calculs, nous transformons automatiquement des programmes utilisant l'arithmétique des nombres à virgule flottante en nous appuyant sur des techniques d'interprétation abstraite. Cette transformation concerne les expressions arithmétique comme l'addition, la multiplication, les fonctions trigonométriques, etc., mais aussi les morceaux de code tel que les affectations, conditionnelles, boucles et fonctions. La syntaxe correspondant à nos commandes est la suivante :

$$\begin{array}{lll} \mathsf{Com} & \ni & c ::= id = e \mid c_1 ; c_2 \mid \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ & & \mid \mathsf{while}_\Phi\ e\ \mathsf{do}\ c \mid \mathtt{nop}\ . \end{array} \quad (11)$$

Pour réaliser notre transformation, nous avons défini un ensemble de règles permettant de réécrire les programmes en des programmes plus précis. Ces règles de transformation ont été implémentées dans un outil appelé Salsa qui prend en entrée un programme initialement écrit dans un langage impératif, et retourne en sortie un programme écrit dans le même langage et numériquement plus précis. Les programmes sont écrits sous forme SSA pour *Static Single Assignment*, chaque variable est écrite uniquement une seule fois dans le code source, ce qui évite les conflits causés par la lecture et l'écriture d'une même variables. Les différentes règles de transformation, données à la figure 3, sont utilisées dans un ordre déterministe c'est-à-dire qu'elles sont appliquées l'une après l'autre. Notre transformation est répétée jusqu'à ce que le programme final ne change plus. Dans notre cas, on optimise une seule variable à la fois nommée variable de référence et notée ϑ. Un programme transformé, p_t, est plus précis que le programme original, p_o, si et seulement si :

- — La variable de référence ϑ correspond mathématiquement à la même expression mathématique dans les deux programmes,
- — Cette variable de référence est plus précise dans p_t que dans p_o.

Nous détaillons maintenant l'ensemble des règles de transformation intra-procédurale qui nous permettent de réécrire les différentes commandes. La transformation de nos programmes utilise un environnement formel δ qui relie des identificateurs à des expressions formelles, $\delta : \mathscr{V} \to Expr$. Tout d'abord, nous avons deux règles pour l'affectation dont la forme est $c \equiv id = e$. La règle $(A1)$ traduit une heuristique permettant de supprimer une affectation du programme source et de la sauvegarder dans la mémoire δ si les conditions suivantes sont bien vérifiées : i) les variables de l'expression e n'appartiennent pas à l'environnement δ, ii) la variable de référence ϑ que nous souhaitons optimiser n'est pas dans la liste noire β, iii) la variable v que l'on souhaite supprimer et mettre dans δ ne correspond pas à la variable de référence ϑ. La seconde règle $(A2)$ permet de substituer les variables déjà mémorisées dans δ dans l'expression e afin d'obtenir une expression plus grosse que nous allons, par la suite, reparenthéser pour en trouver une forme qui est numériquement plus précise. Pour ce qui concerne les séquences de commandes $c_1; c_2$, nous disposons de plusieurs règles de transformation selon la valeur des deux membres c_1 et c_2. Si un des deux membres est égal à nop, nous transformons uniquement l'autre membre (règles $(S1)$ et $(S2)$). Sinon, nous réécrivons les deux membres de la séquence (règle $(S3)$). Le troisième type de transformation de commandes concerne les conditionnelles. Les trois premières règles $(C1)$ à $(C3)$ consistent à évaluer la condition e tout en ayant connaissance statique de sa valeur. Dans le cas où la condition est `vraie` nous transformons la partie `if` et quand elle vaut `faux`, nous transformons la branche `then`. Par ailleurs, si on ne connais pas statiquement la valeur de la condition, nous transformons les deux branches de la conditionnelle. Une dernière règle stipule que les variables qui ont été supprimées alors qu'il ne fallait pas les enlever du programme doivent être ré-injecter dans le corps de la conditionnelle car sinon on rencontre des ennuis à l'exécution car des variables sont non initialisées $(C4)$. Par conséquent, nous mettons ces variables dans la liste noire β pour ne pas les supprimer dans une future utilisation. Quant à la boucle $\mathsf{while}_\Phi\ e\ \mathsf{do}\ c$, nous avons défini deux règles. Une première règle $(W1)$ réécrit le corps de la boucle tandis que l'autre règle intervient lorsque nous rencontrons des variables non définies dans le programme car elles ont été supprimées et sauvegardées dans δ. Comme pour les conditionnelles à ce stade là, nous ré-insérons les variables en question dans le corps de la boucle et nous les rajoutons à la liste noire.

$$\dfrac{\delta' = \delta[id \mapsto e] \quad id \notin \beta}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \texttt{nop}, \delta', C, \beta\rangle}\ (A1) \qquad \dfrac{e' = \delta(e) \quad \sigma^\sharp = [\![C[c]]\!]^\sharp \iota^\sharp \quad \langle e', \sigma^\sharp\rangle \rightsquigarrow^* e''}{\langle id = e, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle id = e'', \delta, C, \beta\rangle}\ (A2)$$

$$\dfrac{}{\langle \texttt{nop}\ ;\ c\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle}\ (S1) \qquad \dfrac{}{\langle c\ ;\ \texttt{nop}\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c, \delta, C, \beta\rangle}\ (S2)$$

$$\dfrac{\begin{array}{r} C' = C\big[[];c_2\big] \quad \langle c_1, \delta, C', \beta\rangle \Rightarrow^*_\vartheta \langle c'_1, \delta', C', \beta'\rangle \quad C'' = C[c'_1;[]] \\ \langle c_2, \delta', C'', \beta'\rangle \Rightarrow_\vartheta \langle c'_2, \delta'', C'', \beta''\rangle \end{array}}{\langle c_1\ ;\ c_2\ , \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'_1\ ;\ c'_2, \delta'', C, \beta''\rangle}\ (S3)$$

$$\dfrac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \texttt{true} \quad \langle c_1, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'_1, \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c'_1), \Psi(\Phi, \delta'), C, \beta\rangle}\ (C1)$$

$$\dfrac{\sigma^\sharp = [\![C[\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]]\!]^\sharp \iota^\sharp \quad [\![e]\!]^\sharp \sigma^\sharp = \texttt{false} \quad \langle c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'_2, \delta', C, \beta\rangle}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \Psi(\Phi, c'_2), \Psi(\Phi, \delta'), C, \beta\rangle}\ (C2)$$

$$\dfrac{\begin{array}{c} Var(e) \cap Dom(\delta) = \emptyset \quad \beta' = \beta \cup Assigned(c_1) \cup Assigned(c_2) \\ \langle c_1, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c'_1, \delta_1, C, \beta_1\rangle \quad \langle c_2, \delta, C, \beta'\rangle \Rightarrow_\vartheta \langle c'_2, \delta_2, C, \beta_2\rangle \quad \delta' = \delta_1 \cup \delta_2 \end{array}}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c'_1\ \mathsf{else}\ c'_2, \delta', C, \beta'\rangle}\ (C3)$$

$$\dfrac{\begin{array}{c} V = Var(e) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta) \setminus V} \\ \langle c';\mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle \end{array}}{\langle \mathsf{if}_\Phi\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle}\ (C4)$$

$$\dfrac{Var(e) \cap Dom(\delta) = \emptyset \quad C' = C[\mathsf{while}_\Phi\ e\ \mathsf{do}\ []] \quad \langle c, \delta, C', \beta\rangle \Rightarrow_\vartheta \langle c', \delta', C', \beta'\rangle}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c', \delta', C, \beta'\rangle}\ (W1)$$

$$\dfrac{\begin{array}{r} V = Var(e) \cup Var(\Phi) \quad c' = AddDefs(V, \delta) \quad \delta' = \delta_{|Dom(\delta) \setminus V} \\ \langle c';\mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta', C, \beta \cup V\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle \end{array}}{\langle \mathsf{while}_\Phi\ e\ \mathsf{do}\ c, \delta, C, \beta\rangle \Rightarrow_\vartheta \langle c'', \delta', C, \beta'\rangle}\ (W2)$$

FIGURE 3. *Règles de transformation pour améliorer la précision de programmes.*

4 Preuve de correction

Pour vérifier la correction de notre transformation, nous introduisons un théorème qui compare deux programmes et montre que le programme le plus précis peut être utilisé à la place de celui moins précis. Notre preuve est basée sur une sémantique opérationnelle classique pour les expressions arithmétiques et les commandes. La comparaison entre les deux programmes nécessite de spécifier une variable de référence ϑ définie par l'utilisateur. Plus précisément, un programme transformé p_t est plus précis qu'un programme d'origine p_o si et seulement si les deux conditions suivantes sont vérifiées :

- La variable de référence ϑ correspond à même expression mathématique dans les deux programmes,
- La variable de référence ϑ est plus précise dans le programme transformé p_t que dans celui d'origine p_o.

Nous utilisons la relation d'ordre $\sqsubseteq \subseteq \mathbb{E} \times \mathbb{E}$ disant qu'une valeur (x, μ) est plus précise qu'une valeur (x', μ') si elles correspondent à la même valeur réelle et si l'erreur μ est plus petite que μ'.

Définition 1 (Comparaison des expressions). Nous considérons $v_1 = (x_1, \mu_1) \in \mathbb{E}$ et $v_2 = (x_2, \mu_2) \in \mathbb{E}$. Nous disons que v_1 est plus précise que v_2, noté $v_1 \sqsubseteq v_2$, si et seulement si $x_1 + \mu_1 = x_2 + \mu_2$ et $|\mu_1| \leq |\mu_2|$. ■

Définition 2 (Comparaison des commandes). Soit c_o et c_t deux commandes, δ_o et δ_t deux environnements formels et ϑ la variable de référence. Nous disons que

$$\langle c_t, \delta_t, C, \beta\rangle \prec_\vartheta \langle c_o, \delta_o, C, \beta\rangle \tag{12}$$

si et seulement si pour chaque $\sigma \in$ Mem,

$$\begin{array}{l} \exists \sigma_o \in \text{Mem}, \langle c_o, \sigma\rangle \to^* \langle \texttt{nop}, \sigma_o\rangle\ , \\ \exists \sigma_t \in \text{Mem}, \langle c_t, \sigma\rangle \to^* \langle \texttt{nop}, \sigma_t\rangle\ , \end{array}$$

- — soit $\sigma_t(\vartheta) \sqsubseteq \sigma_o(\vartheta)$,
- — soit pour chaque $\text{id} \in \text{Dom}(\sigma_o) \setminus \text{Dom}(\sigma_t)$, $\delta_t(\text{id}) = e$ et $\langle e, \sigma\rangle \to^*_e \sigma_o(\text{id})$.

■

La deuxième définition spécifie que :

- Les deux commandes c_o et c_t calculent la même valeur de référence ϑ dans les deux environnements δ_o et δ_t dans une arithmétique exacte,
- La commande transformée est plus précise,
- Si après exécution du programme, une variable est indéfinie dans l'environnement concret δ_t, alors l'expression formelle correspondante est déjà enregistrée dans δ_t.

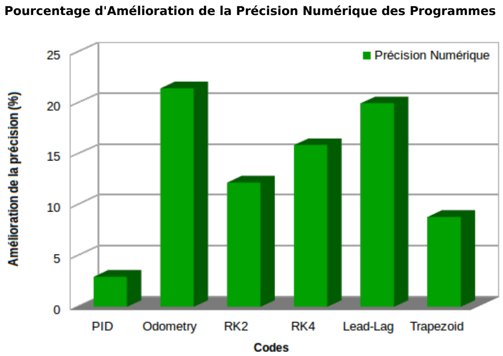


FIGURE 4. *Pourcentage de l'amélioration de la précision numérique des programmes.*

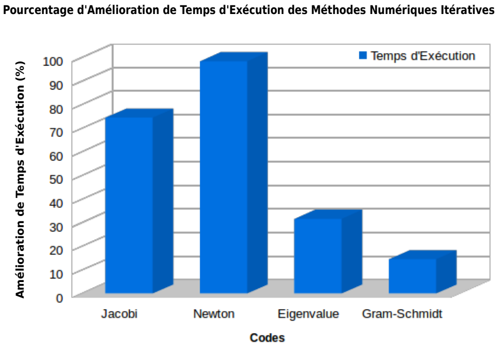


FIGURE 5. *Pourcentage d'amélioration de temps d'exécution des méthodes numériques itératives.*

Théorème 1 (Correction de la transformation des commandes). *Soit c_o le code d'origine, c_t le code transformé, δ_o l'environnement initial, δ_t l'environnement final de la transformation, nous écrivons ainsi*

$$\big(\langle c_o,\delta_o,C,\beta\rangle \Rightarrow_\vartheta \langle c_t,\delta_t,C,\beta\rangle\big) \\ \Longrightarrow \big(\langle c_o,\delta_o,C,\beta\rangle \prec_\vartheta \langle c_t,\delta_t,C,\beta\rangle\big) \ . \quad (13)$$

■

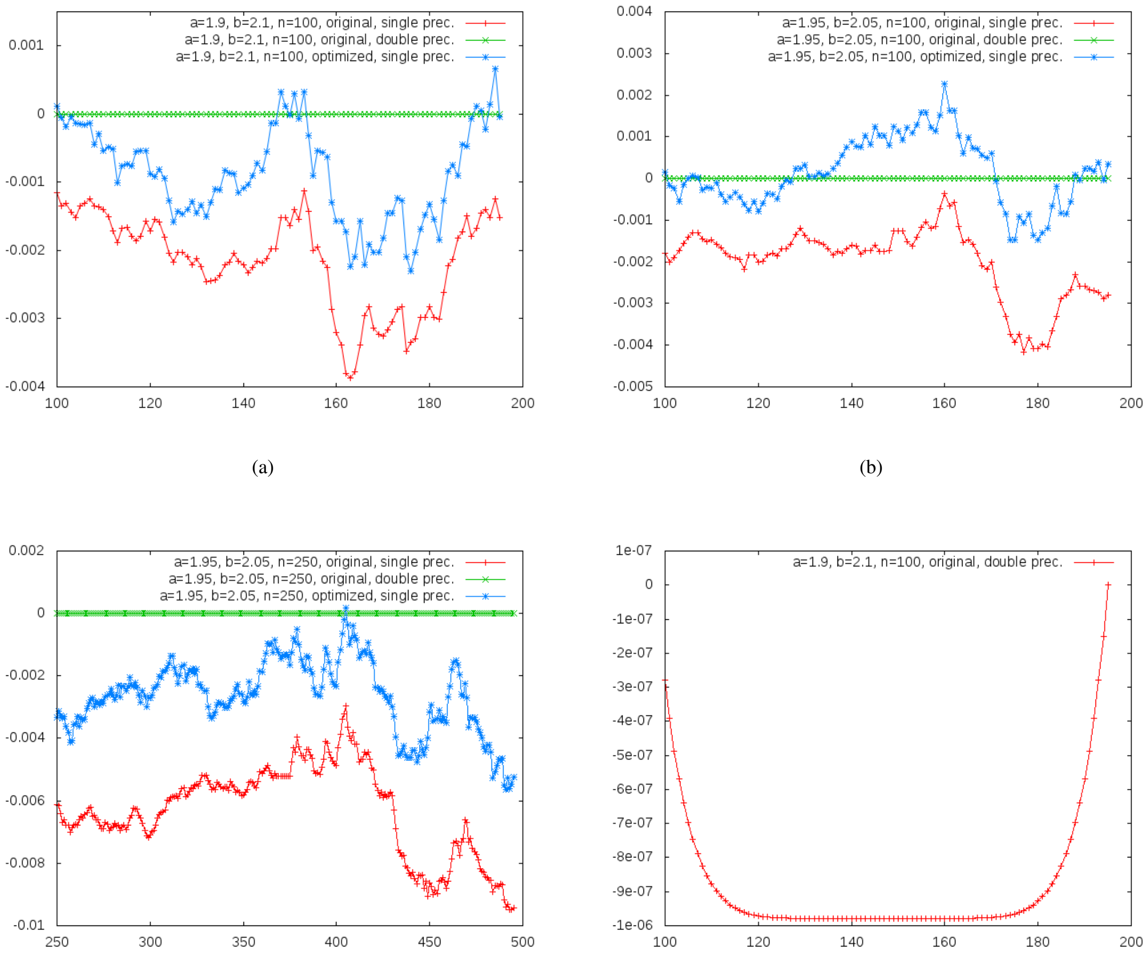
5 Résultats expérimentaux

Nous avons développé un prototype qui transforme automatiquement des bouts de codes contenant des affectations, des conditionnelles, des boucles, des fonctions, etc., écrit dans un langage impératif. De nombreux tests ont été menés afin d'évaluer l'efficacité de notre outil pour les entrées (intervalles) considérées. Les résultats obtenus sont concluants. Les exemples de la Figure 4 illustrent le gain, en pourcentage, de précision numérique sur une suite de programmes provenant des systèmes embarqués et des méthodes d'analyse numérique [7]. Les programmes transformés sont plus précis que ceux de départ, c'est-à-dire, que la nouvelle erreur après transformation est plus petite que l'erreur de départ. Prenant à titre d'exemple le programme qui calcule la position d'un robot à deux roues (`odometry`), la précision a été amélioré de 21%. Si nous observons aussi le système masse-ressort qui consiste à changer la position initiale y d'une masse vers une position désirée y_d, nous remarquons que la précision numérique pour ce programme est améliorée de 19%. Nous nous sommes aussi intéressés à étudier l'impact de l'optimisation de la précision numérique des calculs sur la vitesse de convergence des méthodes numériques itératives. Pour ce faire, nous avons pris une série d'exemples de méthodes telles que les méthodes de Jacobi, Newton, Gram-Schmidt ainsi qu'une méthode de calcul des valeurs propres d'une matrice. En utilisant notre outil, nous avons réussi à réduire le nombre d'itérations nécessaire à la convergence de toutes ces méthodes en améliorant la précision de leurs calculs. Les programmes ont été écrits dans le langage de programmation `C` et compilés avec `GCC` tout en désactivant toutes les optimisations du compilateur (`-o0`). La Figure 5 donne un aperçu sur les expérimentations faites sur les méthodes numériques précédemment citées. Nous accélérons la vitesse de convergence pour ces méthodes avec une moyenne de 20%. Dans le futur, nous souhaitons renouveler cette étude sur des codes réels issus de la physique.

Une autre évaluation de notre méthode de transformation concerne l'impact de l'optimisation de la précision numérique des programmes sur le format des variables utilisées, (simple ou double précision). Cette optimisation offre à l'utilisateur la possibilité de travailler sur des programmes en simple précision tout en étant sûr que les résultats obtenus seront proches des résultats obtenus avec le programme initial exécuté en double précision. Pour cela, nous avons choisi de calculer l'intégrale du polynôme $(x-2)^7$ avec la méthode de `Simpson` et nous nous sommes restreints à étudier le comportement autour de la racine, donc sur l'intervalle $[1.9, 2.1]$ où le polynôme s'évalue très mal dans l'arithmétique des nombres flottants. Nous avons comparé trois programmes, le premier code source écrit en simple précision (32 bits), un deuxième code source en double précision et le troisième programme qui est celui transformé en simple précision (32 bits). Sur la Figure 6 sont illustrés les résultats obtenus. Nous voyons que le programme transformé en 32 Bits est très proche de celui de départ en 64 Bits. L'atout principal de cette comparaison est de permettre à l'utilisateur d'utiliser un format plus compact sans perdre beaucoup d'informations, ce qui permet d'économiser de la mémoire, de la bande passante et du temps de calcul.

6 Conclusion

L'objectif principal de notre travail est d'améliorer la précision numérique des calculs par transformation automatique



(a)

(b)

(c)

(d)

FIGURE 6. *Résultats de la simulation de la méthode des Simpson avec simple, double précision et le programme optimisé utilisant notre outil.*

de programmes. Nous avons défini un ensemble de règles de transformation intra-procédurale et inter-procédurales qui ont été implémentées dans un outil appelé `Salsa`. Notre outil, basé sur les méthodes d'analyse statique par interprétation abstraite, prend en entrée un programme écrit dans un langage impératif et retourne en sortie un autre programme mathématiquement équivalent mais plus précis. Nous avons ensuite démontré la correction de cette approche en faisant une preuve mathématique qui compare les deux programmes, plus précisément, on compare la précision du code source avec celle du code transformé. Cette preuve par induction est appliquée aux différentes constructions du langage supporté par notre outil. Les résultats expérimentaux présentés dans la Section 5 montrent les différentes applications de notre outil.

Une perspective consiste à étendre notre outil pour traiter les codes massivement parallèles. Dans cette direction, nous nous intéressons à résoudre des problèmes spécifiques de la précision numérique comme l'ordre des opérations de calculs dans les programmes parallèles. Nous nous intéressons aussi à explorer le compromis temps d'exécution, performance de calculs, précision numérique ainsi que vitesse de convergence des méthodes numériques. Un point très ambitieux consiste à étudier l'impact de l'amélioration de la précision numérique sur le temps de convergence d'algorithmes de calculs distribués comme ceux utilisés pour le calcul haute performance. De plus, notre intérêt porte sur les problèmes de reproductibilité des résultats, plus précisément, plusieurs exécutions d'un même programme donne des résultats différents et ce à cause de la variabilité de l'ordre d'exécution des expressions mathématiques.

Références

1. ANSI/IEEE. *IEEE Standard for Binary Floating-Point Arithmetic*. SIAM, 2008.
2. E-T. Barr, T. Vo, V. Le, and Z. Su. Automatic detection of floating-point exceptions. In *Symposium on Principles of Programming Languages, POPL '13, 2013*, pages 549–560. ACM, 2013.
3. F. Benz, A. Hildebrandt, and S. Hack. A dynamic program analysis to find floating-point accuracy problems. In *Programming Language Design and Implementation, PLDI '12, 2012*, pages 453–462. ACM, 2012.
4. J. Bertrane, P. Cousot, R. Cousot, F. Feret, L. Mauborgne, A. Miné, and X. Rival. Static analysis by abstract interpretation of embedded critical software. *ACM SIGSOFT Software Engineering Notes*, 36(1) :1–8, 2011.
5. P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252, 1977.
6. P. Cousout and R. Cousot. Systematic design of program transformation frameworks by abstract interpretation. In *Principles of Programming Languages*, pages 178–190. ACM, 2002.
7. N. Damouche, M. Martel, and A. Chapoutot. Intra-procedural optimization of the numerical accuracy of programs. In Manuel Núñez and Matthias Güdemann, editors, *FMICS'15*, volume 9128 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2015.
8. N. Damouche, M. Martel, and A. Chapoutot. Transformation of a PID controller for numerical accuracy. *Electr. Notes Theor. Comput. Sci.*, 317 :47–54, 2015.
9. E. Darulova and V. Kuncak. Sound compilation of reals. In Suresh Jagannathan and Peter Sewell, editors, *POPL'14*, pages 235–248. ACM, 2014.
10. D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and V. Védrine. Towards an industrial use of FLUCTUAT on safety-critical avionics software. In *Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.
11. J. Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.
12. X. Gao, S. Bayliss, and G-A. Constantinides. SOAP : structural optimization of arithmetic expressions for high-level synthesis. In *Field-Programmable Technology, FPT*, pages 112–119. IEEE, 2013.
13. E. Goubault. Static analysis by abstract interpretation of numerical programs and systems, and FLUCTUAT. In *Static Analysis Symposium, SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 1–3. Springer, 2013.
14. E. Goubault and S. Putot. Static analysis of finite precision computations. In *Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *LNCS*. Springer, 2011.
15. A. Ioualalen and M. Martel. A new abstract domain for the representation of mathematically equivalent expressions. In *SAS'12*, volume 7460 of *LNCS*, pages 75–93. Springer, 2012.
16. M. Martel. Accurate evaluation of arithmetic expressions (invited talk). *Electr. Notes Theor. Comput. Sci.*, 287 :3–16, 2012.
17. Matthieu Martel. Propagation of roundoff errors in finite precision computations : A semantics approach. In Daniel Le Métayer, editor, *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, 2002, Proceedings*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002.
18. Matthieu Martel. Semantics of roundoff error propagation in finite precision calculations. *Higher-Order and Symbolic Computation*, 19(1) :7–30, 2006.
19. A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In David A. Schmidt, editor, *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Proceedings*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.
20. David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.
21. J-M. Muller, N. Brisebarre, F. De Dinechin, C-P. Jeannerod, V. Lefèvre, G. Melquiond, N. Revol, D. Stehlé, and S. Torres. *Handbook of Floating-Point Arithmetic*. Birkhäuser, 2010.
22. J.-R. Wilcox P. Panchekha, A. Sanchez-Stern and Z. Tatlock. Automatically improving accuracy for floating point expressions. In *PLDI'15*, pages 1–11. ACM, 2015.
23. A. Solovyev, C. Jacobsen, Z. Rakamaric, and G. Gopalakrishnan. Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. In *FM'15*, volume 9109 of *LNCS*, pages 532–550. Springer, 2015.
24. R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation : A new approach to optimization. *Logical Methods in Computer Science*, 7(1), 2011.

Symbolic Range Analysis of Pointers (short version)*

Maroua Maalej[1]

University of Lyon/LIP/Labex Milyon

Abstract. This short paper is an overview of "Symbolic Range Analysis of Pointers" published in *Code Generation and Optimization, March 2016, Barcelona, Spain.* The full version of the paper [8] describes a new alias analysis using symbolic ranges proposed by Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord and Fernando Magno Quintao Pereira, where we propose to combine symbolic range analysis with alias analysis of pointers.

1 Motivation

Pointer analysis is one of the most fundamental compiler technologies. This analysis lets the compiler distinguish one memory location from others; hence, it provides the necessary information to transform code that manipulates memory. Given this importance, it comes as no surprise that pointer analysis has been one of the most researched topics within the field of compiler construction [6]. This research has contributed to make the present algorithms more precise [4, 11], and faster [5, 10]. Nevertheless, one particular feature of imperative programming languages remains to be handled satisfactorily by the current state-of-the-art approaches: the disambiguation of pointer arithmetics.

1.1 Our analysis in four steps

Step 1: e-SSA form. First of all, we compute the Extended Static Single Assignment (e-SSA) form [1] of our program. This ensures that each variable has a unique definition, and also use copies to propagate information from tests; We are then able to implement our analysis sparsely [2], e.g,. we can assign information directly to variables, instead of pairs of variables and program points.

Step 2: Symbolic range analysis on integers. We run an off-the-shelf *range analysis* parameterized on *symbols.* We borrow notions associated with range analysis, from Nazaré *et al.* [7]. This symbolic range analysis is able to provide us symbolic ranges (like [0,max(0,N)]) for all program integer variables.

Step 3: Abstract interpretation on pointers. We then perform an *abstract interpretation* based constraint propagation on pointer variables. We propagate symbolic ranges of offsets from *abstract location sites.*

* This work was partially supported by the LABEX MILYON (ANR-10-LABX-0070).

Step 4: Answering alias queries. Giving two different pointers of the program, if the abstract values of these pointers have a null intersection, then the two *concrete pointers* do not alias.

2 Example

```
void fill_array (int* p, int L) {
    int i, N = L/2 ;
    for (i = 0; i < N; i++){
        p[i] = 0 ;}
    for ( i = N; i < 2×N ; i++){
        p[i] = 2 × i ;}
}
```

Fig. 1: Simple example of program that fills in an array. We are interested in disambiguating the locations accessed in both loops (for parallelization, loop merge, . . .)

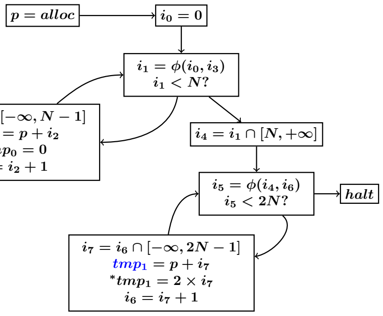


Fig. 2: CFG of program seen in Fig. 1 in e-ssa form.

The symbolic range analysis provides us with the ranges : $R(i_1) = [0, N]$; $R(i_5) = [N, 2N-1]$; $R(i_0) = [0, 0]$; $R(i_3) = [0, N-1]$; $R(i_2) = [0, N-1]$; $R(i_7) = [N, 2N-1]$.

To gather information on pointer variables, we propagate offsets from allocation sites. Let loc_0 be the allocation site associated to pointer p. The abstract values we obtain for p, tmp_0 and tmp_1 are $\mathrm{GR}(p) = \{loc_0 + [0, 0]\}$, $\mathrm{GR}(tmp_0) = \{loc_0 + [0, N-1]\}$ and $\mathrm{GR}(tmp_1) = \{loc_0 + [N, 2N-1]\}$.

Now we can run the alias query on pointers tmp_0 and tmp_1. $\mathrm{GR}(tmp_0)$ and $\mathrm{GR}(tmp_1)$ have a null intersection then tmp_0 and tmp_1 do not alias.

3 Experimental results

We implemented our analysis in the LLVM 3.5 compiler framework, and Figure 3 show the comparison (on classic pointer benchmarks, [9, 12, 3] against the other pointer analyses that are available in LLVM 3.5, namely *basic* and *SCEV*.

The chart in Figure 4 shows how our analysis scales when applied on programs of different sizes. We can analyze 100,000 instructions in about one second.

4 Conclusion

Our technique can disambiguate regions within arrays and C-like structs using the same abstract interpreter. We have achieved precision in our algorithm by combining alias analysis with classic range analysis on the symbolic domain. Our

Program	#Queries	%scev	%basic	%rbaa	%(r + b)
cfrac	89,255	0.87	9.70	16.65	21.03
espresso	787,223	2.39	12.62	28.16	33.04
gs	608,374	15.56	40.67	56.18	59.99
allroots	974	16.32	64.37	79.77	79.88
archie	159,051	0.98	20.57	16.44	28.04
assembler	35,474	2.16	40.31	47.86	55.61
bison	114,025	0.74	10.95	9.56	14.74
cdecl	301,817	13.74	24.80	49.72	50.73
compiler	9,515	0.49	67.27	67.27	69.20
fixoutput	3,778	0.11	88.30	83.17	90.37
football	495,119	3.58	59.20	60.08	65.08
gnugo	13,519	9.23	60.89	78.21	79.29
loader	13,782	2.32	29.55	36.47	46.09
plot2fig	27,372	2.90	24.09	46.45	49.54

Fig. 3: Comparison between three different alias analyses in terms of "no alias" answers / all pointers pairs. **rbaa** is our analysis and **r+b** (our+basic).

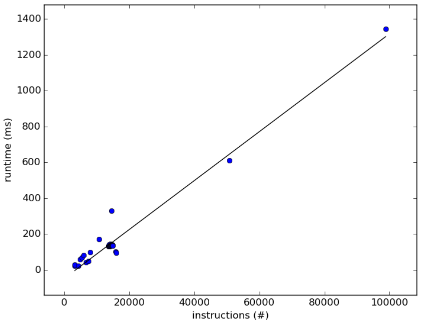


Fig. 4: Runtime of our analysis for the 50 larger benchmarks of LLVM test suite. Each point on the X-axis represents a different benchmark. Benchmarks are ordered by size. R=0.082

analysis is fast, and handles cases that the implementations of pointer analyses currently available in LLVM cannot deal with.

Bibliography

[1] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.

[2] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.

[3] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *PLDI*, pages 177–186. ACM, 1993.

[4] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, pages 290–299. ACM, 2007.

[5] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, pages 265–280, 2011.

[6] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61. ACM, 2001.

[7] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. Validation of memory accesses through symbolic analyses. In *OOPSLA*, pages 791–809. ACM, 2014.

[8] Vitor Paisante, Maroua Maalej, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintao Pereira. Symbolic Range Analysis of Pointers. In *International Symposium of Code Generation and Optmization*, pages 791–809, Barcelon, Spain, March 2016.

[9] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.*, 23(2):105–186, 2001.

[10] L. Shang, X. Xie, and J. Xue. On-demand dynamic symmary-based points-to analysis. In *CGO*, pages 264–274. ACM, 2012.

[11] Qirun Zhang, Xiao Xiao, Charles Zhang, Hao Yuan, and Zhendong Su. Efficient subcubic alias analysis for C. In *OOPSLA*, pages 829–845. ACM, 2014.

[12] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005.

Session du groupe de travail GLACE

Génie Logiciel pour les systèmes Cyber-physiquEs

Software Development Support for Shared Sensing Infrastructures: a Generative and Dynamic Approach

Cyril Cecchinel[1,2], Sébastien Mosser[1,2], and Philippe Collet[1,2]

[1]Université Nice Sophia Antipolis, I3S, UMR 7271, 06900 Sophia Antipolis, France
[2]CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France
{cecchine,mosser,collet}@i3s.unice.fr

Abstract. Sensors networks are the backbone of large sensing infrastructures such as *Smart Cities* or *Smart Buildings*. Classical approaches suffer from several limitations hampering developers' work (*e.g.*, lack of sensor sharing, lack of dynamicity in data collection policies, need to dig inside big data sets, absence of reuse between implementation platforms). This paper presents a tooled approach that tackles these issues. It couples *(i)* an abstract model of developers' requirements in a given infrastructure to *(ii)* timed automata and code generation techniques, to support the efficient deployment of reusable data collection policies on different infrastructures. The approach has been validated on several real-world scenarios and is currently experimented on an academic campus.

Keywords: Sensor Network · Software Composition · Modeling.

1 Introduction

The *Internet of Things* [13] relies on physical objects interconnected between each others, creating a mesh of devices producing information flow. The Gartner group predicts up to 26 billions of things connected to the Internet by 2020. These *things* are organized into sensor networks deployed in *Large-scale Sensing Infrastructures* (LSIs), *e.g.*, *Smart Cities* or *Smart Buildings*, which continuously collect data about our environment. These LSIs implement *Cyber Physical Systems* (CPSs) that monitor ambiant environments.

Facing the problem of managing tremendous amounts of data, a commonly used approach is to rely on sensor pooling [9], [19] and to push data collected by sensors in a central cloud-based platform [15]. Consequently, sensors cannot be exploited at the same time and one needs to rely on data mining solutions to extract and exploit relevant data according to usage scenarios [1], [17]. This approach is adapted for many scenarios where data mining techniques are required, and has the advantage of separating concerns of data collection from data exploitation. Nevertheless, there are many real-life case studies and scenarios where developers need to exploit shared LSIs and implement a diversity of applications that do not need data mining expertise [4]. In this context, the cloud servers create *de facto* silos that isolate datasets from each others and act as a centralized bottleneck. In addition, the computation capabilities of the other layers of the LSI (*e.g.*, the micro-controllers used to pilot the sensors at the hardware level, or the nano-computers acting as network bridges to connect a local sensor infrastructure to the Internet) are under-exploited [10].

To develop software that fully exploits a LSI, the infrastructure must be considered as a white box. But the developer tasks is then more complex as they have to deal with tedious low-level details of implementation out of their main business concerns. This assumes a deep knowledge of micro-controller (sensor platforms) and nano-computer (bridges) programming [4], while a diversity of technological platforms must be handled. In such a situation, programming at the higher level of abstraction and reusing as much code as possible between scenarios and LSIs is of crucial importance. Moreover developers also have to deal with the sharing aspects. It is hard for them as new requirements must be enacted on the LSI and may easily interfere with the other ones.

In this paper, we propose a tooled approach that tackles all these problems and aims at improving reuse, supporting sharing and dynamic data collection policies. A software framework enables developers to specify and program at an appropriate level their sensor exploitation code. It relies on an abstract model of developers' requirements in a given infrastructure so that timed automata and code generation techniques can be combined to support the efficient deployment of data collection policies into a LSI. As a result, several applications can rely on the same sensors, and thus share them. A given code can be reused, translated and deployed on different infrastructures. The framework also ensures that multiple policies can be dynamically composed, so that the generated code automatically handle all requirements and get only relevant data to each consumer.

The remainder of this paper is organized as follows. In Sec. 2, we describe the motivations of our work by introducing a real-life case study and organizing requirements. We present in Sec. 3 the foundations of our framework. In Sec. 4 we assess our approach, providing an illustration, discussing current applications and validating the identified requirements. Sec. 5 discusses related work while Sec. 6 concludes this paper and describes future work.

2 Motivations

In this section, we motivate our work by first introducing the SmartCampus project, then by defining requirements for a development support for shared LSIs. SmartCampus has been deployed on the SophiaTech campus[1] of the University of Nice, located in the Sophia Antipolis technology park. We also introduce a running example extracted from SmartCampus.

2.1 The SmartCampus Project

The SmartCampus project is a prototypical example of an LSI [4]. It acts as an open platform to enable final users (*i.e.*, students, teaching and administrative staff) to build their own innovative services on top of the collected (open)

[1] `http://campus.sophiatech.fr/en/index.php`

data. This project is exactly the class of LSI this work is addressing as it faces the following issues : *(i)* it is not possible to store all the collected data in a big data approach and *(ii)* even if one can afford to store all these data, the targeted developers do not master data mining techniques to properly exploit it. Typically, pieces of software deployed in SmartCampus are driven by functional requirements (*e.g.*, "where to park my car?") and leverage a subset of the available sensors (here the parking lot occupation sensors) to address these requirements. Contrarily to classical systems that use sensor pooling [9, 19], different applications, such as the parking place locator and an emergency system assessing the availability of fire brigade access in the parking lots, rely on the same set of sensors at the very same time. Moreover, developers do not know the kind of hardware deployed physically in the buildings. To support software reuse across different architectures, there is a need to abstract the complexity of using such heterogeneous sensor networks at the proper level of abstraction.

2.2 Supporting Shared Sensing Infrastructure

From the functional analysis of the SmartCampus project (ended in December 2013), we highlighted four requirements with respect to software reuse [4]. These requirements are not only project specific, but also do apply to any IoT-based platform needing to share its sensors on a large scale, *e.g.*, LSIs. This class of system include upcoming smart-building or smart cities wishing to aggregate communities of users to leverage sensors-based system and produce innovative services based on citizen needs.

Pooling and Sharing (R_1). Classical systems rely on sensor polling, with corresponding booking policies. Typical examples of such systems are, for example, the IoT lab platform in France, containing 2,700 sensors deployed in 5 research centers across the country for experiments [9], or the Santander smart city with a closed set of IoT based scenarios [19]. Users book a subset of sensors, work with it and release it afterwards. This setup does not match what is expected in the SmartCampus context. Moreover sensors available in an LSI are classically available through pooling mechanisms [12], [21]. On the one hand, this mechanism is useful when sensors are installed to match a particular setup and deliver a single service. But on the other hand, it is completely irrelevant to the set of scenarios defined by our class of application. Providing a system that only support sharing is also irrelevant, as one needs to deploy critical pieces of software to sensors and be assured that such a critical process will be isolated from other processes (this is similar to the virtual machine isolation requirement in cloud computing).

Yield only relevant data (R_2). To support sharing, classical architectures actually collect data at the minimal available frequency, and store the complete dataset in a cloud-based system, like in Xively [15]. Then, data mining techniques are used to recompute the relevant data from this complete dataset [1], [17]. This faked sharing leads to two type of issues: *(i)* application developers must be aware of the data mining paradigm instead of focusing on their system and *(ii)* as their is no model of what data was expected by the application, it is impossible to reuse the code written for a given LSI into another one. It is worth to note that by yielding only relevant data, developers who want to express mining scenarios will simply define as relevant a wider set of data than classical developers, making the two approaches non-exclusive.

Dynamically support data collection policies (R_3). There is a need to model what kind of data are expected by a given application, in a data collection policy. From a developer perspective, this requirement is critical as it is not reasonable to program specifically for each LSI addressed by the same application, utterly preventing software reuse. In addition applications exploiting the sensors require to change their policies, for example by temporarily increasing some collection frequency during a short period of time. These policies must be enacted dynamically on the LSI to support such changes. Working with a formal definition of what data are expected by the different consumers is the entry point to apply verification and validation techniques on LSIs.

Handling the infrastructure diversity (R_4). As the state of the art relies on artifacts defined at the code level, it is difficult if not impossible to support software reuse across different LSIs. Several approaches leverage operating systems techniques to provide a standard way to program sensors (*e.g.*, TinyOS [14], Contiki [8]). However, these operating systems must be supported by the available sensors. If not, it is up to the developer to manually translate the code from one system to another. In addition, even in the same LSI, hardware obsolescence requires to replace old sensors by new one, often with new hardware due to sensor production life cycle [3]. Thus, it is critical to operate at a code-independent level to express data collection policies.

Summary. Classical approaches are deporting all the sensing intelligence to a cloud-based solution, under-exploiting the computation capabilities of the other layers of the LSI [10]. This also floods the cloud storage with irrelevant data. Even with the help of data mining solutions, the resulting architectures are either centralized with important overhead, or distributed, but still based on an inflexible mining pipeline with identified bottlenecks [17]. Thus, based on the analysis made in the SmartCampus, there is no solution covering all the four requirements expressed by the project. To some extent, this is not surprising. These approaches rely on a strong hypothesis of single consumer and very limited access to the sensors. Thus, only half of R_1 (*i.e.*, pooling) and half of R_2 (*i.e.*, mining) are covered. The two last requirements R_3 & R_4 are covered at the code level, preventing reuse and making maintenance and evolution complex. Considering the emerging class of system that targets communities of developers such as the Fireball project[2], the previous assumptions does not hold anymore.

2.3 Running Example

We introduce here a running example to illustrate the approach. It is a simplification of the use cases identified in the SmartCampus's experimental LSI [4].

[2] `http://www.fireball4smartcities.eu/`

The SmartCampus implementation defines a CPS based on two layers: micro-controllers (sensors and sensor boards) and nano-computer (bridging the sensor network and the Internet). Sensor measurements are sent to a sensor board, which aggregates sensors physically connected to it. The board is usually implemented by a micro-controller that collects data and send them to its associated bridge. A bridge aggregates data coming from multiple sensor boards (thanks to radio or wire-based protocols) and broadcasts on the Internet the received streams to a data collection API, using classical Ethernet connection.

In our example we consider two users, Alice and Bob, who need to use the same sensors to build their own application. Alice develops an application exploiting the associated LSI by collecting data from a temperature sensor every couple of second. Without any specific support, she has to write *(i)* code to be enacted on the different micro-controllers linked to temperature sensors (an infinite loop measuring the temperature every 2 seconds), *(ii)* code to aggregates these data at the bridge level (reading the data sent by the micro-controllers in proprietary representations, and sending it to the cloud-based collector, after having performed data translation from micro-controller format to the collector one), and finally *(iii)* the code that exploits the collected data to implement her application. We claim here that only the latter should be Alice's concern. On his side, Bob develops an application exploiting a temperature sensor each second and a humidity sensor every three seconds. He needs to perform the same kinds of actions as Alice : *(i)* writing the code that reads temperature sensors each second and humidity sensors every 3 seconds, *(ii)* aggregating these data at the bridge level and *(iii)* implementing his application exploiting the collected data.

As simple as this example is, it illustrates the identified requirements. Both users will need to use the same temperature sensor (R_1), and as they have different usages of this sensor, we do not want them to be flooded with non-desirable data (R_2). As the sensor network evolves and has to support new users (*i.e.*, the arrival of Bob), it needs to dynamically adapt the data collection policies (R_3). Finally, as the sensor networks is going to be heterogeneous and composed of different layers (*i.e.*, collection and network), the produced code must automatically fit the infrastructure (R_4).

3 Contribution: The COSmIC Framework

To address the four identified requirements, we propose the *COSmIC* framework, a set of *Composition Operators for Sensing InfrastruCtures.* This section describes the foundations underlying the framework.

3.1 Data Collection Policies as Timed Automata

In sensor networks, automata are commonly used for protocol modeling, and component model approaches [23] are used to develop embedded applications, focusing on the definition of *Interface Automaton* between each components. These automaton-based interfaces enable the different components. We propose to leverage this representation to *(i)* model the data collection policy expressed by the developer, implementing what she expects from an LSI through code generation *(ii)* compose and decompose (dynamically) these policies to handle sharing and infrastructure diversity.

We define a data collection policy $p = \langle Q, \delta, q_0 \rangle$ as a simplification of a classical timed automaton. Q is the set of states defined by the automaton, $\delta : Q \to Q$, its (deterministic) transition function from a given state to another one and finally q_0 its initial state. In real LSIs, tick period is rarely lesser than one second, thus our model assumes a single logical clock that triggers a transition each second. As a policy aims to be indefinitely executed on an LSI, it must be cyclic, and the length of the cycle represents the period of p, denoted as P_p. A given state $q \in Q$ contains an ordered set of actions A implementing the way the developer interacts with the LSI.

In our example, the corresponding policy for Alice p_a is represented by an automaton (depicted in Fig. 1) with two states $\{a_1, a_2\}$.

As a policy needs to be enacted on different platforms, user requirements are translated into a set of basic operations:

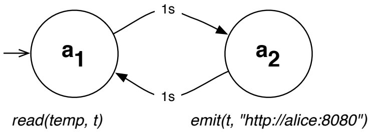
- *read*: Read the value of a *sensor*, *e.g.*, for actions used in our temperature and humidity example.
- *emit*: Send a value to an external *endpoint*, which is usually implemented as a Web service exposing a destination URL for the collected data.

According to this representation and the associated actions, a software developer is able to model what she expects from the LSI for her given use case. The key point is that the developer is completely unaware of the internal implementation of the LSI, and only focuses on reading sensor values and sending data over communication interfaces.

3.2 The Generator Operator (γ)

The designed models are useless if not coupled to code generation algorithms that transform these logical representations into executable code. One of the main issues to tackle here is the variability existing between the different hardware elements that compose an LSI (R_4). For example, at the micro-controller level, a plain Arduino board does not support the *emit* action, whereas an Arduino coupled to an Ethernet communication shield supports it.

We thus define a code generator γ as a couple of functions (pre, do), each consuming as input a policy p. The *pre* function checks a set of preconditions on p to ensure that this policy can be projected to the hardware platform targeted



$$actions(a_1) = \{read(\text{temp}, t)\}$$
$$actions(a_2) = \{emit(t, \text{“http://alice:8080”})\}$$
$$\delta_a(x) \mapsto (x = a_1 \quad a_2) \quad (x = a_2 \quad a_1)$$
$$p_a = (\{a_1, a_2\}, \delta_a, a_1)$$

Fig. 1. Excerpt of a data collection policy

by p. The *do* function takes as input the policy to transform, as well as additional parameters given by the environment. These parameters map logical names to physical elements when relevant (*e.g.*, sensors on an Arduino platform are only identified by the pin number they are plugged in). In a production environment, the generators are not executed by the developer herself. She will only express her needs, and will enact them on the LSI which actually knows its internal infrastructure.

For example, we consider here the policy p_a defined in the previous section, and an Arduino platform as generation target. The corresponding precondition checker assesses the absence of *emit* actions in the following way:

$$pred_{ard}(p) \mapsto \forall q \in Q_b, \nexists emit(\_, \_) \in actions(q) \tag{1}$$

In this case, the checker detects that this policy cannot be deployed on an infrastructure based solely on Arduino, as this hardware does not match the requirements expressed in the policy, *i.e.*, emiting a data to the Internet.

Considering a policy p valid for the Arduino platform (the decomposition operator described in the next section shows how to make p_a valid), the do_{ard} function then visits p to produce the code to be executed on the board, using the Wiring[3] language as target. It also takes as input a *map* acting as a registry (stored in the LSI environment) binding a sensor name to the physical pin that connects it to the board. Lst. 1.1 shows the resulting code for a board coupled to temperature sensor *temp* on pin 9 and an humidity one *hum* on pin 10. The generator maps actions such as $read(temp, t)$ into Wiring code like `v_t = analogRead(9)`.

```
void setup() { // Initialization
  pinMode(9,INPUT); pinMode(10,INPUT);
}
void a1() {
  v_t = analogRead(9); v_h = analogRead(10);
  delay(1000); return a2();
}
void a2() {
  v_t = analogRead(9);
  delay(1000); return a1();
}
void loop() { return a1(); } // Entry point
```

Listing 1.1. Generated code example: $do_{ard}(p)$

Code generators also allow users to reuse their policy for different sensing infrastructures as a given COSmIC policy can be translated to many targets.

3.3 The Decomposition Operator (δ)

Considering a given policy, it has to be decomposed into software artifacts that make sense on the different layers of the LSI, *i.e.*, the micro-controller and bridge layers. For each layer, there may be actions that are incompatible with the hardware. Consequently, a given policy p must be decomposed into n layer-specific sub-policies p'_{layer} (where n is the number of layers) that communicate together and where incompatible actions are substituted by internal communication [20].

[3] Wiring is an open-source framework for micro-controllers (`http://wiring.org.co/`).

This decomposition is performed thanks to a compatibility table T. A function $f_T(a, P)$ applied on this table returns a boolean value reflecting the compatibility of an action a on the platform P[4]. This decomposition process is defined as follows:

$$\delta(p) \equiv \forall a \in p, \forall P \in \mathit{layers}, f_T(a, P) \Rightarrow a \in p'_a \qquad (2)$$

If we consider micro-controllers implemented by Arduino boards and bridges implemented by Raspberry nano-computers, the micro-controller level will not support the emission of data to external endpoints, due to a lack of proper communication interface. In our example, Alice's policy will be decomposed by the operator into two sub-policies: $\delta(p_a) = \{p'_{mic}, p'_{bri}\}$. Consequently, the p_{mic} will read the sensor value and send it to an internal endpoint (substitution of the *emit* action) thanks to the serial communication that links the sensor board to its bridge. This policy is accepted by the $pred_{ard}$ function defined in the Arduino code generator, meaning that p_{mic} can be deployed on such hardware. The p_{bri} policy is executed on the bridge to read the internal communication port and emit the received data to the external endpoint.

3.4 The Composition Operator ($\oplus$)

On a shared LSI, policies designed by different developers will be executed on the very same piece of hardware COSmIC provides a composition operator denoted as $\oplus$ at the automaton level. It composes two given policies and produces a single policy containing an automaton corresponding strictly to the parallel composition of the two inputs.

The $\oplus$-operator assimilates a timed automaton implementing a policy p with a period P_p as a periodic function. The composition of two periodic functions f_1 and f_2 is a periodic function $f = f_1 \circ f_2$ where its period P_f is the least common multiple of P_{f_1} and P_{f_2}. Applied to policies, this means that the composition of two policies p_1 and p_2, denoted as $p = p_1 \oplus p_2$ is a policy with a period $P_p = lcm(P_{p_1}, P_{p_2})$, where each actions of p_1 (respectively p_2) are executed according to P_{p_1} (respectively P_{p_2}). As this $\oplus$-operator is endogenous, it allows a software developer to dynamically reuse a policy by composing it with new incoming policies.

In our example, the policies defined by Alice in p_a and Bob in p_b will exploit the same temperature sensor on the same micro-controller and use the same bridge to emit their values. More details on the composition and decomposition processes are given in Sec. 4.2.

4 Assessment

In this section, we describe the current implementation and its application to our prototypical LSI. Then, we show how COSmIC can be used to model and deploy the running example. We validate our identified requirements through some acceptance criteria and finally discuss threats to validity.

[4] An example of such a table is given in Sec. 4.2.

4.1 Implementation and Application

The initial prototype of the COSmIC framework is available on GitHub[5]. It is implemented with the Scala language (∼ 3500 lines of code) and covers all the concepts presented in Sec. 3. We are currently experimenting COSmIC on Arduino, Raspberry Pi and Cubieboard platforms as part of the SmartCampus project. We also used the FIT IoT-lab platform[6], featuring a pool of over 2700 sensors nodes spread across France, to experiment on the ARM Cortex M3 platform.

To experiment and demonstrate the abstraction of platforms, code generation capabilities, sharing and reuse, we have modeled four identified SmartCampus scenarios and then generated code for each platform type we experiment with:

- *S1 - Late worker detection*: at night, occupied offices are detected by checking if the light is on (*light sensor*) and if there is someone in the office (*presence sensor*);
- *S2 - Fire prevention*: a warning signal on a temperature threshold (*temperature sensor*);
- *S3 - Heat monitoring*: air-conditioning and heating are controlled by checking the ambient air in buildings (*temperature sensor*);
- *S4 - Energy wasting*: To comply with environmental standards, the quality manager wants to monitor light kept on when the building is empty (*light sensor* and *presence sensor*).

Table 1 presents the number of lines of code (LoC) generated for each platform. Every code generator includes a static overhead (template code), specific to the targeted platform. This template provides the implementation of methods called by the COSmIC code generation. The corresponding LoC (italic row on TAB. 1) vary between platforms as some of them are providing more features. The Raspberry Pi template contains only 85 LoC, corresponding to serial reading and value emission on the Internet (a Raspberry Pi cannot read values directly from sensors in the SmartCampus infrastructure). On the other hand, the ARM Cortex M3 template comprises 169 LoC to handle its sensor and network interface. Using a template is efficient as we target low-level platforms, and those functions encapsulate a part of their complexity. For example, a method provided in the ARM Cortex M3 template handles the IPv6 retrieval of sensor measures from a border-router.

We can first observe that without considering the boilerplate code defined in the templates, there is no real difference in terms of LoC between COSmIC and the underlying programming languages. This is not surprising as the design choice of using templates hides low-level details to raise the level of abstraction of each platform. But the key point is that the code written with the COSmIC framework is not a single-target code but actually a model, which can be verified, composed automatically and projected to multiple platforms. Another interesting property is that users do not need to know the underlying platform. For example, if a policy relies only on digital sensors, one can use the Contiki

[5] `http://ace-design.github.io/cosmic/`

[6] `https://www.iot-lab.info/`

Table 1. LoC resulting from scenario generation

	Arduino native	Arduino contiki	Raspberry / Python	ARM Cortex M3 / Python	COSmIC source
Template	*13*	*22*	*85*	*169*	*0*
S_1	6	14	13	11	7
S_2	5	13	11	10	5
S_3	5	13	11	10	7
S_4	6	14	13	11	7
$S = S_1 \oplus S_2 \oplus S_3 \oplus S_4$	63	51	45	39	27
Deployed: S + *Template*	76	73	160	208	N/A

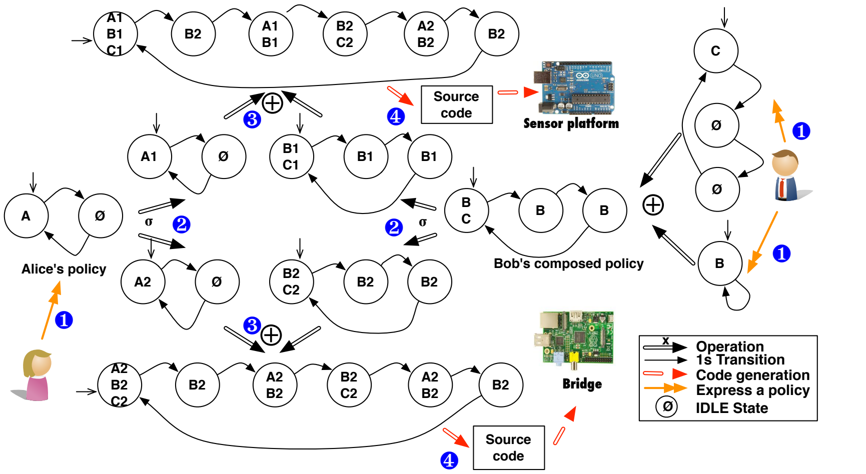


Fig. 2. COSmIC processes on the running example

operating system [8] to use a thread-based implementation of this policy. If a new requirement including values coming from analog sensors needs to be enacted on the same board, Contiki cannot be used anymore and the implementation must be completely rebuilt using only native operations. This is not the case with COSmIC: the two policies will be automatically composed, and it simply implies to change the call to the code generator as the Contiki one will reject the composed policy.

4.2 Illustration

We illustrate the application of the COSmIC operators from policy definition to code generation on the Alice and Bob example (see Sec. 2.3), on the top of the SmartCampus infrastructure[7]. Fig. 2 gives an overview of the different activities.

[7] More details can be found on a companion web page: `https://github.com/ace-design/cosmic/blob/master/publications/ICSR15.md`

❶ *Policies definition.* In a first step, both users have to define their data collection policies in terms of timed automaton. The Alice's timed automaton p_a is already presented in Sec. 3.1. Bob has to express two policies: *(i)* temperature collection policy (p_{bt}) and *(ii)* humidity collection policy (p_{bh}). Bob uses the $\oplus$-operator to build a single policy p_b containing both temperature collection and humidity collection policies.

❷ *Decomposition process.* The next step is related to the decomposition process thanks to the δ-operator. Policies p_a and p_b are global policies that contain incompatible actions for the Arduino micro-controller (e.g. the *emit* action) platform and for the Raspberry nano-computer (e.g. the *read sensor* action). The appropriate compatibility table (Tab. 2) drives the decomposition process, reifying the compatibility of actions per platform. As presented in Sec. 3.3, incompatible actions are substituted by internal communications. After this decomposition process, four layer specific sub-policies are obtained:

$$\delta(p_a) = \{p_{amic}; p_{abri}\} \qquad \delta(p_b) = \{p_{bmic}; p_{bbri}\} \tag{3}$$

❸ *Composition process.* These four sub-policies will be then deployed on the shared infrastructure. The $\oplus$-operator will compose those policies and allow Alice and Bob to exploit the same piece of hardware. p_{amic} is composed with p_{bmic}, and p_{abri} is composed with p_{bbri}:

$$p_{amic} \oplus p_{bmic} = p_{\text{Sensor platform}} \qquad p_{abri} \oplus p_{bbri} = p_{\text{Bridge}} \tag{4}$$

The composition process is an endogenous operation returning a policy that can be reused to be composed, possibly in a dynamic way, with future policies. $p_{\text{Sensor platform}}$ and p_{Bridge} are the policies that will be instantiated on the infrastructure.

❹ *Code generation.* The final step of the deployment process is handled by code generators working directly on the two latter policies. The generated codes are then flashed on the appropriate micro-controllers and bridges using classical LSI deployment tools. At runtime, Alice and Bob will receive sensor values for their application according to their respective needs, although the same sensor is used for both of them.

4.3 Validation

To validate the four requirements presented in Sec. 2, we define an acceptance criterion for each of them and discuss how they are met.

R_1: Pooling and Sharing - More than one application can rely on a given sensor. The illustration in Sec. 4.2 shows that different policies can be enacted on the

Table 2. Excerpt of the COSmIC compatibility table

	Arduino Uno	Raspberry Pi	ARM Cortex M3
read	✓	✗	✓
emit	✗	✓	✓

sensing infrastructure to feed different applications. We performed also this validation on the SmartCampus infrastructure with four scenarios (cf. Sec. 4.1). In this context Table 3 illustrates that the same sensor will be used for different scenarios, validating requirement R_1.

R_2: Yield only relevant data - A given application is only fed with what it expects. As shown in our illustration (Sec. 4.2), a COSmIC user models her data collection policies with timed automaton and triggering of *emit* actions with requested data periodically. The composition operator also maintains this property by construction. It handles two policies p_1 and p_2 respectively T_1 and T_2 periodic, and produces a new lcm(T_1,T_2)-periodic policy. This process is transparent for COSmIC users as her expressed policy will not be modified while she will only receive data as specified in her initial policy. This validates requirement R_2.

R_3: Dynamically support data collection policies - Multiple policies can be dynamically composed. The $\oplus$-operator allows the composition of data collection policies on a sensor network. In the illustration (Sec. 4.2), Bob's policies have been composed into a single one using the $\oplus$-operator. The resulting policy can be used by other operators. Therefore, when a new policy needs to be added to the sensor network, one has just to compose it with the already deployed policy. This endogenous property validates requirement R_3.

R_4: Handling the infrastructure diversity - A given code can be deployed on more than one infrastructure. The infrastructure hardware variability is handled with code generators. These code generators handle a COSmIC DSL input code and produce the code for a given platform. We have successfully modeled and deployed the SmartCampus scenarios on Arduino, Raspberry Pi and ARM Cortex M3 platforms, validating requirement R_4.

4.4 Threats to validity

Scenarios. Our approach is only applied to the SmartCampus context. Even if the corresponding scenarios have been validated through questionnaires and are close to other case studies such as SmartSantander [19], we are aware that we need to step back and introduce more complex scenarios to benchmark COSmIC on a larger scale.

Table 3. Sensor sharing

	Light	Temperature	Presence
Scenario 1 - Late worker detection	✓		✓
Scenario 2 - Fire prevention		✓	
Scenario 3 - Heat monitoring		✓	
Scenario 4 - Energy wasting	✓		✓

Timed automata. Our data collection policies are represented by timed automata. If this approach fits the SmartCampus use case, the combination of different scenarios can lead to a combinatorial explosion, (*e.g.*, collections on a shared sensor at frequencies of one second and one hour would lead to a 3600 states automaton with only two relevant states). The code generation process is impacted by such automata. We currently reduce the size of such automata thanks to a factorization process, but this optimization does not scale with a large number of concurrent scenarios. The use of such automata also impacts the resources. Platforms have to be always powered on, to the detriment of the battery autonomy, to maintain a running clock delivering periodic clock tick. Devising better techniques to handle such cases and providing resource management is part of our future work.

Action execution duration. Our automata represent clocks with a 1 Hz frequency. If the execution of an action is longer than one second, it might be overlapped and aborted by the state transition leading the policy into an inconsistent state. In the future, we plan to use languages based on the formal Clock Constraint Specification Language (CCSL) [6] to determine the duration of action execution and to ensure the temporal correctness of policies.

Deployment of new policies. Our approach handles the dynamic composition of data collection policies and code generation for a given platform. However, we do not support dynamic deployment as some sensor platforms need to be reflashed with a new firmware. When the platform support it, we rely on operating systems (*e.g.*, Contiki) to support this feature.

5 Related Work

Programming sensor networks with specific OS. Several operating systems have been specifically designed for sensing infrastructures, *e.g.*, TinyOS [14] or Contiki [8]. TinyOS is based on a component architecture and comes with its programming language NesC. A developer can create new components or reuse components from the TinyOS's component library to build her own application. Contiki is adapted for networked and resource-constrained devices. Contiki applications can be written and compiled using a specific C compiler. Those OS abstract some complexities of application development, such as memory or energy optimization, but the developer has to be aware of what kind of sensor platforms she is using, directly dealing with their implementation details at a lower level. This leads to a lack of reusability, whereas our approach introduces a generic way to program sensor network. The COSmIC code is written independently from a sensing infrastructure and code generators handle the transformation to a targeted platform.

Sensor network as a database. On top of operating systems deployed on sensing infrastructure, several approaches consider the sensor network itself as a database [7]. Storing the data as close as possible to the sensor producing it instead of pushing everything to the Cloud was demonstrated as cost-efficient and energy saving [22]. The TinyDB system [16] (not maintained since 2005) provides processing mechanisms for sensor querying and data retrieval. It considers a sensor as a micro-database storing their collected data, and allows developers to query sensors according to different criteria (*e.g.*, location). The Cougar system [24] also considers data collected by sensors, and supports users by only expressing queries that are automatically propagated to the sensors. This system does not support sharing (as queries cannot be composed easily), and relies on a centralized engine that computes a collection planning and collects data. On the contrary, COSmIC fully distributes the policies to the different sensors and the infrastructure layers, and supports multiple endpoints for each application.

Model-driven and generative approaches. The model-driven development paradigm has been notably used to design dynamically adaptive systems and to evolve them at runtime [18]. In this approach, the current context model is analyzed at runtime and, if an adaptation needs to be performed, a suitable configuration is built thanks to reference models. The approach can fit lightweight nodes in a sensor network [11]. Our work differs as we do not perform adaptiveness according to the context but design sensor network applications with policies based on a composition equation that can be reused for other compositions or for verification purposes. Exploiting runtime composition is a perspective of our work. The way we generate code is close to the Scalaness approach [5]. It is a type-safe language used to wirelessly program embedded networks running under TinyOS. Two stages are required to program these networks: *(i)* one writes a Scalaness program, which is then *(ii)* translated into Java bytecode. We differ from this approach as we use behavior models to generate code and we do not always have the same destination platforms as we target heterogeneous sensor networks.

6 Conclusions & Perspectives

In this paper we have presented the *COSmIC* Framework used for supporting different developers' collect policies on a shared LSI, generating code deployed at the approriate layer of the LSI. It addresses several limitations of classical approaches, focusing on the sharing of the infrastructure and the production of relevant-only datasets, and allowing software developers to focus on their concerns instead of LSI implementation details. The framework is implemented using the Scala language and preliminary experiments have been conducted on top of the SmartCampus platform [4]. The *COSmIC* framework is a first step for composing policies on an LSI, with a focus on policy definition and composition operators.

Future work aims at extending the approach and making it scale to very large LSIs. First, we will extend this set of operators to build a complete composition algebra, with a formal definition of operator properties (*e.g.*, commutativity, associativity, idempotency), conflict detection mechanisms to prevent inconsistent states (*i.e.*, sending a value before reading it) and a formal support to attach constraints to actions. We also plan to extend these constraints to timed ones, using the TimeSquare toolkit [6] to specify and analyze constraints based on its logical time model and to check them also at runtime. We also plan to enlarge the set of interactions with the LSI by introducing new actions allowing a developer to perform some data computation within the sensor network (*e.g.*, Compute the average value of data coming from different sensors.

For those developers, we will improve the available abstractions by providing a higher level DSL. It will notably hide the creation and management of states and transitions, providing a real focus on what data are collected, processed and used in applications.

The decomposition operator also triggers interesting challenges with respect to the variability of hardware (*i.e.*, Arduino, Phidgets platforms) and facilities (*i.e.*, Supported programming language, resources available) available in the context of LSIs. We plan to use a feature modeling [2] approach to capture this variability, and to bind these models to the generation mechanisms, providing a variable code generation according to the available hardware in a given LSI. Finally, we also plan to support policy composition and variability reasoning at runtime to handle dynamic adaptiveness. We expect the resulting tooled approach to provide an end-to-end support for developers of the massively under-deployment sensing infrastructures.

References

1. Aggarwal, C.C. (ed.): Managing and Mining Sensor Data. Springer (2013)
2. Apel, S., Batory, D.S., Kästner, C., Saake, G.: Feature-Oriented Software Product Lines - Concepts and Implementation. Springer (2013)
3. Buratti, C., Conti, A., Dardari, D., Verdone, R.: An overview on wireless sensor networks technology and evolution. Sensors 9(9), 6869–6896 (2009), `http://www.mdpi.com/1424-8220/9/9/6869`
4. Cecchinel, C., Jimenez, M., Mosser, S., Riveill, M.: An Architecture to Support the Collection of Big Data in the Internet of Things. In: International Workshop on Ubiquitous Mobile Cloud (UMC'14, co-located with SERVICES'14). pp. 1–8. IEEE, Anchorage, Alaska, USA (Jun 2014)
5. Chapin, P.C., Skalka, C., Smith, S.F., Watson, M.: Scalaness/nesT: Type Specialized Staged Programming for Sensor Networks. In: Jrvi, J., Kstner, C. (eds.) GPCE. pp. 135–144. ACM (2013)
6. Deantoni, J., Mallet, F.: TimeSquare: Treat your Models with Logical Time. In: Carlo A. Furia, S.N. (ed.) TOOLS - 50th International Conference on Objects, Models, Components, Patterns - 2012. Lecture Notes in Computer Science - LNCS, vol. 7304, pp. 34–41. Czech Technical University in Prague, in co-operation with ETH Zurich, Springer, Prague, Tchèque, République (May 2012)
7. Diao, Y., Ganesan, D., Mathur, G., Shenoy, P.J.: Rethinking data management for storage-centric sensor networks. In: CIDR 2007, Third Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings. pp. 22–31 (2007)
8. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - a lightweight and flexible operating system for tiny networked sensors. In: Local Computer Networks, 2004. 29th Annual IEEE International Conference on. pp. 455–462 (Nov 2004)
9. Fambon, O., Fleury, E., Harter, G., Pissard-Gibollet, R., Saint-Marcel, F.: Fit iotlab tutorial: hands-on practice with a very large scale testbed tool for the internet of things. In: 10èmes journées francophones Mobilité et Ubiquité (UbiMob). pp. 1–5 (June 2014)
10. Fleurey, F., Morin, B., Solberg, A.: A Model-Driven Approach to Develop Adaptive Firmwares. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS. pp. 168–177. ACM (2011)

11. Fouquet, F., Morin, B., Fleurey, F., Barais, O., Plouzeau, N., Jezequel, J.M.: A Dynamic Component Model for Cyber Physical Systems. In: Proceedings of the 15th ACM SIGSOFT Symposium on Component Based Software Engineering. pp. 135–144. CBSE '12, ACM, New York, NY, USA (2012)
12. Gluhak, A., Krco, S., Nati, M., Pfisterer, D., Mitton, N., Razafindralambo, T.: A Survey on Facilities for Experimental Internet of Things Research. IEEE Communications Magazine 49(11), 58–67 (Dec 2011), http://hal.inria.fr/inria-00630092
13. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. Future Generation Comp. Syst. 29(7), 1645–1660 (2013)
14. Levis, P., Madden, S., Polastre, J., Szewczyk, R., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., Culler, D.: Tinyos: An operating system for sensor networks. In: in Ambient Intelligence. Springer Verlag (2004)
15. LogMeIn: Xively (May 2014), http://xively.com/
16. Madden, S.R., Franklin, M.J., Hellerstein, J.M., Hong, W.: Tinydb: An acquisitional query processing system for sensor networks. ACM Trans. Database Syst. 30(1), 122–173 (Mar 2005), http://doi.acm.org/10.1145/1061318.1061322
17. Mahmood, A., Ke, S., Khatoon, S., Xiao, M.: Data mining techniques for wireless sensor networks: A survey. IJDSN 2013 (2013)
18. Morin, B., Barais, O., Jezequel, J., Fleurey, F., Solberg, A.: Models@run.time to Support Dynamic Adaptation. Computer 42(10), 44–51 (2009)
19. Sanchez, L., Galache, J., Gutierrez, V., Hernandez, J., Bernat, J., Gluhak, A., Garcia, T.: Smartsantander: The meeting point between future internet research and experimentation and the smart cities. In: Future Network Mobile Summit (FutureNetw), 2011. pp. 1–8 (June 2011)
20. Stickel, M.E.: A Unification Algorithm for Associative-Commutative Functions. J. ACM 28(3), 423–434 (Jul 1981)
21. Tonneau, A.S., Mitton, N., Vandaele, J.: A Survey on (mobile) wireless sensor network experimentation testbeds. In: DCOSS - IEEE International Conference on Distributed Computing in Sensor Systems. Marina Del Rey, California, États-Unis (May 2014), http://hal.inria.fr/hal-00988776
22. Tsiftes, N., Dunkels, A.: A database in every sensor. In: Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems. pp. 316–332. SenSys '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/2070942.2070974
23. Völgyesi, P., Maróti, M., Dóra, S., Osses, E., Lédeczi, Á.: Software Composition and Verification for Sensor Networks. Sci. Comput. Program. 56(1-2), 191–210 (2005)
24. Yao, Y., Gehrke, J.: The cougar approach to in-network query processing in sensor networks. SIGMOD Rec. 31(3), 9–18 (Sep 2002), http://doi.acm.org/10.1145/601858.601861

Modeling and Verification of Functional and Non Functional Requirements of Ambient Self Adaptive Systems

M. Ahmad[a,b], N. Belloir[a], J. M. Bruel[b]

[a] *University of Pau and the Pays of the Adour, LIUPPA*
64000 Cedex, France
[b] *University of Toulouse, CNRS/IRIT*
F-31062 Toulouse Université Cedex, France

Abstract

Self Adaptive Systems modify their behavior at run-time in response to changing environmental conditions. For these systems, Non Functional Requirements play an important role, and one has to identify as early as possible the requirements that are adaptable. We propose an integrated approach for modeling and verifying the requirements of Self Adaptive Systems using Model Driven Engineering techniques. For this, we use Relax, which is a Requirements Engineering language which introduces flexibility in Non Functional Requirements. We then use the concepts of Goal Oriented Requirements Engineering for eliciting and modeling the requirements of Self Adaptive Systems. For properties verification, we use OMEGA2/IFx profile and toolset. We illustrate our proposed approach by applying it on an academic case study.

Keywords: Requirements Engineering, Model Driven Engineering, Relax, Dynamic Adaptive Systems, Domain Specific Language, Non Functional Requirements, Properties Verification, Goal Oriented Requirements Engineering

1. Introduction

As applications continue to grow in size, complexity, and heterogeneity, it becomes increasingly necessary for computing based systems to dynamically self adapt to changing environmental conditions. These systems are called Dynamically Adaptive Systems (DASs) [41]. Example applications that require DASs capabilities include automotive systems, telecommunication systems, environmental monitoring, and power grid management systems. In this context, an adaptive system is a set of interacting or interdependent entities, real or abstract, forming an integrated whole that together are able to respond to environmental changes or changes in the interacting parts. A Self Adaptive Systems (SAS), like other systems, has goals that must be satisfied and, whether these goals are explicitly identified or not, system requirements should be formulated to

Email addresses: manzoor.ahmad@univ-pau.fr (M. Ahmad), nicolas.belloir@univ-pau.fr (N. Belloir), bruel@irit.fr (J. M. Bruel)

Preprint submitted to Elsevier *November 26, 2014*

guarantee goal satisfaction. This fundamental principle has served systems development well for several decades but is founded on an assumption that goals are fixed. In general, goals can remain fixed if the environment in which the system operates is stable [40]. The distributed nature of SAS and changing environmental factors (including human interaction) makes it difficult to anticipate all the explicit states in which the system will be during its lifetime.

It is generally accepted that errors in requirements are very costly to fix [29]. The avoidance of erroneous requirements is particularly important for the emerging class of systems that need to adapt dynamically to changes in their environment. Many such DASs are being conceived for applications that require a high degree of assurance [36], in which an erroneous requirement may result in a failure at run-time that has serious consequences. The requirement for high assurance is not unique to DASs, but the requirement for dynamic adaptation introduces complexity of a kind not seen in conventional systems where adaptation, if it is needed at all, can be done off-line. The consequent dynamic adaptation complexity is manifested at all levels, from the services offered by the run-time platform, to the analytical tools needed to understand the environment in which the DASs must operate.

Requirements Engineering (RE) is concerned with what a system ought to do and within which constraints it must do it. RE for SAS, therefore, must address what adaptations are possible and what constrains how those adaptations are carried out. In particular, questions to be addressed include: what aspects of the environment are relevant for adaptation? Which requirements are allowed to vary or evolve at run-time and which must always be maintained? In short, RE for SAS must deal with uncertainty because the expectations on the environment frequently vary over time. We identify the uncertainty in requirements of these systems and show how to verify it.

We are of the view that, on one hand, requirements for SAS should consider the notion of uncertainty while defining it; on the other hand, there should be a way to verify these requirements as early as possible, even before the development of these systems starts. In order to handle the notion of uncertainty in SAS, RE languages for these systems should include explicit constructs for identifying the point of flexibility in its requirements [41]. In this context, we provide an integrated approach to achieve this objective. We have used two approaches for defining and modeling requirements, i.e. Goal Oriented Requirements Engineering (GORE) techniques are used to define and model the requirements of SAS [23, 44, 28, 43] and SysML is used to specify the system and to provide a link with the requirements.

We propose a model based requirements modeling and verification process for SAS that takes into account the uncertainty in requirements of these systems. We provide some tools to implement our apporach and then apply it on an academic case study. The notion of goals is added to take into account the advantages offered by GORE. Requirements verification is done using model checking technique.

This paper is organized as follows: In section 2, we describe the background and the concepts which form the basis of this work, section 3 shows the state of the art regarding RE for SAS and properties verification of these systems, section 4 shows and illustrate our proposed approach through an example and the tools that we have developed, section 5 shows the case study that we used for the validation of our approach, and section 6 concludes the paper and shows the future work.

2. Background

2.1. RELAX

Relax is an RE language for DASs in which explicit constructs are included to handle uncertainty. For example, the system might wish to temporarily Relax a non-critical requirement in order to ensure that critical requirements can still be met. The need for DASs is typically due to two key sources of uncertainty. First is the uncertainty due to changing environmental conditions, such as sensor failures, noisy networks, malicious threats, and unexpected (human) input; the term environmental uncertainty is used to capture this class of uncertainty. A second form of uncertainty is behavioral uncertainty, which refers to situations where the requirements themselves need to change. It is difficult to know all requirements changes at design time and, in particular, it may not be possible to enumerate all possible alternatives [41].

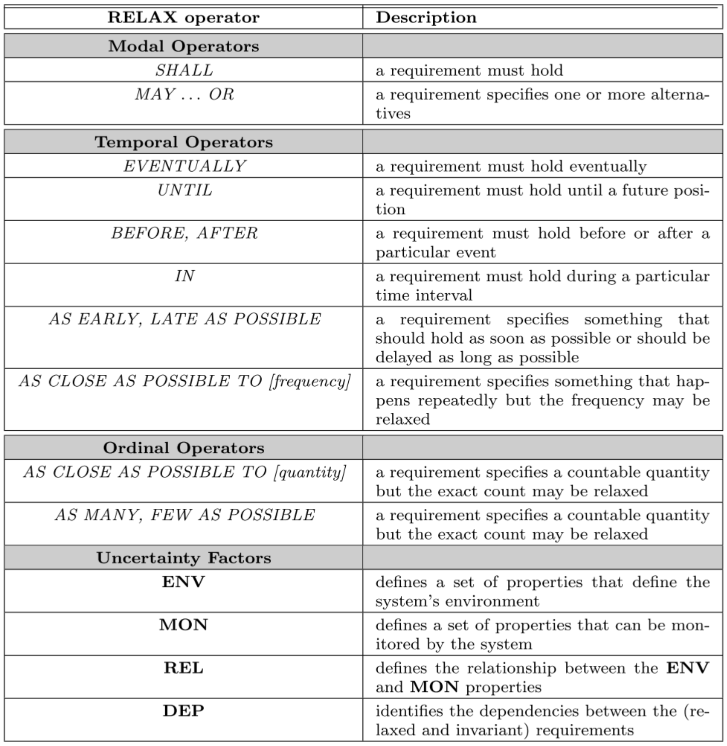
2.1.1. RELAX Vocabulary

The vocabulary of Relax is designed to enable the analysts to identify the requirements that may be Relax-ed when the environment changes. Relax addresses both types of uncertainties. Relax also outlines a process for translating traditional requirements into Relax requirements. The only focal point is for the requirement engineers to identify the point of flexibility in their requirements. Relax identifies two types of requirements: one that can be Relax-ed in favor of other ones called variant or Relax-*ed* and other that should never change called *invariant*. It is important to note that the decision of whether a requirement is invariant or not is an issue for the system stakeholders, aided by the requirements engineer.

Relax takes the form of a structured natural language, including operators designed specifically to capture uncertainty [40]; their semantics is also defined. Figure 1 shows the set of Relax operators, organized into modal, temporal, ordinal operators and uncertainty factors. The conventional modal verb *SHALL* is retained for expressing a requirement with Relax operators providing more flexibility in how and when that functionality may be delivered. More specifically, for a requirement that contributes to the satisfaction of goals that may be temporarily left unsatisfied, the inclusion of an alternative, temporal or ordinal Relax-ation modifier will define the requirement as Relax-able.

2.1.2. RELAX Grammar

The syntax of Relax expressions is defined by the grammar shown in Figure 2. Parameters of Relax operators are typed as follows: p is an atomic proposition, e is an event, t is a time interval, f is a frequency and q is a quantity. An event is a notable occurrence that takes place at a particular instant in time. A time interval is any length of time bounded by two time instants. A frequency defines the number of occurrences of an event within a given time interval. If the number of occurrences is unspecified, then it is assumed to be one. A quantity is something measurable, meaning it can be enumerated. In particular, a Relax expression φ is said to be quantifiable if and only if there exists a function Δ such that $\Delta(\varphi)$ is a quantity. A valid Relax expression



RELAX operator	Description
Modal Operators	
SHALL	a requirement must hold
MAY ... OR	a requirement specifies one or more alternatives
Temporal Operators	
EVENTUALLY	a requirement must hold eventually
UNTIL	a requirement must hold until a future position
BEFORE, AFTER	a requirement must hold before or after a particular event
IN	a requirement must hold during a particular time interval
AS EARLY, LATE AS POSSIBLE	a requirement specifies something that should hold as soon as possible or should be delayed as long as possible
AS CLOSE AS POSSIBLE TO [frequency]	a requirement specifies something that happens repeatedly but the frequency may be relaxed
Ordinal Operators	
AS CLOSE AS POSSIBLE TO [quantity]	a requirement specifies a countable quantity but the exact count may be relaxed
AS MANY, FEW AS POSSIBLE	a requirement specifies a countable quantity but the exact count may be relaxed
Uncertainty Factors	
ENV	defines a set of properties that define the system's environment
MON	defines a set of properties that can be monitored by the system
REL	defines the relationship between the **ENV** and **MON** properties
DEP	identifies the dependencies between the (relaxed and invariant) requirements

Figure 1: Relax Operators [41]

$$\begin{aligned}
\varphi := true \mid & false \mid p \mid SHALL\ \varphi \\
& \mid MAY\ \varphi_1 OR\ MAY\ \varphi_2 \\
& \mid EVENTUALLY\ \varphi \mid \varphi_1 UNTIL\ \varphi_2 \\
& \mid BEFORE\ e\ \varphi \mid AFTER\ e\ \varphi \mid IN\ t\ \varphi \\
& \mid AS\ CLOSE\ AS\ POSSIBLE\ TO\ f\ \varphi \\
& \mid AS\ CLOSE\ AS\ POSSIBLE\ TO\ q\ \varphi \\
& \mid AS\ \{EARLY, LATE, MANY, FEW\} \\
& AS\ POSSIBLE\ \varphi
\end{aligned}$$

Figure 2: Relax Grammar [41]

is any conjunction of statements $s_1 \dots s_m$ where each s_i is generated by the grammar.

The semantics of Relax expressions is defined in terms of Fuzzy Branching Temporal Logic (FBTL) [24]. FBTL can describe a branching temporal model with uncertain temporal and logical information. It is the representation of uncertainty in FBTL that makes it suitable as a formalism for Relax.

2.1.3. RELAX Process

Figure 3 shows the Relax process. The conventional process of requirement discovery has been applied to get *SHALL* statements. Relax process is then used to identify the requirements as invariant and Relax-ed.

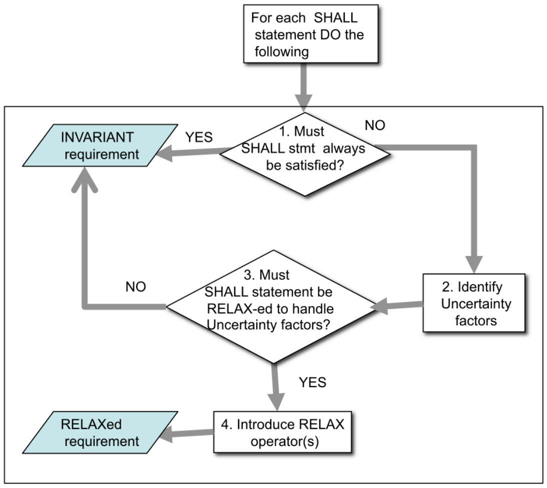

For each SHALL statement DO the following
1. Must SHALL stmt always be satisfied?
YES
NO
INVARIANT requirement
2. Identify Uncertainty factors
3. Must SHALL statement be RELAX-ed to handle Uncertainty factors?
NO
YES
4. Introduce RELAX operator(s)
RELAXed requirement

Figure 3: Relax Process [41]

First of all, for each *SHALL* statement, we check whether it must always be satisfied or not. Then for each potentially Relax-able requirement, we identify the uncertainty factors. Here also the observable properties of the environment are identified. The *ENV*/*MON* relationship is made explicit by *REL*, and *DEP* is used to identify the inter-dependencies between requirements. Then we check whether the *SHALL* statement should be Relax-ed to handle uncertainty factors or not, here we analyze the uncertainty factors to determine if sufficient uncertainty exists in the environment that makes absolute satisfaction of the requirement problematic or undesirable. If so, then this *SHALL* statement needs to proceed to the next step for introducing Relax operators. If, however, the analysis reveals no uncertainty in its scope of the environment, then the requirement is potentially always satisfiable and therefore identified as an invariant.

After the application of Relax process on traditional requirements, we obtain invariant and Relax-ed requirements. Relax-ed requirements support a high degree of flexibility that goes well beyond the original requirements. Once the requirements engineer determines that indeed a level of flexibility can be tolerated, then the downstream developers, including the designers and programmers have the flexibility to incorporate the most suitable adaptive mechanisms to support the desired functionality. These decisions may be made at design time and/or run time [10, 21].

2.2. SysML/KAOS

The SysML/Kaos [22] model is an extension of the SysML[1] requirements model with concepts of the Kaos goal model [27]. SysML is an extension of Uml[2], so it provides concepts to represent requirements and to relate them to other model elements, allowing the definition of traceability links between re-

[1] http://www.omgsysml.org/

[2] http://www.omg.org/spec/UML/

quirements and system models. The SysML/Kaos meta-model is implemented as a new profile, importing the SysML profile.

2.2.1. SysML

SysML is a general purpose modeling language for systems engineering applications. SysML is a Uml profile that represents a subset of Uml 2.0 with extensions. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. These systems may include hardware, software, information, processes, personnel, and facilities. In particular, the language provides graphical representations with a semantic foundation for modeling system requirements, behavior, structure, and parametrics, which is used to integrate with other engineering analysis models.

SysML includes a graphical construct to represent text based requirements and relate them to other model elements. The requirements diagram captures requirements hierarchies and requirements derivation, and the <<***satisfy***>> and <<***verify***>> relationships allow a modeler to relate a requirement to a model element, e.g. <<***block***>>, that satisfies or verifies the requirements. The requirement diagram provides a bridge between typical requirements management tools and system models.

2.2.2. KAOS

Kaos is a methodology for RE enabling analysts to build requirements models and to derive requirements documents from Kaos models. The first key idea behind Kaos is to build a model for the requirements, i.e. for describing the problem to be solved and the constraints that must be fulfilled by any solution provider. Kaos has been designed : *(i)* To fit problem descriptions by allowing to define and manipulate concepts relevant to problem description ; *(ii)* To improve the problem analysis process by providing a systematic approach for discovering and structuring requirements ; *(iii)* To clarify the responsibilities of all the project stakeholders ; *(iv)* To let the stakeholders communicate easily and efficiently about the requirements

2.2.3. Why SysML/KAOS?

SysML and Kaos have some advantages and weak points, but these are complementary to each other based on the following points : *(i)* Requirements description: A textual description in SysML and a description in the form of goals in Kaos ; *(ii)* Relation between requirements: SysML has <<***contain***>> and <<***derive***>> relations; these relations do not have precise semantics which leads to confusion. Kaos has refinement relations AND/OR ; *(iii)* Traceability relations: <<***satisfy***>> and <<***verify***>> relations in SysML allow to define traceability. Kaos does not have explicit relations ; *(iv)* Tools: A number of tools exist for SysML; most of them are open source. Kaos propose a proprietary tool called Objectiver[3].

Traditionally, requirements are divided into Functional Requirements (FRs) and Non Functional Requirements (NFRs). Due to the complexity of systems, NFRs should be processed much earlier; at the same level of abstraction as FRs which will allow taking into account these properties for the evaluation of

[3] http://www.objectiver.com/

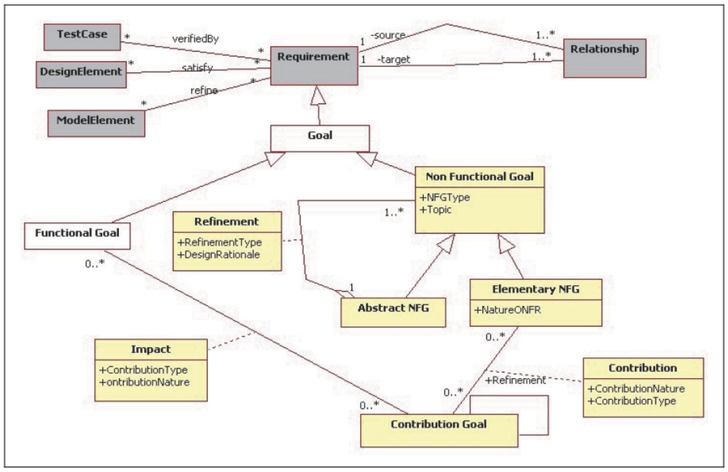


Figure 4: SysML/Kaos Meta Model [22]

alternate options, risk and conflict analysis. The benefit of SysML is that it allows throughout the development cycle to relate requirements to other model elements, thus ensuring continuity from the requirements phase to the implementation phase. However, the proposed concepts of requirements in SysML are not as rich as in the other RE methods (especially GORE). SysML/Kaos is the result of motivation to benefit from the contributions of SysML, while ensuring a more precise definition of the concepts. SysML/Kaos is inpired from the work of [13] and [18]. The SysML/Kaos model allows both FRs [26] and NFRs [22] to be modeled.

2.2.4. SysML/KAOS Meta-Model

Figure 4 shows the extended meta-model of SysML/Kaos; non functional concepts are represented as yellow boxes, the gray boxes represent the SysML concepts. The instantiation of the meta-model allows us to obtain a hierarchy of NFRs in the form of goals. Non Functional Goals (NFGs) are organized in refinement hierarchies. The meta-class *Non Functional Goal* represents the Non Functional Goal (NFG), it is specified as a sub-class of the meta-class *Goal* which itself is a sub-class of the meta-class *Requirement* of SysML. An NFG represents a quality that the future system must have in order to satisfy a Functional Requirement (FR). The *NFGTYPE* specifies the type of NFG and the attribute *TOPIC* represents the domain concept concerned by this type of requirement. An NFG can thus be represented with the following syntax: *NFGType [Topic]*. An NFG is either an *Abstract NFG* or an *Elementary NFG*. A goal that cannot be further refined is an *Elementary NFG*. The refinement of an *Abstract NFG* by either abstract or elementary goals is represented by the *Association Class Refinement*. An *Abstract NFG* may contain several combinations of sub goals (abstract or elementary). The relationship *Refinement* becomes an *Association Class* between an *Abstract NFG* and its sub goals. It can be specialized to represent And/Or goal refinements. At the end of the refinement process, it is necessary to identify and express the various alternative ways to satisfy the *Elementary NFGs*. For that, the SysML/Kaos meta-model considers the concept of the meta-class *Contribution Goal*. A *Contribution Goal* captures a possible way to satisfy an *Elementary NFG*. The *Association Class Contribution* describes the characteristics of the contribution. It provides two properties: *ContributionNature* and *ContributionType*. The first one specifies whether the contribution is *positive* or *negative*, whereas the second one specifies whether the contribution is *direct* or *indirect*. A *positive* (or *negative*) contribution helps positively (or negatively) to the satisfaction of an *Elementary NFG*. A *direct contribution* describes an explicit contribution to the *Elementary NFG*. An *indirect contribution* describes a kind of contribution that is a direct contribution to a given goal but induces an unexpected contribution to another goal. Finally, the concept of *Impact* is used to connect NFGs to Functional Goals (FGs). It captures the fact that a *Contribution Goal* has an effect on FGs.

2.3. The OMEGA2 UML/SysML Profile and IFx Toolset

Formal methods provide tools to verify the consistency and correctness of a specification with respect to the desired properties of the system. For this reason, we use these methods to prove some of the properties of the system before the system development even starts. We use OMEGA2/IFx profile and toolset for the properties verification and model simulation of our case study.

2.3.1. The OMEGA2 Profile

OMEGA2 profile [32] is an executable UML/SysML profile used for the formal specification and validation of critical real-time systems. It is based on a subset of UML 2.2/SysML 1.1 containing the main constructs for defining the system structure and behavior.

The OMEGA2 UML/SysML profile defines the semantics of UML/SysML elements providing the means to model coherent and unambiguous system models. In order to make the models verifiable, it presents as extension the *observers* mechanism for specifying dynamic properties of models. The OMEGA2 UML/SysML Profile is implemented by the IFx toolbox which provides static analysis, simulation and timed automaton based model checking [14] techniques for validation.

The architecture of an OMEGA2 model is described in Class/Block Definition Diagrams by classes/blocks with their relationships. Each class/block defines properties and operations, as well as a state machine. The hierarchical structure of a model is defined in composite structures/Internal Block Diagram (IBD): parts that communicate through ports and connectors. For the SysML block diagram (Block Definition Diagram (BDD)), the following concepts are taken into account: blocks and their relationships (association, aggregation, generalization), the interfaces, the basic types, signals.

For the system behavior, the OMEGA2 profile takes into account the following concepts: State machines (excluding: history states, entry point, exit point, junction) and Actions; for this, the profile defines a concrete syntax. This syntax is used for example to define operation bodies and transition effects in state machines. The textual action language is compatible with the UML 2.2 action meta-model and implements its main elements: object creation and destruction, operation calls, expression evaluation, variable assignment, signal output, return action as well as control flow structuring statements.

For specifying and verifying dynamic properties of models, OMEGA2 uses the notion of *observers*. *Observers* are special classes/blocks monitoring runtime state and events. They are defined by classes/blocks stereotyped with <<*observer*>>. They may have local memory (attributes) and a state machine describes their behavior. States are classified as <<*success*>> and <<*error*>> states to express the (non)satisfaction of safety properties. The main issue in modeling *observers* is the choice of events which trigger their transitions.

The trigger of an *observer* transition is a match clause specifying the type of event (e.g., receive), some related information (e.g., the operation name) and observer variables that may receive related information (e.g., variables receiving the values of operation call parameters). Besides events, an observer may access any part of the state of the Uml model: object attributes and state, signal queues.

2.3.2. IFx Toolset

OMEGA2 models can be simulated and properties can be verified using the IFx toolset [11]. The IFx toolset provides verification which ensures the automatic process of verifying whether an OMEGA2 Uml/SysML model satisfies (some of) the properties (i.e. *observers*) defined on it. The verification method employed in IFx is based on systematic exploration of the system state space (i.e., enumerative model checking). The IFx toolset also provides simulation which designates the interactive execution of an OMEGA2 Uml/SysML model. The execution can be performed step-by-step, random, or guided by a simulation scenario (for example an error scenario generated during a verification activity).

The IFx toolset relies on a translation of Uml/SysML models towards a simple specification language based on an asynchronous composition of extended timed automata, the IF language[4], and on the use of simulation and verification tools available for IF. The translation takes an input model in XML Metadata Interchange (XMI) 2.0 format. The compiler verifies the set of well-formedness rules imposed by the profile and generates an IF model that can be further reduced by static analysis techniques. This model is subject to verification that either validates the model with respect to its properties or produces a list of error scenarios that can be further debugged using the simulator. The OMEGA2/IFx approach has been applied for the verification and validation of industry grade models [20] providing interesting results.

3. State of the Art

Different road map papers on Software Engineering (SE) for SAS [21, 19] discuss the state of the art, its limitations, and identify critical challenges. [21] presents a research road map for SE of SAS focusing on four views, which are identified as essential: requirements, modeling, engineering, and assurances. The focus is on development methods, techniques, and tools that seem to be required to support the systematic development of complex software systems with dynamic self adaptive behavior. The most recent road map paper [19]

[4] http://www-if.imag.fr/

discusses four essential topics of self-adaptation: design space for self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation for SAS.

3.1. Requirements Engineering for Self Adaptive Systems

An SAS is able to modify its behavior according to changes in its environment. As such, an SAS must continuously monitor changes in its context and react accordingly. But here the question arises as to what aspects of the environment the SAS should monitor. Clearly, the system cannot monitor everything and exactly what should the system do if it detects a less than optimal pattern in the environment? Presumably, the system still needs to maintain a set of high level goals that should be maintained regardless of the environmental conditions. But non critical goals could well be Relax-ed, thus allowing the system a degree of flexibility during or after adaptation. It is important to identify these properties as early as possible.

Levels of Requirement Engineering for Modeling (LoREM) [23] is an approach for modeling the requirements of Dynamic Adaptive Systems (DAS) using i* goal models [42]. The i* goal models are used to represent the stakeholder objectives, non-adaptive system behavior (business logic), adaptive behavior, and adaptation mechanism needs of DAS. Each of these i* goal models addresses the three RE concerns (conditions to monitor, decision-making procedure, and possible adaptations) from a specific developers perspective.

Awareness Requirements (AwReqs) [35] are requirements that talk about the success or failure of other requirements. More generally, AwReqs talk about the states requirements can assume during their execution at run-time. AwReqs are represented in a formal language and can be directly monitored by a requirements monitoring framework.

Claims [39, 38] were applied as markers of uncertainty to record the rationale for a decision made with incomplete information in DASs. The work in [33] integrates Relax and Claims to assess the validity of Claims at runtime while tolerating minor and unanticipated environmental conditions that can otherwise trigger adaptations.

Relax can be used in goal oriented modeling approaches for specifying and mitigating sources of uncertainty in DASs [12]. AutoRELAX [34], is an approach that generates Relax-ed goal models that address environmental uncertainty by identifying which goals to Relax, which Relax operators to apply, and the shape of the fuzzy logic function that defines the goal satisfaction criteria. AutoRELAX also requires an executable specification of the DAS, such as a simulation or a prototype, which applies the set of utility functions to measure how well the DAS satisfies its requirements in response to adverse conditions. For the experimental setup of AutoRELAX, a null hypothesis is defined which states that there is no difference between a Relax-ed and an unRelax-ed goal model.

Fuzzy Live Adaptive Goals for Self-Adaptive Systems (FLAGS) [7] is an innovative goal model which deals with the challenges posed by SAS. Goal models have been used for representing systems requirements, and also for tracing them onto their underlying operationalization.

The state of the art regarding RE for SAS shows different approaches from the point of view of its complementarity with Relax. The different steps in

LoREM are interesting but our focus is on Relax-ed requirements as we want to identify the uncertainty in the requirements of DASs. Regarding AwReqs, in future work, we want to integrate this concept into our approach using Monitor-Analyze-Plan-Execute (MAPE) [25] feedback loop that operationalizes the system's adaptability mechanisms. Claims are also subject to uncertainty, in the form of unanticipated environmental conditions and unreliable monitoring information, that can adversely affect the behavior of the DAS if it spuriously falsifies a claim. A Claim can also be monitored at run time to prove or disprove its validity [39], thereby triggering adaptation to reach more desirable system configurations if necessary. Claims therefore complement Relax.

3.2. Properties Verification of SAS

For the properties verification of SAS, we use the OMEGA2/IFx profile and toolset which was developed in our team. The advantage of the OMEGA2 profile is that it provides the notion of *observers* for specifying and verifying dynamic properties of models. In terms of properties verification, there exists a number of techniques. In the following, we give a description of some of it.

[9] presents a verification approach based on MEDISTAM-RT, which is a methodological framework for the design and analysis of real-time systems and timed traces semantics, to check the fulfillment of NFRs. It only focuses on safety and timeliness properties, to assure the correct functioning of Ambient Assisted Living (AAL) systems and to show the applicability of this methodology in the context of this kind of system.

[5] introduces a profile named Timed UML and RT-LOTOS Environment (TURTLE) which extends the Uml class and activity diagrams with composition and temporal operators. TURTLE is a real-time Uml profile with a formal semantics expressed in Real Time Language Of Temporal Ordering Specifications (RTLOTOS) [17]. With its formal semantics and toolkit, TURTLE enables a priori detection of design errors through a combination of simulation and verification/validation techniques.

In [26], the authors propose an extension to SysML with concepts from the goal model of the Kaos method (SysML/Kaos) with rules to derive a formal B [1] specification from this goal model. The B formal method is a complete method that supports a large segment of the software development life cycle: specification, refinement and implementation.

In MEDISTAM-RT, the focus is on safety and timeliness properties, we do not treat any specific type of properties. We verify those requirements that are of interest for adaptation in SAS. In TURTLE, design errors can be detected through simulation and verification, that's the reason why we plan to explore the complementarity of this approach with our approach. The use of formal methods like B can help avoid the state space explosion problem which is inherent in model checking techniques. We have worked on studying the complementarity of these two approaches and we plan to integrate it in our approach in the future work.

4. Proposed Approach

In this section, we introduce the overall view of our proposed approach [2]. We show our contribution then we describe the overall process of our approach.

To illustrate our proposed approach, we use requirements from the barbados Car Crash Crisis Management System (bCMS) case study. At the end, we show the integrated tooling environment that we developed to validate our approach.

4.1. Contribution

To properly define the scope of our contribution , it is necessary to identify the work we have done. Firstly, we have found that although the use of traditional process of SysML/Kaos was interesting for modeling the requirements of SAS, it does not take into account the notion of uncertainty. On the other hand, Relax is a process adapted to identify and highlight the uncertainty, but it does not provide tools for its implementation. Finally, the verification techniques used for these models does not take into account the uncertainty posed by these systems. Based on this observation, we contributed towards the definition of an integrated tool based process. For this, we developed support for Relax. Then we developed rules to transform requirements addressed by Relax to SysML/Kaos, using model transformation techniques. Finally, we integrated formal verification techniques i.e. OMEGA2/IFx in the process. To reduce the risk of state space explosion problem [15] when we take into account the whole system using OMEGA2/IFx, we limited its use to verify only adaptable properties. We present in detail the work and the overall process in the next section.

4.2. The Proposed Approach

In the following, each step of the proposed approach is explained with associated input and output. Figure 5 shows the overall view of our proposed approach.

1. The overall approach that we proposed takes requirements as input. These requirements are elicited in the form of *SHALL* statements by a requirement engineer which are then divided into FRs and NFRs.
2. We apply Relax process (section 2.1.3) on these FRs and NFRs to get those requirements that are associated with the adaptability features of SAS called Relax-ed requirements and those that are fixed called invariant requirements.
3. The resulting Relax-ed requirements are then formalized using an editor that we developed called Relax COOL editor. This editor takes into account the uncertainty factors associated with each Relax-ed requirement. Xtext[5] is used for the development of this editor.
4. At this point, we use a process for the conversion of Relax-ed requirements into goal concepts i.e. SysML/Kaos. We use a correlation table (section 4.3.1) for the correspondence between Relax-ed requirements and SysML/Kaos concepts [4]. For this purpose, we have developed a tool called Relax2SysML/Kaos editor, which is based on Atlas Transformation Language (ATL) transformations. For the time being, the tool helps in mapping the Relax concepts to SysML/Kaos concepts but not the inverse.

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

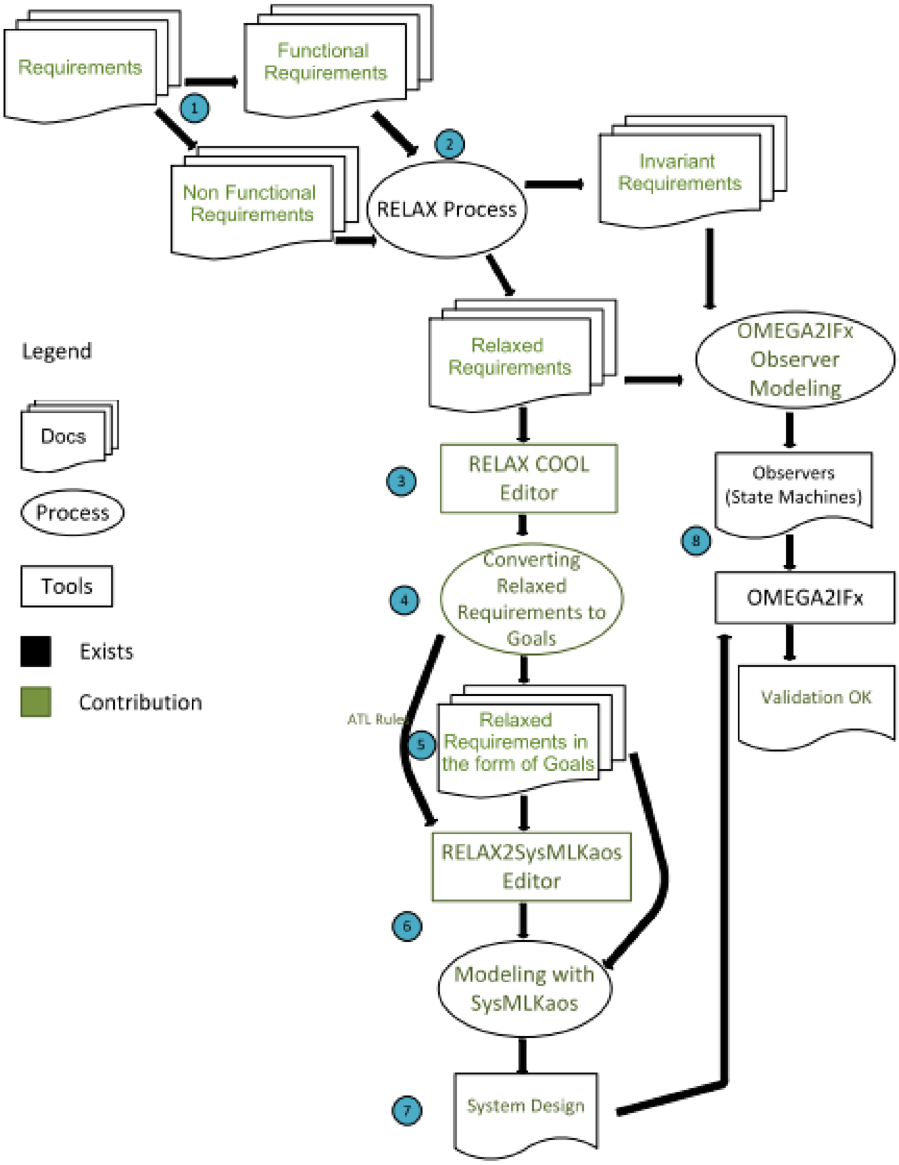


Figure 5: Overall View of Our Approach

5. At this step, we have a full list of Relax-ed requirements with uncertainty factors converted into SysML/Kaos goal concepts.
6. The non functional Relax-ed requirements in the form of SysML/Kaos goal concepts can now be modeled with the help of SysML/Kaos editor.
7. This step shows the system design. The Relax-ed requirements of the SAS are now modeled and we have a snapshot of the system design.
8. Once we have the system design, we use the OMEGA2/IFx *observers* to

verify the properties of SAS. The input to this step are the OMEGA2/IFx *observers* which are the RELAX-ed and invariant requirements. The verification either results in the fulfillment of all the properties or if there is an error produced during verification, it can be simulated through the interactive simulation interface of the IFx toolset in order to identify the source of the error and then subsequently correct it in the model.

4.3. Integration of the Approaches

Here we present how we defined the convergence between different methods used in our approach.

4.3.1. Relationship b/w RELAX, SysML/KAOS and SysML

In our integrated approach, we take benefit of SYSML/KAOS while modeling RELAX-ed requirements of SAS. In Figure 6, we show how several key concepts are taken into account in the selected approaches. Most of the time, the concepts are not fully covered (e.g. <<***satisfy***>>) for monitoring in SYSML, this stereotype is used between a block and a requirement), but we have indicated in the Figure 6 the closest mechanism that supports the concepts. The concepts are taken from RELAX and are then compared with the other approaches.

- In SYSML/KAOS, requirements are described in the form of goals; SYSML describes requirements in textual form; RELAX requirements are also in textual form with an enhanced version.

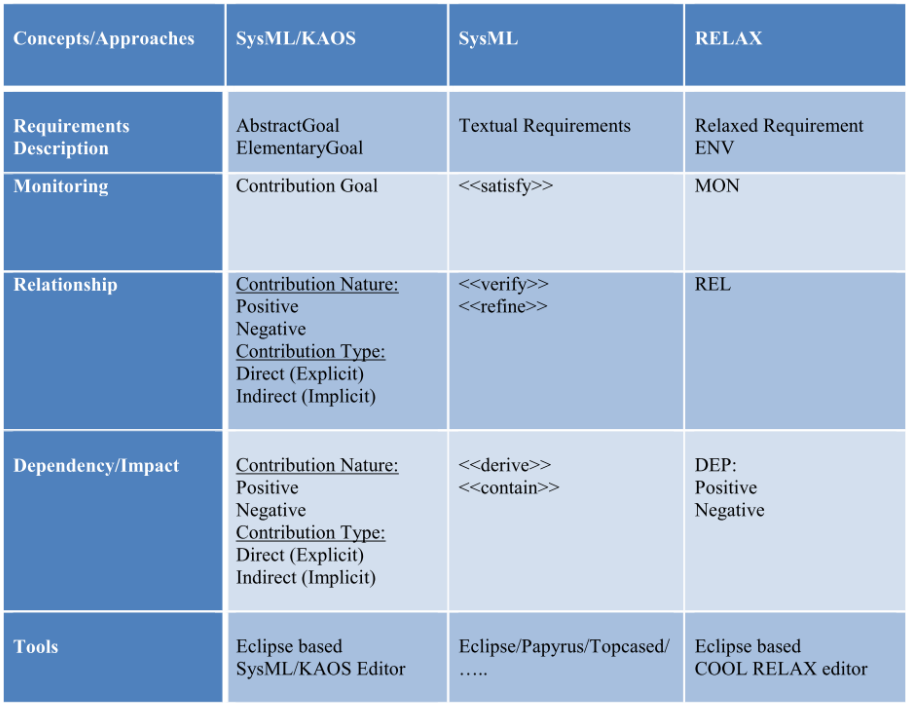
- To deal with monitoring, SYSML/KAOS has the *Contribution Goal* concept which is used to satisfy an *Elementary NFG*, SYSML has <<***satisfy***>> which is used when a <<***block***>> satisfies a <<***requirement***>> while for RELAX, we have the concept of *MON* which is used to measure the environment, i.e. *ENV*.

- SYSML/KAOS has the concept of *Contribution* which is an *Association Class* between *Contribution Goal* and *Elementary NFG*. *Contribution* describes the characteristics of the contribution. It provides two properties: *ContributionNature* and *ContributionType*. SYSML has <<***verify***>> and <<***refine***>> relationships while for RELAX, we have *REL* variable which identifies the relationship between *ENV* and *MON* or more precisely how *MON* achieves *ENV*.

- For Dependency/Impact, SYSML/KAOS describes it as an *Impact* of a *Contribution Goal* on a Functional Goal (FG). It also has the same two properties, i.e. *ContributionNature* and *ContributionType*. This impact can be *positive* or *negative* and *direct* or *indirect*. In SYSML, we have the concept of <<***derive***>> which shows the dependency between requirements, RELAX has *positive* and *negative* dependency which shows the dependency of a RELAX-ed requirement on other requirements.

- For the tools available for each approach, SYSML/KAOS has a tool called SYSML/KAOS editor, SYSML has a number of tools e.g. eclipse[6], pa-

[6] `http://www.eclipse.org/`



Concepts/Approaches	SysML/KAOS	SysML	RELAX
Requirements Description	AbstractGoal ElementaryGoal	Textual Requirements	Relaxed Requirement ENV
Monitoring	Contribution Goal	<<satisfy>>	MON
Relationship	Contribution Nature: Positive Negative Contribution Type: Direct (Explicit) Indirect (Implicit)	<<verify>> <<refine>>	REL
Dependency/Impact	Contribution Nature: Positive Negative Contribution Type: Direct (Explicit) Indirect (Implicit)	<<derive>> <<contain>>	DEP: Positive Negative
Tools	Eclipse based SysML/KAOS Editor	Eclipse/Papyrus/Topcased/ …..	Eclipse based COOL RELAX editor

Figure 6: Relationship b/w SysML/Kaos SysML and Relax

pyrus[7], topcased[8] etc. and for Relax, we have developed eclipse based Relax COOL editor [8]. We have also developed Relax2SysML/Kaos editor which does the mapping between Relax uncertainty factors and SysML/Kaos goal concepts.

4.3.2. Uncertainty Factors/Impacts

Relax Uncertainty factors, especially *ENV* and *MON*, are particularly important for documenting whether the system has means for monitoring the important aspects of the environment. By collecting these *ENV* and *MON* attributes, we can build up a model of the environment in which the system will operate, as well as a model of how the system monitors its environment. Having said this, SysML/Kaos can complement Relax by injecting more information in the form of *positive/negative* and *direct/indirect* impacts. The grammar of Relax acts as a meta-model for our Relax COOL editor, while SysML/Kaos has extended the meta-model of SysML with goal concept. As both meta-models are close to the SysML meta-model, we have bridged Relax and SysML/Kaos using our proposed approach.

[7] http://www.papyrusuml.org

[8] http://www.topcased.org/

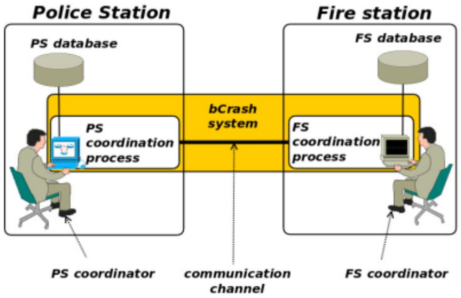


Figure 7: bCMS Case Study Overall View

4.3.3. Verification of Ambient System's Properties through Formal Methods

Using our proposed approach, we provide a strong consistency between models. This can be ensured thanks to the use of formal methods that provide verification tools for the properties verification and model simulation of SAS. We have integrated OMEGA2/IFx for properties verification and model simulation of these systems in our proposed approach. By doing this, we bridge the gap between the requirements phase and the initial formal specification phase.

4.4. Proposed Approach Illustration

To illustrate our approach, we use the bCMS[9] case study. Here is an excerpt of the case study.

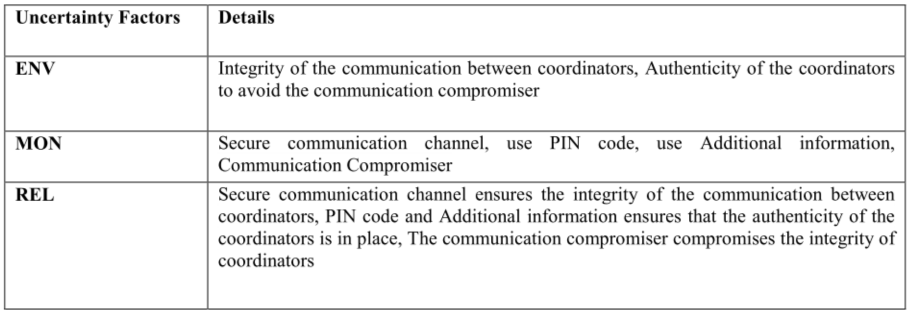
The bCMS is a distributed crash management system that is responsible for coordinating the communication between a Fire Station Coordinator (FSC) and a Police Station Coordinator (PSC) to handle a crisis in a timely manner. Information regarding the crisis as it pertains to the tasks of the coordinators is updated and maintained during and after the crisis. There are two collaborative sub-systems. Thus, the global coordination is the result of the parallel composition of the (software) coordination processes controlled by the two (human) distributed coordinators. There is no central database; fire and police stations maintain separate databases and may only access information from the other database through the bCMS system. Each coordination process is hence in charge of adding and updating information in its respective database. Figure 7 shows the overall view of the bCMS case study.

We have chosen an (illustrative) subset of the bCMS requirements. The requirements are shared and numbered in a shared document[10].

We have first applied the RELAX process on bCMS requirements to get invariant and RELAX-ed requirements. For RELAX-ed requirements, all the uncertainty factors were identified. Then using the correlation in Figure 6, we have modeled the bCMS system requirements with the SYSML/KAOS approach. In [3], we have modeled some more requirements of the bCMS case study. Following are the invariant and RELAX-ed requirements that were identified:

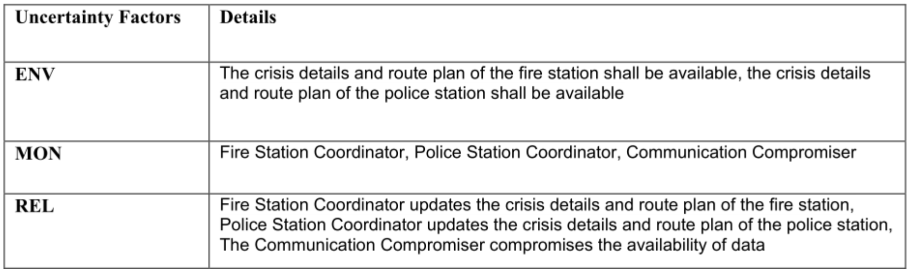
- Invariant requirements: R1, R2, R3.

[9] Available at http://cserg0.site.uottawa.ca/cma2013re/CaseStudy.pdf
[10] Available at http://goo.gl/uscP5



Uncertainty Factors	Details
ENV	Integrity of the communication between coordinators, Authenticity of the coordinators to avoid the communication compromiser
MON	Secure communication channel, use PIN code, use Additional information, Communication Compromiser
REL	Secure communication channel ensures the integrity of the communication between coordinators, PIN code and Additional information ensures that the authenticity of the coordinators is in place, The communication compromiser compromises the integrity of coordinators

Figure 8: Integrity RELAX-ed Requirement Uncertainty Factors



Uncertainty Factors	Details
ENV	The crisis details and route plan of the fire station shall be available, the crisis details and route plan of the police station shall be available
MON	Fire Station Coordinator, Police Station Coordinator, Communication Compromiser
REL	Fire Station Coordinator updates the crisis details and route plan of the fire station, Police Station Coordinator updates the crisis details and route plan of the police station, The Communication Compromiser compromises the availability of data

Figure 9: Availability RELAX-ed Requirement Uncertainty Factors

- Relax-ed Requirements: R4, R8.

Figure 8 shows the uncertainty factors associated with the *Integrity* R4 (*The system shall ensure that the integrity of the communication between coordinators regarding crisis location, vehicle number, and vehicle location is preserved AS CLOSE AS POSSIBLE TO 99.99% of the time.*) Relax-ed requirement.

Figure 9 shows the uncertainty factors associated with the *Availability* R8 (*The crisis details and route plan of the fire station and the police station shall be available with the exception of AS CLOSE AS POSSIBLE To 30 minutes for every 48 hours when no crisis is active.*) Relax-ed requirement.

Figure 10 shows the high level goal model of the bCMS case study. The goal at the highest level is identified as *Security[bCMS System]*, which is an *Abstract NFG* and is AND-refined into two sub goals using refinement by type: *Integrity[bCMS System]* and *Availability[bCMS System]*. The goal *Availability[bCMS System]* is an *Abstract NFG* and is AND-refined until we reach the *Elementary Goals*. The goal *"The crisis details and route plan of the fire station shall be available[bCMS System]"* is satisfied by the *Contribution Goal Fire Station Coordinator* having a *direct* and *positive* contribution on it and the goal *"The crisis details and route plan of the police station shall be available[bCMS System]"* is satisfied by the *Contribution Goal Police Station Coordinator* which also has a *direct* and *positive* contribution on its accomplishment. The *Contribution Goal Communication Compromiser* contributes directly and negatively towards the satisfaction of the two *Elementary Goals* whose objective is to disrupt the response to the crisis for some personal gain.

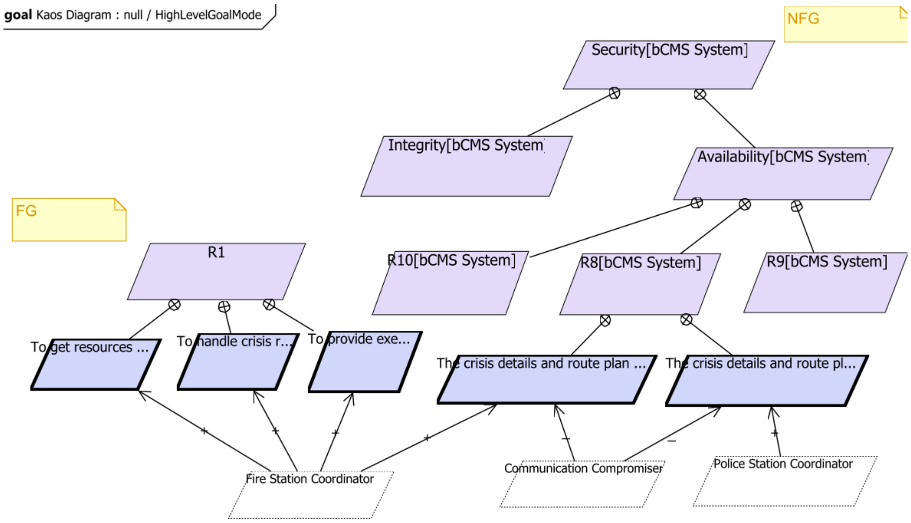
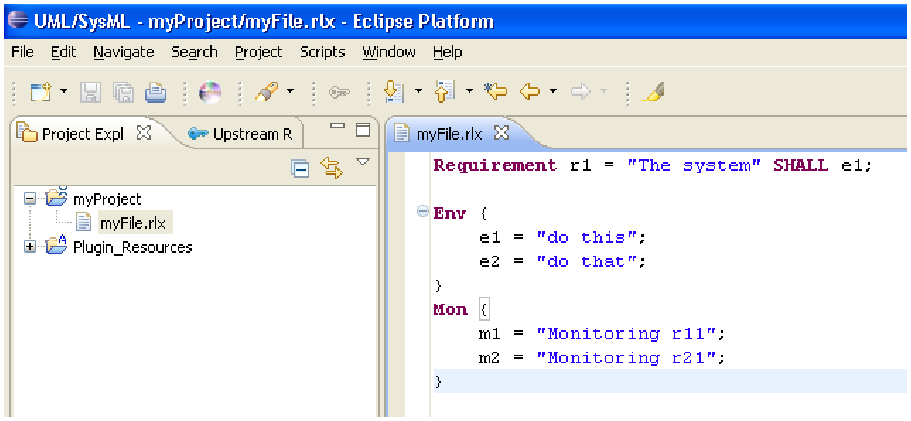


Figure 10: High level goal model



```
Requirement r1 = "The system" SHALL e1;

Env {
    e1 = "do this";
    e2 = "do that";
}
Mon {
    m1 = "Monitoring r11";
    m2 = "Monitoring r21";
}
```

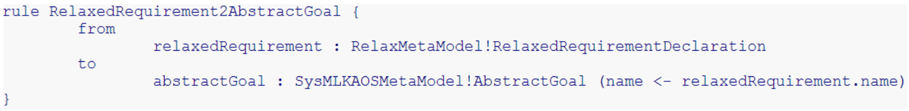
Figure 11: RELAX File

4.5. Tools Support

In this section, we introduce the tools that implements our proposed approach.

4.5.1. RELAX Editor

For the generation of RELAX editor, Xtext is used. Xtext is a framework for the development of Domain Specific Languages (DSL) and other textual programming languages and helps in the development of an Integrated Development Environment (IDE) for the DSL. Some of the IDE features that are either derived from the grammar or easily implementable are: syntax coloring, model navigation, code completion, outline view, and code templates. The RELAX grammar is used as a meta-model for this editor which is generated by



```
rule RelaxedRequirement2AbstractGoal {
        from
                relaxedRequirement : RelaxMetaModel!RelaxedRequirementDeclaration
        to
                abstractGoal : SysMLKAOSMetaModel!AbstractGoal (name <- relaxedRequirement.name)
}
```

Figure 12: Relaxed Requirement to Abstract Goal Mapping

Xtext that we call Relax.ecore. Figure 11 shows an example of the Relax file with uncertainty factors. The Relax file is represented with an extension *.rlx*. Once we have the *.rlx* file, we can transform it into an XMI model. The XMI model can then be manipulated and will serve us for the model transformation from Relax to SysML/Kaos as explained in the next section.

4.5.2. RELAX to SysML/Kaos Transformation

In our approach, we want to transform Relax-ed requirements uncertainty factors into SysML/Kaos goal concepts. This transformation will help in taking into account the adaptability features associated with SAS in the form of uncertainty factors of Relax-ed requirements and then modeling these requirements in SysML/Kaos. In this way, we can benefit from the advantages offered by GORE. For this purpose, the Relax and SysML/Kaos meta-models are used.

ATL Rules

ATL is a model transformation language and toolkit. It provides a way to produce a number of target models from a set of source models. An ATL transformation program is composed of rules that define how source model elements are matched and navigated to create and initialize the elements of the target models. The generation of target model elements is achieved through the specification of transformation rules.

Mapping between RELAX and SysML/Kaos Elements

Here, we present the relationship between Relax and SysML/Kaos elements. The Relax abstract syntax is defined in the Relax meta-model. In turn, the SysML/Kaos abstract syntax is defined in the SysML/Kaos meta-model.

Figure 6 shows the mapping between the two concepts. For the ATL transformation rules, a Relax-ed requirement is mapped to an *Abstract Goal* as shown in Figure 12, an *ENV* is mapped to an *Elementary Goal* and *MON* is mapped to *Contribution Goal*. Figure 13 shows the generated SysML/Kaos model after the application of ATL rules. Figure 14 shows the SysML/Kaos model opened in the editor.

5. Proof of Concepts

In this section, we apply our approach on an academic Ambient Assisted Living (AAL) case study. The goal of AAL solutions is to apply ambient intelligence technology to enable people with specific demands, e.g. handicapped or elderly, to live in their preferred environment [9]. In order to achieve this goal, different kinds of AAL systems can be proposed and most of them pose

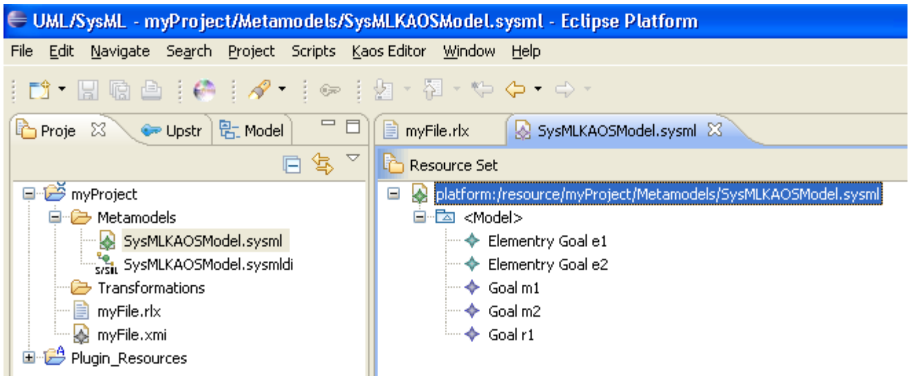
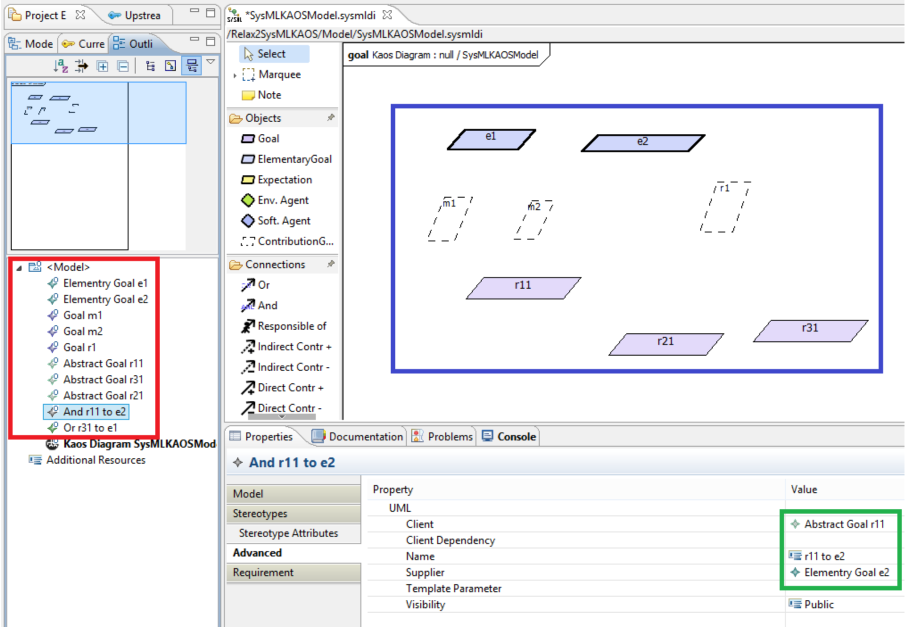


Figure 13: SysML/Kaos Model



Property	Value
UML	
Client	Abstract Goal r11
Client Dependency	
Name	r11 to e2
Supplier	Elementry Goal e2
Template Parameter	
Visibility	Public

Figure 14: Generated SysML/Kaos Model using ATL Transformations

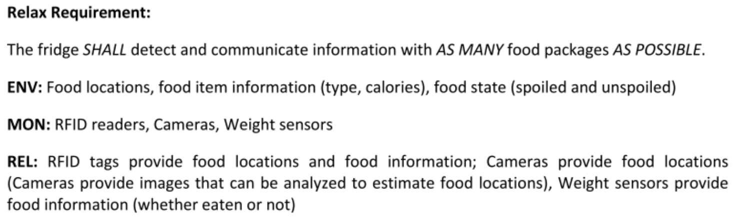
reliability issues and describe important constraints upon the development of software systems [16]. We model the requirements of an AAL[11] home which ensures the health of a *Patient* like the one studied by research teams at the IUT of Blagnac[12]. We then show the verification of some of the properties of the AAL system.

[11] http://www.iese.fraunhofer.de/fhg/iese/projects/med_projects/aal--lab/index.jsp

[12] http://mi.iut-blagnac.fr/

Mary is a widow. She is 65 years old, overweight and has high blood pressure and cholesterol levels. Mary gets a new intelligent fridge. It comes with 4 temperature and 2 humidity sensors and is able to read, store, and communicate RFID information on food packages. The fridge communicates with the Ambient Assisted Living (AAL) system in the house and integrates itself. In particular, it detects the presence of spoiled food and discovers and receives a diet plan to be monitored based on what food items Mary is consuming.
An important part of Mary's diet is to ensure minimum liquid intake. The intelligent fridge partially contributes to it. To improve the accuracy, special sensor-enabled cups are used: some have sensors that beep when fluid intake is necessary and have a level to monitor the fluid consumed; others additionally have a gyro detecting spillage. They seamlessly coordinate in order to estimate the amount of liquid taken: the latter informs the former about spillages so that it can update the water intake level. However, Mary sometimes uses the cup to water flowers. Sensors in the faucets and in the toilet also provide a means to monitor this measurement.

Figure 15: AAL Case Study



Relax Requirement:

The fridge *SHALL* detect and communicate information with *AS MANY* food packages *AS POSSIBLE*.

ENV: Food locations, food item information (type, calories), food state (spoiled and unspoiled)

MON: RFID readers, Cameras, Weight sensors

REL: RFID tags provide food locations and food information; Cameras provide food locations (Cameras provide images that can be analyzed to estimate food locations), Weight sensors provide food information (whether eaten or not)

Figure 16: RELAX Requirement Example

5.1. Requirements Modeling of the AAL Case Study

Figure 15 shows an excerpt of the case study which highlights the need to ensure *Patient*'s health in the AAL home. Advanced smart homes, such as Mary's AAL, rely on adaptivity to work properly. For example, the sensor-enabled cups may fail, but since maintaining a minimum of liquid intake is a life-critical feature, the AAL should be able to respond by achieving this requirement in some other way [41].

Figure 16 shows an example of Relax-ed requirement from the Mary's AAL home, which results from the application of the Relax process on the traditional requirement: *The Fridge shall read, store and communicate RFID information on food packages.* [30] shows the application of Relax process on some of the requirements of the AAL case study.

5.1.1. High Level Goal Model

Figure 17 shows the high level goal model of the AAL. From the AAL system problem statement, we have identified *Reliability [AAL system]* as a non functional high level goal. In fact, one of the expected qualities of the system is to run reliably. This is very important for several reasons and particularly because

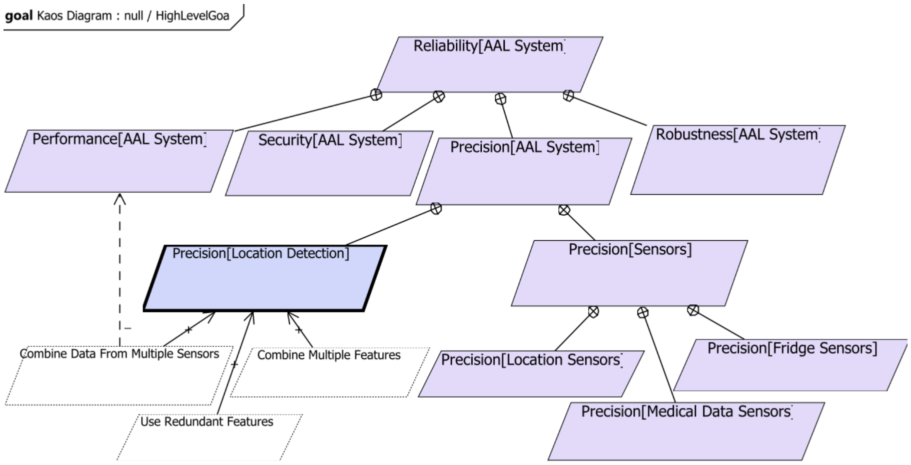

goal Kaos Diagram : null / HighLevelGoa
Reliability[AAL System
Performance[AAL System]
Security[AAL System]
Precision[AAL System]
Robustness[AAL System
Precision[Location Detection]
Precision[Sensors]
Combine Data From Multiple Sensors
Combine Multiple Features
Use Redundant Features
Precision[Location Sensors
Precision[Fridge Sensors]
Precision[Medical Data Sensors
+
+
+
–

Figure 17: High Level Goal Model

frequent visits from a technician could be a factor of disturbance for *Mary* and unfeasible due to the large number of AAL houses across the world. The high level goal *Reliability [AAL System]* is AND-refined into four sub goals using refinement by type: *Precision [AAL System]*, *Security [AAL System]*, *Robustness [AAL System]* and *Performance [AAL System]*. Each sub goal can be further refined until the refinement stops and we reach an *Elementary Goal* which can then be assigned to a *Contribution Goal*. The sub goal *Precision [AAL System]* is AND-refined into two sub goals: *Precision [Location Detection]* and *Precision [Sensors]* using refinement by subject. The sub goal *Precision [Sensors]* is then AND-refined into three *Elementary NFGs* using refinement by subject. The sub goal *Precision [Location Detection]* can be satisfied by a *positive* and *direct* contribution by one of the following *Contribution Goals*: *combine data from multiple sensors*, *combine multiple features* and *use redundant features*. The *Contribution Goal combine data from multiple sensors*, contribute indirectly and negatively to the satisfaction of the sub goal *Performance [AAL System]*.

5.1.2. Low Level Goal Model

Figure 18 shows the security goal model of AAL. In order to further extract new goals from the AAL system, we identify another goal, *Security [fridge data]*, which is an *Abstract NFG* that can be AND-refined into three sub goals using refinement by type: *Confidentiality [fridge data]*, *Integrity [fridge data]* and *Availability [fridge data]*. Similarly, the sub goal *Availability [fridge data]* can be refined into two sub goals using refinement by subject: *Availability [Storing RFID information]* and *Availability [Sensors data]*. The *Contribution Goal having high-end sensors* contributes directly and positively to the goal *Availability [Sensors data]*, and may contribute indirectly and positively to *Integrity [fridge data]*.

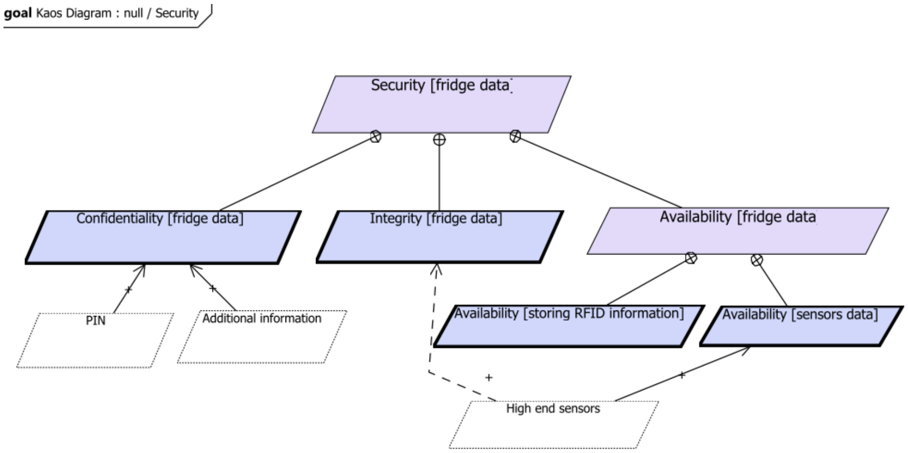

goal Kaos Diagram : null / Security
Security [fridge data
Confidentiality [fridge data]
Integrity [fridge data]
Availability [fridge data
Availability [storing RFID information]
Availability [sensors data]
PIN
Additional information
High end sensors
+
+
+
+

Figure 18: Security Goal Model

5.2. Properties Verification of the AAL system with OMEGA2/IFx Profile and Toolset

The specification and verification of NFRs in the early stages of the AAL development cycle is a crucial issue [31]. In this section, we show how we used OMEGA2/IFx [37] for the properties verification and model simulation of AAL system.

5.2.1. Modeling the AAL system with OMEGA2 Profile

We start by taking into account the structural part of the AAL system. Those parts are considered that are concerned with the daily calorie intake of the *Patient* in the AAL house. The AAL system is composed of *Fridge* and *Patient*; these parts are modeled along with the interaction that takes place between them. The *Fridge* partially contributes to the minimum liquid intake of the *Patient*; it also looks at the calorie consumption of the *Patient* as the *Patient* needs not to exceed it after a certain threshold.

Figure 19 shows the main Internal Block Diagram (IBD). The important parts of the AAL system are *Patient* and *Fridge*. A *Fridge* in turn is composed of *Display*, *Alarm*, *Controller*, and *Food* blocks. Figure 20 shows the IBD for the *Fridge* block. Each of the four blocks' behaviors is modeled in a separate State Machine Diagram (SMD). The *Food* block contains information about the *Food* items in the *Fridge*, the calories contained in each item, the total number of calories the *Patient* has accumulated and the calorie threshold that should not be surpassed. The *Fridge Display* is used to show the amount of calories consumed by the *Patient*. The *Alarm* is activated in case the *Patient's* calorie level surpasses a certain threshold.

Figure 21 shows the SMD for the *Patient* block. Here, the exchange of information between *Patient* and *Fridge* takes place. The number and quantity of each item present in the *Fridge* is identified. If a certain product still present in the *Fridge* is chosen by the *Patient* then the information is communicated with the *Fridge* and the list is updated. Otherwise the *Fridge* is empty and the *Patient* will wait to be refilled. Also, if the *Alarm* of the *Fridge* is raised due to

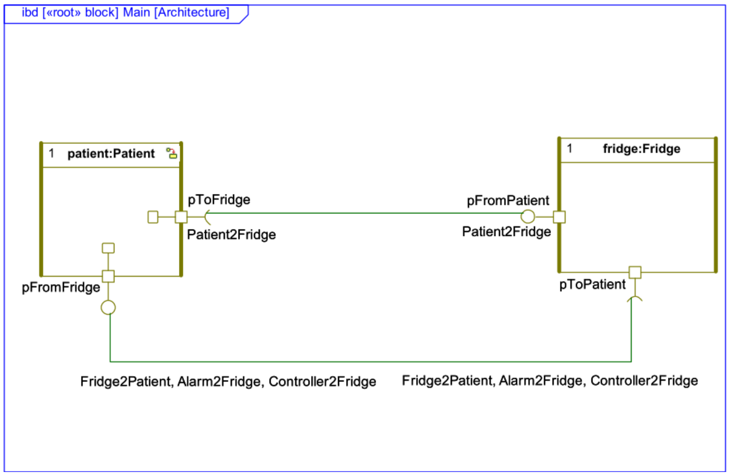

ibd [«root» block] Main [Architecture]
1 patient:Patient
pToFridge
Patient2Fridge
pFromFridge
1 fridge:Fridge
pFromPatient
Patient2Fridge
pToPatient
Fridge2Patient, Alarm2Fridge, Controller2Fridge
Fridge2Patient, Alarm2Fridge, Controller2Fridge

Figure 19: Main Internal Block Diagram

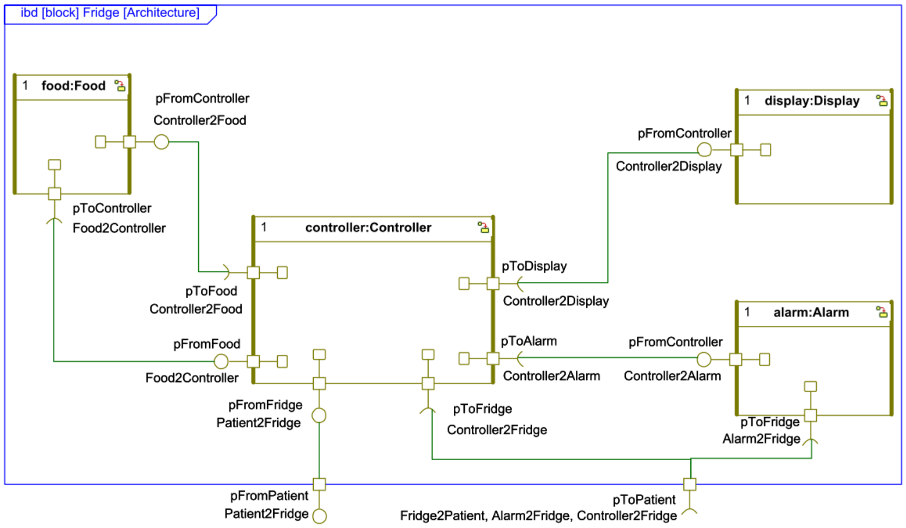


Figure 20: Fridge Internal Block Diagram

high intake of calories, the *Patient* stops eating and waits for the system to be unblocked.

5.2.2. Properties Verification of the AAL system

Below are the properties to be verified [4].

The Fridge SHALL detect and communicate with Food packages

Property 1: The Fridge SHALL detect and communicate information with AS MANY Food packages AS POSSIBLE. A Relax-ed version of this requirement with all the uncertainty factors is shown in Figure 16.

The satisfaction of this requirement contributes to the balanced diet of the

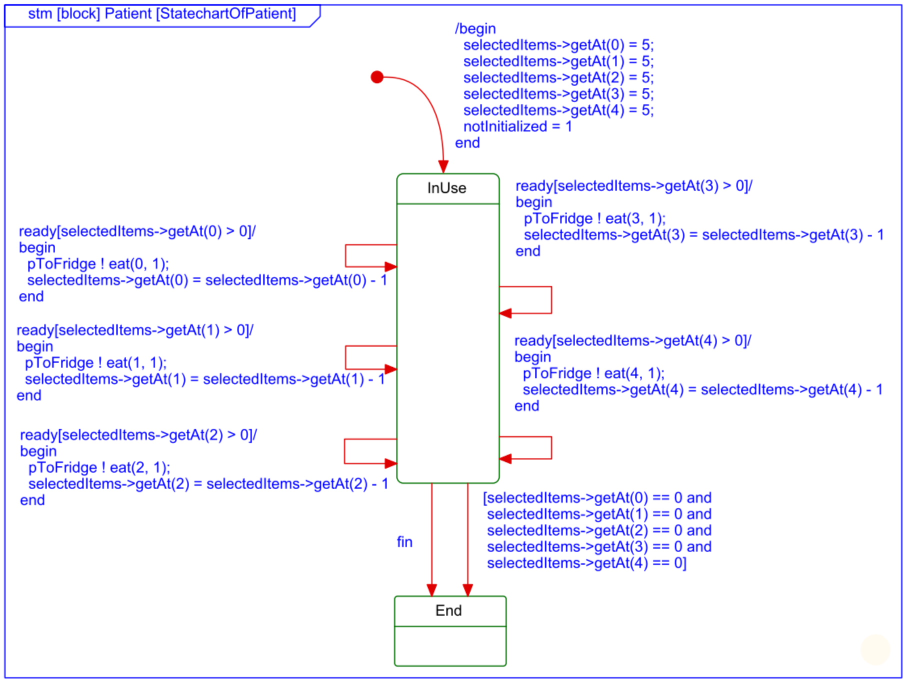

stm [block] Patient [StatechartOfPatient]
/begin
selectedItems->getAt(0) = 5;
selectedItems->getAt(1) = 5;
selectedItems->getAt(2) = 5;
selectedItems->getAt(3) = 5;
selectedItems->getAt(4) = 5;
notInitialized = 1
end
InUse
ready[selectedItems->getAt(3) > 0]/
begin
pToFridge ! eat(3, 1);
selectedItems->getAt(3) = selectedItems->getAt(3) - 1
end
ready[selectedItems->getAt(0) > 0]/
begin
pToFridge ! eat(0, 1);
selectedItems->getAt(0) = selectedItems->getAt(0) - 1
end
ready[selectedItems->getAt(1) > 0]/
begin
pToFridge ! eat(1, 1);
selectedItems->getAt(1) = selectedItems->getAt(1) - 1
end
ready[selectedItems->getAt(4) > 0]/
begin
pToFridge ! eat(4, 1);
selectedItems->getAt(4) = selectedItems->getAt(4) - 1
end
ready[selectedItems->getAt(2) > 0]/
begin
pToFridge ! eat(2, 1);
selectedItems->getAt(2) = selectedItems->getAt(2) - 1
end
[selectedItems->getAt(0) == 0 and
selectedItems->getAt(1) == 0 and
selectedItems->getAt(2) == 0 and
selectedItems->getAt(3) == 0 and
selectedItems->getAt(4) == 0]
fin
End

Figure 21: Patient State Machine Diagram

Patient. The choice of this property for verification is motivated by the fact that it is important for the AAL system to know about as many *Food* items present in the *Fridge* as possible. Figure 22 shows the SMD of the *Property 1*. The trigger for this property is an *observer* transition which is a *match* clause specifying the type of event (e.g., send), some related information (e.g., eat) and *observer* variable (e.g., p) that may send related information. The first task is to identify the number of items consumed by the *Patient* and the total number of items in the *Fridge*. Then the identity of the *Patient* is verified, if the person is identified as the *Patient*, then the next step is to calculate the number of items consumed. After this, the number of items left in the *Fridge* is calculated which is equal to the sum of all the items present in the *Fridge*. Then in the last step, we calculate if ((total number of items - number of items consumed - number of items left) >-1) and ((total number of items - number of items consumed - number of items left) <1), it means that we have reached the <<*success*>> state by having information about all the items present in the *Fridge*, i.e. it should be 0 (which means that there is no information loss). Inversely, if it is less than -1 or greater than 1, then it means that we are missing information about some of the items present in the *Fridge* and the *observer* passes into the <<*error*>> state.

We now consider the invariant requirement. *Property 2: The Alarm SHALL be raised instantaneously if the total number of calories surpasses the maximum calories allowed for the Patient.*

Figure 23 shows the SMD for *property 2*. This property ensures that the *Patient* should stop eating as soon as the total number of calories surpasses the

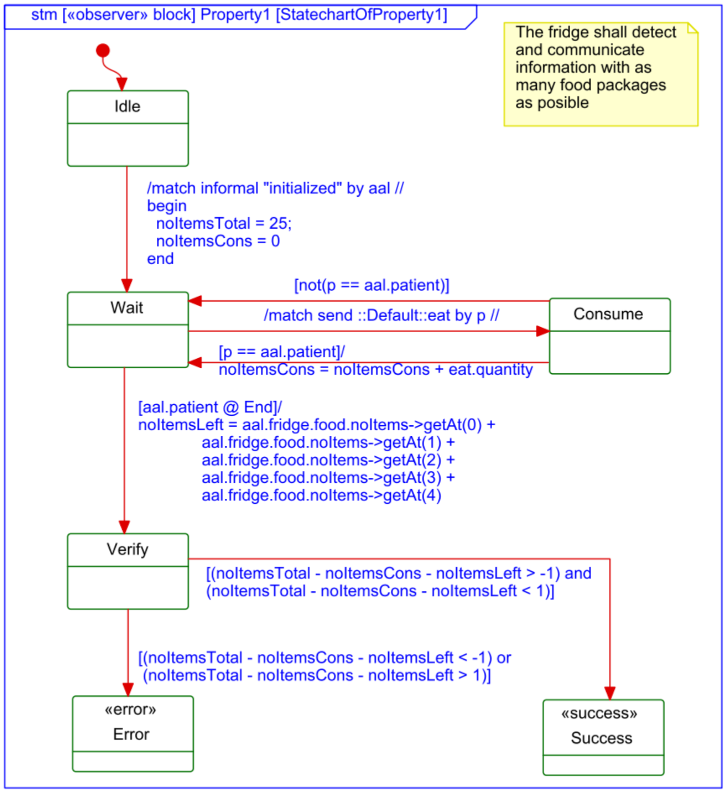

stm [«observer» block] Property1 [StatechartOfProperty1]
The fridge shall detect and communicate information with as many food packages as posible
Idle
/match informal "initialized" by aal //
begin
noItemsTotal = 25;
noItemsCons = 0
end
[not(p == aal.patient)]
Wait
/match send ::Default::eat by p //
Consume
[p == aal.patient]/
noItemsCons = noItemsCons + eat.quantity
[aal.patient @ End]/
noItemsLeft = aal.fridge.food.noItems->getAt(0) +
aal.fridge.food.noItems->getAt(1) +
aal.fridge.food.noItems->getAt(2) +
aal.fridge.food.noItems->getAt(3) +
aal.fridge.food.noItems->getAt(4)
Verify
[(noItemsTotal - noItemsCons - noItemsLeft > -1) and
(noItemsTotal - noItemsCons - noItemsLeft < 1)]
[(noItemsTotal - noItemsCons - noItemsLeft < -1) or
(noItemsTotal - noItemsCons - noItemsLeft > 1)]
«error»
Error
«success»
Success

Figure 22: Property1 State Machine Diagram

maximum calories allowed and that the *Alarm* should be raised. This requirement implies that the *Alarm* shall be immediately raised as soon as the total number of calories equals or surpasses the maximum calories allowed for the *Patient*. If it happens then the *Patient* should stop eating and we will reach a <<*success*>> state but if the *Patient* continues to eat, it means that we are reaching an <<*error*>> state.

5.2.3. Verification Results

Until now, the AAL system is modeled along with the properties to be verified on the model. We now show how to verify these properties using the IFx toolset. The AAL2 model is first exported into *AAL2.xmi* and then using the IFx toolset the *AAL2.xmi* is compiled into *AAL2.if*. The *AAL2.if* is compiled into an executable file i.e. *AAL2.x*. While verifying the AAL model, the model checker has found several error scenarios. Any of the error scenarios can then be loaded through the interactive simulation interface of the IFx toolset to trace back the error in the model and then correct it.

In order to debug a model, firstly we import it into the simulator. We check the states of the *observers* in order to identify which property has not been satisfied. In this case, *Property 2* fails. While checking the state of the entire system for this property, we discover that the <<*error*>> state contained

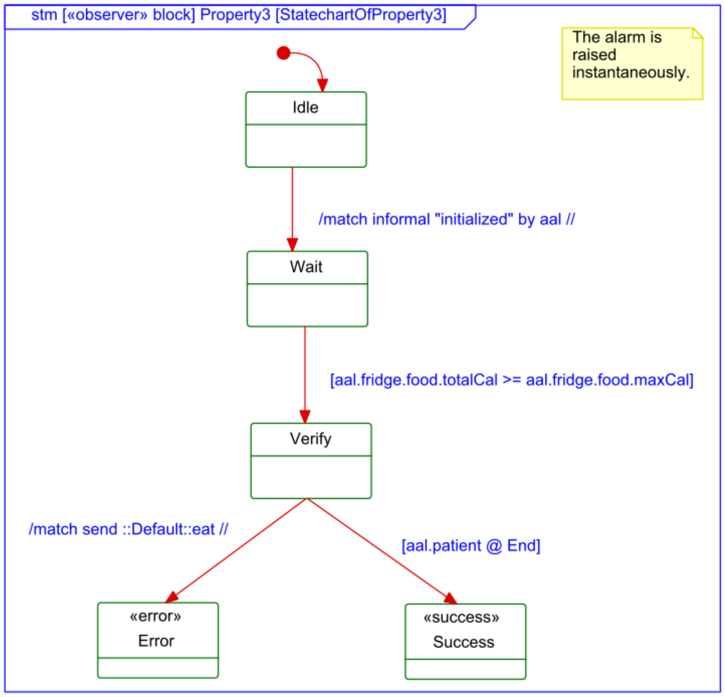

stm [«observer» block] Property3 [StatechartOfProperty3]
The alarm is raised instantaneously.
Idle
/match informal "initialized" by aal //
Wait
[aal.fridge.food.totalCal >= aal.fridge.food.maxCal]
Verify
/match send ::Default::eat //
[aal.patient @ End]
«error»
Error
«success»
Success

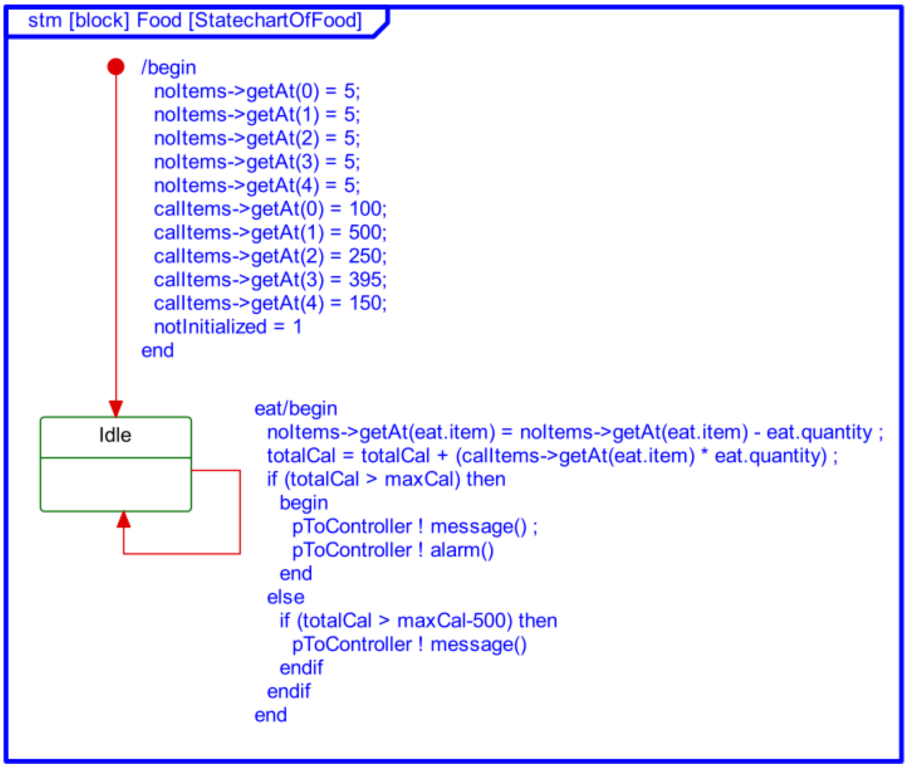
Figure 23: Property2 State Machine Diagram

the maximum allowed number of calories for the total number of calories consumed and subsequently eat requests are sent by the *Patient*. This implies that the *Alarm* function of the intelligent *Fridge* doesn't function properly which is strictly linked to its *Food* process. One can observe in the SMD of the *Food* block (Figure 24) that the *Alarm* is raised only if the total number of consumed calories is strictly superior to the maximum allowed; a condition which doesn't satisfy the request that the *Alarm* is raised as soon as possible. The correction consists of raising the *Alarm* also in case the total number of consumed calories is equal to the maximum allowed threshold. Once this error is corrected in the SMD of the *Food* block, the verification succeeds.

6. Conclusion and Future Work

The context of this research work is situated in the field of SE for SAS. This work resides in the very early stages of the software development life cycle i.e. at the RE phase. The overall contribution is to propose an integrated approach for modeling and verifying the requirements of SAS using Model Driven Engineering (MDE) techniques. It takes requirements as input and then by applying various processes and tools, we integrate the notion of uncertainty in requirements which we model using GORE techniques. Once we have the system design, we then introduce a mechanism for the properties verification of SAS.

We used Relax which is an RE language for SAS and which can introduce flexibility in NFRs to adapt to any changing environmental conditions. The



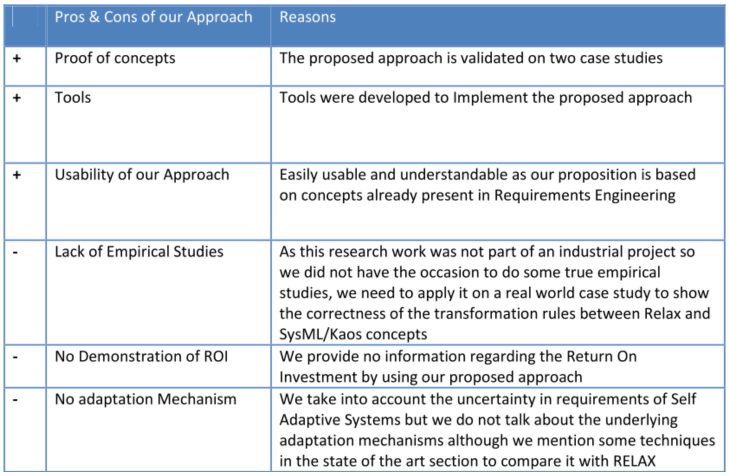
```
/begin
  noItems->getAt(0) = 5;
  noItems->getAt(1) = 5;
  noItems->getAt(2) = 5;
  noItems->getAt(3) = 5;
  noItems->getAt(4) = 5;
  calItems->getAt(0) = 100;
  calItems->getAt(1) = 500;
  calItems->getAt(2) = 250;
  calItems->getAt(3) = 395;
  calItems->getAt(4) = 150;
  notInitialized = 1
end
```

```
eat/begin
  noItems->getAt(eat.item) = noItems->getAt(eat.item) - eat.quantity ;
  totalCal = totalCal + (calItems->getAt(eat.item) * eat.quantity) ;
  if (totalCal > maxCal) then
    begin
      pToController ! message() ;
      pToController ! alarm()
    end
  else
    if (totalCal > maxCal-500) then
      pToController ! message()
    endif
  endif
end
```

Figure 24: Food State Machine Diagram

essence of Relax for SAS is that it provides a way to relax certain requirements against other requirements in situations where the resources are constrained or priority must be given to requirements. For this purpose we have developed a tool called Relax COOL editor which is used to automate the formalization of SAS requirements by taking into account the different uncertainty factors associated with each Relax-ed requirement. We then use SysML/Kaos which is an extension of the SysML requirements model with concepts of the Kaos goal model. Here, invariant requirements are captured by the concept of FGs whereas Relax-ed requirements are captured by the concept of NFGs. We have provided a correlation table that helps in mapping the Relax and SysML/Kaos concepts. Using this table, the Relax-ed requirements are then transformed into SysML/Kaos goal concepts. This mapping is done using ATL, which is a model transformation technique and which takes as input a source model and transforms it into a target model. We have developed a tool called Relax2SysML/Kaos editor which is capable of modeling the Relax-ed requirements in the form SysML/Kaos goal concepts. We provide a mechanism to verify some adaptable and invariant properties of the SAS using formal method technique OMEGA2/IFx. In order to validate our proposed approach, we have applied it to an academic Ambient Assisted Living case study.

Our work resides within the framework of self adaptation, but we do not treat the development of self adaptation mechanisms. We help SAS developers



	Pros & Cons of our Approach	Reasons
+	Proof of concepts	The proposed approach is validated on two case studies
+	Tools	Tools were developed to Implement the proposed approach
+	Usability of our Approach	Easily usable and understandable as our proposition is based on concepts already present in Requirements Engineering
-	Lack of Empirical Studies	As this research work was not part of an industrial project so we did not have the occasion to do some true empirical studies, we need to apply it on a real world case study to show the correctness of the transformation rules between Relax and SysML/Kaos concepts
-	No Demonstration of ROI	We provide no information regarding the Return On Investment by using our proposed approach
-	No adaptation Mechanism	We take into account the uncertainty in requirements of Self Adaptive Systems but we do not talk about the underlying adaptation mechanisms although we mention some techniques in the state of the art section to compare it with RELAX

Figure 25: Pros and Cons of our Proposed Approach

by providing a mechanism for identifying the uncertainty associated with the requirements of these systems. Figure 25 shows a table with the pros and cons of our proposed approach.

In terms of the future work, we have applied our approach to an academic case study. The next step is to apply it to a real industrial case study, which will confront it to more rigorous and varied evaluation criteria such as its usability and its performance.

We plan to investigate the adaptation mechanism techniques so that we can incorporate it in our proposed approach. Our approach takes into account the uncertainty in requirements of SAS, we model it using SysML/Kaos and then we verify it but we do not talk about the underlying adaptation mechanisms.

For the time being, our Relax2SysML/Kaos tool is capable of mapping the Relax concepts to SysML/Kaos concepts but not the inverse. A natural follow up of our work is to investigate how we could make it a two-way process. This would help in taking into account the information modeled in SysML/Kaos that we can not capture in Relax.

The verification of Relax-ed requirements in our proposed approach is done using OMEGA2/IFx. To take into account, the complexity of large systems, we can do the validation of their requirements at execution time. A promising approach to managing complexity in run-time environments is to develop adaptation mechanisms that leverage software models, referred to as Models@run.time [10]. Research on Models@run.time seeks to extend the applicability of models and abstractions to the run-time environment, with the goal of providing effective technologies for managing the complexity of evolving software behavior while it is executing [6].

In our proposed approach, for the properties verification using OMEGA2/IFx, we model the *observers* and then we check these observers against the system design to see if the properties are verified or not. Right now, we model these *observers* as an SMD. We would like to automate this process of *observers* modeling by automatically generating it from Relax-ed and invariant requirements.

The use of model checking techniques used by OMEGA2/IFx exposes us to the problem of state space explosion which is inherent in these techniques. We handle this problem in our proposed approach by only injecting Relax-ed or invariant requirements, i.e. those requirements that are of interest for SAS. But we can counter this problem using formal methods like B. There are already some works done for the mapping between SysML/Kaos and B in this regard. In [26], a method is defined for bridging the gap between the requirements analysis level (Extended SysML) and the formal specification level (B). This method derives the architecture of B specifications from SysML goal hierarchies. We believe that using proof based formal methods like B can help in overcoming the state space explosion problem associated with model checking techniques.

7. References

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

[2] Manzoor Ahmad. *Modeling and Verification of Functional and Non Functional Requirements of Ambient, Self Adaptive Systems.* PhD thesis, Mathématique Informatique Télécommunications, University of Toulouse Mirail, France, 2013.

[3] Manzoor Ahmad, João Araújo, Nicolas Belloir, Rgine Laleau Jean M. Bruel, Christophe Gnaho, and Farrida Semmak. Self-Adaptive Systems Requirements Modelling: four Related Approaches Comparison. In *Comparing *Requirements* Modeling Approaches Workshop (CMA@RE), RE 2013*, Rio de Janeiro Brazil, 2013. IEEE Computer Society Press.

[4] Manzoor Ahmad, Iulia Dragomir, Jean M. Bruel, Iulian Ober, and Nicolas Belloir. Early analysis of ambient systems sysml properties using omega2-ifx. In *3rd International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH'13)*, 2013.

[5] Ludovic Apvrille, Jean P. Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. TURTLE: A Real-Time UML Profile Supported by a Formal Validation Toolkit. *IEEE Trans. Softw. Eng.*, 30(7), 2004.

[6] Uwe Aßmann, Nelly Bencomo, Betty H. C. Cheng, and Robert B. France. Models@Run.Time (Dagstuhl Seminar 11481). *Dagstuhl Reports*, 1(11), 2011.

[7] Luciano Baresi, Liliana Pasquale, and Paola Spoletini. Fuzzy Goals for Requirements-Driven Adaptation. In *Proceedings of the 2010 18th IEEE International Requirements Engineering Conference*, RE '10, pages 125–134, Washington, DC, USA, 2010. IEEE Computer Society.

[8] Jérémy Bascans, Jérémy Walczak, Jérôme Zeghoudi, Manzoor Ahmad, Jacob Geisel, and Jean M. Bruel. COOL RELAX Editor, M2ICE Project, Universit de Toulouse le Mirail, 2013.

[9] Kawtar Benghazi, María Visitación Hurtado, María Luisa Rodríguez, and Manuel Noguera. Applying formal verification techniques to ambient assisted living systems. In *OnTheMove Workshop (OTM '09)*. Springer-Verlag Berlin Heidelberg 2009, 2009.

[10] Gordon S. Blair, Nelly Bencomo, and Robert B. France. Models@ Run.Time. *Computer*, 42(10):22–27, 2009.

[11] Marius Bozga, Susanne Graf, Ileana Ober, Iulian Ober, and Joseph Sifakis. The IF Toolset. In *Formal Methods for the Design of Real-Time Systems (FMDRTS '04)*. Springer Berlin/Heidelberg, 2004.

[12] Betty H. C. Cheng, Pete Sawyer, Nelly Bencomo, and Jon Whittle. A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] Lawrence Chung, Brian A. Nixon, Eric Yu, and John Mylopoulos. *Non-Functional Requirements in Software Engineering.* Springer, 1st edition, 1999.

[14] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, London, 1999.

[15] Edmund M. Clarke, William Klieber, Milo Novek, and Paolo Zuliani. Model Checking and the State Explosion Problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer Berlin Heidelberg, 2012.

[16] Jane Cleland-Huang, Raffaella Settimi, Xuchang Zou, and Peter Solc. Automated Classification of Non-Functional Requirements. *Requir. Eng.*, 12(2):103–120, May 2007.

[17] Jean P. Courtiat, Celso A. S. Santos, Christophe Lohr, and B. Outtaj. Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique. *Comput. Commun.*, 23(12), 2000.

[18] Luiz Marcio Cysneiros and Julio Cesar Sampaio do Prado Leite. Nonfunctional requirements: From elicitation to conceptual models. In *IEEE Transactions On Software Engineering*, volume 30, pages 328–350, 2004.

[19] Rogério de Lemos et al. Software engineering for self-adaptive systems: A second research roadmap.

[20] Iulia Dragomir, Iulian Ober, and David Lesens. A Case Study in Formal System Engineering with SysML. In *17th International Conference on Engineering of Complex Computer Systems (ICECCS '12)*. IEEE, 2012.

[21] Betty H.C. Cheng et al. Software Engineering for Self-Adaptive Systems. chapter Software Engineering for Self-Adaptive Systems: A Research Roadmap, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[22] Christophe Gnaho and Farida Semmak. Une Extension SysML pour l'ingénierie des Exigences Non-Fonctionnelles Orientée But. In *Ingénierie des Systèmes d'Information.* Lavoisier Paris FRANCE, 2010.

[23] HeatherJ. Goldsby, Pete Sawyer, Nelly Bencomo, Betty H.C. Cheng, and Danny Hughes. Goal-Based Modeling of Dynamically Adaptive System Requirements. In *Proceedings of the 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, ECBS '08, Washington, DC, USA, 2008. IEEE Computer Society.

[24] Seong ick Moon, K.H. Lee, and Doheon Lee. Fuzzy branching temporal logic. In *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 2004.

[25] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1), 2003.

[26] Rgine Laleau, Farida Semmak, Abderrahman Matoussi, Dorian Petit, Ahmed Hammad, and Bruno Tatibouet. A First Attempt to Combine SysML Requirements Diagrams and B. *Innovations in Systems and Software Engineering*, 6, 2010.

[27] Axel V. Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 1st edition edition, 2009.

[28] Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Yijun Yu. Towards Requirements-Driven Autonomic Systems Design. In *Proceedings of the 2005 workshop on Design and evolution of autonomic application software*, DEAS '05, pages 1–7, New York, NY, USA, 2005. ACM.

[29] Robyn R. Lutz. Targeting Safety-Related Errors during Software Requirements Analysis. *J. Syst. Softw.*, 34(3), 1996.

[30] Jean-Michel Bruel Manzoor Ahmad. A comparative study of relax and sysml/kaos. Technical report, Instituit de Recherche en Informatique de Toulouse, University Toulouse II Le Mirail, France, 2014.

[31] Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living Assistance Systems: an Ambient Intelligence Approach. In *Proceedings of the 28th international conference on Software engineering (ICSE '06)*. ACM, 2006.

[32] Iulian Ober and Iulia Dragomir. OMEGA2: A New Version of the Profile and the Tools. In *15th International Conference on Engineering of Complex Computer Systems (ICECCS '10)*. IEEE, 2010.

[33] Andres J. Ramirez, Betty H. C. Cheng, Nelly Bencomo, and Pete Sawyer. Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time. In RobertB. France, Jrgen Kazmeier, Ruth Breu, and Colin Atkinson, editors, *Model Driven Engineering Languages and Systems*, volume 7590 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[34] Andres J. Ramirez, Erik M. Fredericks, Adam C. Jensen, and Betty H. C. Cheng. Automatically RELAXing a Goal Model to Cope with Uncertainty. In *Proceedings of the 4th international conference on Search Based Software Engineering*, SSBSE'12, pages 198–212, Berlin, Heidelberg, 2012. Springer-Verlag.

[35] Vítor E. S. Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness Requirements for Adaptive Systems. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self- Managing Systems*, SEAMS '11, pages 60–69, New York, NY, USA, 2011. ACM.

[36] Eric P. Kasten Seyed M. Sadjadi, Philip K. McKinley. Architecture and Operation of an Adaptable Communication Substrate. In *Proceedings. The Ninth IEEE Workshop on Future Trends of Distributed Computing Systems FTDCS'03*, 2003.

[37] Verimag and Irit. *OMEGA2-IFx for UML/SysML v2.0, Profile and Toolset, User Manual Document v1.1*, 2011.

[38] Kristopher Welsh and Pete Sawyer. Understanding the Scope of Uncertainty in Dynamically Adaptive Systems. In *Requirements Engineering: Foundation for Software Quality*. Springer Berlin Heidelberg, 2010.

[39] Kristopher Welsh, Pete Sawyer, and Nelly Bencomo. Towards Requirements Aware Systems: Run-Time Resolution of Design-Time Assumptions. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011.

[40] Jon Whittle, Pete Sawyer, Nelly Bencomo, and Betty H. C. Cheng. A Language for Self-Adaptive System Requirements. In *International Workshop on Service-Oriented Computing: Consequences for Engineering Requirements, 2008. SOCCER '08.*, 2008.

[41] Jon Whittle, Pete Sawyer, Nelly Bencomo, Betty H. C. Cheng, and Jean M. Bruel. RELAX: Incorporating Uncertainty into the Specification of Self-Adaptive Systems. In *Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE*, RE '09, pages 79–88, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Eric S. K. Yu. Towards Modeling and Reasoning Support for Early-Phase Requirements Engineering. In *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*. IEEE Computer Society, 1997.

[43] Yijun Yu, Alexei Lapouchnian, Sotirios Liaskos, John Mylopoulos, and Julio C. S. P. Leite. From Goals to High-Variability Software Design. In *Proceedings of the 17th international conference on Foundations of intelligent systems*, ISMIS'08, pages 1–16, Berlin, Heidelberg, 2008. Springer-Verlag.

[44] Yijun Yu, Julio C.S.P. Leite, and John Mylopoulos. From Goals to Aspects: Discovering Aspects from Requirements Goal Models. In *Proceedings of the Requirements Engineering Conference, 12th IEEE International*, RE '04, pages 38–47, Washington, DC, USA, 2004. IEEE Computer Society.

Session commune aux groupes de travail GLE et RIMEL

Génie Logiciel Empirique — Rétro-Ingénierie, Maintenance et Evolution des Logiciels

Developers' Perception of Co-Change Patterns: An Empirical Study

Luciana L. Silva*†, Marco Tulio Valente*, Marcelo Maia‡ and Nicolas Anquetil§
*Department of Computer Science, Federal University of Minas Gerais, Brazil - {luciana.lourdes,mtov}@dcc.ufmg.br
†Federal Institute of Triangulo Mineiro, Brazil
‡Faculty of Computing, Federal University of Uberlandia, Brazil - marcmaia@facom.ufu.br
§RMoD Project-Team - INRIA Lille Nord Europe - nicolas.anquetil@inria.fr

***Abstract*—Co-change clusters are groups of classes that frequently change together. They are proposed as an alternative modular view, which can be used to assess the traditional decomposition of systems in packages. To investigate developer's perception of co-change clusters, we report in this paper a study with experts on six systems, implemented in two languages. We mine 102 co-change clusters from the version history of such systems, which are classified in three patterns regarding their projection to the package structure: Encapsulated, Crosscutting, and Octopus. We then collect the perception of expert developers on such clusters, aiming to ask two central questions: (a) what concerns and changes are captured by the extracted clusters? (b) do the extracted clusters reveal design anomalies? We conclude that Encapsulated Clusters are often viewed as healthy designs and that Crosscutting Clusters tend to be associated to design anomalies. Octopus Clusters are normally associated to expected class distributions, which are not easy to implement in an encapsulated way, according to the interviewed developers.**

I. Introduction

In his seminal paper on modularity and information hiding, Parnas developed the principle that modules should hide "difficult design decisions or design decisions which are likely to change" [1]. Nonetheless, Parnas' criteria to decompose systems into modules are not widely used to assess whether—after years of maintenance and evolution—the modules of a system were indeed able to confine changes. In other words, developers typically do not evaluate modular designs using historical data on software changes. Instead, modularity is evaluated most of the times under a structural perspective, using static measures of size, coupling, cohesion, etc [2]–[4]. Less frequently, semantic relations, normally extracted from source code vocabularies, are used [5]–[8].

In previous work [9], [10], we proposed the Co-Change Clustering technique to assess modularity using the history of software changes, widely available nowadays from version control repositories. Co-change clusters are sets of classes that frequently changed together in the past. They are computed applying a cluster algorithm over a co-change graph, which is a graph that represents the co-change relations in a system [11], [12]. A co-change relation between two classes is enabled whenever they are changed by the same commit transaction [13]. We also proposed the usage of distribution maps—a well-known software visualization technique [14]—to reason on co-change patterns, which are recurrent projections of co-change clusters over package structures. However, we did not evaluate to what extent co-change clusters reflect well-defined concerns, according to software developers. We also did not evaluate whether they are useful instruments to detect design anomalies, specially when the clusters crosscut the package structure or have most classes in one package and very few ones in other packages.

To reveal developers' view on the usage of Co-Change Clustering, we report in this paper an empirical study with seven experts on six systems, including one closed-source and large information system implemented in Java and five open-source software tools implemented in Pharo (a Smalltalk-like language). We mine 102 co-change clusters from the version histories of such systems, which are then classified in three patterns regarding their projection over the package structure: Encapsulated Clusters (clusters that when projected over the package structure match all co-change classes in such packages), Crosscutting Clusters (clusters whose classes are spread over several packages, touching few classes in each one), and Octopus Clusters (clusters that have most classes in one package and some "tentacles" in other packages). From the initially computed clusters, 53 clusters (52%) are covered by the proposed co-change patterns. We analyze each of these clusters with developers, asking them two overarching questions: (a) what concerns and changes are captured by the cluster? (b) does the cluster reveal design flaws? Our intention with the first question is to evaluate whether co-change clusters capture cohesive concerns that changed frequently during the software evolution. With the second question, we aim to evaluate whether co-change clusters—specially the ones classified as Crosscutting and Octopus clusters—can reveal design (or modularity) flaws.

Our contributions with this paper are twofold. First, we report a new experience on using Co-Change Clustering in a new set of systems, including both a real-world commercial information system implemented in Java and five open-source systems implemented in a second language (Pharo). Second, we report, summarize, and discuss the developer's views on co-change clusters.

We start by summarizing our technique for extracting co-change clusters (Section II) and by defining the co-change patterns considered in this paper (Section III). Then, we present the research method and steps followed in the study (Section IV). Section V reports the developers' view on co-change clusters and the main findings of our study. Section VI puts in perspective our findings and the lessons learned with the study. Section VII discusses threats to validity and Section IX concludes.

II. Co-Change Clustering

This section presents the method we follow to extract co-change graphs (Section II-A) and then co-change clusters (Section II-B). A detailed description of the steps presented in this section is available in previous work [9], [10].

A. *Co-Change Graphs*

A co-change graph is an undirected graph $\{V, E\}$, where V is a set of classes and E is a set of edges. An edge connects two classes (vertices) C_i and C_j if there is a transaction (commit) in the Version Control System that contains C_i and C_j, for $i \neq j$. The weight of an edge represents the number of commits including C_i and C_j.

To extract co-change graphs commit data is preprocessed. First, we discard commits that only change artifacts like script files, documentation, configuration files, etc. Because our focus is on co-changes involving classes. We also remove testing classes, because co-changes between tested and testing classes are usually expected and for this reason are less important. We also remove highly scattered commits, i.e., commits that change a massive number of classes. These commits represent very particular maintenance tasks (e.g., a change in comments on license agreements), which are not recurrent.

Second, from the remaining commits, we select the ones whose textual description refers to a valid maintenance task-ID in a tracking system, like Bugzilla, Jira, etc. When the same task-ID is found in multiple commits, we merge such commits, and consider just the merged commits in the co-change graph. However, it is common to have a significant number of commits not linked to issue reports [9], [15]. Therefore, we apply a time window to select the remaining commits. We merge commits by the same developer when they are performed under a given time interval. In this way, we handle the scenario when the developer commits multiple times when performing the same maintenance task. If such commits are not handled considered, we could miss relevant co-change relations.

Finally, co-change graphs are post-processed pruning edges whose weights is less than a given support threshold. The reason is that such edges are not relevant for our purpose of modeling recurrent maintenance tasks.

As described in this section, the pre and post-processing steps—as well as the clustering step discussed in the next section—depend on some thresholds. The concrete threshold values used in this paper are presented in Section IV-C.

B. *Co-Change Clusters*

Co-change clusters are set of classes in a co-change graph that frequently changed together. Co-change clusters are extracted automatically using a graph clustering algorithm designed to handle sparse graphs, as is typically the case of co-change graphs [9], [11], [12]. More specifically, we use the Chameleon clustering algorithm, which is an agglomerative and hierarchical clustering algorithm recommended to sparse graphs [16]. Chameleon consists of two phases. In the first phase, a sparse graph is extracted from the original graph (a co-change graph in our case) and a graph partitioning algorithm divides the data set (classes) into sets of clusters. In the second phase, an agglomerative hierarchical mining algorithm is applied to merge the clusters retrieved in the first phase. This algorithm maximizes the number of edges within a cluster (internal similarity) and minimizes the number of edges among clusters (external similarity).

Chameleon requires the number of clusters *M* as an input parameter in the first phase. An inappropriate value of *M* may lead to poor clusters. For this reason, we run Chameleon multiple times varying *M*'s value. After each execution, the previous tested value is decremented by one and the clusters smaller than a minimal threshold are discarded. The goal is to focus on groups of classes that may be used as alternative modular views. Therefore, it is not reasonable to consider clusters with a small number of classes.

After pruning the small clusters, a clustering quality function is computed over the remaining clusters, to provide an overall score for the clusters generated by a given M value. This function combines measures of the clusters cohesion (tight clusters) and cluster separation (highly separated clusters). A detailed description of these measures is out of the scope of this paper and is available elsewhere [9], [10].

III. Co-Change Patterns

In this section, we propose three co-change patterns aiming to represent common instances of co-change clusters. The patterns are defined by projecting clusters over the package structure of an object-oriented system, using distribution maps. Distribution maps are a software visualization technique that represents classes as small squares in large rectangles, which represent packages [14]. The color of the classes represent a property; in our specific case, the co-change cluster.

Co-change patterns are defined using two metrics originally proposed for distribution maps: focus and spread. First, *spread* measures how many packages are touched by a cluster q. Second, *focus* measures the degree the classes in a co-change cluster dominate their packages. For example, if a cluster touches all classes of a package, its focus is one. In formal terms, the *focus* of a co-change cluster q is defined as follows:

$$focus(q) = \sum_{p_i \in P} touch(q, p_i) * touch(p_i, q)$$

where

$$touch(q, p) = \frac{|q \cap p|}{|p|}$$

The measure $touch(q, p_i)$ represents the number of classes in a cluster q located in package p_i divided by the number of classes in p_i that are included in at least one co-change cluster. Similarly, $touch(p_i, q)$ is the number of classes in package p_i that are a member of cluster q divided by the number of classes in q. Focus ranges between 0 and 1, where 1 means that the cluster entirely dominates the packages it touches.

Using focus and spread, we propose three patterns of co-change clusters, as follows:

Encapsulated: An Encapsulated co-change cluster q dominates all classes of the packages it touches, i.e.,

$$Encapsulated(q), \text{if } focus(q) == 1$$

Figure 1 shows two examples of Encapsulated Clusters.[1] All classes in Cluster 9 (blue) are located in the same package, which only has classes in this cluster. Similarly, Cluster 10 (green) has classes located in three packages. Moreover, these three packages do not have classes in other clusters.

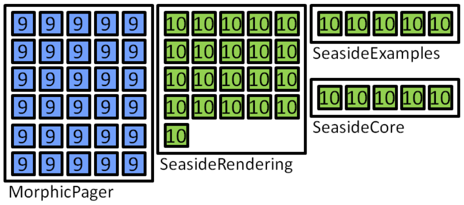


Fig. 1. Encapsulated clusters (Glamour)

Crosscutting: Conceptually, a Crosscutting Cluster is spread over several packages but touches few classes in each one. In practical terms, we propose the following thresholds to represent a Crosscutting cluster q:

$$Crosscutting(q), \quad \text{if } spread(q) \geq 4 \;\; \wedge \;\; focus(q) \leq 0.3$$

Figure 2 shows an example of Crosscutting cluster. Cluster 8 (red) is spread over seven packages, but does not dominate any of them (its focus is 0.14).

Octopus: Conceptually, an Octopus Cluster q has two subclusters: a body B and a set of tentacles T. The body has most classes in the cluster and the tentacles have a very low focus, as follows:

$$\begin{aligned} Octopus(q, B, T) = \text{if} \quad & touch(B, q) > 0.50 \, \wedge \\ & focus(T) \leq 0.25 \, \wedge \\ & focus(q) > 0.3 \end{aligned}$$

By requiring $focus(q) > 0.3$, a cluster cannot be classified as Crosscutting and Octopus, simultaneously.

Figure 3 shows an Octopus cluster, whose body has 22 classes, located in one package. The cluster has a single tentacle class. When considered as an independent sub-cluster, this tentacle has focus 0.005. Finally, the whole Octopus has focus 0.78, which avoids its classification as Crosscutting.

As usual in the case of metric-based rules to detect code patterns [17], [18], the proposed strategies to detect co-change patterns depend on thresholds to specify the expected spread and focus values. To define such thresholds we based on our previous experiences with co-change clusters extracted for open-source Java-based systems [9], [10]. Typically, low focus values are smaller than 0.3 and high spread values are greater or equal to four packages.

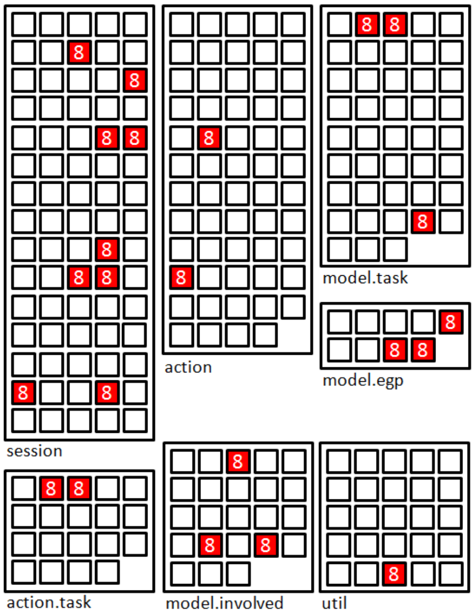


Fig. 2. Crosscutting cluster (SysPol)

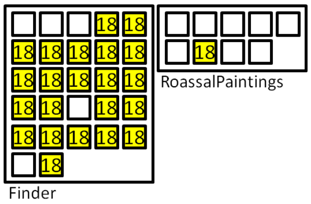


Fig. 3. Octopus cluster (Moose)

IV. Study Design

In this section we present the questions that motivated our research (Section IV-A). We also present the dataset (Section IV-B), the thresholds selection (Section IV-C), the steps we followed to extract the co-change clusters (Section IV-D), and to conduct the interviews (Section IV-E).

A. Research Questions

With this research, our *goal* is to investigate from the point of view of expert developers and architects the concerns represented by co-change patterns. We also evaluate whether these patterns are able to indicate design anomalies, in the *context* of Java and Pharo object-oriented systems. To achieve these goals, we pose three research questions in the paper:

RQ #1: To what extent do the proposed co-change patterns cover real instances of co-change clusters?

RQ #2: How developers describe the clusters matching the proposed co-change patterns?

[1] All examples used in this section are real instances of co-change clusters, extracted from the subject systems used in this paper, see Section IV-B.

RQ #3: To what extent do the clusters matching the proposed co-change patterns indicate design anomalies?

With RQ #1, we check whether the proposed strategy to detect co-change patterns match a representative set of co-change clusters. With the second and third RQs we collect and organize the developers' view on co-change patterns. Specifically, with the second RQ we check how developers describe the concerns and requirements implemented by the proposed co-change patterns. With the third RQ, we check whether clusters matching the proposed co-change patterns—specially the ones classified as Crosscutting and Octopus—are usually associated to design anomalies.

B. Target Systems

To answer our research questions, we investigate the following six systems: (a) SysPol, which is a closed-source information system implemented in Java that provides many services related to the automation of forensics and criminal investigation processes; the system is currently used by one of the Brazilian state police forces (we are omitting the real name of this system, due to a non-disclosure agreement with the software organization responsible for SysPol's implementation and maintenance); (b) five open-source systems implemented in Pharo [19], which is a Smalltalk-like language. We evaluate the following Pharo systems: Moose (a platform for software and data analysis), Glamour (an infrastructure for implementing browsers), Epicea (a tool to help developers share untangled commits), Fuel (an object serialization framework), and Seaside (a framework for developing web applications).

Table I describes these systems, including information on number of lines of code (LOC), number of packages (NOP), number of classes (NOC), number of commits extracted for each system, and the time frame considered in this extraction.

TABLE I
TARGET SYSTEMS

System	LOC	NOP	NOC	Commits	Period
SysPol	63,754	38	674	9,072	10/13/2010 - 08/08/2014
Seaside	26,553	28	695	5,741	07/17/2013 - 12/08/2014
Moose	33,967	36	505	2,417	01/21/2013 - 11/17/2014
Fuel	5,407	6	136	2,009	08/05/2013 - 12/03/2014
Epicea	26,260	9	222	1,400	08/15/2013 - 11/15/2014
Glamour	21,076	24	452	3,213	02/08/2013 - 11/27/2014

C. Thresholds Selection

For this evaluation, we use the same thresholds of our previous experience with Co-Change Clustering [9], [10]. We reused the thresholds even for the Pharo systems, because they are decomposed in packages and classes, as in Java.

SysPol is a closed-source system developed under agile development guidelines. In this project, the tasks assigned to the development team usually have an estimated duration of one working day. For this reason, we set up the time window threshold used to merge commits as one day, i.e., commits performed in the same calendar day by the same author are merged. Regarding the Pharo systems, developers have more freedom to select the tasks to work on as common in open-source systems. Moreover, they usually only commit after finishing and testing a task (as explained to us by Pharo's leading software architects). However, Pharo commits are performed per package. For example, a maintenance task that involves changes in classes located in packages P_1 and P_2 is concluded using two different commits: a commit including the classes located in P_1 and another containing the classes in P_2. For this reason, in the case of the five Pharo systems, we set up the time window used to merge commits as equal to one hour. On the one hand, this time interval is enough to capture all commits related to a given maintenance task, according to Pharo architects. On the other hand, developers usually take more than one hour to complete a next task after committing the previous one. Finally, we randomly selected 50 change sets from one of the Pharo systems (Moose) to check manually with one of system's developer. He confirmed that all sets refer to unique programming task.

D. Extracting the Co-Change Clusters

We start by preprocessing the extracted commits to compute co-change graphs. Table II presents four measures: (a) the initial number of commits considered for each system; (b) the number of discard operations targeting commits that do not change classes or change a massive number of classes; (c) the number of merge operations targeting commits referring to the same Task-ID in the tracking system or performed under the time window thresholds; (d) the number of change sets effectively used to compute the co-change graphs. By change sets we refer to the commits used to create the co-change graphs, including the ones produced by the merge operations.

TABLE II
PREPROCESSING FILTERS AND NUMBER OF CO-CHANGE CLUSTERS

System	Commits	Discard Ops	Merge Ops	Change Sets
SysPol	9,072	1,619	1,447	1,951
Seaside	5,741	1,725	1,421	1,602
Moose	2,417	289	762	856
Fuel	2,009	395	267	308
Epicea	1,400	29	411	448
Glamour	3,213	2,722	1,075	1,213

After applying the preprocessing and post-processing filters, we use the ModularityCheck[2] tool to compute the co-change clusters [20]. Table III shows the number of co-change clusters computed for each system (102 clusters, in total).

TABLE III
NUMBER OF CO-CHANGE CLUSTERS PER SYSTEM

System	# clusters	System	# clusters
SysPol	20	Fuel	8
Seaside	25	Epicea	14
Moose	20	Glamour	15

Figure 4 shows the distribution of the densities of the co-change clusters extracted for each system. Density is a key property in co-change clusters, because it assures that there is

[2] http://aserg.labsoft.dcc.ufmg.br/modularitycheck

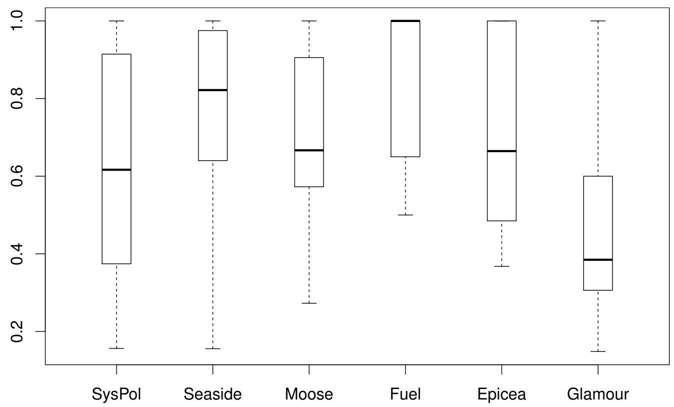


Fig. 4. Co-change clusters density

a high probability of co-changes between each pair of classes in the cluster. The SysPol's clusters have a median density of 0.63, whereas the co-change graph extracted for this system has a density of 0.20. The clusters of the Pharo systems have a median density ranging from 0.39 (Glamour) to 1.00 (Fuel), whereas the highest density of the co-change graphs for these systems is 0.11 (Fuel).

E. Interviews

Table IV describes the number of expert developers we interviewed for each system and how long they have been working on the systems. In total, we interviewed seven experienced developers. In the case of SysPol, we interviewed the lead architect of the system, who manages a team of around 15 developers. For Moose, we interviewed two experts. For the remaining Pharo systems, we interviewed a single developer per system.

TABLE IV
EXPERT DEVELOPERS' PROFILE

System	Developer ID	Experience (Years)	# Emails/Chats
SysPol	D1	2	44
Seaside	D2	5	0
Moose	D3;D4	4.5	6
Fuel	D5	4.5	0
Epicea	D6	2.5	2
Glamour	D7	4	1

We conducted face-to-face and Skype semi-structured interviews with each developer using open-ended questions [21]. During the interviews, we presented all co-change clusters that matched one of the proposed co-change patterns to the developers. We used Grounded Theory [22] to analyze the answers and to organize them into categories and concepts (sub-categories). The interviews were transcribed and took approximately one and half hour (with each developer; both Moose developers were interviewed in the same session). In some cases, we further contacted the developers by e-mail or text chat to clarify particular points in their answers. Table IV also shows the number of clarification mails and chats with each developer. For Moose, we also clarified the role of some classes with its leading architect (D8) who has been working for 10 years in the system.

V. Results

In this section, we present the developer's perceptions on the co-change clusters, collected when answering the proposed research questions.

A. To what extent do the proposed co-change patterns cover real instances of co-change clusters?

To answer this RQ, we categorize the co-change clusters as Encapsulated, Crosscutting, and Octopus, using the definitions proposed in Section III. The results are summarized in Table V. As we can observe, the proposed co-change patterns cover from 35% (Epicea) to 72% (Seaside) of the clusters extracted for our subject systems. We found instances of Octopus Clusters in all six systems. Instances of Encapsulated Clusters are found in all systems, with the exception of SysPol. By contrast, Crosscutting Clusters are less common. In the case of four systems (Seaside, Moose, Fuel, and Epicea) we did not find a single instance of this pattern.

TABLE V
NUMBER AND PERCENTAGE OF CO-CHANGE PATTERNS

System	Encapsulated	Crosscutting	Octopus	Total
SysPol	0 (0%)	8 (40%)	5 (25%)	13 (65%)
Seaside	7 (28%)	0 (0%)	11 (44%)	18 (72%)
Moose	5 (25%)	0 (0%)	1 (5%)	6 (30%)
Fuel	3 (37%)	0 (0%)	1 (13%)	4 (50%)
Epicea	3 (21%)	0 (0%)	2 (14%)	5 (35%)
Glamour	4 (27%)	1 (7%)	2 (20%)	7 (47%)
Total	22 (41.5%)	9 (17%)	22 (41.5%)	53 (52%)

In summary, we found 53 co-change clusters matching one of the proposed co-change patterns (52%). The remaining clusters do not match the proposed patterns because their spread and focus do not follow the thresholds defined in Section III.

B. How developers describe the clusters matching the proposed co-change patterns?

To answer this RQ, we presented each cluster categorized as Encapsulated, Crosscutting, or Octopus to the developer of the respective system and asked him to describe the central concerns implemented by the classes in these clusters.

Encapsulated Clusters: The codes extracted from developers' answers for clusters classified as Encapsulated are summarized in Table VI. The table also presents the package that encapsulates each cluster. The developers easily provided a description for 21 out of 22 Encapsulated clusters. A cluster encapsulated in the `Core` package of Moose (Cluster 16) is the only one the developers were not able to describe by analyzing only the class names. Therefore, in this case we asked the experts to inspect the commits responsible to this cluster and they concluded that the co-change relations are due to *"several small refactorings applied together"*. Since these refactorings are restricted to classes in a single package, they were not filtered out by the threshold proposed to handle highly scattered commits.

Analyzing the developers' answers, we concluded that all clusters in Table VI include classes that implement clear and

TABLE VI
CONCERNS IMPLEMENTED BY ENCAPSULATED CLUSTERS

System	Cluster	Packages	Codes
Seaside	1	Pharo20ToolsWeb	Classes to compute information such as memory and space status
	2	ToolsWeb	Page to administrate Seaside applications
	3	Pharo20Core	URL and XML enconding concerns
	4	Security, PharoSecurity	Classes to configure security strategies
	5	JSONCore	JSON renderer
	6	JavascriptCore	Implementation of JavaScript properties in all dialects
	7	JQueryCore	JQuery wrapper
Glamour	8	MorphicBrick	Basic widgets for increasing performance
	9	MorphicPager	Glamour browser
	10	SeasideRendering, SeasideExamples, SeasideCore	Web renderer implementation
	11	GTInspector	Object inspector implementation
Moose	12	DistributionMap	Classes to draw distribution maps
	13	DevelopmentTools	Scripts to use Moose in command line
	14	MultiDimensionsDistributionMap	Distribution map with more than one variable
	15	MonticelloImporter	Monticello VCS importer
	16	Core	Several small refactoring applied together
Fuel	17	Fuel	Serializer and materializer operations
	18	FuelProgressUpdate	Classes that show a progress update bar
	19	FuelDebug	Implementation of the main features of the package
Epicea	20	Hiedra	Classes to create vertices and link them in a graph
	21	Mend	Command design pattern for modeling change operations
	22	Xylem	Diff operations to transform a dictionary X into Y

well-defined concerns. For this reason, we classify all clusters in a single category, called *Specific Concerns*. For example, in Seaside, Cluster 4 has classes located in two packages: Security and PharoSecurity. As indicated by their names, these two packages are directly related to security concerns. In Glamour, Cluster 10 represents planned interactions among Glamour's modules, as described by Glamour's developer:

“The Rendering *package has web widgets and rendering logic. The* Presenter *classes in the* Rendering *package represent an abstract description for a widget, which is translated into a concrete widget by the* Renderer. *Thus, when the underlying widget library (*Core*) is changed, the* Renderer *logic is also changed. After that, the* Examples *classes have to be updated.”* (D7)

Crosscutting Clusters: Table VII presents the codes extracted for the Crosscutting Clusters detected in SysPol, which concentrates 8 out of 9 Crosscutting Clusters considered in our study. We identified that these Crosscutting Clusters usually represent *Assorted Concerns* (category) extracted from the following common concepts:

Assorted, Mostly Functional Concerns. In Table VII, 7 out of 8 Crosscutting Clusters express SysPol's functional concerns. Specifically, four clusters are described as a collection of several concerns (the reference to several is underlined, in Table VII). In the case of these clusters, the classes in a package tend to implement multiple concerns; and a concern tend to be implemented by more than one package. For example, SysPol's developer made the following comments when analyzing one of the clusters:

*“*CreateArticle *is a quite generic use case in our system. It is usually imported by other use cases. Sometimes, when implementing a new use case, you must first change classes associated to* CreateArticle*”* (D1, on Cluster 2)

“These classes represent a big module that supports Task *related features and that contain several use cases. Classes related to* Agenda *can change with* Task *because there are* Tasks *that can be saved in a* Agenda*”* (D1, on Cluster 4)

We also found a single Crosscutting Cluster in Glamour, which has 12 classes spread across four packages, with focus 0.26. According to Glamour's developer *“These classes represent changes in text rendering requirements that crosscut the rendering engine and their clients”* (D7).

Assorted, Mostly Non-functional Concerns. Cluster 8 is the only one which expresses a non-functional concern, since its classes provide access to databases (and also manipulate Article templates).

Therefore, at least in SysPol, we did not find a strong correspondence between recurrent and scattered co-change relations—as captured by Crosscutting Clusters—and classical crosscutting concerns, such as logging, distribution, persistence, security, etc. [23], [24]. This finding does not mean that such crosscutting concerns have a well-modularized implementation in SysPol (e.g., using aspects), but that their code is not changed with frequency.

Octopus Clusters: From the developers' answers, we observed that all Octopus represent *Partially Encapsulated Concerns* (category), as illustrated by the following clusters:[3]

- In Moose, there is an Octopus (see Figure 3) whose body implement browsers associated to Moose panels (Finder) and the tentacle is a generic class for visualization, which is used by the Finder to display visualizations inside browsers.

[3] Due to the lack of space, we do not report the Octopus clusters' codes. We provide this content at http://aserg.labsoft.dcc.ufmg.br/ICSME15-CoChanges

TABLE VII
CONCERNS IMPLEMENTED BY CROSSCUTTING CLUSTERS IN SYSPOL

ID	Spread	Focus	Size	Codes
1	9	0.26	45	Several concerns, search case, search involved in crime, insert conduct
2	9	0.22	29	Several concerns, seizure of material, search for material, and create article
3	10	0.20	31	Requirement related to the concern *analysis*, including review analysis and analysis in flagrant
4	12	0.15	31	Several classes are associated to the *task* and *consolidation* concerns
5	15	0.29	35	Subjects related to create article and to prepare expert report
6	9	0.22	24	Several concerns in the model layer, such as criminal type and indictment
7	7	0.14	24	Features related to people analysis, insertion, and update
8	4	0.30	12	Access to the database and article template

- In Glamour, there is an Octopus whose body implement a declarative language for constructing browsers and the tentacles are UI widgets. Changes in this language (e.g., to support new types of menus) propagate to the `Renderer` (to support the new menu renderings).

C. To what extent do clusters matching the proposed co-change patterns indicate design anomalies?

We also asked the developers whether the clusters are somehow related to design or modularity anomalies, including bad smells, misplaced classes, architectural violations, etc.

Encapsulated Clusters: In the case of Encapsulated Clusters, design anomalies are reported for a single cluster in Glamour (Cluster 9, encapsulated in the `MorphicPager` package, as reported in Table VI). Glamour's developer made the following comment on this cluster:

"The developer who created this new browser did not follow the guidelines for packages in Glamour. Despite of these classes define clearly the browser creation concern, the class `GLMMorphicPagerRenderer` *should be in the package* `Renderer` *and the class* `GLMPager` *should be in the package* `Browser`*"* (D7, on Cluster 9)

Interestingly, this cluster represents a conflict between structural and logical (or co-change based) coupling. Most of the times, the two mentioned classes changed with classes in the `MorphicPager` package. Therefore, the developer who initially implemented them in this package probably favoured this logical aspect in his decision. However, according to Glamour's developer there is a structural force that is more important in this case: subclasses of `Renderer`, like `GLMMorphicPagerRenderer` should be in their own package; the same for subclasses of `Browser`, like `GLMPager`.

Crosscutting Clusters: SysPol's developer explicitly provided evidences that six Crosscutting clusters (67%) are related to *Design Anomalies* (category), including three kind of problems (concepts):

Low Cohesion/High Coupling (two clusters). For example, Cluster 2 includes a class, which is "one of the classes with the highest coupling in the system." (D1)

High Complexity Concerns (two clusters). For example, Cluster 4 represents "a difficult part to understand in the system and its implementation is quite complex, making it hard to apply maintenance changes." (D1)

Package Decomposition Problems (two clusters). For example, 27 classes in Cluster 1 "include implementation logic that should be in an under layer." (D1)

SysPol's developer also reported reasons for *not* perceiving a design problem in the case of three Crosscutting Clusters (Clusters 6, 7, and 8). According to the developer, these clusters have classes spread over multiple architectural layers (like `Session`, `Action`, `Model`, etc), but implementing operations related to the same use case. According to the developer, since these layers are defined by SysPol's architecture, there is no alternative to implement the use cases without changing these classes.

Octopus Clusters: SysPol's developer provided evidences that two out of five Octopus Clusters in the system are somehow related to design anomalies. The design anomalies associated to Octopus Clusters are due to *Package Decomposition Problems*. Moreover, a single Octopus Cluster among the 17 clusters found in the Pharo tools is linked to this category. For example, in Epicea, one cluster includes "some classes located in the `Epicea` package, which should be moved to the `Ombu` package". It is worth mentioning that these anomalies were unknown to the developers. They were detected after inspecting the clusters to comprehend their concerns.

In contrast, the developers did not find design anomalies in the remaining 16 Octopus clusters detected in the Pharo systems. As an example from Moose, the developer explained as follows the Octopus associated to Cluster 18 (see Figure 3):

"The propagation starts from RoassalPaintings to Finder. Whenever something is added in the RoassalPaintings, it is often connected with adding a menu entry in the Finder." (D8)

Interestingly, the propagation in this case happens from the tentacle to the body classes. It is a new feature added to `RoassalPaintings` that propagates changes to the body classes in the `Finder` package. Because Roassal (a visualization engine) and Moose (a software analysis platform) are different systems, it is more complex to refactor the tentacles of this octopus.

VI. DISCUSSION

In this section, we put in perspective our findings and the lessons learned with the study.

A. Applications on Assessing Modularity

On the one hand, we found that Encapsulated Clusters typically represent well-designed modules. Ideally, the higher the number of Encapsulated Clusters, the higher the quality of a module decomposition. Interestingly, Encapsulated Clusters are the most common co-change patterns in the Pharo software tools we studied, which are generally developed by high-skilled developers. On the other hand, Crosscutting Clusters in SysPol tend to reveal design anomalies with a precision of 67% (at least in our sample of eight Crosscutting Clusters). Typically, these anomalies are due to concerns implemented using complex class structures, which suffer from design problems like *high coupling/low cohesion*. They are not related to classical non-functional concerns, like logging, persistence, distribution, etc. We emphasize that the differences between Java (SysPol) and Pharo systems should not be associated exclusively to the programming language. For example, in previous studies we evaluated at least two Java-based systems without Crosscutting Clusters [10]. In fact, SysPol's expert associates the modularity problems found in the system to a high turnover in the development team, which is mostly composed by junior developers and undergraduate students.

Finally, developers are usually skeptical about removing the octopus' tentacles, by for example moving their classes to the body by inserting a stable interface between the body and the tentacles. For example, Glamour's developer made the following comments when asked about these possibilities:

"Unfortunately, sometimes it is difficult to localize changes in just one package. Even a well-modularized system is a system after all. Shielding changes in only one package is not absolutely possible." (D7)

B. The Tyranny of the Static Decomposition

Specifically for Crosscutting Clusters, the false positives we found are due to maintenance tasks whose implementation requires changes in multiple layers of the software architecture (like user interface, model, persistence, etc). Interestingly, the expert developers usually view their static software architectures as dominant structures. Changes that crosscut the layers in this architecture are not perceived as problems, but as the only possible implementation solution in face of their current architectural decisions. During the study, we referred to this recurrent observation as the *tyranny of the static decomposition*. We borrowed the term from the "tyranny of the dominant decomposition" [25], normally used in aspect-oriented software development to denote the limitations of traditional languages and modularization mechanisms when handling crosscutting concerns.

In future work, we plan to investigate this tyranny in details, by arguing developers if other architectural styles are not possible, for example centered on domain-driven design principles [26]. We plan to investigate whether information systems architected using such principles are less subjected to crosscutting changes, as the ones we found in SysPol.

C. (Semi-)Automatic Remodularizations

Modular decisions deeply depend on software architects expertise and also on particular domain restrictions. For example, even for SysPol's developer it was difficult to explain and reason about the changes captured by some of the Crosscutting Clusters detected in his system. For this reason, the study did not reveal any insights on techniques or heuristics that could be used to (semi-)automatically remove potential design anomalies associated to Crosscutting Clusters. However, co-change clusters readily meet the concept of virtual separation of concerns [27], [28], which advocates module views that do not require syntactic support in the source code. In this sense, co-change clusters can be an alternative to the Package Explorer, helping developers to comprehend the spatial distribution of changes in software systems.

D. Limitations

We found three co-change clusters that are due to floss refactoring, i.e., programming sessions when the developer intersperses refactoring with other kinds of source code changes, like fixing a bug or implementing a new feature [29]. To tackle this limitation, we can use tools that automatically detect refactorings from version histories, like Ref-Finder [30] and Ref-Detector [31]. Once the refactorings are identified, we can remove co-change relations including classes only modified as prescribed by the identified refactorings.

Furthermore, 49 co-change clusters have no matched the proposed patterns (48%). We inspected them to comprehend why they were not categorized as Encapsulated, Crosscutting, or Octopus. We observed that 32 clusters are well-confined in packages, i.e., they touch a single package but their focus is lower than 1.0. This behavior does not match Encapsulated pattern because these clusters share the packages they touch with other clusters. Moreover, we also identified 14 clusters similar to Octopus, i.e., they have bodies in a package and arms in others. However, some clusters have bodies smaller and others have arms tinier than the threshold settings. The remaining three clusters touch very few classes per package but their spread is lower than the threshold settings.

Finally, we did not look for false negatives, i.e., sets of classes that changed together, but that are not classified as co-change clusters by the Chameleon graph clustering algorithm. Usually, computing false negatives in the context of architectural analysis is more difficult, because we depend on architects to generate golden sets. Specifically in the case of co-change clusters, we have the impression that expert developers are not completely aware of the whole set of changes implemented in their systems and on the co-change relations established due to such changes, making it more challenging to build a golden set of co-change clusters.

VII. Threats to Validity

First, we evaluated six systems, implemented in two languages (Java and Pharo) and related to two major domains (information systems and software tools). Therefore, our results may not generalize to other systems, languages, and application domains (external validity). Second, our results may reflect personal opinions of the interviewed developers on software architecture and development (conclusion validity).

Anyway, we interviewed expert developers, with large experience, and who are responsible for the central architectural decisions in their systems. Third, our results are directly impacted by the thresholds settings used in the study (internal validity). We handled this threat by reusing thresholds from our previous work on Co-Change Clustering, which were defined after extensive experimental testings. Furthermore, thresholds selection is usually a concern in any technique for detecting patterns in source code. Finally, there are threats concerning the way we measured the co-change relations (construct validity). Specifically, we depend on pre and posprocessing filters to handle commits that could pollute the co-change graphs with meaningless relations. For example, during the analysis with developers, we detected three co-change clusters (6%) that are motivated by small refactorings. Ideally, it would be interesting to discard such commits automatically. Finally, we only measured co-change relations for classes. However, SysPol has other artifacts, such as XML and XHTML files, which are not considered in the study.

VIII. Related Work

Semantic Clustering is a technique based on Latent Semantic Indexing (LSI) and clustering to group source code artifacts that use similar vocabulary [6], [7]. Therefore, Co-Change and Semantic Clustering are conceptually similar techniques, sharing the same goals. However, they use different data sources (commits vs vocabularies) and processing algorithms (Chamaleon graph clustering algorithm vs LSI). Moreover, Semantic Clustering was not evaluated in the field, using developers' views on the extracted clusters.

Ball et al. introduced the concept of co-change graphs [11] and Beyer and Noack improved this concept proposing a graph visualization technique to reveal clusters of frequently co-changed artifacts [12]. Their approach clusters all software artifacts which are represented as vertices in the co-change graphs. In contrast, we prune several classes during the co-change graph and co-change cluster extraction phases. We also compute co-change clusters using a clustering algorithm designed for sparse graphs, as is typically the case of co-change graphs. Finally, our focus is not on software visualization but on using co-change clusters to support modularity analysis.

Co-Change mining is used to predict changes [13], [32], to support program visualization [12], [33], to reveal logical dependencies [34], [35], to improve defect prediction techniques [36], and to detect bad smells [37]. Zimmermann et al. propose an approach that uses association rules mining on version histories to suggest possible future changes (e.g., if class `A` usually co-changes with `B`, and a commit only changes `A`, a warning is raised recommending to check whether `B` should not be changed too) [13]. However, co-change clusters are coarse-grained structures, when compared to the set of classes in association rules. They usually have more classes (at least, four classes according to our thresholds) and are detected less frequently. Therefore, they are better instruments to help developers on program comprehension.

Bavota et al. investigate how the various types of coupling aligns with developer's perceptions [38]. They consider coupling measures based on structural, dynamic, semantic, and logical information and evaluate how developers rate the identified coupling links. Their results suggest that semantic coupling seems to better reflect the developers' mental model that represents interactions between entities. However, the developers interviewed in the study evaluated only pairs of classes. In contrast, we retrieve co-change clusters having four or more classes.

Vanya et al. use co-change clusters to support systems partitioning, aiming to reduce coupling between parts of a system [39]. Their approach differs from ours because co-change clusters containing files from the same part of the system are discarded. Beck and Diehl compare and combine logical and structural dependencies to retrieve modular designs [40], [41]. They report clustering experiments to recover the architecture of ten Java programs. Their results indicate that logical dependencies are interesting when substantial evolutionary data is available. In this paper, we considered all commits available for the evaluated systems (almost 24K commits).

IX. Conclusion

One of the benefits of modularity is managerial, by allowing separate groups to work on each module with little need for communication [1]. However, this is only possible if modules confine changes; crosscutting changes hamper modularity by making it more difficult to assign work units to specific teams. In this paper, we evaluate in the field a technique called Co-Change Clustering, proposed to assess modularity using logical (or evolutionary) coupling, as captured by co-change relations. We concluded that Encapsulated Clusters are very often linked to healthy designs and that 67% of Crosscutting Clusters are associated to design anomalies. Octopus Clusters are normally associated to expected class distributions, which are not easy to implement in an encapsulated way, according to the interviewed developers.

As future work, we plan to conduct a large scale study to analyze whether the occurrence of co-change patterns is associated to programming languages. Particularly, we intend to consider well-known projects implemented in other languages, such as C/C++. Also, we plan to investigate the effect that alternative software architecture styles, e.g., architectures based on domain-driven principles, have on co-change patterns. Specifically, we plan to check whether such architectures can avoid the generation of clusters matching Crosscutting or Octopus patterns. We also plan to compare and contrast the results of Co-Change Clustering with the ones generated by Semantic Clustering [5], [6].

Acknowledgments

This research is supported by grants from FAPEMIG, CAPES, and CNPq. We would like to thank the developers of SysPol and the developers of the evaluated Pharo systems for accepting to participate in our study.

References

[1] D. L. Parnas, “On the criteria to be used in decomposing systems into modules.” *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972.

[2] B. S. Mitchell and S. Mancoridis, “On the automatic modularization of software systems using the Bunch tool,” *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 193–208, 2006.

[3] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[4] N. Anquetil, C. Fourrier, and T. C. Lethbridge, “Experiments with clustering as a software remodularization method,” in *6th Working Conference on Reverse Engineering (WCRE)*, 1999, pp. 235–255.

[5] G. Santos, M. T. Valente, and N. Anquetil, “Remodularization analysis using semantic clustering,” in *1st CSMR-WCRE Software Evolution Week*, 2014, pp. 224–233.

[6] A. Kuhn, S. Ducasse, and T. Girba, “Enriching reverse engineering with semantic clustering,” in *12th Working Conference on Reverse Engineering (WCRE)*, 2005, pp. 133–142.

[7] A. Kuhn, S. Ducasse, and T. Gírba, “Semantic clustering: Identifying topics in source code,” *Information and Software Technology*, vol. 49, no. 3, pp. 230–243, 2007.

[8] G. Bavota, M. Gethers, R. Oliveto, D. Poshyvanyk, and A. de Lucia, “Improving software modularization via automated analysis of latent topics and dependencies,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 1, pp. 1–33, 2014.

[9] L. L. Silva, M. T. Valente, and M. Maia, “Assessing modularity using co-change clusters,” in *13th International Conference on Modularity*, 2014, pp. 49–60.

[10] ——, “Co-change clusters: Extraction and application on assessing software modularity,” *Transactions on Aspect-Oriented Software Development (TAOSD)*, pp. 1–37, 2015.

[11] T. Ball, J. K. A. A. Porter, and H. P. Siy, “If your version control system could talk ...” in *ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

[12] D. Beyer and A. Noack, “Clustering software artifacts based on frequent common changes,” in *13th International Workshop on Program Comprehension (IWPC)*, 2005, pp. 259–268.

[13] T. Zimmermann, P. Weissgerber, S. Diehl, and A. Zeller, “Mining version histories to guide software changes,” *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, 2005.

[14] S. Ducasse, T. Gîrba, and A. Kuhn, “Distribution map,” in *22nd IEEE International Conference on Software Maintenance (ICSM)*, 2006, pp. 203–212.

[15] C. Couto, P. Pires, M. T. Valente, R. Bigonha, and N. Anquetil, “Predicting software defects with causality tests,” *Journal of Systems and Software*, pp. 1–38, 2014.

[16] G. Karypis, E.-H. S. Han, and V. Kumar, “Chameleon: hierarchical clustering using dynamic modeling,” *Computer*, vol. 32, no. 8, pp. 68–75, 1999.

[17] R. Marinescu, “Detection strategies: metrics-based rules for detecting design flaws,” in *20th IEEE International Conference on Software Maintenance (ICSM)*, 2004, pp. 350–359.

[18] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2006.

[19] O. Nierstrasz, S. Ducasse, and D. Pollet, *Pharo by Example*. Square Bracket Associates, 2010.

[20] L. L. Silva, D. Felix, M. T. Valente, and M. Maia, “ModularityCheck: A tool for assessing modularity using co-change clusters,” in *5th Brazilian Conference on Software: Theory and Practice*, 2014, pp. 1–8.

[21] U. Flick, *An Introduction to Qualitative Research*. SAGE, 2009.

[22] J. Corbin and A. Strauss, “Grounded theory research: Procedures, canons, and evaluative criteria,” *Qualitative Sociology*, vol. 13, no. 1, pp. 3–21, 1990.

[23] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, “Aspect-oriented programming,” in *11th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 1241. Springer Verlag, 1997, pp. 220–242.

[24] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An overview of AspectJ,” in *15th European Conference on Object-Oriented Programming (ECOOP)*, ser. LNCS, vol. 2072. Springer Verlag, 2001, pp. 327–355.

[25] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr., “N degrees of separation: Multi-dimensional separation of concerns,” in *21st International Conference on Software Engineering (ICSE)*, 1999, pp. 107–119.

[26] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Prentice Hall, 2003.

[27] S. Apel and C. Kästner, “Virtual separation of concerns - A second chance for preprocessors,” *Journal of Object Technology*, vol. 8, no. 6, pp. 59–78, 2009.

[28] C. Kästner, S. Apel, and M. Kuhlemann, “Granularity in software product lines,” in *30th International Conference on Software Engineering (ICSE)*, 2008, pp. 311–320.

[29] E. Murphy-Hill, C. Parnin, and A. P. Black, “How we refactor, and how we know it,” in *31st International Conference on Software Engineering (ICSE)*, 2009, pp. 287–297.

[30] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, “Template-based reconstruction of complex refactorings,” in *IEEE International Conference on Software Maintenance (ICSM)*, 2010, pp. 1–10.

[31] N. Tsantalis, V. Guana, E. Stroulia, and A. Hindle, “A multidimensional empirical study on refactoring activity,” in *Conference of the Center for Advanced Studies on Collaborative Research (CASCON)*, 2013, pp. 132–146.

[32] M. P. Robillard and B. Dagenais, “Recommending change clusters to support software investigation: An empirical study,” *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 22, no. 3, pp. 143–164, 2010.

[33] M. D’Ambros, M. Lanza, and M. Lungu, “Visualizing co-change information with the evolution radar,” *IEEE Transactions on Software Engineering*, vol. 35, no. 5, pp. 720–735, 2009.

[34] A. Alali, B. Bartman, C. D. Newman, and J. I. Maletic, “A preliminary investigation of using age and distance measures in the detection of evolutionary couplings,” in *10th Working Conference on Mining Software Repositories (MSR)*, 2013, pp. 169–172.

[35] G. A. Oliva, F. W. Santana, M. A. Gerosa, and C. R. B. de Souza, “Towards a classification of logical dependencies origins: a case study,” in *12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution (EVOL/IWPSE)*, 2011, pp. 31–40.

[36] M. D’Ambros, M. Lanza, and R. Robbes, “On the relationship between change coupling and software defects,” in *16th Working Conference on Reverse Engineering (WCRE)*, 2009, pp. 135–144.

[37] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. de Lucia, and D. Poshyvanyk, “Detecting bad smells in source code using change history information,” in *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2013, pp. 11–15.

[38] G. Bavota, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, “An empirical study on the developers’ perception of software coupling,” in *35th International Conference on Software Engineering (ICSE)*, 2013, pp. 692–701.

[39] A. Vanya, L. Hofland, S. Klusener, P. van de Laar, and H. van Vliet, “Assessing software archives with evolutionary clusters,” in *16th IEEE International Conference on Program Comprehension (ICPC)*, 2008, pp. 192–201.

[40] F. Beck and S. Diehl, “Evaluating the impact of software evolution on software clustering,” in *17th Working Conference on Reverse Engineering (WCRE)*, 2010, pp. 99–108.

[41] ——, “On the impact of software evolution on software clustering,” *Empirical Software Engineering*, vol. 18, no. 5, pp. 970–1004, 2013.

How Do Developers React to API Evolution? The Pharo Ecosystem Case

André Hora*†, Romain Robbes‡, Nicolas Anquetil†, Anne Etien†, Stéphane Ducasse†, Marco Tulio Valente*
*ASERG Group, Department of Computer Science (DCC)
Federal University of Minas Gerais, Brazil
{hora, mtov}@dcc.ufmg.br
†RMod team, Inria Lille Nord Europe
University of Lille, CRIStAL, UMR 9189, Villeneuve d'Ascq, France
{firstName.lastName}@inria.fr
‡PLEIAD Lab, Department of Computer Science (DCC)
University of Chile, Santiago, Chile
rrobbes@dcc.uchile.cl

***Abstract*—Software engineering research now considers that no system is an island, but it is part of an ecosystem involving other systems, developers, users, hardware, ... When one system (*e.g.,* a framework) evolves, its clients often need to adapt. Client developers might need to adapt to functionalities, client systems might need to be adapted to a new API, client users might need to adapt to a new User Interface. The consequences of such changes are yet unclear, what proportion of the ecosystem might be expected to react, how long might it take for a change to diffuse in the ecosystem, do all clients react in the same way? This paper reports on an exploratory study aimed at observing API evolution and its impact on a large-scale software ecosystem, Pharo, which has about 3,600 distinct systems, more than 2,800 contributors, and six years of evolution. We analyze 118 API changes and answer research questions regarding the magnitude, duration, extension, and consistency of such changes in the ecosystem. The results of this study help to characterize the impact of API evolution in large software ecosystems, and provide the basis to better understand how such impact can be alleviated.**

I. Introduction

As frameworks evolve, client systems often need to adapt their source code to use the updated API. To facilitate this time-consuming task, frameworks should be backward-compatible and include deprecated methods. In practice, researchers have found that frameworks are backward-incompatible [1] and deprecation messages are often missing [2]. To deal with these problems, some approaches have been developed to help client developers. This can be done, for example, with the support of specialized IDEs [3], the help of experts [4], or the inference of change rules [1], [5], [6], [7], [8].

Commonly, these approaches are evaluated on small-scale case studies. In practice, many software systems are part of a larger software ecosystem, which often exists in organizations, or open-source communities [9]. In this context, it is hard to predict the real impact of API evolution. For example, in ecosystems, API deprecation may affect hundreds of clients, with several of these clients staying in an inconsistent state for long periods of time or do not reacting at all [2]. This suggests that the impact of API evolution may be large and sometimes unknown; in this context, managing API evolution is a complex and risky task [10].

To support developers to better understand the real impact of API evolution and how it could be alleviated, software ecosystems should also be studied. In that respect, a first large-scale study was performed by one of the authors of this paper, Robbes *et al.* [2], to verify the impact of deprecated APIs on a software ecosystem. However, API evolution is not restricted to deprecation. It may imply, for example, a better API design that improves code legibility, portability, performance, security, etc. But are client developers aware of such evolving APIs? How frequent and how broad is the impact on clients? The aforementioned study analyzes the adoption of a specific group of changes, methods explicitly annotated as deprecated. But this introduces a bias as people will probably notice more readily changes documented and checked by the compiler (explicit deprecation) than changes not advertised. Therefore, there is still space for analyzing the adoption of more generic API changes (not explicitly marked as deprecated).

In this paper, we analyze the impact of API changes, not related to explicit API deprecation, on client systems. We set out to discover (i) to what extent API changes propagate to client systems, and (ii) to what extent client developers are aware of these changes. Our goal is to better understand, at the ecosystem level, to what extent client developers are affected by the evolution of APIs, and to reason about how it could be alleviated. Thus, we investigate the following research questions to support our study:

- RQ1 (Magnitude): How many systems react to API changes in an ecosystem and how many developers are involved?
- RQ2 (Duration): How long does it take for systems to react to API changes?
- RQ3 (Extension): Do all the systems in an ecosystem react to API changes?
- RQ4 (Consistency): Do systems react to an API change in the same way?

In this study we cover the Pharo[1] software ecosystem, which has about 3,600 distinct systems, more than 2,800 contributors and six years of evolution, and we analyze 118 API changes. We also compare our results with the Robbes *et al.* study on API deprecation on Pharo [2] to better understand how these two types of API evolution affect client systems.

The contributions of this paper are summarized as follows:

1) We provide a large-scale study, at the ecosystem level, to better understand to what extent client developers are impacted by API changes that are not marked as deprecated.
2) We provide a comparison between our results and the ones of the previous API deprecation study [2].

Structure of the paper: In Section II, we present the API changes considered in this study. We describe our experiment design in Section III. We present and discuss the experiment results in Section IV. We present the implications of our study in Section V, and the threats to the validity in Section VI. Finally, we present related work in Section VII, and we conclude the paper in Section VIII.

II. API Changes

A. Definition

In this work, we focus on API changes related to method replacements and improvements, following the line studied by several researches in the context of framework migration [1], [5], [6], [7], [11], [12], [13], [8]. Next, we define and provide examples on the two types of API changes considered in this paper.

Method replacements: In this type of API change, one or more methods in the old release are replaced by one or more methods in the new release. For example, in a one-to-one mapping, the method LineConnection.end() was replaced by LineConnection.getEndConnector() from JHotDraw 5.2 to 5.3 [1]. In another case, in a one-to-many mapping, the method CutCommand(DrawingView) was replaced by CutCommand(Alignment, DrawingEditor) *and* UndoableCommand(Command) [1]. In both examples, the removed methods have not been deprecated; they were simply dropped, causing clients to fail.

Method improvements: In this type of API change, one (or more) method in the old release is improved, producing one (or more) new method in the new release. For example, in Apache Ant, the method to close files was improved to centralize the knowledge on closing files [13], producing a one-to-one mapping where calls to InputStream.close() should be replaced by FileUtils.close(InputStream). In this case, both solutions to close files are available in the new release, *i.e.,* both methods can be used. However, the latter is the suggested one in order to improve maintenance. In the Moose platform[2], a convention states that calls to MooseModel.root() and MooseModel.add(MooseModel) should be replaced by MooseModel.install() when adding models. Again, all the methods are available to be used, but MooseModel.install() is the suggested one to improve code legibility.[3]

These types of API changes are likely to occur during framework evolution, thus their detection is helpful for client developers. Recently, researchers proposed techniques to automatically infer rules that describe such API changes [1], [6], [7], [11], [12], [13]. In this study, we adopt our previous approach [7] in order to detect API changes, which covers both aforementioned cases.

B. Detecting API Changes

In our approach, API changes are automatically produced by applying the *association rules* data mining technique [14] on the set of method call changes between two versions of one method. We produce rules in the format old-call → new-call, indicating that the old call should be replaced by the new one. Each rule has a *support* and *confidence*, indicating the frequency that the rule occurs in the set of analyzed changes and a level of confidence. We also use some heuristics to filter rules that are more likely to represent relevant API changes; for example, rules can be ranked by confidence, support or occurrences in different revisions. Please, refer to our previous study [7] for an in-depth description about how the rules are generated.

III. Experiment Design

A. Selecting the Clients: Pharo Ecosystem

For this study, we select the ecosystem built around the Pharo open-source development community. Our analysis included six years of evolution (from 2008 to 2013) with 3,588 systems and 2,874 contributors. There are two factors influencing this choice. First, the ecosystem is concentrated on two repositories, SqueakSource and SmalltalkHub, which gives us a clear inclusion criterion. Second, we are interested in comparing our results with the previous work of Robbes *et al.* [2]; using the same ecosystem facilitates this comparison.

The Pharo ecosystem: Pharo is an open-source, Smalltalk-inspired, dynamically typed language and environment. It is currently used in many industrial and research projects.[4] The Pharo ecosystem has several important projects. For example, Seaside[5] is a web-development framework, a competitor for Ruby on Rails as the framework of choice for rapid web prototyping. Moose is an open-source platform for software and data analysis. Phratch, a visual and educational programming language, is a port of Scratch to the Pharo platform. Many other projects are developed in Pharo and hosted in the SqueakSource or SmalltalkHub repositories.

The SqueakSource and SmalltalkHub repositories: SqueakSource and SmalltalkHub repositories are the basis for the software ecosystem that the Pharo community have built over the years. They are the *de facto* platform for sharing open-source code for this community offering a nearly complete view of

[1] http://www.pharo.org, verified on 25/03/2015

[2] http://www.moosetechnology.org, verified on 25/03/2015

[3] See the mailing discussion in: http://goo.gl/Ul3Sha, verified on 25/03/2015

[4] http://consortium.pharo.org, verified on 25/03/2015

[5] http://www.seaside.st, verified on 25/03/2015

the Pharo software ecosystem. The SqueakSource repository is also partially used by the Squeak open-source development community. SmalltalkHub was created after SqueakSource by the Pharo community to be a more scalable and stable repository. As a consequence, many Pharo projects migrated from SqueakSource to SmalltalkHub, and nowadays, new Pharo projects are concentrated in SmalltalkHub.

Transition between SqueakSource and SmalltalkHub: We detected that 211 projects migrated from SqueakSource to SmalltalkHub while keeping the same name and copying the full source code history. We count these projects only once: we only kept the projects hosted in SmalltalkHub, which hosts the version under development and the full code history.

In theory, the migration was done automatically by a script provided by SmalltalkHub developers, thus keeping the metadata such as project name. However, to increase our confidence in the data, we calculated the Levenshtein distance between the projects in each repository to detect cases of similar but not equal project names. We detected that 93 systems had similar names (*i.e.,* Levenshtein distance = 1). By manually analyzing each of these systems, we detected that most of them are in fact distinct projects, *e.g.,* “AST” (from abstract syntax tree) and “rST” (from remote smalltalk). However, 14 systems are the same project with a slightly different name, *e.g.,* “Keymapping” in SqueakSource was renamed to “Keymappings” in SmalltalkHub. In these cases, again, we only kept the projects hosted in SmalltalkHub, as they represent the version under development and include the full source code history.

B. Selecting the Frameworks: Pharo Core

Pharo core frameworks: The frameworks from which we applied associations rules mining to extract the API changes come from the Pharo core. They provide a set of APIs, including collections, files, sockets, unit tests, streams, exceptions, graphical interfaces, etc. (they are Pharo’s equivalent to Java’s JDK). Such frameworks are available to be used by any system in the ecosystem.

We took into account all the versions of Pharo core since its initial release, *i.e.,* versions 1.0, 1.4, 2.0, and 3.0. Table I shows the number of classes and lines of code in each version. The major development effort between versions 1.0 and 1.4 was focused on removing outdated code that came from Squeak, the Smalltalk dialect Pharo is a fork of, explaining the drop in number of classes and lines of code.

TABLE I
PHARO CORE VERSIONS SIZE.

Version	1.0	1.4	2.0	3.0
Classes	3,378	3,038	3,345	4,268
KLOC	447	358	408	483

Generating a list of API changes: We adopted our previous approach [7], described in Section II, to generate a list of API changes. We set out to produce rules with a minimum support of 5, and a minimum confidence of 50%. The minimum support at 5 states that a rule has a relevant amount of occurrences in the framework, and the minimum confidence at 50% yields a good level of confidence (as an example, Schäfer *et al.* [12] use a confidence of 33% in their approach to detect evolution rules). Moreover, the thresholds reduce the number of rules to be manually analyzed.

This process produced 344 rules that were manually analyzed with the support of documentation and code examples to filter out incorrect or noisy ones. For example, the rule SortedCollection.new() → OrderedCollection.new() (*i.e.,* Java’s equivalent to SortedSet and List, respectively) came out from a specific refactoring on a specific framework but clearly we cannot generalize this change for clients, so this rule was discarded. This filtering produced 148 rules.

Filtering the list of API changes by removing deprecation: Naturally, some of the API changes inferred by our approach are related to API deprecation. As such cases were studied by Robbes *et al.* [2], they are out of the scope of this paper. For this purpose, we first extracted all methods marked as deprecated found in the analyzed evolution of Pharo core; this produced 1,015 API deprecation. By discarding the API changes related to API deprecation, *our final list includes 118 API changes*.

From these API changes, 59 are about method suggestion (*i.e.,* both methods are available to be used by the client; *cf.* Section II) and 59 are about method replacement (*i.e.,* the old method is removed, so it is not available to be used). Furthermore, 10 out of the 118 API changes involved the evolution of internal APIs of the frameworks which, in theory, should not affect client systems. By internal API, we mean a public component that should only be used internally by the framework, *i.e.,* not by client systems. For instance, in Eclipse, the packages named with *internal* include public classes that is not part of the API provided to the clients [15], [16].

In Table II, we present some examples of API changes. The first API change improves code legibility, as it replaces two method calls by a single, clearer one. The second example replaces a method with a more robust one, that allows one to provide a different behavior when the intersection is empty. The third is an usage convention: Pharo advises not to use Object.log() methods, to avoid problems with the *log* function. Finally, the fourth one represents a class and method replacement due to a large refactoring: ClassOrganizer.default() does not exist anymore; ideally, it should have been marked as deprecated.

TABLE II
EXAMPLE OF API CHANGES.

id	API change (old-call → new-call)
1	ProtoObject.isNil() *and* Boolean.ifTrue(*) → ProtoObject.ifNil(*)
2	Rectangle.intersect(*) → Rectangle.intersectIfNone(*,*)
3	Object.logCr(*) → Object.traceCr(*)
4	ClassOrganizer.default() → Protocol.unclassified()

Assessing reactions of API changes in the ecosystem: When analyzing the reaction of the API changes in the ecosystem, we do not consider the frameworks from which we discovered the API changes, but only the client systems hosted at SqueakSource and SmalltalkHub, as described in Subsection III-A. To verify a reaction to API change in these systems, we need to detect when the change was available. We consider that an API change is available to client systems from the moment it was discovered in the framework. All commits in the client systems after this moment that remove a method call from the old API and add a call to the new API are considered to be reactions to the API change.

Notice that, in this study, we assess commits in the ecosystem that applied the prescribed API change (*i.e.,* the removals and additions of method call according to the rule we inferred). In the API deprecation study [2], the authors were primarily interested in the removals of calls to deprecated methods, but did not consider their replacement.

IV. Results

A. Magnitude of Change Propagation

RQ1. How many systems react to the API changes in an ecosystem and how many developers are involved?

1) Results: In this research question we verify the frequency of reactions and we quantify them in number of systems, methods, and developers.

Frequency of reactions: From the 118 API changes, 62 (53%) caused reactions in at least one system in the ecosystem. Moreover, from these API changes, 5 are internal, meaning client developers also use internal parts of frameworks to access functionalities not available in the public interfaces. We see in the next research questions that many systems take time to react to API changes. Hence, some API changes may not have been applied by all systems yet.

These reactions involved 178 (5%) client systems and 134 (4.7%) distinct developers. We show the distribution of such data (*i.e.,* the API changes that caused change propagation) in the box plots shown in Figure 1.

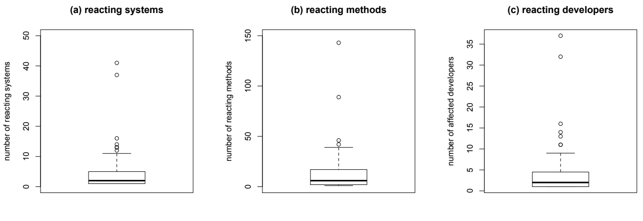


Fig. 1. Box plots for (a) systems, (b) methods, and (c) developers reacting to API changes.

Reacting systems: Figure 1a shows the distribution of reacting systems: the 1st quartile is 1 (bottom of the box), the median is 2 (middle of the box), the 3rd quartile is 5 (*i.e.,* 25% of the API changes cause reactions in 5 or more systems, forming the top of the box in the box plot), and the maximum is 11 (*i.e.,* it marks the highest number of reacting systems that is not considered an outlier, forming the top whisker of the box). The API change isNil().ifTrue(*) → ifNil(*) caused the largest reaction, 41 systems; this change is depicted as the dot at the top of the box plot in Figure 1a (in a box plot all outliers are shown as dots).

Reacting methods: For methods (Figure 1b), the 1st quartile is 2, the median is 6, the 3rd quartile is 17, and the maximum is 39. These results show that some systems reacted several times to the same API change: the median system reaction is 2 while the median method reaction is 6. For example, the API change isNil().ifTrue(*) → ifNil(*) caused reaction in 41 systems, but 89 methods.

Reacting developers: The number of developers impacted by API changes is shown in Figure 1c, as the number of commit authors that react to the API changes. In this case, the 1st quartile is 1, the median is 2, the 3rd quartile is 5, and the maximum is 11. The median at 2 shows that many change propagations involve few developers while the 3rd quartile at 5 shows that some of them involve several developers. The API change isNil().ifTrue(*) → ifNil(*), for example, involved a large number of developers (37). Overall, the distribution of the number of developers involved in the change is similar to the number of systems, implying that it is common that only one developer from a given system reacts to the API changes.

2) Comparison with API deprecation: Our magnitude results are different when we compare to explicit API deprecation. In the previous study there was a higher level of reaction to API changes. In the present study, 62 API changes caused reactions while in the API deprecation case, 93 deprecated entities caused reactions, *i.e.,* 50% more. The median of reactions in our study is 2, whereas it is 5 in the case of API deprecation. This is expected, since deprecated methods produce warning messages to developers while in the case of API changes no warning is produced.

Another difference relies on the number of developers involved in the reaction. In our study, it is common that *one* developer reacts to the API changes while in the API deprecation study it is more common that *several* developers of the same system react. One possible explanation is again that changes involving deprecated methods are usually accompanied by warnings, thus they can be performed by any client developer. In contrast, the API changes evaluated in this work can only be performed by developers that previously know them. This confirms that reacting to an API change is not trivial, thus, sharing this information among developers is important.

These results compared to the previous study reinforce the need to explicitly annotate API deprecation. More people gain knowledge of the changes and more systems/methods are adapted.

B. Duration of Change Propagation

RQ2. How long does it take for systems to react to API changes?

1) Results: A quick reaction to API changes is desirable for clients to benefit sooner from the new API. Next, we evaluate the reaction time of the ecosystem.

We calculate the reaction time to an API change as the number of days between its creation date (*i.e.,* the first time it was detected in the framework) and the first reaction in the ecosystem. As shown in Figure 2a, the minimum is 0 days, the 1st quartile is 5 days, the median is 34 days, the 3rd quartile is 110 days. The 1st quartile at 5 days shows that some API changes see a reaction in few days: this is possible if developers work both on frameworks and on client systems, or coordinate API evolution via mailing lists [17].

In contrast, the median at about 34 days and the 3rd quartile at 110 days indicate that some API changes take a long time to be applied. In fact, as Pharo is a dynamically typed language, some API changes will only appear for developers at runtime which can explain the long time frame.

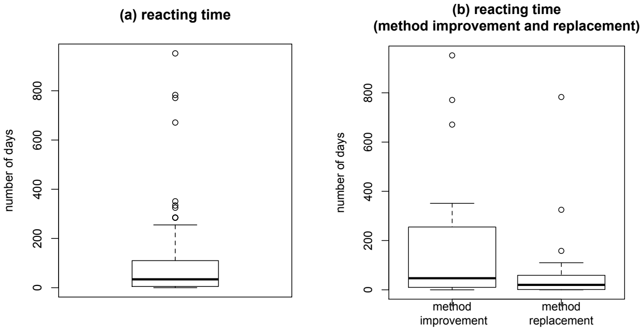


Fig. 2. Box plots for reaction time of (a) all API changes and (b) separated by method improvement and replacement, both in number of days.

In addition, we analyze the reaction time considering the two categories of API changes, method improvement and replacement, as shown in Figure 2b. For the API changes about improvement, the 1st quartile is 10 days, the median is 47 days, the 3rd quartile is 255 days, and the maximum is 351 days. In contrast, for the API changes about replacement, the 1st quartile is 1 days, the median is 20 days, the 3rd quartile is 59 days, and the maximum is 110 days.

Therefore, the reaction time for the API changes due to method improvements is longer than the ones about replacement, implying that the former is harder to be detected by client developers. This is explained by the fact that in the case of method improvements, the old method is still valid, so client developers are not forced to update their code. However, they would benefit if such API changes are suggested to them beforehand. In practice, many developers are simply not aware.

In summary, the results show that the reaction to API changes is not quick. Client developers need some time to discover and apply the changes; this time is longer for API changes related to method improvements.

2) Comparison with API deprecation: The reaction time of the API changes considered in our study is longer when we compare to the reaction time of API deprecation. In the API deprecation case, the 1st quartile is 0 days, the median is 14 days, and the 3rd quartile is 90 days (compared to 5, 34 and 110 days, respectively, in our API changes). Clearly, the reaction to deprecated APIs is faster than in the case of API changes. This is facilitated by the warning messages produced by deprecated methods.

If we compare method improvement in this study with method explicit deprecation in the previous study, we see it takes more than 3 times longer (47 days against 14) to react without explicit deprecation. To complement the result in section IV-A, explicit deprecation allows more developers to know of the change and more quickly.

C. Extension of Change Propagation

RQ3. Do all the systems in an ecosystem react to API changes?

1) Results: In the previous subsection we concluded that some systems take a long time to react to an API change. Here, we see that other systems do not react at all. To determine whether all systems react to the API changes, we investigate all the systems that are potentially affected by them, *i.e.,* that feature calls to the old API.

Table III shows that 2,188 (61%) client systems are potentially affected by the API changes, involving 1,579 (55%) distinct developers. Moreover, we detected that 112 API changes (out of 118), including the 10 internal API changes, potentially affected systems in the ecosystem. In the rest of this subsection, we analyze the distribution of such data.

TABLE III
EXTENSION OF CHANGE PROPAGATION.

Number of affected...		
Systems	**Methods**	**Developers**
2,188	107,549	1,579

Affected systems and methods: Figures 3a and 3b show the distribution of systems and methods affected by API changes in the ecosystem. We note that the number of affected systems and methods are much higher than those that actually react to API changes (as shown in Figure 1). The 1st quartile of affected systems is 15 compared to only 1 system reacting (methods: 59 compared to 2). The median of affected systems by an API change is 56.5 compared to only 2 systems reacting to it (methods: 253 compared to 2). The 3rd quartile of affected systems is 154.5 compared to 5 systems reacting (methods: 744.5 compared to 17).

Relative analysis: The relative analysis of reacting and affected systems produces a better overview of the impact. In that respect, comparing the ratio of reacting systems to the ratio of affected systems gives the distribution shown in Figure 4a. It shows that a very low number of systems react: the median is 0%, the 3rd quartile is 3%, the maximum is 7%. We investigate possible reasons for this low amount of reactions.

In an ecosystem, a possibly large amount of the systems may be stagnant, or even dead [2]. Thus, we first investigate the hypothesis in which systems that did not react either died before the change propagation started or were stagnant. A system is dead if there are no commits to its repository

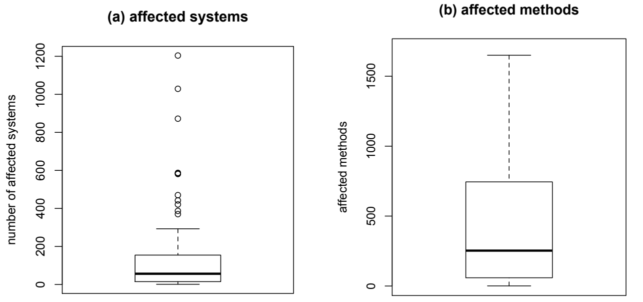


Fig. 3. Box plots for (a) systems and (b) methods affected by API changes.

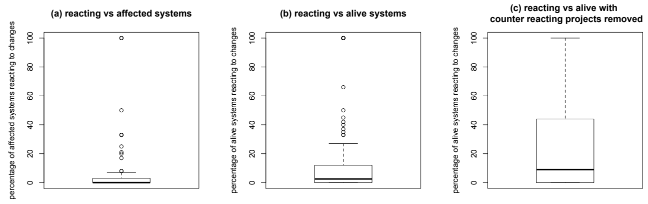


Fig. 4. Box plots for ratios of: (a) reacting affected systems; (b) reacting alive systems; and (c) reacting alive systems, removing counter reactions.

after the API change that triggered the change propagation. A system is stagnant if a minimal number of commits (less than 10) was performed after the API change. Thus, removing dead or stagnant systems (*i.e.,* keeping alive systems only) produces the distribution shown in Figure 4b: the median is 2.5%, the 3rd quartile is 12%, and the maximum is 27%.

A second reason why a system would not react to a change is when it is using another version of the framework, one in which the API did not change. This may occur when a system does not have the manpower to keep up-to-date with the evolution and freezes its relationship with a version that works [2]. To estimate this effect, we measure the number of systems that actually add more calls to the old API change, *i.e.,* they are counter reacting to the API evolution. Thus, removing these systems from the alive ones gives the distribution shown in Figure 4c: the median is 9%, the 3rd quartile is 44%, and the maximum is 100%. This new distribution reveals that many systems do not update to the new framework versions, even after filtering out dead, stagnant, and counter-reacting systems. As a result, the effort of migrating to newer versions becomes more expensive over time due to change accumulation.

2) Comparison with API deprecation: The presented reacting ratios are very different when compared to the API deprecation study. For the *ratio of reacting and affected systems*, the 1st quartile is 13%, the median is 20%, and the 3rd quartile is 31% in the API deprecation case (compared to 0%, 3% and 7%, respectively, in our API changes), which confirms the difference between both types of API evolution. These percentages increase in the other ratio comparisons. For the *ratio of reacting and alive without counter reacting systems*, the 1st quartile is 50%, the median is 66%, and the 3rd quartile is 75% for API deprecation (compared to 0%, 9% and 44%, respectively, in our API changes). Clearly, client systems react more to API deprecation. However, our results show that reactions to API changes are not irrelevant.

D. Consistency of Change Propagation

RQ4. Do systems react to an API change in the same way?

1) Results: The API changes analyzed in the previous research questions described the main way the analyzed frameworks evolved. However, some API changes may allow multiple replacements [2]. For example, Table IV shows three examples of API changes extracted from the analyzed frameworks, and their reactions by the ecosystem.

TABLE IV
EXAMPLES OF API CHANGES; THE NUMBERS SHOW THE CONFIDENCE OF THE REPLACEMENT IN THE ECOSYSTEM.

Old call (framework)	**New call**	
	Framework	**Ecosystem**
doSilently()	suspendAllWhile()	80% suspendAllWhile()
Pref.menuFont()	Fonts.menuFont()	40% Fonts.menuFont() 40% ECPref.menuFont()
HashAlgorithm.new()	SHA1.new()	63% HashFunction.new() 30% SHA1.new()

The first API change, doSilently() → suspendAllWhile(), is mostly followed by the ecosystem, presenting a confidence of 80% (*i.e.,* 80% of the commits that removed the old call also added the new call). For the second API change, Preferences.standardMenuFont() → StandardFonts.menuFont(), the ecosystem reacts with two possible replacements, both with confidence of 40%. For the third API change, SecureHashAlgorithm.new() → SHA1.new(), the ecosystem also reacts with two possible replacements: a main one with confidence of 63% and an alternative one with 30%.[6] Notice that, in this case, the main replacement is not the one extracted from the analyzed framework, *i.e.,* it is HashFunction.new() instead of SHA1.new().

To better understand such cases, we analyze the consistency of the API changes by verifying the reactions of the ecosystem. *Consistency of main and alternative replacements in the ecosystem:* Figure 5a presents the confidence distribution of the main and alternative replacements in the ecosystem. For the main replacement, the 1st quartile is 36%, the median is 60%, and the 3rd quartile is 100%. For the alternative replacement, the 1st quartile is 20%, the median is 25%, and the 3rd quartile is 31%. These results show that alternative replacements are found in the ecosystem (such as the second and third examples in Table IV), but with less confidence than the main ones. Thus, alternative replacements explain a minority of the cases where affected systems do not react to the prescribed API changes.

[6] Main and alternative replacements of API changes in the ecosystem are determined by verifying how the ecosystem replaces the old calls. This is done by applying our approach described in Section II in the ecosystem itself.

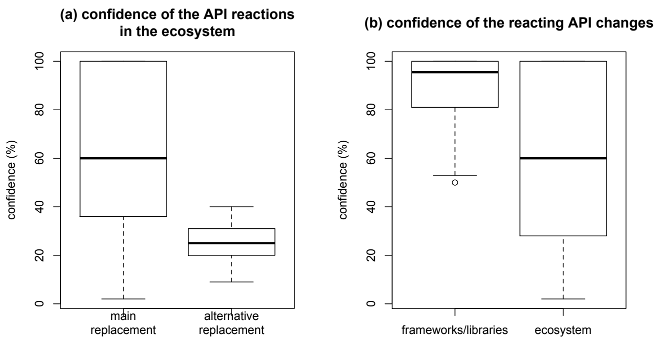


Fig. 5. Box plots for the confidence of (a) the reaction in the ecosystem (main and alternative replacements) and (b) the reacting API changes (frameworks/libraries and ecosystem).

Consistency of API changes in the frameworks and in the ecosystem: Figure 5b compares the confidence distribution of the 62 reacting API changes both in the analyzed frameworks and in the ecosystem. In the analyzed frameworks, the minimum is 53%, the 1st quartile is 81%, the median is 95%, and the 3rd quartile is 100% (recall that a minimum confidence of 50% was adopted to generate the API changes). In the ecosystem, for the *same* API changes, the minimum is 2%, the 1st quartile is 28%, the median is 60%, and the 3rd quartile is 100%.

There is a difference in the confidence: the API changes are more consistently followed by the frameworks than by the ecosystem. This suggests that many replacements are not resolved in a uniform manner in the ecosystem: client developers may adopt other replacements in addition to the prescribed ones (such as the second and third examples in Table IV); method calls may be simply dropped, so they disappear without replacements; and developers may replace the old call by local solutions. Thus, this result provides evidence that API changes can be more confidently extracted from frameworks than from clients (*i.e.,* the ecosystem).

2) Comparison with API deprecation: For the main replacement in the API deprecation study, the confidence of the 1st quartile is 46%, the median is 60%, and the 3rd quartile is 80% (compared to 36%, 60%, and 100%, respectively, in our study). Note that the distribution of the main replacement is mostly equivalent in both cases. In fact, in the case of API deprecation, it is common the adoption of alternative replacements and home-grown solutions due to empty warning messages.

V. Summary and Implications

In summary, our study shows that 53% (62 out of 118) of the analyzed API changes caused reaction in only 5% of the systems and affected 4.7% of the developers. Overall, the reaction time of API changes is not quick (median 34 days). Client developers, naturally, need some time to discover and apply the new API; this time is even longer in the case of method improvement. In contrast, a large amount of systems are potentially affected by the API changes: 61% of the systems and 55% of the developers. In fact, the number of affected systems are much higher than those that actually react to API changes. As a result, the effort of porting to newer versions becomes more expensive due to change accumulation.

The answers to our research questions allow us to formulate the following implications of our study.

Deprecation mechanisms should be more adopted: Half of the API changes analyzed in this work (59 out of 118) are about method replacements. It means that such API changes are probably missing to use deprecation mechanisms. Ideally, they should have been marked as deprecated by the framework developers. In fact, in large frameworks, developers may not know whether their code is used by clients: this may cause a growth [2] or a lack in the use of deprecation [1], [15].

In our study, this lack of deprecation was mainly due to large refactorings in the frameworks. For instance, the framework for dealing with files completely changed after Pharo 1.4. As a result, some APIs missed to be marked as deprecated; *e.g.,* in the Moose migration to Pharo 3.0, a developer noticed this issue and commented[7]: "In FileSystem, ensureDirectory() was renamed to ensureCreateDirectory() without a deprecation", the framework developer then answered: "Fill up a bug entry and we will add this deprecation. Good catch". In fact, for such cases, asking in Question and Answer sites[8] or mailing lists[9] is the current alternative for client developers.

Based on these results we conclude the following:

Many deprecation opportunities are missed by the developers (we found at least 59 instances in our study). Recommenders can be built to remind API developers about these missed opportunities.

Client developers use internal parts of frameworks: Internal APIs are unstable and unsupported interfaces [16], so they should not be used by clients. However, all the internal APIs (*i.e.,* 10 cases) analyzed in this work are used by clients. From such, 5 caused the clients to react as in the frameworks. Thus, our results reinforce (at large-scale and ecosystem level) previous studies [11], [18], [16], showing that client systems use internal parts of frameworks to access functionalities not available in the public interfaces for a variety of reasons.

Based on these results we conclude the following:

Internal APIs are sometimes used by client developers in the ecosystem under analysis. Recommenders can be build to help API developers identify often used internal APIs; those are candidates to be public APIs to keep clients using stable and supported interfaces.

Replacements are not resolved in a uniform manner: Many replacements are not resolved uniformly in the ecosystem. Clients may adopt other replacements in addition to the

[7] http://forum.world.st/moving-moose-to-pharo-3-0-td4718927.html, verified on 25/03/2015

[8] Coordination via Question and Answer sites: http://stackoverflow.com/questions/15757529/porting-code-to-pharo-2-0, verified on 25/03/2015

[9] Coordination via mailing lists: http://goo.gl/50q2yZ, http://goo.gl/k9Fl0K, http://goo.gl/SkMORX, verified on 25/03/2015

prescribed ones; method calls may be simply dropped; and developers may replace the old call by local solutions. In this context, some studies propose the extraction of API changes from frameworks (*e.g.,* [11]) other propose the extraction from clients (*e.g.,* [12]).

Based on these results we conclude the following:

> There is no clear agreement on the best extraction source to detect API changes: frameworks or client systems. This study reinforces frameworks as a more reliable source.

Reactions to API changes can be partially automated: As we observed, many systems do not react to the API changes because they are not aware. Moreover, in the case of large client systems, the adaptation may take a long time and is costly if done manually.

Based on these results we conclude the following:

> Most of the API changes that we found in this work can be implemented as rules in static analysis tools such as FindBugs [19], PMD [20], and SmallLint [21]. These rules could help client developers to keep their source code up-to-date with the new APIs.

VI. Threats to Validity

Construct Validity: The construct validity is related to whether the measurement in the study reflects real-world situations. In our study, the main threat is the quality of the data we analyze and the degree of involved manual analysis.

Software ecosystems present some instances of duplication (around 15% of the code [22]), where packages are copied from a repository to another (*e.g.,* a developer keeping a copy of a specific framework version). This may overestimate the number of systems reacting to an API change.

Smalltalk (and Pharo) is a dynamically typed language, so the detection of API change reaction may introduce noise as systems may use unrelated methods with the same name. This means that an API change that uses a common method name makes change propagation hard to be detected. This threat is alleviated by our manual filtering of noisy API changes.

Another factor that alleviates this threat is our focus on specific evolution rules (*i.e.,* a specific replacement of one or more calls by one or more calls). For the first three research questions, we include only commits that are removing an old API *and* adding a new API to detect an API reaction. Requiring these two conditions to be achieved, decreases—or in some cases eliminates—the possibility of noise. For the fourth research question, we require the presence of the methods that contain a call to the old API. In this case, the noise could have been an issue, however, this threat is reduced since we discarded the API changes involved with common methods, *i.e.,* the noisy ones.

We also identify two threats regarding the comparison with the API deprecation study [2]. First, the time interval studied is not the same one: we analyzed the ecosystem evolution in the period from 2008 to 2013 while the API deprecation study analyzed from 2004 to 2011. Second, the way API changes are selected is different: while we deprecation study simply collected the list of API deprecation, we inferred the API changes from commits in source code repository; these API changes were manually validated by the authors of the paper with the support of documentation and code examples to eliminate incorrect and noisy ones. For these reasons, we can not claim that this is an exact comparison. Parts of the differences observed may be due to other factors.

Internal Validity: The internal validity is related to uncontrolled aspects that may affect the experimental results. In our study, the main threat is the possible errors in the implementation of our approach.

Our tool to detect API changes has been (i) used by several members of our laboratory to support their own research on frameworks evolution, and (ii) divulged in the Moose reengineering mailing list, so that developers of this community can use it; thus, we believe that these tasks reduce the risks of this threat.

External Validity: The external validity is related to the possibility to generalize our results. In our study, the main threat is the representativeness of our case studies.

We performed the study on a single ecosystem. It needs to be replicated on other ecosystems in other languages to characterize the phenomenon of change propagation more broadly. Our results are limited to a single community in the context of open-source; closed-source ecosystems, due to differences in the internal processes, may present different characteristics. However, our study detected API change reactions in thousands of client systems, which makes our results more robust.

The Pharo ecosystem is a Smalltalk ecosystem, a dynamically typed programming language. Ecosystems in a statically typed programming language may present differences. In particular, we expect static type checking to reduce the problem of noisy API changes for such ecosystems.

As an alternative to our choice of ecosystem, we could have selected a development community based on a more popular language such as Java or C++. However, this would have presented several disadvantages. First, deciding which systems to include or exclude would have been much more challenging. Second, the potentially very large size of the ecosystem could prove impractical. We consider the size of the Pharo ecosystem as a "sweet spot": with about 3,600 distinct systems and more than 2,800 contributors, it is large enough to be relevant.

VII. Related Work

A. Software Ecosystems Analysis

Software ecosystem is an overloaded term, which has several meanings. There are two principal facets: the first one focuses on the business aspect [23], [24], and the second on the artefact analysis aspect, *i.e.,* on the analysis of multiple, evolving software systems [25], [9], [2]. In this work we use the latter one; we consider an ecosystem to be "*a collection of software projects which are developed and co-evolve in the same environment*" [9]. These software systems have common underlying components, technology, and social norms [25].

Software ecosystems have been studied under a variety of aspects. Jergensen *et al.* [25] study the social aspect of ecosystems by focusing on how developers move between projects in the software ecosystems. The studies of Lungu *et al.* [26] and Bavota *et al.* [10] aim to recover dependencies between the software projects of an ecosystem to support impact analysis. Lungu *et al.* [27] focus on the software ecosystems analysis through interactive visualization and exploration of the systems and their dependencies. Gonzalez-Barahona *et al.* [28] study the Debian Linux distribution to measure its size, dependencies, and commonly used programming languages.

Recently, Mens *et al.* [29] proposed the investigation of similarities between software ecosystems and natural ecosystems found in ecology. In this context, they are studying the GNOME and the CRAN ecosystems to better understand how software ecosystems can benefit from biological ones. German *et al.* [30] also analyze the evolution of the CRAN ecosystem, investigating the growth of the ecosystem, and the differences between core and contributed packages.

In the context of API evolution and ecosystem impact analysis, McDonnell *et al.* [31] investigate API stability and adoption on a small-scale Android ecosystem. In such study, API changes are derived from Android documentation. They have found that Android APIs are evolving fast while client adoption is not catching up with the pace of API evolution. Our study does not rely on documentation but on source code changes to generate the list of APIs to answer different questions. Moreover, our study investigates the impact of API evolution on thousands of distinct systems.

In a large-scale study, Robbes *et al.* [2] investigate the impact of a specific type of API evolution, API deprecation, in an ecosystem that includes more than 2,600 projects; such ecosystem is the same that is used in our work. Our study considers API changes that were not marked as deprecated. Thus, there is no overlap between the changes investigated in our work and the ones investigated by that work. In fact, these studies complement each other to better characterize the phenomenon of change propagation at the ecosystem level.

B. API Evolution Analysis

Many approaches have been developed to support API evolution and reduce the efforts of client developers. Chow and Notkin [4] present an approach where the API developers annotate changed methods with replacement rules that will be used to update client systems. Henkel and Diwan [3] propose CatchUp!, a tool that uses an IDE to capture and replay refactorings related to the API evolution. Hora *et al.* [7], [8] present tools to keep track of API evolution and popularity.

Kim *et al.* [32] automatically infer rules from structural changes. The rules are computed from changes at or above the level of method signatures, *i.e.,* the body of the method is not analyzed. Kim *et al.* [33] propose a tool (LSDiff) to support computing differences between two system versions. In such study, the authors take into account the body of the method to infer rules, improving their previous work [32] where only method signatures were analyzed. Nguyen *et al.* [34] propose LibSync that uses graph-based techniques to help developers migrate from one framework version to another. Using the learned adaptation patterns, the tool recommends locations and update operations for adapting due to API evolution.

Dig and Johnson [15] help developers to better understand the requirements for migration tools. They found that 80% of the changes that break client systems are refactorings. Cossette *et al.* [35] found that, in some cases, API evolution is hard to handle and needs the assistance of an expert.

Some studies address the problem of discovering the mapping of APIs between different platforms that separately evolved. For example, Zhong *et al.* [36] target the mapping between Java and C# APIs while Gokhale *et al.* [37] present the mapping between JavaME and Android APIs.

VIII. Conclusion

This paper presented an empirical study about the impact of API evolution, in the specific case of methods unrelated to API deprecation. The study was done in the context of a large-scale software ecosystem, Pharo, with about 3,600 distinct systems. We analyzed 118 important API changes from frameworks, and we found that 53% impacted other systems. We reiterate the most interesting conclusions from our experiment results:

- API changes can have a large impact on the ecosystem in terms of client systems, methods, and developers. Client developers need some time to discover and apply the new API, and the majority of the systems do not react at all.
- API changes can not be marked as deprecated because framework developers are not aware of their use by clients. Moreover, client developers can use internal APIs to access functionalities not available in the public interfaces.
- Replacements can not be resolved in a uniform manner in the ecosystem. Thus, API changes can be more confidently extracted from frameworks than from clients.
- Most of the analyzed API changes can be implemented as rules in static analysis tools to reduce the adaptation time or the amount of projects that are not aware about a new/better API.
- API changes knowledge can be concentrated in a small amount of developers. API changes and deprecation can present different characteristics, for example, reaction to API changes is slower and less clients react.

As future work, we plan to extend this research to analyze ecosystems based on statically typed languages. Thus, the results presented in our study will enable us to compare reactions of statically and dynamically typed ecosystems to better characterize the phenomenon of change propagation.

Acknowledgment

This research was supported by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico - Brasil), ANR (Agence Nationale de la Recherche - ANR-2010-BLAN-0219-01) and FAPEMIG.

REFERENCES

[1] W. Wu, Y.-G. Gueheneuc, G. Antoniol, and M. Kim, "Aura: a hybrid approach to identify framework evolution," in *International Conference on Software Engineering*, 2010.

[2] R. Robbes, M. Lungu, and D. Röthlisberger, "How do developers react to API deprecation? The case of a smalltalk ecosystem," in *International Symposium on the Foundations of Software Engineering*, 2012.

[3] J. Henkel and A. Diwan, "Catchup!: Capturing and replaying refactorings to support API evolution," in *International Conference on Software Engineering*, 2005.

[4] K. Chow and D. Notkin, "Semi-automatic update of applications in response to library changes," in *International Conference on Software Maintenance*, 1996.

[5] A. Hora, N. Anquetil, S. Ducasse, and S. Allier, "Domain Specific Warnings: Are They Any Better?" in *International Conference on Software Maintenance*, 2012.

[6] S. Meng, X. Wang, L. Zhang, and H. Mei, "A history-based matching approach to identification of framework evolution," in *International Conference on Software Engineering*, 2012.

[7] A. Hora, A. Etien, N. Anquetil, S. Ducasse, and M. T. Valente, "APIEvolutionMiner: Keeping API Evolution under Control," in *Software Evolution Week (European Conference on Software Maintenance and Working Conference on Reverse Engineering)*, 2014.

[8] apiwave, "Discover and track APIs," http://apiwave.com, 2015.

[9] M. Lungu, "Reverse Engineering Software Ecosystems," *PhD thesis, University of Lugano, Switzerland (October 2009)*, 2009.

[10] G. Bavota, G. Canfora, M. D. Penta, R. Oliveto, and S. Panichella, "The evolution of project inter-dependencies in a software ecosystem: the case of Apache," in *International Conference on Software Maintenance*, 2013.

[11] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *International Conference on Software engineering*, 2008.

[12] T. Schäfer, J. Jonas, and M. Mezini, "Mining framework usage changes from instantiation code," in *International Conference on Software engineering*, 2008.

[13] A. Hora, N. Anquetil, S. Ducasse, and M. T. Valente, "Mining System Specific Rules from Change Patterns," in *Working Conference on Reverse Engineering*, 2013.

[14] M. Zaki and W. Meira Jr, "Fundamentals of data mining algorithms," 2012.

[15] D. Dig and R. Johnson, "The role of refactorings in API evolution," in *International Conference on Software Maintenance*, 2005.

[16] J. Businge, A. Serebrenik, and M. G. van den Brand, "Eclipse API usage: the good and the bad," *Software Quality Journal*, 2013.

[17] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz, "A Quantitative Analysis of Developer Information Needs in Software Ecosystems," in *European Conference on Software Architecture Workshops*, 2014.

[18] J. Boulanger and M. Robillard, "Managing concern interfaces," in *International Conference on Software Maintenance*, 2006.

[19] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," in *Object Oriented Programming Systems Languages and Applications*, 2004.

[20] T. Copeland, *PMD Applied*. Centennial Books, 2005.

[21] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object Systems*, vol. 3, 1997.

[22] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *International Conference on Software Engineering*, 2012.

[23] D. G. Messerschmitt and C. Szyperski, "Software ecosystem: understanding an indispensable technology and industry," *MIT Press Books*, vol. 1, 2005.

[24] S. Jansen, S. Brinkkemper, and M. Cusumano, *Software Ecosystems: Analyzing and Managing Business Networks in the Software Industry*. Edward Elgar Pub, 2013.

[25] C. Jergensen, A. Sarma, and P. Wagstrom, "The onion patch: migration in open source ecosystems," in *European Conference on Foundations of Software Engineering*, 2011.

[26] M. Lungu, R. Robbes, and M. Lanza, "Recovering inter-project dependencies in software ecosystems," in *International Conference on Automated Software Engineering*, 2010.

[27] M. Lungu, M. Lanza, T. Gîrba, and R. Robbes, "The small project observatory: Visualizing software ecosystems," *Science of Computer Programming*, vol. 75, no. 4, 2010.

[28] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, 2009.

[29] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik, "Studying evolving software ecosystems based on ecological models," in *Evolving Software Systems*, T. Mens, A. Serebrenik, and A. Cleve, Eds. Springer Berlin Heidelberg, 2014.

[30] D. M. German, B. Adams, and A. E. Hassan, "The evolution of the R software ecosystem," in *European Conference on Software Maintenance and Reengineering*, 2013.

[31] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *International Conference on Software Maintenance*, 2013.

[32] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *International Conference on Software Engineering*, 2007.

[33] M. Kim and D. Notkin, "Discovering and Representing Systematic Code Changes," in *International Conference on Software Engineering*, 2009.

[34] H. A. Nguyen, T. T. Nguyen, G. Wilson, Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," in *International Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[35] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *International Symposium on the Foundations of Software Engineering*, 2012.

[36] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang, "Mining API mapping for language migration," in *International Conference on Software Engineering*, 2010.

[37] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," in *International Conference on Software Engineering*, 2013.

Impact of Developer Turnover on Quality in Open-Source Software

Matthieu Foucault
U. of Bordeaux, LaBRI, France
mfoucaul@labri.fr

Marc Palyart
UBC, Canada
mpalyart@cs.ubc.ca

Xavier Blanc
U. of Bordeaux, LaBRI, France
xblanc@labri.fr

Gail C. Murphy
UBC, Canada
murphy@cs.ubc.ca

Jean-Rémy Falleri
U. of Bordeaux, LaBRI, France
falleri@labri.fr

ABSTRACT

Turnover is the phenomenon of continuous influx and retreat of human resources in a team. Despite being well-studied in many settings, turnover has not been characterized for open-source software projects. We study the source code repositories of five open-source projects to characterize patterns of turnover and to determine the effects of turnover on software quality. We define the base concepts of both external and internal turnover, which are the mobility of developers in and out of a project, and the mobility of developers inside a project, respectively. We provide a qualitative analysis of turnover patterns. We also found, in a quantitative analysis, that the activity of external newcomers negatively impact software quality.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics—*process metrics*

Keywords

Mining software repositories, qualitative analysis, software metrics

1. INTRODUCTION

Throughout the evolution of a project, the team contributing to it evolves, with collaborators joining, leaving, or changing their role in the project. This phenomenon of continuous influx and retreat of human resources is called *turnover*. Turnover has been studied in managerial science and human-computer interaction research, with several theories regarding its impact. The most common theory holds that turnover has a negative impact on performance and on the quality of the work, due to a loss of experience [22]. Other theories suggest that turnover has (1) a positive impact since the most dissatisfied members leave the team, and that only the most motivated ones stay in it [27], (2) helps renew experience and knowledge on the team [40], and (3) increases social interactions [10].

In the software development context, developer turnover has been analyzed by Mockus [30] on one industrial project. He found that developers leaving the project had a negative impact on quality but that new members had no effect on it. Our work extends these findings made on an industrial software project by looking at five large open-source software projects. These projects are interesting to study given their extensive use and low barriers to entry and exit for collaborators [15].

To study turnover in open-source, we introduce activity metrics that measure external and internal turnover. By splitting a software project into different modules, we are able to measure not only the arrivals and departures of developers from the project (i.e. external turnover), but also the movement of developers within the project (i.e. internal turnover).

Based on the concepts we define in this paper, we quantify the level of turnover, both external and internal, in open-source software projects. We quantify turnover by measuring the amount of changes performed in the source code by newcomers or leavers, instead of measuring the actual number of developers joining or leaving, as there is a great disparity between developers in open-source projects. We then perform an empirical study on 5 large open-source projects (Angular.JS, Ansible, Jenkins, JQuery and Rails) to provide insights on the relationship among developer turnover and software quality, where quality was measured based on the density of bug-fixing commits. The extraction process for bug-fixing commits is performed manually, to reduce the risk of errors produced by automatic approaches [5, 7, 21], thus limiting the number of projects that can be considered in this paper.

We provide the following contributions, for the five open-source projects mentioned above:

- We provide a curated set of bugs.
- We provide metrics to measure turnover.
- We show the importance of the turnover phenomenon in open-source projects.
- We observe several trends of internal and external turnover.
- We show that there is a relationship between turnover and quality of software modules.

This paper is structured as follows: Section 2 presents the theory and related work. Turnover metrics are defined in

Section 3. Our research questions are detailed in Section 4 and the methodology we used to build our dataset in Section 5. Our results are then presented in Section 6. Section 7 presents an overview of the main threats to the validity of these results, and finally, Section 8 concludes and presents trails for future work. We produced a replication package which allows to reproduce and extend the results presented in this study. This package, which has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations, is presented Section 10.

2. THEORY & RELATED WORK

As the literature contains different and sometimes contradictory opinions on turnover, we first describe all its possible interpretations. We then present existing work on turnover in collaborative communities and finally research specific to software development.

2.1 Turnover Perception

Member turnover, initially defined as the rate at which individuals leave a project, can be extended to all the changes made to the development team of a project. These modifications of the team can be either external, (i.e., a member leaves or joins the team) or internal (i.e., a member changes her role in the team). Distinct theories regarding the impact of turnover, whether it is external or internal, suggest that it has both positive and negative aspects on a team.

2.1.1 External Turnover

The most common vision holds that external turnover negatively impacts employee performance [22, 43]. Departures lead to a loss of experience and knowledge, but also disrupt the social network and environment of those who remain [4, 11]. Moreover, it induces devoting resources and time to recruit and train new employees.

A second vision considers turnover as a good opportunity for organizations, as leavers are those most dissatisfied with the current organization, and those who remain enjoy better conditions and performance [27].

A last perspective sees moderate levels of turnover as the best organizational performance [3]. When there is no turnover, experience and knowledge are not renewed, and become obsolete and parochial [40]. Introduction of new people is a solution to overcome this situation, as their vision is less established and less redundant with respect to the knowledge possessed by the current team.

2.1.2 Internal Turnover

Internal turnover was defined in traditional organizations as the number of employees who changed function within an organization [20]. Motivations behind such actions are opportunities for career moves to increase income and autonomy as well as getting new responsibilities and expressing new skills [41]. Kanter et al. pointed out that members had lower aspirations and involvements in their work when mobility was blocked [26]. Thus, internal mobility is commonly supported to maintain members commitment to the organization.

2.2 Turnover in Collaborative Platforms

Turnover has been studied in online communities and collaborative platforms where participants are free to enter or leave at any moment without any cost. In the English Wikipedia, high turnover is even the norm with sixty percent of editors contributing only for a single day [32]. Ransbotham et al. suggested that collaboration success can be reached thanks to moderate levels of turnover [36], provided that the level of novel knowledge exceeds the loss of existing knowledge held by departing people. Similarly, Dabbish et al. discovered that membership turnover might bring fresh levels of activity and liveliness in a community which leads to increased participation [10]. Inversely, Qin et al. observed that departures of WikiProjects contributors has a negative effect on the community and causes social capital losses [35].

2.3 Turnover in Software Development

Developer turnover in open-source software projects was studied mainly to understand developers motivations to contribute. Yu et al. suggested that personal expectation plays a role in project retention, and that turnover is partially explained by dissatisfaction [45]. Hynninen et al. conducted a survey with developers and suggested that their departures from a project can be a manifestation of low organizational commitment [23]. A study from Schilling et al. unveiled that the level of development experience and knowledge is strongly associated with retention [38]. According to Sharma et al., past activity, age and size of a project as well as developer tenures are important predictors of turnover [39]. These observations are consistent with other classical theoretical models related to job satisfaction [44].

Measures of knowledge loss were suggested by Izquierdo-Cortazar et al [24]. These measures include the evolution of *orphan* lines of code lastly edited by a developer who left the team. They showed that while in some projects, developers devote efforts to maintain code introduced by former developers, in others, they seek to eliminate such code. Robles et al. designed a methodology to compute generations of joining and leaving developers [37]. Finally, Fronza et al. propose a wordle to visualize the level of cooperation of a team and mitigate the knowledge loss due to turnover [17].

Hall et al. conducted a survey with practitioners to unveil that turnover may be related to project success, but however did not define turnover metrics computable by analyzing the history of the project [19]. Mockus found that while departures of members impact the software quality because of the loss of knowledge and experience, newcomers are not responsible for an increase of defects, possibly because they are not assigned to important changes [30]. Mockus also found a relationship between turnover and productivity in commercial projects [29].

Mens et al. explored developer turnover in the Gnome ecosystem with concepts and metrics similar as the ones we use in this paper [28]. They looked at developer turnover at a coarser grain: in their study, internal turnover refers to the mobility of developers between projects of the Gnome ecosystem, while external turnover in their case was associated to developers entering or leaving the Gnome ecosystem. Our study differs from theirs as it we look at a finer granularity: we measure mobility of developer between modules of a project, and in and out of a project. In their study they sought for possible patterns of developer turnover, with the conclusion that this is a highly project-specific phenomenon. They did not, seek for a relationship between turnover and code quality.

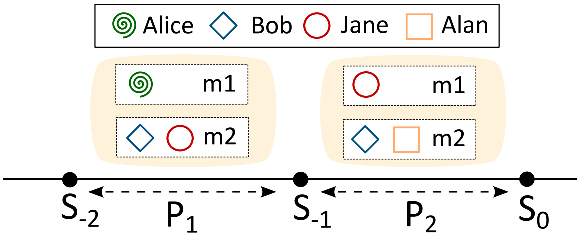


Figure 1: Example of fictive software project containing two modules.

3. TURNOVER METRICS

A software project can have many kinds of turnover. To be able to study different aspects of turnover, we introduce five metrics that can be computed from the source code history of a software project.

3.1 Setup and Requirements

In order to compute turnover metrics, we need to define the periods over which turnover will be computed, as well as how contributors are identified.

3.1.1 *Period Selection*

We compute developer turnover by comparing the contributors of software modules in two consecutive time periods: P_1 and P_2. These two periods are therefore delimited by three snapshots of the project history: S_0, S_{-1} and S_{-2} such that P_1 is delimited by S_{-2} and S_{-1} and that P_2 is delimited by S_{-1} and S_0 (see Figure 1).

In practice, S_0 is the snapshot for which we want to compute turnover metrics. The selection of the two other snapshots can be based on different approaches. One could consider either the prior releases of the software, snapshots such that periods P_1 and P_2 have the same duration, or snapshots such that periods P_1 and P_2 have the same overall activity in the repository. We study in Section 5 the impact of these choices on turnover computation.

3.1.2 *Software Modules and Contributors*

Developer turnover is relative to the software project's structure. A developer who has only worked on a few modules in the system that suddenly contributes to more, should be considered as new, or inexperienced, as she moves to new parts of the code base.

We therefore consider that a software project is composed of a finite set M of software modules developed by a finite set of developers who submit their code modifications by sending commits to a shared code repository. Each module is defined by a finite set of source code files. When a developer modifies one of the files of a software module by committing her work, she is contributing to that module. A developer contributes to the software project as soon as she contributes once to any module of the software.

To illustrate all our definitions, we rely on the fictitious software depicted in Figure 1, which is composed of two modules developed over two periods P_1 and P_2. A total of four developers participated to this software between the S_{-2} and S_0 snapshots.

Given a module m, $D_{m,P}$ is the set of developers who made at least one contribution to m during the period P. We obtain with our example $D_{m2,P_1} = \{\text{Bob, Jane}\}$ and $D_{m2,P_2} = \{\text{Bob, Alan}\}$.

D_t is the set of developers who made at least one contribution to the software during the period t, i.e. $D_P = \bigcup_{m}^{M} D_{m,P}$. From our example, we have $D_{P_1} = \{\text{Alice, Bob, Jane}\}$ and $D_{P_2} = \{\text{Bob, Jane, Alan}\}$.

3.2 Turnover Actors and Metrics

We now provide formal definitions for the sets of developers involved in turnover, and the metrics associated to them. We consider two kinds of developer turnover: external and internal turnover. The developers involved in each kind of turnover are considered to be either newcomers or leavers. Finally, we define stayers, i.e. the developers contributing to both studied periods.

We consider as newcomers the developers who joined the team of a module in the period P_2, whereas leavers are the developers who left the team of a module within the period P_1. This difference between the periods is due to the fact that the intent of our metrics is to evaluate the impact of turnover on the quality of the software at the S_0 snapshot. Thus, newcomers of the P_2 period may influence its quality as their first contributions on a module were between S_{-1} and S_0, and leavers of the P_1 period may influence its quality as the loss of knowledge their departure induce will be perceptible after they left, i.e., after the S_{-1} snapshot.

3.2.1 *External Turnover*

External turnover refers to the movement of developer in and out of a project.

External newcomers of a module m are the developers who contributed to the module between S_{-1} and S_0, but did not contribute to any module of the project between S_{-2} and S_{-1} (i.e., during the P_1 period). The set of external newcomers is noted EN_{m,P_1,P_2} and is computed as follows:

$$EN_{m,P_1,P_2} = D_{m,P_2} - D_{P_1}$$

In Figure 1, we observe that Alan is a newcomer in $m2$, and that he did not work on any module during P_1. He is therefore an external newcomer, and thus $EN_{m2,P_1,P_2} = \{\text{Alan}\}$.

External leavers of a module m refer to developers who worked on the module during P_1 but did not contribute to the project at all in P_2. The set of external leavers is noted EL_{m,P_1,P_2} and is computed as follows:

$$EL_{m,P_1,P_2} = D_{m,P_1} - D_{P_2}$$

We observe that only Alice contributed to $m1$ during P_1 but was inactive on the project in P_2. Consequently, $EL_{m1,P_1,P_2} = \{\text{Alice}\}$.

3.2.2 *Internal Turnover*

Internal turnover refers to movements of developers inside a project. Even though some developers contribute to a project in both periods P_1 and P_2, they may not work on the same modules in the two periods.

Internal newcomers are the developers who contributed to m in P_2, but not in P_1. However, they contributed to at least one other module than m in this period. They are noted IN_{m,P_1,P_2} and are computed as follows:

$$IN_{m,P_1,P_2} = (D_{m,P_2} - D_{m,P_1}) \cap D_{P_1}$$

Following the previous illustrations, we obtain here $IN_{m1,P_1,P_2} = \{\text{Jane}\}$ and $IN_{m2,P_1,P_2} = \emptyset$.

Internal leavers refer to developers who ceased to contribute to a module m but are still active in the project. This set is noted IL_{m,P_1,P_2} and is computed as follows:

$$IL_{m,P_1,P_2} = (D_{m,P_1} - D_{m,P_2}) \cap D_{P_2}$$

We observe that only Jane modified $m1$ during P_2 but not in P_1, while working on $m2$ during P_1. Consequently, $IL_{m2,P_1,P_2} = \{\text{Jane}\}$.

3.2.3 Stayers

Finally, stayers are the developers who contributed to a module m in both P_1 and P_2. We define the set of stayers for a given module as:

$$St_{m,P_1,P_2} = D_{m,P_1} \cap D_{m,P_2}$$

3.2.4 Metric Definitions

The intention of our metrics is to quantify the impact that the different turnover actors may have on a module's quality at the snapshot S_0 of the project. Due to the large inequalities in the involvement of developers in open-source projects, we cannot quantify turnover by counting the number (or ratio) of developer in each of the categories defined above. Filtering the developers by considering only core or paid contributors is not a viable alternative either. Indeed, peripheral developers as a group still produce a significant amount of contributions, and ignoring these contributions may significantly impact our measurements. Therefore, to measure the impact that each category of turnover actors have on the source code, we use the activity of developers, i.e., the amount of source code they produce.

For a given module m, developer d and period t, we define $A_{m,d,t}$ as the activity of the developer, which we measure using the code churn, i.e. the number of lines of code added or deleted by can be measured with the number of file modifications she performed on the module, or the code churn (i.e., the total number of lines added or deleted) of such modifications. In this paper we only present results obtained using the code churn as an activity measure. However, results obtained with the number of modifications are similar, and are available online (see Section 9).

The five metrics we define are the internal and external leavers activity (ILA and ELA, resp.), the internal and external newcomers ratio (INA and ENA, resp.), and the stayers activity (SA):

$$ILA_{m,P_1,P_2} = \sum\nolimits_{d \in IL_{m,P_1,P_2}} A_{m,d,P_1}$$

$$ELA_{m,P_1,P_2} = \sum\nolimits_{d \in EL_{m,P_1,P_2}} A_{m,d,P_1}$$

$$INA_{m,P_1,P_2} = \sum\nolimits_{d \in IN_{m,P_1,P_2}} A_{m,d,P_2}$$

$$ENA_{m,P_1,P_2} = \sum\nolimits_{d \in EN_{m,P_1,P_2}} A_{m,d,P_2},$$

$$StA_{m,P_1,P_2} = \sum\nolimits_{d \in St_{m,P_1,P_2}} avg(A_{m,d,P_1}, A_{m,d,P_2})$$

4. RESEARCH QUESTIONS

To the best of our knowledge we found no previous study that looked at trends of developer turnover in open-source software projects. Hence the first objective of our study is to seek for such trends, starting with a global view of turnover at the project level, and then focusing on developer turnover on module thanks to the metrics previously defined.

More formally, we seek to answer the following two research questions:

RQ1 Using the concepts of external newcomers and leavers at the project level, is turnover an important phenomenon (in terms of number of developers involved) in open-source software projects?

RQ2 Looking deeply into the project at the module level, is there any patterns regarding the contributions of persistent, internal and external developers?

By answering the aforementioned research questions, we provide an overview of developer turnover both at the project and at the module levels. We then go further by exploring the relationship between developer turnover at the module level and software quality, which we measure based on bugfix information.

We then answer the following research question:

RQ3 Using the turnover metrics at the module level, is there any relationship with the quality of the software modules?

5. DATASET CONSTRUCTION

Although many automatic techniques are often used to build large datasets, they are all imprecise to a certain extent. Instead of having a dataset with dozens of project containing approximate measures, we chose to focus on the reliability of the information extracted from the dataset. In particular, to answer our research questions, our dataset must meet several requirements:

- The author of each contribution must be clearly identified.
- The source code of the project must be organized into modules.
- A measure of quality must be available for each module.

Each of these criteria is addressed in current research, and software engineering researchers are still developing techniques to extract reliable information from software repositories, as we detail below.

5.1 Authors Identification

Centralized VCS.

The first issue regarding the identification of authors is related to the version control system (VCS) used by the project. In centralized VCSs such as Subversion, a developer must enter her credentials to commit her code to the central repository. Given the large number of contributors to open-source projects, assigning credentials to each of them would be unwieldy, and contributions are therefore submitted via patches, and applied by core developers who have credentials for the repository.

This issue is fixed by the use of decentralized VCSs such as Git, which are able to distinguish the original author of a commit and the developer who added it to the main repository (i.e., the committer) [6]. However, automatically selecting a large number of Git repositories (from hosting platforms such as GitHub) would not be a suitable process in our case as a non negligible amount of large Git repositories are simply mirrors of Subversion repositories. Well known examples of such repositories include the gcc compiler project, or most of the projects hosted by the Apache Software Foundation (eg. the httpd server). Moreover, even if a project currently uses Git as a VCS, it may not have been so for all its development history. It is not uncommon for a project, especially older projects, to migrate its code base from one VCS to another throughout its history. This is the case of two projects selected in our dataset, Rails and Jenkins, which originally used Subversion and then migrated to Git. We manually searched commit messages for contents such as "Patch sent by Alice" to determine if at one point these projects were still using Subversion or if they did migrate to Git, and only include the history subsequent to this migration in our analyses.

Identity Merging.

Even when the identity of each contribution's author is reliable, it is possible that a single developer has several identities in the VCS, because of typos, changes in the configuration of the Git client, or a change of email address for instance. This issue is addressed by identity merging, for which Goeminne and Mens address a comprehensive review [18]. Following their recommendations, we use a semi-automatic process which is based on their simple algorithm which has a very high recall. To counter the low precision of the algorithm, we manually review the results of the identity merge algorithm and remove false positive merges.

5.2 Quality Measurement

Our study aims to evaluate the quality of project' modules for a given snapshot. In most software engineering studies the quality of software projects is assessed by looking at the number of bugs fixed by the developers.

Bug Fix Identification.

In order to measure the amount of these bugs, the state-of-the-art technique used in studies mining software repositories consists in parsing the commit messages, looking for the identifier of a bug stored in the project's bugtracker (e.g., "Bug #42") [46]. However, recent work raised concerns regarding the precision and recall of this automatic process, due to the misclassification of issues in the bugtrackers, or imprecision of algorithms linking bugs to source code [21, 5, 7].

Some approaches remove these concerns by only considering information stored in the VCS, and assume that the number of bug-fixing commits is a fair representation of the actual number of bugs within a software module. Unfortunately, to the best of our knowledge, no automatic approach has a satisfactory precision to produce reliable statistics. For instance, among the best automatic approaches, the ones developed by Tian et al. [42] and Mockus et al. [31] have a precision of only 53% and 61%, respectively, in the evaluated benchmarks, which in our case would have unpredictable effect on the number of bug-fixing commits identified, and would be a non-negligible bias to our study.

As we did not find a suitable automatic approach we chose to manually analyze commits to constitute our dataset to the detriment of the number of projects that we were able to include in it. Our manual approach therefore aims to identify commits that are true bugfixes.

Maintenance Branches.

To have measures which are representative of the quality of the code at a given snapshot or release, we need to isolate post-release bugfixes from development bugfixes. Post-release bugfixes for a snapshot S_0 are commits that fix a bug which was in the project's code at the snapshot S_0, while development bugfixes performed after the snapshot S_0 may have been introduced between the snapshot S_0 and the time of the bugfix. If the development history of a project is linear (i.e. if all the commits are performed on a single branch), isolating one category of commits from the other may be cumbersome and imprecise. Therefore, another constraint is added to the projects to include in our dataset: the release S_0 must have a dedicated maintenance branch, sometimes called long time support (or LTS) branch, where the only commits performed in it aim to improve the code quality of the release S_0. These maintenance branches differ from development branches. They usually do not contain new features. The operations performed in such branches are mainly bug-fixing, documentation, optimizations, or compatibility updates related to third party dependencies (e.g., the 2.3.x maintenance branch of Rails contains updates related to new versions of the Ruby programming language). Moreover, we restrict our search to maintenance branches where no commit was performed for the past six months, in order to have branches where most of the bugs were had time to get fixed.

Bug Fix Classification.

Our definition of a bug-fixing commit includes any semantic changes to the source code which fixes an unwanted behavior. The type of bugs considered includes any arithmetic or logic bug (e.g., division by zero, infinite loops, etc.), resource bugs (e.g., null pointer exceptions, buffer overflows, etc.), multi-threading issues such as deadlocks or race conditions, interfacing bugs (e.g., wrong usage of a particular API, incorrect protocol implementation or assumptions of a particular platform, etc), security vulnerabilities, as well as misunderstood requirements and design flaws.

We identified bug-fixing commits manually, discarding commits where new features are implemented. We choose to ignore commits where performance optimizations are performed, as we consider performance issues as a different aspect of code quality. Moreover, we also ignore commits that resolve compatibility issues due to the evolution of a third-party dependency, as these bugfixes are not due to the lack of quality of the changed code, but to the modification of an external requirement. Finally, it occurs that bug-fixing commits are lated discarded by the developers due to a regression introduced by the bugfix. In such cases, the developers perform a "revert" operation of such commits, and we ignore both the "revert" and the "reverted" commits.

We consider that bug-fixing commits are atomic, in the way that we do not consider the possibility that a bug-fixing commit may in fact include two bug-fixes. Moreover, if a bug-fixing commits affects two modules, the number of bug-fixing commits will be incremented in both modules.

5.3 Code Modularization

In this study, we use metrics that target software modules. Breaking a software system into modules is known to be a hard task that requires some subjective choices [33]. We consider two heuristics for determining software modules, such as its organization within files and directories or the co-change activity. We present here the different sets of modules based on these heuristics.

5.3.1 Using the Directory Structure

The first modularization approach we consider is based on the directory structure of the system, in which software modules are defined to be either a file or a directory, with the possibility to include or not its subdirectories. We chose not to simply extract a modularization based on the directory structure, instead we manually inspected the directory structure of each project to select a suitable level of granularity so that a module includes similar features, based on file and directories names, and on the information found in projects configuration files. To overcome the bias of having a single judge for the module decomposition, we asked three members of our research group (three PhD students in software engineering) to provide, for each of the five projects in the corpus, a list of software modules. The three judges then met to merge their results. They agreed on the granularity of most of the modules of projects such as JQuery, Angular.JS and Ansible, while agreement on Jenkins and Rails was initially reached by only two judges, the third one having chosen a coarser granularity. As the decisions made by the judges may be different than the developers of the projects, we tried to contact their core developers to confirm our decompositions, using the official mailing lists and/or IRC channels of each project. Unfortunately, we did not obtained any answers.

5.3.2 Using the Co-change Activity

The second modularization technique we use considers that source code files that are changed together (i.e. in the same commit) belong to the same module, regardless of the directory structure of the project. We use an automated process that consists in building the co-change graph of the project, which is a weighted, undirected graph where each vertex is a source code file of the project, and the weight of an edge is equal to the number on commits where both files were modified together. To determine the modules, we used two algorithms aiming at building communities in a graph [9, 34]. Both algorithms produced a relatively low number of modules (less than ten) in the projects developed in Javascript (Angular.JS and JQuery), which is due to the fact that Javascript projects tend to have fewer, larger files compared to projects in languages such as Java. Therefore, these decomposition allow to produce statistical results on only three projects. As the results obtained with this modularization algorithms are similar to the ones obtained with the manual decomposition based on directories structure, they are not presented in this paper. However, they are available in our additional results online (see Section 9).

5.4 Periods Selection

The computation of turnover metrics for a snapshot S_0 relies on the choice of two periods P_1 and P_2 (Figure 1).

To choose a suitable size for the periods P_1 and P_2, we measured the impact of these periods on the sets of turnover actors (i.e. internal and external leavers and newcomers). The length of the periods P_1 and P_2 may impact the resulting sets of actors, especially if the periods are too short, in which case we may consider as newcomers or leavers developers who stopped contributing to the project for a period of time before re-starting. To assess the impact of this choice we have tested four configurations for the lengths of the periods: one release-based configuration where S_0, S_{-1} and S_{-2} are three following releases of the project, and three time-based configurations where P_1 and P_2 both last for 1, 3 and 6 months.

Using $|P_1| = |P_2|$ may limit our vision in the past. This may for example result in considering some developers as newcomers because they were inactive for sometime, but the length of P_1 is not sufficient to see their previous contributions. On the other hand, if we looked at the whole history of the project to check whether developers are newcomers or leavers, we may consider as stayers developers who did not contribute to the project for several years. To quantify the impact of the length of P_1 and P_2, we compute two versions of each turnover set:

- A version with *limited visibility*, where $|P_1| = |P_2|$.
- A version with *full visibility*, where:
  - S_{-2} = is the beginning of the Git repository when computing the sets of newcomers.
  - S_0 is the most recent release available in the project when computing the sets of leavers.

To decide which period size is suitable for our analyses, we chose to measure the similarity between sets of turnover actors computed with limited and full visibility, using the Sorensen-Dice quotient of similarity, which is equal to 1 when two sets are identical, and 0 when they are disjoint [12]. The selected period size is the first period size where the median Dice coefficient is, for all projects and actors sets, above a threshold of 0.75.

The distributions of Dice coefficients obtained for each project are available online (See Section 9). For each period size $|P|$, project and category of turnover actors (e.g., external newcomers), we have a distribution of Dice coefficient, as we computed one Dice coefficient for each module. These distribution show that, with a period of one month, several sets of developers have large differences between limited and full visibility, the worst case being with Angular.JS where sets of external newcomers computed with limited visibility have no intersection with sets computed with full visibility. With $|P| = 3$ months, the distributions are closer to a dice coefficient of 1, but there are still cases where the median Dice coefficient is below the threshold of 0.75, especially with internal turnover. With $|P| = 6$ months, most of the sets of turnover actors are identical whether we use limited or full visibility. Only few modules have a dice coefficient of zero, and the median Dice coefficient for all projects and categories of turnover actors is above the threshold of 0.75 The release period configuration is not stable as the length of time between two releases depends on the roadmap of the project and on the features that are developed.

Therefore, we chose to use the 6 months period for the remainder of our analysis: all the results presented in this paper consider that $|P_1| = |P_2| = 6$ months.

5.5 Resulting Dataset

Our dataset, listed in Table 1 includes five projects, written in four different programming languages. The selected

Table 1: The projects included in our dataset.

Project	Language	Release (S_0)	#Bugfixes	LoC	#Modules
Angular.JS	JavaScript	1.0.0	147	11,041	26
Ansible	Python	1.5.0	62	50,553	29
Jenkins	Java	1.509	74	79,774	60
JQuery	Javascript	1.8.0	46	5,306	23
Rails	Ruby	2.3.2	390	33,919	46

releases are minor releases (i.e., no breaking changes have been performed in the selected development period) in Ansible, JQuery, and Rails. They are major release in Angular.JS and Jenkins. The selected releases are, with the exception of the one in Ansible, considered to be long term supported (LTS) releases. For these LTS releases, bug-fixing commits are backported from the main development branch even after subsequent releases are available. In Ansible, although the maintenance of the 1.5.x releases stopped a couple of week before the availability of the 1.6.0 release, it was performed simultaneously with the development of the 1.6.0 release. This dataset is available online (see Section 9) and can be reused for future studies.

6. RESULTS

6.1 Turnover at the Project Level (RQ1)

In order to characterize developer turnover at the project level we look at the number of external newcomers, external leavers and stayers during the life of each project.

Developers Volatility.

Since we defined $|P_1| = |P_2| = 6$ months we compute the different sets of actors by starting with S_0= 12 months after the earliest version of the project when we know that Git was used as a VCS, and move S_0 toward the end of the project by steps of two weeks. The resulting numbers are presented in Figure 2.

We can observe two types of phases during the life of a project. The first phase that we call the “enthusiastic” phase can only be seen in Angular.JS and Ansible since we are missing the beginning of the other projects as we excluded from the study the period when they were using SVN. During the “enthusiastic” phase (2011-2014 for Angular.JS and 2013-06/2014 for Ansible) the number of newcomers is constantly superior to the number of leavers. At some point projects switch to the second phase that we call the “alternating” phase where either the number of newcomers or leavers is higher than the other one.

In all projects, the number of newcomers and leavers is quite high. Throughout the histories of these projects, at least 80% of developers are either newcomers or leavers. Overall this confirms that turnover in open-source software projects is an important phenomenon.

Stayers Conversion and Motivations.

The number of stayers increase mainly during the “enthusiastic” phase and stay fairly stable during the “alternating” phase. To further understand the evolution of the population of stayers we use the notion of conversion rate that is usually found in marketing. In our case the conversion rate represents the proportion of newcomers that the project was able to keep long enough so they could become stayers. It is equal to the number of developers who were at least once stayer divided by the number of developers in the whole history of the project we look at. The conversion rates for each project are between 8% (Ansible) and 19% (Jenkins and JQuery). Even if it is not in the same proportion for each project we observed that only a low ratio of newcomers become stayers.

To better understand what make developers stay in their project we looked at the top 10 stayers of each projects: developers who were in the stayers set the highest number of times over the project history. We searched their Github and LinkedIn profiles as well as their personal web pages to understand their motivation. We found four categories:

- Developers who are paid by the company that develops the project. For example 7 out of the top 10 stayers of Angular.JS work at Google which maintains the framework.
- Developers who are paid by a company that use the project for their business. It is the case 6 times in the top 10 stayers of Ansible.
- Developers who are consultants on the technology developed within the project. For example 5 out of the top 10 stayers of Rails are consultants.
- Developers who contribute on their spare time without direct or indirect financial interest. Out of the 50 top stayers we looked at only 2 fit that category.

In conjunction to these categories developers were sometimes also the initial creators of the project (6 developers out of 50).

6.2 Patterns of Contributions (RQ2)

The visualizations in Figure 3 represents the turnover metrics[1] computed with $|P_1| = |P_2| = 6$ months and where the S_0 snapshots are the releases mentioned in Table 1 (these releases are also indicated in Figure 2 via vertical lines). We use these visualizations to observe the different patterns of contributions.

In Angular.JS, most of the activity is due to stayer or external leavers. The high amount of external leavers activity is in fact due to the contributions of a single developer, a major contributor who was inactive in the six months prior to the release of Angular.JS 1.0.0.

In Ansible, all categories of developers have similar levels of activity, and all contributed to a wide range of modules. This differs from other projects, especially for external newcomers: all but one module has external newcomers, and these developers often have an important activity. We looked more closely at the module where external newcomers were the most active, which is the module containing “cloud” plugins for Ansible. Among the newcomers making the most contributions, one was hired at Ansible, Inc., and two worked at Rackspace, a managed cloud computing company, and developed an Ansible plugin for the Rackspace cloud storage. These developers were most probably paid to do their contributions, which explains this high level of activity, not present with most of the newcomers.

The last three projects, Jenkins, JQuery and Rails, exhibit the same patterns. In these three projects, internal newcomers are active in most of the modules, while exter-

[1] Figure 3 also contains information related to bugfixes, which are discussed in the next research question.

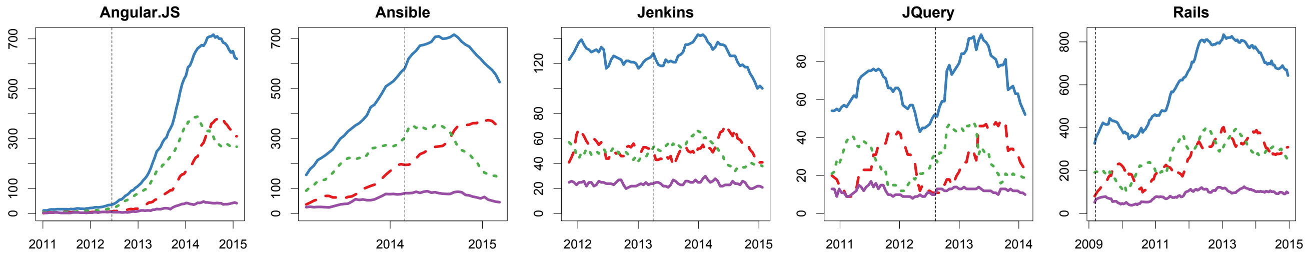


Figure 2: Evolution of developer turnover. The plain blue line (on top) represents the total number of developers, the plain purple line (on the bottom) the number of stayers, the green dotted line the number of external newcomers and the red dashed line the number of external leavers.

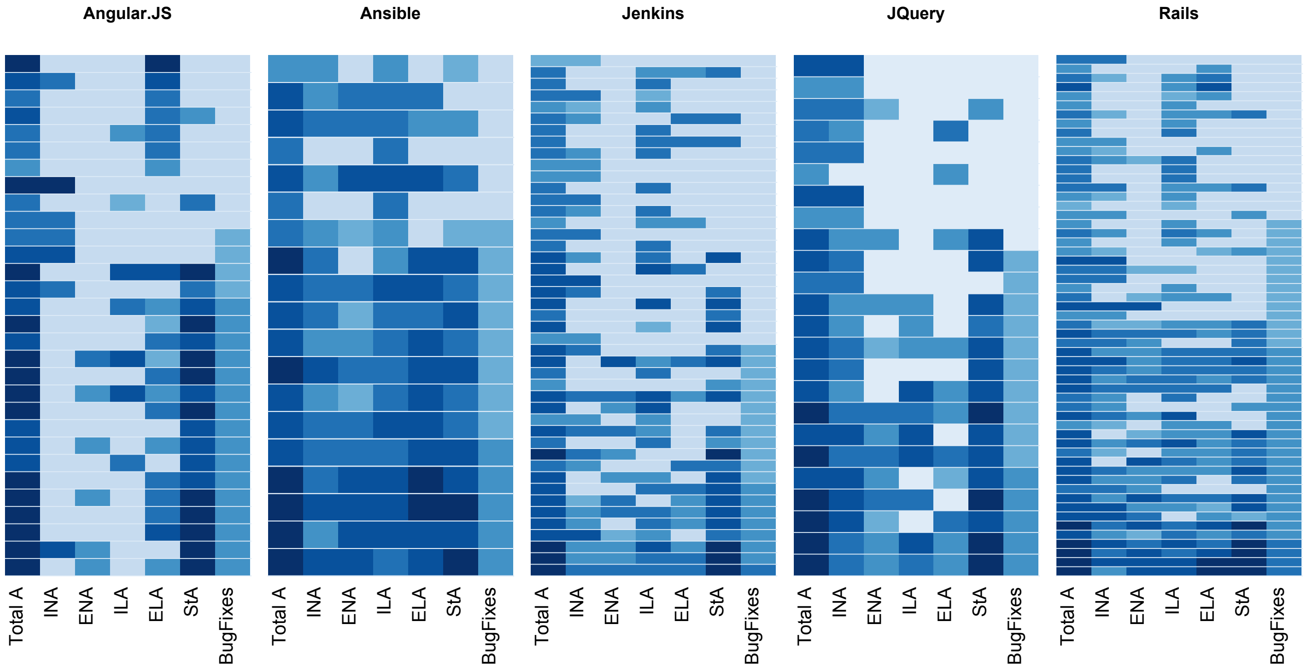


Figure 3: Visualization of developers activity and the quantity of bugfixes for each module. Each horizontal line of blocks represents a module. The darker the color, the higher the metric value.

nal newcomers and leavers are more focused, and do not contribute to more than half of the modules.

Overall there is no module that was changed exclusively by external newcomers. In all the projects of our corpus, the external newcomers always contributed to modules with either internal internal or permanent developers.

6.3 Developer Turnover and Software Module Quality (RQ3)

To answer our third research question, related to the relationship between module turnover and software quality, we use the bug-related information extracted from our dataset (same configuration as RQ2 for P_1 and P_2). We perform Spearman correlation tests between each turnover metric and our quality metric. The quality metric we use is the density of bugfixes per module (i.e., the number of commits that fixed bugs divided by the size of the module). These bugfixes are extracted from the maintenance branch associated to S_0, meaning that there is a high probability that they indeed fix defects that occur in S_0.

Table 2 presents the results of these correlation tests for each project and metric. Correlation coefficients vary from -1 to 1, which corresponds to a perfect negative and positive correlation, respectively. Also, a correlation coefficient of 0 reveals an absence of correlation. We used bootstrap with the BCa statistic to compute 95% confidence intervals of the correlation coefficients [13, 14]. If both ends of a confidence interval are either positive or negative (results highlighted in bold), this means that there is a strong probability that there is a positive or negative correlation, respectively, between the turnover metric and the density of bug-fixing commits.

To have a deeper understanding of the observed correlations, Figure 3 presents a graphical visualization of the turnover metrics and the number of bugfixes. In that Figure, each project is presented by a matrix where each column represents a metric, and each line represents a component of the project. The cell of the matrix then represents the value of the corresponding metrics and darker colors represent higher values.

Table 2: Spearman correlation coefficients between turnover metrics and the density of bug-fixing commits per module. Confidence intervals are computed using bootstrap.

Project	INA	ILA	ENA	ELA	StA	Overall Activity
Angular.JS	[-0.41 , 0.37]	[-0.36 , 0.25]	**[0.12 , 0.69]**	[-0.56 , 0.16]	**[0.23 , 0.83]**	[-0.16 , 0.7]
Ansible	[-0.27 , 0.73]	[-0.31 , 0.65]	[-0.21 , 0.68]	[-0.3 , 0.7]	[-0.15 , 0.76]	[-0.27 , 0.75]
Jenkins	[-0.3 , 0.28]	[-0.18 , 0.42]	**[0.3 , 0.75]**	[-0.05 , 0.51]	**[0.05 , 0.63]**	[-0.01 , 0.6]
JQuery	[-0.1 , 0.69]	**[0.13 , 0.81]**	[-0.02 , 0.73]	[-0.4 , 0.44]	**[0.09 , 0.84]**	**[0.14 , 0.8]**
Rails	[-0.01 , 0.52]	[-0.24 , 0.3]	**[0.09 , 0.57]**	[-0.23 , 0.3]	**[0.14 , 0.58]**	**[0.03 , 0.51]**

The most important information in the results presented in Table 2 is that there is a positive correlation between the External Newcomer Activity and the density of bugfixes. Almost all of the projects exhibit a quite strong correlation. Only Ansible exhibits a weak correlation but, looking at Figure 3, this is certainly due to the fact that external newcomers contributed to almost all of the components, even to the ones that were not the target of bugfixes. This is consistent with the theories exposed in Section 2, which suggest that external turnover has a negative effect on the quality of a team's work. External Leavers Activity on the other hand do not show any statistically significant correlation with bugfix density in Table 2, and the two columns seem completely independent in Figure 3.

Although Table 2 shows three statistically significant correlations between the Internal Leaver and Newcomer Activity and bugfixes, their interpretation when looking at Figure 3 is unclear. As discussed with the previous research question, internal newcomers contribute to the majority of the modules, even the ones without any bugfix.

Finally, as expected, there is a correlation between the activity of persistent developers and the density of bugfixes. This then raises the question of the relative importance of the turnover metrics regarding the software quality, and especially for the External Newcomer Activity (ENA) metrics, as there is no correlation of the internal turnover metrics. To measure how important is ENA we therefore built multiple linear regression models including other metrics, such as the size of modules or the number of developers who contributed to it. Unfortunately, it did not produce exploitable results, due to the low R-squared of the resulting models, and multicollinearity issues exposed by high variance inflation factors of the predictors. We therefore cannot provide sound answer to that point.

7. THREATS TO VALIDITY

The validity of the results presented above is exposed to several threats that we present here.

7.1 External Validity

The generalization of the results is our first concern. On one hand, we selected projects that use different programming languages and that have hundreds of developers. On the other hand, the study was performed on only five projects that were manually selected. To overcome this threat, further studies have to be performed, to confirm and improve the findings presented in this paper. A barrier to achieve these studies is to build curated datasets, following the requirements presented in Section 5.

7.2 Internal Validity

Our metrics assume that the only way developers contribute to a project is by modifying its source code. This is an approximation, as developers can modify other files such as build and documentation files. A project is not confined to its version control system: other types of repositories, such as bug-tracking system or mailing lists, might reveal that some developers considered as newcomers or leavers might be in fact persistent contributors of the project. Turnover metrics based on multiple kinds of repositories are left for future work.

It should be noted that our results should not be interpreted as if external leavers and newcomers developers introduced more bugs than internal. We do not provide or have any information on who introduce bugs because there is, to the best of our knowledge, no reliable algorithm that can identify the author of a bug.

We did not find a reliable way to identify developers with push rights to the repositories. Hence, we could not determine the impact of this feature on the different patterns of turnover. However it should be noted that the projects in our dataset mainly follow a pull-request workflow (a popular approach on Github). With this workflow, even if a developer has push rights she will create a pull-request in the project when making a contribution to benefit from the review mechanism. Thus, except for the developers in charge of merging the pull-requests the other core members do not need to have push rights.

7.3 Construct Validity

In addition, we identify several threats to construct validity from the previous study. On GitHub, developers can submit pull requests, so that the project leaders, who have write permission on the repository, can add their contributions to the project. As the identity of the initial author is maintained through the *pull* operation, she is identifiable even though she does not have access to the main repository. However, as shown in [25] it may happen that a developer discussed with a pull request author to agree on its acceptance. Even though this developer spent time to fix or improve the pull request content, all the credits will go to the pull request author. This may also introduce a bias in the results.

Identifying software modules in a project is not a straightforward task and might be subject to interpretation. Since we could not get the confirmation from the different development teams some modules in our decomposition might be split or merged in comparison to what the development teams would have defined.

Related to the same threat the quality of the software architecture can have an impact on the metrics. Retention could appear higher in well modularized systems than poorly designed systems where one fix might require changes to many modules. The fact that the results produced with the decomposition based on co-change activity overlaps strongly with the manual decomposition based on directories shows that the impact is negligible for this dataset.

We deliberatively did not rely on the information provided by bugtrackers, as several studies showed that their use can introduce an important bias [21]. The drawback of our technique is that the number of bug-fixing commits may not reveal the actual number of bugs that appeared in the software modules. There may exists bugs that are tedious to fix and remain to be resolved. In addition, the manual analysis has some limits due to the subjective evaluation to decide whether or not a commit is a bug-fixing commit. We only went through a maintenance branch to collect such commits for each project, although it potentially exists bug-fixing commits from the main development branch that have not been backported to the maintenance branch. Finally some bug-fixing commits may fix bugs that were not introduced in the current release but in one of the older releases.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose and investigate metrics to measure turnover in open-source software projects. Our metrics measure how the structure of a group of developers is changing, both internally and externally, for a given period of a software project. We used these metrics on five open-source projects with two objectives: to observe the turnover phenomenon, and to evaluate its relationship with software quality.

We observed that the five open-source projects in our corpus, chosen because of their popularity and success, have a high turnover. This observation disagrees with the conclusions of Hall et al. that recommend to control turnover to improve the success of industrial projects [19]. Our results then suggest that turnover and success may have a different relationship in open-source projects.

Looking at the module level, we show some very interesting turnover patterns. These patterns reveal that the projects of our corpus act differently regarding turnover. For instance, in some projects modules receive contributions only by internal developers with no contribution from stayers. These patterns also show that in all projects external newcomers always work with either permanent or internal developers, who hopefully supervise them. Such an observation opens the room for rules or guidelines that will define how newcomers should be supervised, and how they should contribute to modules of a project [8].

We also found that external turnover has a negative impact on the quality of the modules. This result is consistent with theories that suggest that external turnover has a negative effect on the quality of a team's work. However, it differs from the ones of Mockus [30], as in our case newcomers have a relationship with quality and leavers do not have such relationship, while it was the opposite in Mockus' study. On the other hand, internal turnover has almost no effect. Our observations therefore do not confirm the theories that suggest that internal turnover is beneficial. These findings can be reused by researchers when using software metrics based on the activity of developers on the source code: as the activity of external newcomers has a stronger relationship with quality than the activity of other categories of developers, this may be the only activity worth considering.

Finally, our study and findings lead the way for many kinds of future work:

- Our findings are based on observations made on software modules, with manual observation of patterns and using correlation between the density of bug-fixing commits and the activity of the different categories of turnover actors. As there are no related work that performed such observations on open-source projects, our study needs to be replicated on more projects, which is facilitated by our replication package (Section 10).
- The main limitation of our metrics is the fact that they require a selection of periods. In particular, we shown that the length of the chosen periods has a major impact on the measures, and we therefore provide some insights showing that a time period is adequate in the case of open-source project, with good results with 6 months periods. We then plan to overcome this limitation by developing continuous metrics for turnover, where the discretization of the history is not necessary.
- Our results regarding turnover patterns suggest that the observed patterns are impacted by the motivation of developers, which mainly depends on the fact that they are paid or not. This hypothesis can be evaluated only if it is possible to distinguish paid contributors from volunteers. We then plan to identify the employee of the developers, and then to analyze its relationship with turnover metrics.
- Independently of whether developers are paid or volunteers, they may be core member of the projects, and thus have a higher retention level than other developers, as well as a higher level of activity in the project. Identifying core members of a project may help us understanding the impact of developer turnover on software quality.

9. AUXILIARY MATERIAL

Due to the space constraint of this paper, part of our results are available online [2]. This page includes results regarding the period selection, as well as more detailed versions of Figure 2 and Figure 3.

10. REPLICATION PACKAGE

The dataset built using the methodology presented in Section 5, the code necessary to extract the metrics, as well as additional results are available online, in a replication package that has been successfully evaluated by the Replication Packages Evaluation Committee and found to meet expectations [2]. We describe here the technical aspects of this package, the data produced by the executed software and the installation process of the replication package.

10.1 The Diggit Tool

The software of our replication package relies on the Diggit tool, which supports analysis of Git repositories and which helps manage the analysis process [1]. Diggit manages a set of Git repositories. On each repository, Diggit applies several user specified *analyses*. When all user-specified analyses have been applied, Diggit applies global analyses (called *joins* in the tool) that use the results of all the previously applied analyses to produce final results. Diggit is used within a *diggit folder*, which contains various configuration files, including the list of Git repositories to clone and analyse, the list of analyses and joins to perform, and additional information that may be used by analyses. This tool is developed in Ruby by two of the authors of this paper; we used version 2.0.2.

10.2 Package Installation and Usage

The replication package is distributed as a VirtualBox virtual machine image. This image is based on a minimal version of a Linux Ubuntu on which only the requirements to replicate the study were installed. The list of commands required to install our replication package from a fresh install of a minimal Ubuntu is also available online.

The package consists of a set of diggit analyses, that are all loaded in a diggit folder in the VM image (this folder can also be generated with a script). The first step of the replication is to clone the five Git repositories to be analyzed using the `dgit clone` command. Then, the `dgit analyses perform` command allows to run, for each of the cloned repositories, the following analyses:

- Extraction of the number of lines of code of each file at release S_0, which is used to compute bug-fixing commits density.
- Computation of the number of lines of code and the number of bug-fixing commits of each module (the list of modules is stored in a configuration file).
- Computation of the activity of each developer on each module.
- Computation of the activity of developers at the project level.

The data produced for each repository is then aggregated by a global analysis that uses the R programming language and produces all the results presented in this paper and in the additional results available online.

10.3 Replication Data

Our replication package also provides data that can be reused for future studies. It includes the information described in Section 5 which is given as input to the analyses described above, and activity information, which can be reused to compute other metrics than developer turnover (such as code ownership for instance [16]).

For each repository, the data provided as input of the analyses is the following:

- The author renaming information.
- The lists of modules extracted with the different modularization techniques.
- The commit id of the S_0 release.
- The commit ids of the bug-fixing commits performed in S_0 release's maintenance branch.

This data is stored in a single JSON file and thus can be easily reused.

Besides the final results provided by the global analysis that are presented in this paper, each analysis produces intermediary results that are stored in a *MongoDB* database. The data stored in this database consist in a monthly measure of activity (code churn) for each developer and module, and each month prior to the S_0 release up to the start of the Git history of the repository.

11. REFERENCES

[1] The diggit git repository analysis tool. `https://github.com/jrfaller/diggit`. Accessed: 2015-07-15.

[2] Replication package - impact of developer turnover on quality in open-source software. `http://se.labri.fr/a/FSE15-foucault`. Accessed: 2015-07-15.

[3] M. A. Abelson and B. D. Baysinger. Optimal and dysfunctional turnover: Toward an organizational level model. *Academy of Management Review*, 9(2):331–341, Apr. 1984.

[4] L. Argote and D. Epple. Learning curves in manufacturing. *Science*, 247(4945):920–924, Feb. 1990.

[5] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT symposium on The Foundations of Software Engineering (ESEC/FSE)*, page 121–130, 2009.

[6] C. Bird, P. Rigby, E. Barr, D. Hamilton, D. German, and P. Devanbu. The promises and perils of mining git. In *6th IEEE International Working Conference on Mining Software Repositories, 2009. MSR '09*, pages 1 –10, May 2009.

[7] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère. Empirical evaluation of bug linking. In *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR 2013)*, pages 1–10, Mar. 2013.

[8] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. Who is going to mentor newcomers in open source projects? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 44, 2012.

[9] A. Clauset, M. E. Newman, and C. Moore. Finding community structure in very large networks. *Physical review E*, 70(6):066111, 2004.

[10] L. Dabbish, R. Farzan, R. Kraut, and T. Postmes. Fresh faces in the crowd: Turnover, identity, and commitment in online groups. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, page 245–248. ACM, 2012.

[11] G. G. Dess and J. D. Shaw. Voluntary turnover, social capital, and organizational performance. *Academy of Management Review*, 26(3):446–456, July 2001.

[12] L. R. Dice. Measures of the amount of ecologic association between species. *Ecology*, 26(3):297, July 1945.

[13] B. Efron. Bootstrap methods: another look at the jackknife. *The Annals of Statistics*, page 1–26, 1979.

[14] B. Efron. Better bootstrap confidence intervals. *Journal of the American Statistical Association*, 82(397):171–185, 1987.

[15] K. Fogel. *Producing Open Source Software: How to Run a Successful Free Software Project.* O'Reilly Media, first edition, Feb. 2013. http://www.producingoss.com/.

[16] M. Foucault, J.-R. Falleri, and X. Blanc. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, page 39:1–39:9. ACM, 2014.

[17] I. Fronza, A. Janes, A. Sillitti, G. Succi, and S. Trebeschi. Cooperation wordle using pre-attentive processing techniques. In *2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 57–64, May 2013.

[18] M. Goeminne and T. Mens. A comparison of identity merge algorithms for software repositories. *Science of Computer Programming*, 78(8):971–986, 2013.

[19] T. Hall, S. Beecham, J. Verner, and D. Wilson. The impact of staff turnover on software projects: The importance of understanding what makes software practitioners tick. In *Proceedings of the 2008 ACM SIGMIS CPR Conference on Computer Personnel Doctoral Consortium and Research*, SIGMIS CPR '08, page 30–39. ACM, 2008.

[20] D. S. Hamermesh, W. H. J. Hassink, and J. C. v. Ours. Job turnover and labor turnover: A taxonomy of employment dynamics. Open Access publications from Tilburg University 12-86873, Tilburg University, 1996.

[21] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, page 392–401, 2013.

[22] M. A. Huselid. The impact of human resource management practices on turnover, productivity, and corporate financial performance. *Academy of Management Journal*, 38(3):635–672, June 1995.

[23] P. Hynninen, A. Piri, and T. Niinimaki. Off-site commitment and voluntary turnover in GSD projects. In *2010 5th IEEE International Conference on Global Software Engineering (ICGSE)*, pages 145–154, Aug. 2010.

[24] D. Izquierdo-Cortazar. Relationship between orphaning and productivity in evolution and GIMP. 2008.

[25] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, page 92–101. ACM, 2014.

[26] R. M. Kanter. The impact of hierarchical structures on the work behavior of women and men. *Social Problems*, 23(4):415–430, Apr. 1976.

[27] D. Krackhardt and L. W. Porter. When friends leave: A structural analysis of the relationship between turnover and stayers' attitudes. *Administrative Science Quarterly*, 30(2):242–61, Jan. 1985.

[28] T. Mens, M. Claes, P. Grosjean, and A. Serebrenik. Studying evolving software ecosystems based on ecological models. In *Evolving Software Systems*, pages 297–326. Springer Berlin Heidelberg, Jan. 2014.

[29] A. Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, page 67–77. IEEE Computer Society, 2009.

[30] A. Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, page 117–126. ACM, 2010.

[31] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00)*, ICSM '00, page 120–. IEEE Computer Society, 2000.

[32] K. Panciera, A. Halfaker, and L. Terveen. Wikipedians are born, not made: A study of power editors on wikipedia. In *Proceedings of the ACM 2009 International Conference on Supporting Group Work*, GROUP '09, page 51–60. ACM, 2009.

[33] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[34] P. Pons and M. Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences - ISCIS 2005*, number 3733 in Lecture Notes in Computer Science, pages 284–293. Springer Berlin Heidelberg, 2005.

[35] X. Qin, M. Salter-Townshend, and P. Cunningham. Exploring the relationship between membership turnover and productivity in online communities. 2014.

[36] S. Ransbotham and G. C. Kane. Online communities: Explaining rises and falls from grace in wikipedia. *MIS Q.*, 35(3):613–628, Sept. 2011.

[37] G. Robles and J. M. Gonzalez-Barahona. Contributor turnover in libre software projects. In *Open Source Systems*, number 203 in IFIP International Federation for Information Processing, pages 273–286. Springer US, Jan. 2006.

[38] A. Schilling, S. Laumer, and T. Weitzel. Who will remain? an evaluation of actual person-job and person-team fit to predict developer retention in FLOSS projects. In *2012 45th Hawaii International Conference on System Science (HICSS)*, pages 3446–3455, Jan. 2012.

[39] P. N. Sharma, J. Hulland, and S. Daniel. Examining turnover in open source software projects using logistic hierarchical linear modeling approach. In *Open Source Systems: Long-Term Sustainability*, number 378 in IFIP Advances in Information and Communication Technology, pages 331–337. Springer Berlin Heidelberg, Jan. 2012.

[40] J. D. Shaw, N. Gupta, and J. E. Delery. Alternative conceptualizations of the relationship between voluntary turnover and organizational performance. *Academy of Management Journal*, 48(1):50–68, Feb. 2005.

[41] J. D. Thompson. Organizations in action: Social science bases of administrative theory. SSRN Scholarly Paper ID 1496215, Social Science Research Network, 1967.

[42] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Software Engineering (ICSE), 2012 34th International Conference on*, page 386–396, 2012.

[43] Z. Ton and R. S. Huckman. Managing the impact of employee turnover on performance: The role of process conformance. Jan. 2008.

[44] S. G. Westlund and J. C. Hannon. Retaining talent: Assessing job satisfaction facets most significantly related to software developer turnover intentions.

Journal of Information Technology Management, 19(4):1–15, 2008.

[45] Y. Yu, A. Benlian, and T. Hess. An empirical study of volunteer members' perceived turnover in open source software projects. In *2012 45th Hawaii International Conference on System Science (HICSS)*, pages 3396–3405, Jan. 2012.

[46] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *International Workshop on Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007*, page 9, May 2007.

Titre de la contribution : Automating the Extraction of Model-based Software Product Lines from Model Variants

Auteurs : Jabier Martinez, Tewfik Ziadi, Tegawendé Bissyandé, Jacques Klein and Yves Le Traon

We address the problem of automating 1) the analysis of existing similar model variants and 2) migrating them into a software product line. Our approach, named MoVaPL, considers the identification of variability and commonality in model variants, as well as the extraction of a CVL-compliant Model-based Software Product Line (MSPL) from the features identified on these variants. MoVaPL builds on a generic representation of models making it suitable to any MOF-based models. We apply our approach on variants of the open source ArgoUML UML modeling tool as well as on variants of an In-flight Entertainment System. Evaluation with these large and complex case studies contributed to show how our feature identification with structural constraints discovery and the MSPL generation process are implemented to make the approach valid (i.e., the extracted software product line can be used to regenerate all variants considered) and sound (i.e., derived variants We address the problem of automating 1) the analysis of existing similar model variants and 2) migrating them into a software product line. Our approach, named MoVaPL, considers the identification of variability and commonality in model variants, as well as the extraction of a CVL-compliant Model-based Software Product Line (MSPL) from the features identified on these variants. MoVaPL builds on a generic representation of models making it suitable to any MOF-based models. We apply our approach on variants of the open source ArgoUML UML modeling tool as well as on variants of an In-flight Entertainment System. Evaluation with these large and complex case studies contributed to show how our feature identification with structural constraints discovery and the MSPL generation process are implemented to make the approach valid (i.e., the extracted software product line can be used to regenerate all variants considered) and sound (i.e., derived variants which did not exist are at least structurally valid).which did not exist ar- at least structurally valid).

Tracking the Software Quality of Android Applications along their Evolution

Geoffrey Hecht[1,2], Omar Benomar[2], Romain Rouvoy[1], Naouel Moha[2], Laurence Duchien[1]
[1] University of Lille / Inria, France
[2] Université du Québec à Montréal, Canada
geoffrey.hecht@inria.fr, benomar.omar@courrier.uqam.ca, romain.rouvoy@inria.fr,
moha.naouel@uqam.ca, laurence.duchien@inria.fr

***Abstract*—Mobile apps are becoming complex software systems that must be developed quickly and evolve continuously to fit new user requirements and execution contexts. However, addressing these requirements may result in poor design choices, also known as *antipatterns*, which may incidentally degrade software quality and performance. Thus, the automatic detection and tracking of antipatterns in this apps are important activities in order to ease both maintenance and evolution. Moreover, they guide developers to refactor their applications and thus, to improve their quality. While antipatterns are well-known in object-oriented applications, their study in mobile applications is still in its infancy. In this paper, we analyze the evolution of mobile apps quality on $3,568$ versions of 106 popular Android applications downloaded from the Google Play Store. For this purpose, we use a tooled approach, called PAPRIKA, to identify 3 object-oriented and 4 Android-specific antipatterns from binaries of mobile apps, and to analyze their quality along evolutions.**

***Keywords*—Android, antipattern, mobile app, software quality.**

I. Introduction

Software evolve over time, inevitably, to cope with the introduction of new requirements, to adapt to new environments, to fix bugs, or to improve software design. However, regardless of the type of changes performed, the software quality may deteriorate as a result of software aging [38].

Software quality declines along evolutions because of the injection of poor design and implementation choices into software. Code smells and antipatterns are symptoms of such bad choices [17]. Additionally, the introduction of antipatterns may affect software maintainability [53], [54], increase the change-proneness [26], the fault-proneness [27], and the complexity [50] of software modules. Therefore, the existence of antipatterns in software is a relevant indicator of its quality.

The research community recognized this issue and proposed several techniques to detect code smells and antipatterns [34], [37]. Most of them are concerned with identifying possible deviations from *Object-Oriented* (OO) principles, such as functional decomposition and encapsulation. Despite the availability of software development kits for mobile apps, specific code smells and antipatterns may still emerge, due to the limitations and constraints on resource like memory, CPU or screen sizes. Mobile apps may also exhibit OO code smells and antipatterns, such as Blob class, which may decrease their quality as they evolve. To the best of our knowledge, most of the proposed techniques and methods for code smells and antipatterns detection in mobile applications do not involve large empirical data to give a genuine picture of the issue, rather they detect antipatterns on a single version of few specific open source systems [49]. This is all the more true in the case of techniques for Android applications as the research field is in its infancy. Also, OO antipatterns and mobile-specific ones are rarely studied together in order to assess the software quality of Android applications.

Regarding software quality along software evolutions, most techniques are based on the analysis of variation of software metrics over time. For instance, studies have been performed on the distribution of bug introduction to software along its evolutions [36] or on an aggregation of metrics as a quality indicator [20]. However, these techniques were not applied to mobile applications.

In this paper, we present a fully automated approach that monitors the evolution of mobile apps and assesses their quality by considering antipatterns. To this end, we propose a software quality estimation technique based on the consistency between software artifacts size and OO and Android antipatterns. To identify these antipatterns from Android applications, we use a metrics-based detection technique. Rather than evaluating each application independently, our approach leverages the whole dataset to detect and evaluate the evolution of antipatterns in each application. The antipatterns detection and quality tracking are automatic processes included in PAPRIKA, our tooled approach. Typically, we intend to answer the two following research questions:

- **RQ1:** Can we infer software quality evolution trends across different versions of Android applications?
- **RQ2:** How does software quality evolve in Android applications?

The rationale behind our approach is the close relationship between antipatterns and software artifacts size. A recent large empirical study [45] showed that most antipatterns are introduced at the creation of software artifacts. The same study indicates that some antipatterns are introduced as a result of several changes performed on software artifacts along evolutions. Based on these findings, we propose to study the variations between software artifacts sizes and the number of antipatterns contained in them. Typically, we intend to track the lack of correlation between the number of antipatterns and the software size along evolutions. Our study is based on a dataset made of 106 Android applications and $3,568$ versions. We choose popular applications from the Google Play Store in order to be representative of complex and up-to-date applications. However, most of these applications are not open-source. Thus, it was necessary to detect the antipatterns at the bytecode granularity. In this way, our approach could be used by both developers and app store providers to track the apps quality evolution at both global and local scales.

The main contributions of this paper are: (1) an approach to track and evaluate the quality of android apps along their evolutions via the detection of antipatterns, (2) the observation of particular relationships between these antipatterns in the case of mobile applications, (3) the identification of 5 quality evolution trends and their possible causes inferred by the analysis of our dataset and, finally (4) an empirical study involving over 3500 version on which we apply our tooled approach.

This paper is organized as follows: Section II gives a brief background on Android apps and bytecode-based techniques. Section III discusses different contributions related to our work. Our automatic tooled approach, PAPRIKA, is detailed in Section IV. The results of the application of our approach are presented in Section V. Section VI concludes the paper.

II. BACKGROUND ON ANDROID PACKAGE AND BYTECODE

This section provides a short overview of the specificities of *Android Application Package* (APK) and Dalvik bytecode.

Android apps are distributed using the APK file format. APK files are archive files in a ZIP format, which are organized as follows: 1) the file `AndroidManifest.xml` describes application metadata including *name*, *version*, *permissions* and *referenced library files* of the application, 2) the directory `META-INF` that contains meta-data and certificate information, 3) an `asset` and a `res` directory containing non-compiled resources, 4) a `lib` directory for eventual native code used as library, 5) a `resources.arsc` file for pre-compiled resources, and 6) a `.dex` file containing the compiled application classes and code in *dex* file format [5]. While Android apps are developed using the Java language, they use the Dalvik Virtual Machine as a runtime environment. The main difference between the *Java Virtual Machine* (JVM) and the Dalvik Virtual Machine is that Dalvik is register-based, in order to be memory efficient compared to the stack-based JVM [5]. The resulting bytecode compiled from Java sources and interpreted by the Dalvik Virtual Machine is therefore different.

Disassembler exists for the Dex format [8] and tools to transform the bytecode into intermediate languages or even Java are numerous [11], [14], [21], [42]. However, there is an important loss of information during this transformation for all the existing approaches. For instance, additional algorithms have to be used to infer the type of local variables or to determine the type of branches as `for`, `while` and `if` constructions are replaced by `goto` instructions in the bytecode [5], [14]. Some dependencies are also absent from the Dex files, resulting in phantom classes, which cannot be analyzed without the source code. And, of course, the native code included in the `lib` directory cannot be decompiled with these tools. It is also important to note that around 30% of all the mobile apps distributed on the Google Play Store are obfuscated [52] in order to prevent reverse-engineering. The ProGuard tool used to obfuscate code is even pre-installed on the beta of Android Studio provided by Google to replace Eclipse ADT [3]. It is likely that code obfuscation will be even more common in the future. With obfuscation, most classes and methods are renamed, often with just one or two alphabetical characters, leading to the loss of most of lexical properties. Fortunately, the application structure is preserved and classes from the Android Framework are not renamed, thus allowing to retrieve some information from the classes that inherit them.

III. RELATED WORK

In this section, we discuss the relevant literature about analysis and antipatterns detection in mobile apps and related work on software evolution.

Mobile apps are mostly developed using OO languages, such as Java or Objective-C. Since their definition by Chidamber and Kemerer [19], OO metrics have gained popularity to assess software quality. Numerous works validated OO metrics to be efficient quality indicators [12], [15], [28], [43]. This has lead to the creation of tooled approaches, such as DECOR [35] or IPLASMA [31], which use OO metrics to detect code smells and antipatterns in OO applications. Most of the code smells and antipatterns, like *long method* or *blob class*, detected by these approaches are inspired by the work of Fowler [23] and Brown *et al.* [17]. These approaches are compatible with Java, but since they were mostly developed before the emergence of mobile apps they are not taking into account the specificities of Android apps and are not compatible with Dex bytecode.

With regard to mobile apps, Linares-Vásquez *et al.* [29] used DECOR to perform the detection of 18 different OO antipatterns in mobile apps built using *Java Mobile Edition* (J2ME) [6]. This large-scale study was performed on $1,343$ apps and shows that the presence of antipatterns negatively impacts the software quality metrics, in particular metrics related to fault-proneness. They also found that some antipatterns are more common with certain categories of Java mobile apps. Specifically in Android apps, Verloop [49] used popular Java refactoring tools, such as PMD [7] or JDEODORANT [44] to detect code smells, like *large class* or *long method* in open-source software. They found that antipatterns tend to appear at different frequencies in classes that inherit from the Android Framework (called core classes) compare to classes which are not (called non-core classes). For example, *long method* was detected twice as much in core classes in term of ratio. However, they did not considered Android-specific antipatterns in both of these studies.

The detection and the specification of mobile-specific antipatterns are still considered as open issues. Reimann *et al.* [39] propose a catalog of 30 quality smells dedicated to Android. These code smells are mainly originated from the good and bad practices documented online in Android documentations or by developers reporting their experience on blogs. They are concerning various aspect like implementations, user interfaces or database usages. They are reported to have a negative impact on properties, such as efficiency, user experience or security. We chose to detect some of these code smells with our approach, which are presented in Section IV-D. We selected antipatterns that can be detected by static analysis and despite code obfuscation. Reimann *et al.* are also offering the detection and correction of code smells via the REFACTORY tool [40]. This tool can detect the code smells from an EMF model. The source code can be converted to EMF if necessary. However, we have not been yet able to execute this tool on an Android app. Moreover, there is no evidence that all the antipatterns of the catalog are detectable using this approach.

Concerning the analysis of Android apps and the study of their specificities, the SAMOA [33] tool allows developers to analyze their mobile apps from the source code. The tool collects metrics, such as the number of packages, lines of code, or the cyclomatic complexity. It also provides a way to visualize external API calls as well as the evolution of metrics along versions and a comparison with other analyzed apps. They performed this analysis on 20 applications and discovered that they are significantly different from classical software systems. They are smaller, and make an intensive usage of external libraries, which leads to a more complex code to understand during maintenance activities. Ruiz *et al.* [41] analyzed Android packages to understand the reuse of classes in Android apps. They extract bytecode and then analyze class signatures for this purpose. They discovered that software reuse via inheritance, libraries, and frameworks is prevalent in mobile apps compared to regular software. Xu [52] also examined APKs of $122,570$ applications and determines that developer errors are common in manifest and permissions. He also analyzed the apps' code and observed that Java reflection and code obfuscation are widely used in mobile apps, making reverse-engineering harder. He also noticed the heavy usage of external libraries in its corpus of analyzed apps. Nonetheless, antipatterns were not considered as part of these studies.

Much work has been done concerning the assessment of software quality throughout the evolution of OO non-mobile applications. Zhu *et al.* [56] monitor software quality throughout the evolution by considering three different modularity views: Package, Structural and Semantic. They assess software quality by computing the deviation trends between the considered modular views to indicate on quality evolution. The idea behind their approach is that if the considered different views are well aligned, then software quality is good. They studied the quality evolution of three open source Java systems. Zhang and Kim [55] propose to study software evolution quality by monitoring the number of defects using *Statistical Process Control* (SPC) and control charts (c-charts). They examined over 60 c-charts representing different Eclipse and Gnome components and identified 6 common quality evolution patterns. The quality evolution patterns serve as indicators of symptomatic situations that development teams should address. For instance, the *Roller Coaster* pattern, which represents large variations of defect numbers, suggests that software quality is unstable and that better management and planning is required to ensure high and consistent quality. Tufano *et al.* [45] conducted a large scale empirical study to investigate the code smells introduction by developers by analyzing the change history of software projects. The authors aim at identifying how code smells are introduced in software along its evolutions. Their major findings are that most code smells are introduced when files are created and new features are developed or existing ones are enhanced. Van Emden and Moonen [48] propose an approach to detect code smells in Java programs. They distinguish between two type of smells: primitive smells which are detected directly from code and derived smells which are deduced from the context. The detected antipatterns are then presented to the developers for inspection and quality assurance using visualization. In this approach, the developers are left with the responsibility of assessing the quality by analyzing the visualization. Our approach provides the developers an estimation of software quality based on antipatterns.

Unlike the above mentioned contributions, and regardless of software quality along evolutions, some studies analyze software evolution to abstract higher level information with the purpose of comprehension. Barry *et al.* [13] propose a method to identify software evolution patterns. The pattern identification is based on software volatility information. Volatility is approximated by computing the amplitude, dispersion, and periodicity of software changes at regular intervals in the software history. Each period is defined by a volatility class, and sequence analysis is applied to reveal similar patterns in time. Xing and Stroulia [51] present an approach for understanding evolution phases and styles of object-oriented systems. The authors use a structural differencing algorithm to compare changing system class models over time, and gather the system's evolution profile. The resulting sequence of structural changes is then analyzed to gain insight about the system's evolution. Finally, Benomar *et al.* [16] investigate the automatic detection of software development phases by studying software evolutions. They use a search-based technique to identify software evolution periods having similar development activities. Search heuristics, such as development rate, importance and type of changes are used to define and understand software evolution.

Our proposed approach aims at identifying hotspots in terms of software quality along evolutions. We consider the analysis of mobile apps evolution and apply a novel approach to detect antipatterns, which we use to estimate the software quality. The proposed approach is automatic and takes as input versions of mobile apps and monitors their quality along evolutions.

IV. PAPRIKA: A TOOLED APPROACH TO DETECT SOFTWARE ANTI-PATTERNS

In this section, we introduce the key components of PAPRIKA, our tooled approach for analyzing the design of mobile apps in order to detect software antipatterns.

A. *Overview of the Approach*

PAPRIKA builds on a four-step approach, which is summarized in Figure 1. As a first step, PAPRIKA parses the APK file of the mobile app under analysis to extract some metadata (*e.g.*, app name, package) and a representation of the code. Additional metadata (*e.g.*, rating, number of downloads) are also extracted from the Google Play Store and passed as arguments. This representation is then automatically visited to compute a model of the code (including classes, methods, attributes) as a graph annotated with a set of raw quality metrics (cf. Section V). As a second step, this model stored into a graph database (cf. Section IV-C). The third step consists in querying the graph to detect the presence of common antipatterns in the code of the analyzed apps (cf. Section IV-D). Paprika is built from a set of components fitting these steps in order to leverage different analyzers, databases or antipatterns detection mechanisms. Each analyzed APK is automatically stored into the database. Thus the database can contains a version history for each analyzed application. Finally, the last step use this version history to compute a software quality evolution score (cf. Section IV-E).

B. Step 1: Collecting Metrics from Application Artifacts

Input: One APK file and its corresponding metadata.

Output: A Paprika quality model including entities, properties and metrics.

Description: This steps consists in generating a model of the mobile app and extracting the raw quality metrics from an input artifact. This model is built incrementally, while analyzing the bytecode, and complemented with properties collected from the Google Play Store. From this representation, Paprika builds a model based on eight entities: App, Class, Method, Attribute, Variable,ExternalClass and External-Method. ExternalClass and ExternalMethod represents entities from the Java API, the Android framework or third-party libraries. The properties described in Table I are attached as attributes to these entities, while they are linked together by the relationships reported in Table II.

TABLE I. List of Paprika properties.

Name	Entities	Comments
name	All	Name of the entity
app_key	All	Unique id of an application
rating	App	Rating on the store
date_download	App	APK download date
date_analysis	App	Date of the analysis
package	App	Name of the main package
size	App	APK size (MB)
developer	App	Developer name
category	App	Category in the store
price	App	Price in the store
nb_download	App	Number of downloads from the store
parent_name	Class	For inheritance
modifier	Class Variable Method	public, protected or private
type	Variable	Object type of the variable
full_name	Method	method_name#class_name
return_type	Method	Return type of the method
position	Argument	Argument position in the method signature

TABLE II. List of Paprika relationships.

Name	Entities
APP_OWNS_CLASS	App – Class
CLASS_OWNS_METHOD	Class – Method
CLASS_OWNS_ATTRIBUTE	Class – Attribute
METHOD_OWNS_ARGUMENT	Method – Argument
EXTENDS	Class – Class
IMPLEMENTS	Class – Class
CALLS	Method – (External)Method
USES	Method – Variable

Paprika proceeds with the extraction of metrics for each entity. The 34 metrics currently available in Paprika are reported in Table III. Paprika supports two kinds of metrics: *OO* and *Android-specific*. Boolean metrics are used to determine different kinds of entities, whereas integers are used for counters or when the metrics are aggregated. Contrary to the properties, metrics often require computation or to process the bytecode representation. For example, it is necessary to browse the inheritance tree in order to determine if a class inherits from some Android Framework-specific fundamentals classes, which include:

- *Activity* represents a single screen on the user interface. Activity may start others activities from the same or a different application;
- *Service* is a task that runs in the background to perform long-running operations or to work for remote processes;
- *Content provider* manages shared data between apps;
- *Broadcast receiver* can listen and respond to system-wide broadcast announcements from the system or other apps;
- *Application* is used to maintain a global application state.

Some composite metrics, such as ClassComplexity, require more computation based on other raw metrics, thus they are computed at the end of the process.

Implementation: We use the Soot framework [47] and its Dexpler module [14] to analyze APK artifacts. Soot converts the Dalvik bytecode of mobile apps into a Soot internal representation, which is similar to the Java language. Soot can also be used to generate the call graph of the mobile app. This model is built incrementally by visiting the internal representation of Soot, and complemented with properties collected from the Google Play Store. Then, Paprika proceeds with the extraction of metrics for each entity by exploring the Soot model. In order to optimize performance and to reduce execution time, these steps are not executed sequentially, but in an opportunistic way while visiting the Soot model.

Compared to traditional approaches for antipattern detection [49], using bytecode analysis instead of source code analysis raises some technical issues. For example, we cannot directly access widely-used metrics, such as the number of lines of codes or the number of declared locals of a method. Therefore, we use abstract metrics, that are approximations of the missing ones, like the number of instructions to approximate the number of lines of code.

Moreover, as mentioned previously, many applications available on Android markets are obfuscated to optimize size and make reverse-engineering harder. Most methods, attributes, and classes are therefore renamed with single letters. Thus, we cannot rely on lexical data to compute some quality metrics and we have to apply some bypass strategies. For instance, to determine the presence of getters/setters, we are not observing the method names, rather we focus on the number and types of instructions as well as the variables accessed by the method.

C. Step 2: Converting Paprika Model as a Graph Model

Input: A Paprika quality model with entities, properties and metrics.

Output: A software quality graph model stored in a database.

Description: We aim at providing a scalable solution to analyze mobile apps at large. We also wants to keep an history of each version analysis. Therefore, we use a graph database as a flexible yet efficient solution to store and query the app model annotated with quality metrics extracted by Paprika.

Since this kind of database is not depending on a rigid schema, the Paprika model is almost as it is described in

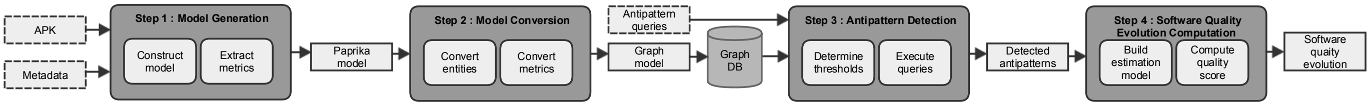


Fig. 1. Overview of the PAPRIKA approach to detect software antipatterns in mobile apps and analyze their evolution.

TABLE III. LIST OF PAPRIKA METRICS

Name	Type	Entities	Comments
NumberOfClasses	OO	App	
NumberOfInterfaces	OO	App	
NumberOfAbstractClasses	OO	App	
NumberOfMethods	OO	Class	
DepthOfInheritance	OO	Class	Integer value, minimum is 1.
NumberOfImplementedInterfaces	OO	Class	
NumberOfAttributes	OO	Class	
NumberOfChildren	OO	Class	
ClassComplexity	OO	Class	Sum of methods complexity, Integer value
CouplingBetweenObjects	OO	Class	Chidamber and Kemerer [19], Integer value
LackofCohesionInMethods	OO	Class	LCOM2 [19], Integer value
IsAbstract	OO	Class, Method	
IsFinal	OO	Class, Variable, Method	
IsStatic	OO	Class, Variable, Method	
IsInnerClass	OO	Class	
IsInterface	OO	Class	
NumberOfParameters	OO	Method	
NumberOfDeclaredLocals	OO	Method	Can be different from source code
NumberOfInstructions	OO	Method	Related to number of lines in source code
NumberOfDirectCalls	OO	Method	Numbers of calls made by the method
NumberOfCallers	OO	Method	Numbers of called made by other methods
CyclomaticComplexity	OO	Method	McCabe [32], Integer value
IsGetter	OO	Method	Computed to bypass obfuscation
IsSetter	OO	Method	Computed to bypass obfuscation
IsInit	OO	Method	Constructor
IsSynchronized	OO	Method	
NumberOfActivities	Android	App	
NumberOfBroadcastReceivers	Android	App	
NumberOfContentProviders	Android	App	
NumberOfServices	Android	App	
IsActivity	Android	Class	
IsApplication	Android	Class	
IsBroadcastReceiver	Android	Class	
IsContentProvider	Android	Class	
IsService	Android	Class	

the previous section. All PAPRIKA entities are represented by nodes, their attributes and metrics are properties attached to these nodes. The relationships between entities are represented by one-way edges.

Implementation: We selected the graph database NEO4J [10] and we used its Java-embedded version. We chose NEO4J because, when combined with the CYPHER [9] query language, it offers good performance on large-scale datasets, especially when embedded in Java [25]. Furthermore, NEO4J is also able to contain a maximum of 2^{35} nodes and relationships, which match our scalability requirements. Finally, NEO4J offers a straightforward conversion from the PAPRIKA quality metrics model to the graph database.

D. Step 3: Detecting Anti-patterns from Graph Queries

Input: A graph database containing a model of the mobile apps to analyze and the antipatterns queries.

Output: Software antipatterns detected in the applications.

Description: Once the model loaded and indexed by the graph database, we use the database query language to detect common software antipatterns. Entity nodes which implements antipatterns are returned as results for all analyzed applications. The results are grouped by versions.

Implementation: We use the CYPHER query language [9] to detect common software antipatterns as illustrated in Listings 1 and 2.

All OO antipatterns are detected using a threshold to identify abnormally high value from others commons values. To define such thresholds, we collect all the values of a specific metric for the whole dataset and we identify outliers. We use a Tukey Box plot [46] for this task. All values superior to the upper whisker are considered as very high whereas all values inferior to the lower one are very low. The upper border of the box represents the *first quartile* (Q1) whereas the lower border is the *third quartile* (Q3), the distance between Q1 and Q3 is called the *interquartile range* (IQR). The upper whisker value is given by the formula $Q3 + 1.5 \times IQR$, which is equal to 15 for the number of methods in our example for Blob Class as described in Listing 1. It means that if the number of methods exceeds 15, then it is considered as an outlier and can be tagged as a class containing a high number of methods. By combining the three thresholds, we are able to detect Blob classes. The usage of this statistical method allows us to set thresholds that are specific to the input dataset, consequently results may vary depending on the mobile apps included in the analysis process. Thus, the thresholds are representative of all applications in the dataset and not only the currently analyzed application.

Currently, PAPRIKA supports 7 antipatterns, including 4 Android-specific antipatterns:

Blob Class (BLOB) - OO: A Blob class, also know as *God class*, is a class with a large number of attributes and/or methods [17]. The Blob class handles a lot of responsibilities compared to other classes. Attributes and methods of this class are related to different concepts and processes, implying a very low cohesion. Blob classes are also often associated with numerous data classes. Blob classes are hard to maintain and increase the difficulty to modify the software. In PAPRIKA, classes are identified as Blob classes whenever the metrics `numbers_of_attributes`, `number_of_methods` and `lack_of_cohesion_in_methods` are *very high*. The CYPHER query for this antipattern is described in Listing 1.

Listing 1. CYPHER query to detect a Blob class.

```
MATCH (cl:Class)
WHERE cl.lack_of_cohesion_in_methods > 20
  AND cl.number_of_methods > 15
  AND cl.number_of_attributes > 8
RETURN cl
```

Long Method (LM) - OO: Long methods are implemented with much more lines of code than other methods. They are often very complex, and thus hard to understand and maintain. These methods can be split into smaller methods to fix the problem [23]. PAPRIKA identifies a *long method* when the number of instructions for one method is very high.

Complex Class (CC) - OO: A *complex class* is a class containing complex methods. Again, these classes are hard to understand and maintain and need to be refactored [23]. The class complexity is calculated by summing the internal methods complexities. The complexity of a method can be calculated using McCabe's Cyclomatic Complexity [32].

Member Ignoring Method (MIM) - Android: In Android, when a method does not access to an object attribute, it is recommended to use a static method. The static method invocations are about 15%–20% faster than a dynamic invocation [2], [18]. PAPRIKA can detect such methods since the access of an attribute by a method is extracted during the analysis phase. The CYPHER query for this antipattern is described in Listing 2. This request explores node properties to return non-static methods that are not constructors and relations to detect methods that are not using any class variable nor calling other methods. Such detected methods could have been made static to increase performance without any other consequences on the implementation.

Listing 2. Cypher query to detect Member Ignoring Method.

```
MATCH (m:Method)
WHERE NOT HAS(.`is_static`)
AND NOT HAS(m.is_init)
AND NOT m-[:USES]->(:Variable)
AND NOT (m)-[:CALLS]->(:Method)
RETURN m
```

Leaking Inner Class (LIC) - Android: In Java, non-static inner and anonymous classes are holding a reference to the outer class, whereas static inner classes are not. This could provoke a memory leak in Android systems [18], [30]. Given that the PAPRIKA model contains all inner classes of the application and a metric to identify static class is attached to classes, *leaking inner classes* are detected by a dedicated query in PAPRIKA.

UI Overdraw (UIO) - Android: The Android Framework API provides two methods in the `Canvas` Class (`clipRect() quickReject()`) to avoid the overdraw of non-modified part of the UI. The usage of these method is recommended by Google to increase app performance and improve user experience [1]. Thus, we are using the data stored in PAPRIKA database external API calls to detect views which not use one of these methods.

Heavy Broadcast Receiver (HBR) - Android: Similarly to services, android broadcast receivers can trigger ANR when their `onReceive()` methods perform lengthy operations [4]. Therefore, we are using the same heuristic to detect them.

E. Step 4: Computing Software Quality using Anti-patterns

Input: The Android application detected antipatterns.

Output: Evolution of software quality score throughout the mobile applications versions.

Description: This step consists of scoring each version of the mobile application. The score serves as an estimation of the mobile app quality in a particular version. Our estimation of software quality is based on the consistency between applications size and the number of detected antipatterns. Typically, we compute a quality score for each version of the application and track the values of the score along the entire history of the application (successive versions).

Implementation: To compute our software quality score, we first build an estimation model using linear regression. The linear regression model represents the relationship between the number of antipatterns and the size of an application. The linear regression is computed by minimization of squared residuals. The number of classes is used at explanatory variable for *BLOB* (r=0.90), *LM* (r=0.91), *CC* (r=0.92), *MIM* (r=0.80). The number of inner classes is used for *LIC* (r=0.80). The number of views and the number of broadcast receivers are respectively used for *UIO* (r=0.95) and *HBR* (r=0.88). Then, the software quality score of an application at a particular version, with a value for the number of antipatterns and a value of size, is computed as the additive inverse of the residual. A larger positive residual value suggests worst software quality because it means that the current observed version of the application has more antipatterns with respect to its size than the norm (linear regression). Conversely, a larger negative residual value implies better quality because of the lower number of antipatterns in the current observed version. Figure 2 ilustrates how the quality score is estimated from the linear regression computed from our dataset.

Additionally, software quality score at any version is computed as an aggregation of the software quality scores of the preceding versions. The aggregation is performed by computing the average of previous versions scores. Hence, the

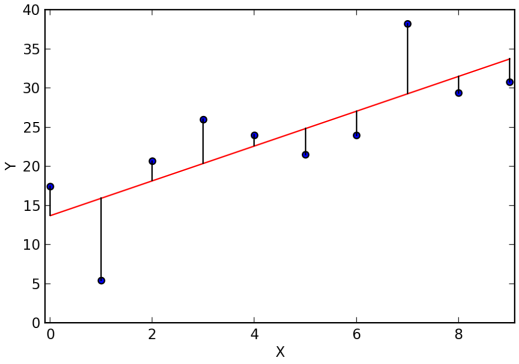


Fig. 2. Example of residuals for linear regression used for quality estimation.

effect of past versions is propagated along the evolution of the application. The scores are computed for each antipattern.

V. Empirical Study

A. Dataset

As mentioned before, this study is based on a large dataset of Android applications downloaded from the Google Play Store. The fact that Paprika operates on the applications' binaries allowed us to gather a wide variety of Android applications. The only challenge we had in data collection was to find a sufficient number of versions of applications so that our tracking of software quality makes sense.

We consider 106 Android applications that differ both in internal attributes, such as their size, and external attributes from the perspective of end users. Our tooled approach, Paprika, is used to track software quality of Android applications. To this end, we collected several versions of each applications to form a total of $3,568$ versions, which we used to estimate software quality. These apps were collected between June 2013 and June 2014 from the top 150 of each category of the Google Play Store. Each app has a minimum of 20 versions. The detailed list of versions is located at https://goo.gl/MktCYM. We present the empirical data from different viewpoints, to illustrate the diversity of our dataset.

1) Category: We classify the downloaded applications according to the category they belong to. All the 24 categories of Google Play Store are represented in our dataset. Figure 3 shows the number of applications per category. For example, *Twitter* app belongs to the category *Social* in the Google Play Store. Also, the proportions revealed by Figure 3 are representative of those found in the Store. In particular, we notice that the majority of apps belong to one of four categories: *Social*, *Comunication*, *Productivity* and *Photography*.

2) Ranking: We describe our dataset from the end users rankings viewpoint. The distribution of applications, according to their ranking, is presented in Figure 4. We observe that most of the applications are ranked above 4.0 (around 90%) and values follow a normal distribution. Applications ranking scores are based on a Likert scale and thus, are computed as the mean of ordinale type values. Although, this is not desirable from measurement theory perspective [22], these scores give some insight about the feeling of end-users towards these applications.

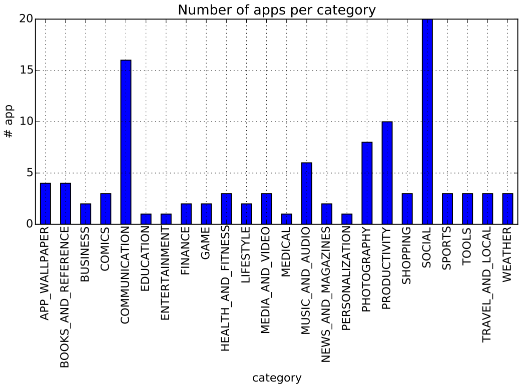


Fig. 3. Distribution of the Android application with regards to their category.

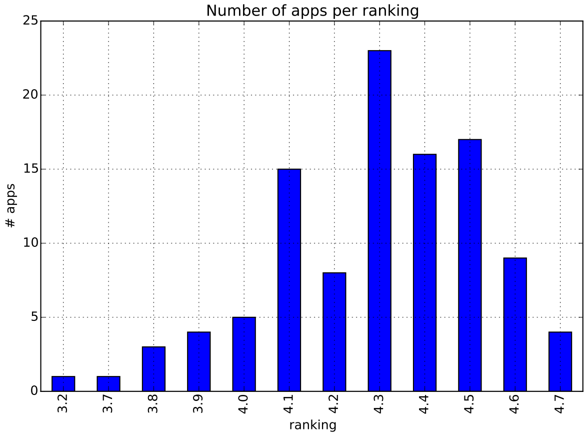


Fig. 4. Distribution of the Android application with regards to their ranking.

3) Download: The number of downloads of Android application is also a good indicator of its popularity among end-users. Our dataset includes applications that vary in number of downloads. The number of downloads is one of the metrics that characterizes android applications in the Google Play Store and is advertised as a token of popularity among end users, just as the ranking score.

An interesting finding about applications downloads is its correlation with the application size. Figure 5 plots the average number of classes in applications versus the number of downloads. This suggests that larger applications are more popular, probably providing more valuable features to the end-users compared to small apps. This observation also supports the claim that mobile applications are becoming complex software systems.

4) Versions: The number of versions of an application may indicate its maturity and the more versions the more stable, reflected in the appreciation of end-users. However, we can only collect information about relative time of versions (their order in time basically) and no information of the absolute time or the interval periods between versions. Therefore, we cannot speculate on the relationship between frequency of releases

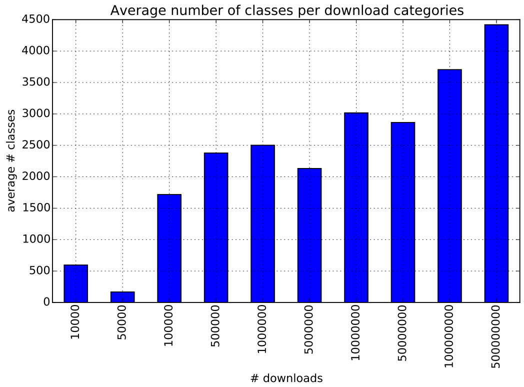


Fig. 5. Average number of classes in applications per number of downloads.

and end users ranking scores.

5) Size: The Android applications analyzed in this study vary in sizes. Among all the versions considered, the sizes in terms of number of classes range from 8 to over $9,000$ classes. In general, larger applications have more versions, better ranking scores and are more downloaded. We also calculated the size in terms of special classes, such as number of activities, number of views, etc.

B. Analysis Results

We applied PAPRIKA on all the 106 mobile apps and generated evolution graphs for each application and each antipattern. Overall, we analyzed over 700 graphs representing the evolution of the quality scores discussed in Section IV. As suspected, the evolution of quality varies from one application to another and thus there is no general trend that reveals itself across all applications. This is due to the disparity between mobile apps in terms of antipatterns and size ratios. For instance, CAMERA360 app has 0.75 MIM antipatterns per class on average while TO DO CALENDAR PLANNER app has 2.5 MIM antipatterns per class. However, we could observe relationships across different antipatterns evolution scores and report some evolution trends of mobile apps quality.

1) Relationships between antipatterns: Our estimation of quality is based on the correlation between software antipatterns and software entities. Some antipatterns have better correlation with the number classes in mobile apps, such a *BLOB*, others are more closely related to the number of views (*e.g.*, *UIO*), the number of inner classes (*e.g.*, *LIC*) or the number of brodcast receivers (*e.g.*, *HBR*).

We observe that software evolution graphs based on antipatterns, which correspond to the same type of software entities, are similar. The type of software entity with which an antipattern is correlated comes directly from the antipattern definition. Here, we consider that antipatterns *LM* and *MIM* are correlated to classes although, in reality, they are correlated to the methods of classes. This approximation is supported by the correlation score computed between number of classes and *LM* and *MIM* antipatterns (0.8 for both antipatterns). Figure 6 illustrates the similarity of the evolution score for *BLOB*, *CC*, *LM* and *MIM* antipatterns for the IMO app. Notice that the 4 quality scores drop and rise in the same manner.

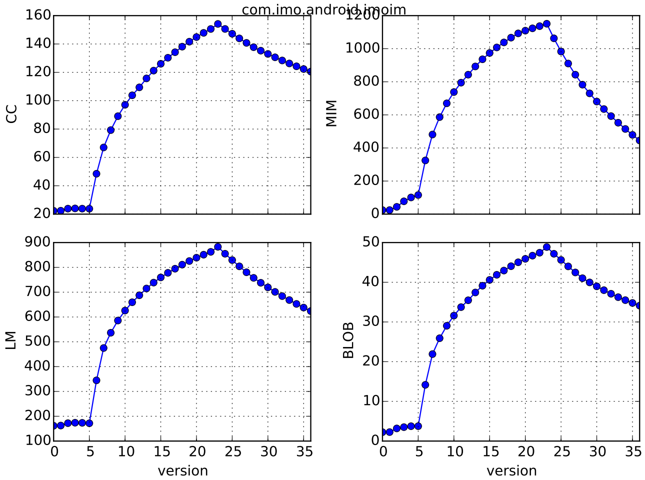


Fig. 6. Software quality scores of IMO app based on *BLOB*, *CC*, *LM* and *MIM* antipatterns.

In some special cases, we observe that the quality score based on *BLOB* and the one based on *CC* or *LM* evolve in opposite manner. Figure 7 shows the evolution of the quality scores based on *BLOB* and on *LM* for RETRO CAMERA app. This situation happens when *BLOB* antipattern "absorbs" *CC* and/or *LM* antipatterns. Often, a *BLOB* class is also a complex class (*CC*) and contains long complex methods (*LM*). Such classes tend to reduce the complexity of surrounding classes (*CC*) as well as reducing the length of their methods (*LM*).

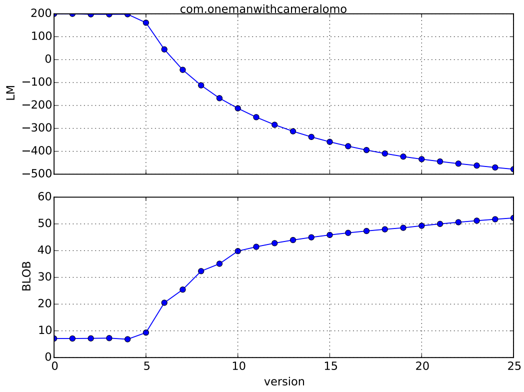


Fig. 7. Software quality scores of RETRO CAMERA app based on *BLOB* and *LM* antipatterns.

In the case of quality scores based on LIC, UIO and HBR, the evolution trends are different. These antipatterns are correlated to inner classes, views and broadcast receivers, which are special types of classes. They heavily depend on the mobile app implementation and hence have different evolutionary paths than the rest of the application. In particular, the number of views and the number broadcast receivers do not necessarily change during large periods of the evolution.

2) Quality Evolution Trends: From the analysis of software quality evolution graphs of the apps and antipatterns considered in our study, we have established 5 major quality evolution trends to answer RQ1 : Can we infer software quality evolution trends across different versions of Android applications?

A) Constant Decline: In general, application size increases along evolutions and new antipatterns are introduced. This evolution trend implies that no action has been done to fix the introduced antipatterns. Hence, quality declines because new antipatterns are introduced without fixing older ones. This is the most encountered quality evolution trend and can be present for either part of the application's evolution or for the entire evolution. Figure 8 shows the constant decline in quality based on LM antipattern.

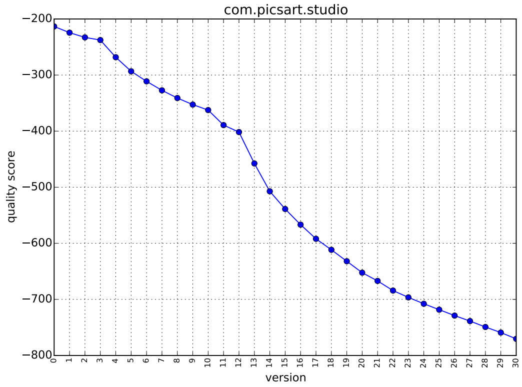


Fig. 8. Software quality scores of PICSART app based on LM antipattern.

B) Constant Rise: Despite the fact that application size evolves along evolutions, there are situations where its evolution seems controlled and less arbitrary. Such situations where development teams follow rigorous programming standards and where software evolution is well structured, exhibit a constant increase in quality over time regardless of changes to applications size. For example, FLIPBOARD app size is changed twice in the 26 considered versions, on versions 19 and 24, but it remains constant otherwise. Figure 9 depicts the constant rise in software quality.

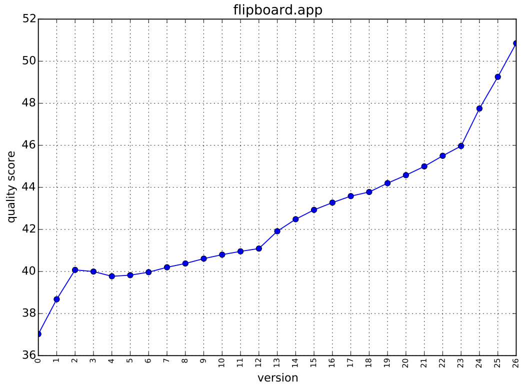


Fig. 9. Software quality scores of FLIPBOARD app based on CC antipattern.

C) Stability: This evolution trend represents periods where software quality is constant. The ratio of introduced antipatterns versus application size is comparable along several consecutive versions. Usually, this trend appears during a finite period, such as in the first 10 versions of FIREFOX app in the LIC based quality shown in Figure 10.

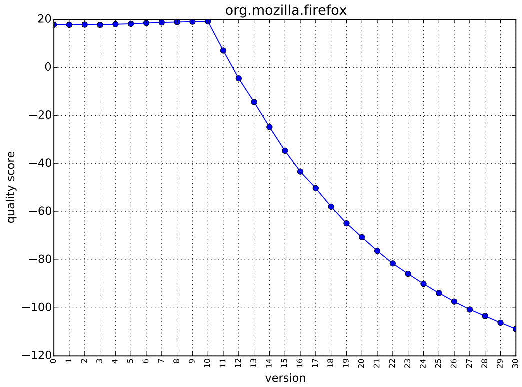


Fig. 10. Software quality scores of FIREFOX app based on LIC antipattern.

D) Sudden Decline: There are cases where the quality score drops abruptly at a particular version. It indicates a turning point in the evolution of the mobile app and is accompanied by a large variation in application size. This drop in quality at a certain version is propagated to the following versions and quality remains low as shown in Figure 11. The rapid decline in quality score of EVERNOTE app, happens in versions 5 and 6, then quality stabilizes, but stays low until version 10 to start a constant rise trend.

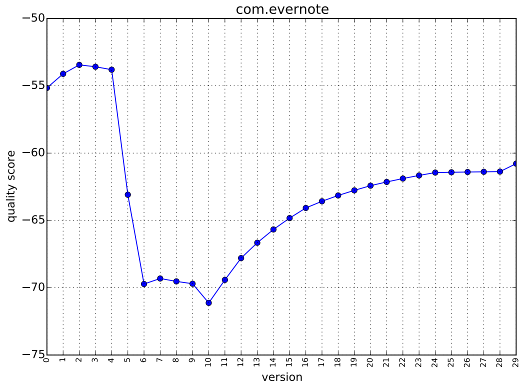


Fig. 11. Software quality scores of EVERNOTE app based on BLOB antipattern.

E) Sudden Rise: It represents the inverse of the previous trend (*D*) and is characterized by a rapid increase in quality score. Figure 12 illustrates the sudden rise trend in LIC antipattern-based quality of SKYPE app at version 5. Similarly, we observe the propagation of the good quality to successive versions (constant good quality). The rapid increase in quality occurs at the same time of major application size change. This suggests that developers perform refactorings to improve the code structure and enforce solid programming principles, which reflects on quality.

C. Case Study: Twitter

This section presents the quality evolution analysis of TWITTER app as an example of how we can utilize our tooled approach, PAPRIKA, to answer RQ2 : How does software quality evolve in Android applications?

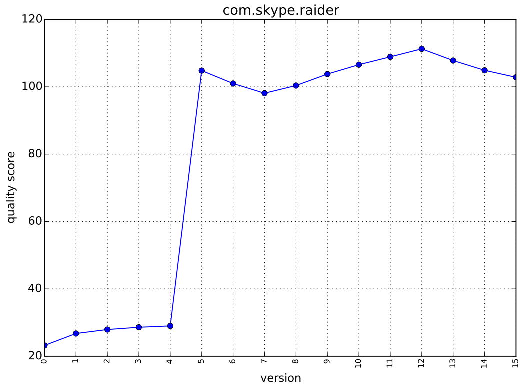


Fig. 12. Software quality scores of SKYPE app based on LIC antipattern.

TWITTER app is given a ranking score of 4.1 by end users. Also, it has been downloaded over $100M$ times, which indicates its high popularity. One question worth asking: how does TWITTER quality evolved over time? To this end, we applied PAPRIKA on the 75 versions of the app collected.

On the one hand, we investigate the changes in terms of number of classes that TWITTER app undergoes during its evolution. Figure 13 plots the number of classes per version. From the size information, we observe several situations of interest, which we will use throughout this case study. First, there is a large addition of classes from version 9 to 12 (from about 530 to over $2,800$). Second, the app size constantly rises from versions 19 to 47 and from versions 48 to 74. However, there is a size drop in the middle of this growth in both cases (versions 31 and 59), which suggests minor refactorings. Finally, TWITTER app size declines drastically at version 48 (from about $4,700$ to below $3,000$). This implies that major refactorings were performed at version 48.

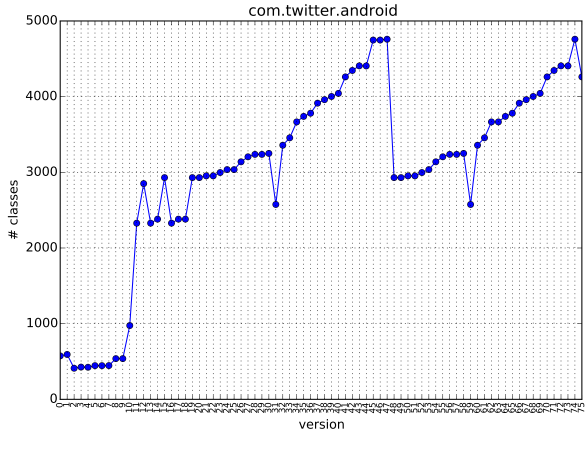


Fig. 13. Size changes of TWITTER app during evolution.

On the other hand, we analyze the evolution of the quality scores computed by PAPRIKA. For instance, Figure 14 presents the CC antipattern-based quality score per version. We focus on what happens to quality at the situations of interest of app size explained before. First, the large addition of classes to the app after version 9 corresponds to a rise in quality, which indicates that new additions occurred with better design that improved quality. This amelioration continues until version 31 when quality starts a constant decline trend. This coincides with the minor refactorings performed at that version as shown in Figure 13. The same pattern happens again at version 59, which is consistent with the app size situations. Finally, the sudden drop in size at version 48 enhanced the quality. This indicates that many antipatterns problems were resolved while doing these major refactorings. The quality keeps rising until the following refactorings at version 59 as mentioned before.

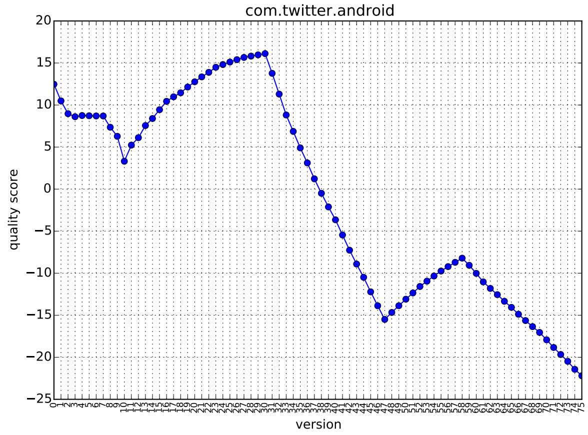


Fig. 14. Software quality scores of TWITTER app based on CC antipattern.

VI. CONCLUSION AND FUTURE WORK

In this paper, we introduced PAPRIKA, a tooled approach to monitor the evolution of mobile apps quality based on antipatterns. The process is fully automatic and takes as input mobile apps in the form of Android application packages. PAPRIKA is robust to code obfuscation and identifies antipatterns from applications' bytecode. The antipattern detection is based on software metrics computed by the tool. We consider 3 Object Oriented antipatterns and 4 Android antipatterns. The detected antipatterns are utilized in the evaluation of mobile apps quality. First, we empirically compute a baseline of software quality from the large set of mobile applications collected. Then, the quality of a mobile app is estimated as the deviation from the baseline. In this manner, we analyze the evolution of a mobile quality by propagating the quality at a particular version to its subsequent versions. We believe that our tooled approach could be useful for both Android app developers and App Store providers since PAPRIKA can help them to evaluate the quality of one or thousands of app versions. As future work, we intend to analyze the evolution of external attributes of mobile apps and investigate their potential correlation between mobile apps quality. An external attribute of mobile apps could be the end users impressions. We would use sentiment analysis [24] to track the evolution of end users feelings towards apps along with apps quality evolution.

Acknowledgments

The authors thank Kevin Allix and Jacques Klein from the University of Luxembourg for their help with the dataset. This study is supported by NSERC and FRQNT research grants.

References

[1] Android Performance Patterns: Overdraw, Cliprect, QuickReject. https://youtu.be/vkTn3Ule4Ps?list=PLWz5rJ2EKKc9CBxr3BVjPTPoDPLdPIFCE. [Online; accessed May-2015].

[2] Android Performance Tips. http://developer.android.com/training/articles/perf-tips.html. [Online; accessed May-2015].

[3] Android studio. https://developer.android.com/sdk/installing/studio.html. [Online; accessed May-2015].

[4] AntiPattern: freezing a UI with Broadcast Receiver. http://gmariotti.blogspot.fr/2013/02/antipattern-freezing-ui-with-broadcast.html. [Online; accessed May-2015].

[5] Dalvik bytecode. https://source.android.com/devices/tech/dalvik/dalvik-bytecode.html. [Online; accessed May-2015].

[6] Java platform, micro edition (java me). http://www.oracle.com/technetwork/java/embedded/javame/index.html. [Online; accessed May-2015].

[7] Pmd. http://pmd.sourceforge.net/. [Online; accessed November-2014].

[8] Smali: An assembler/disassembler for android's dex format. https://code.google.com/p/smali. [Online; accessed May-2015].

[9] CYPHER. http://neo4j.com/developer/cypher-query-language. [Online; accessed May-2015].

[10] NEO4J. http://neo4j.com. [Online; accessed May-2015].

[11] Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar. [Online; accessed May-2015].

[12] K. Aggarwal, Y. Singh, A. Kaur, and R. Malhotra. Empirical analysis for investigating the effect of object-oriented metrics on fault proneness: a replicated case study. *Software process: Improvement and practice*, 14(1):39–62, 2009.

[13] E. Barry, C. Kemerer, and S. Slaughter. On the uniformity of software evolution patterns. In *Proc. of the International Conference on Software Engineering*, pages 106–113, May 2003.

[14] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In *Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis*, pages 27–38. ACM, 2012.

[15] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, 1996.

[16] O. Benomar, H. Abdeen, H. Sahraoui, P. Poulin, and M. A. Saied. Detection of Software Evolution Phases based on Developement Activities. In *Proc. of the 23rd International Conference on Program Comprhension*, page To appear. IEEE, 2015.

[17] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau. *AntiPatterns: refactoring software, architectures, and projects in crisis*. Wiley New York, 1. auflage edition, 1998.

[18] M. Brylski. Android Smells Catalogue. http://www.modelrefactoring.org/smell_catalog, 2013. [Online; accessed May-2015].

[19] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[20] N. Drouin, M. Badri, and F. Tour. Analyzing Software Quality Evolution using Metrics: An Empirical Study on Open Source Software. *Journal of Software*, 8(10), 2013.

[21] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *USENIX security symposium*, volume 2, page 2, 2011.

[22] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[23] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[24] E. Guzman and W. Maalej. How Do Users Like This Feature? A Fine Grained Sentiment Analysis of App Reviews. In *Proc of the 22nd IEEE International Requirements Engineering Conference (RE)*, pages 153–162, Aug 2014.

[25] F. Holzschuher and R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In *Proc. of the Joint EDBT/ICDT 2013 Workshops*, pages 195–204. ACM, 2013.

[26] F. Khomh, M. Di Penta, and Y. Guéhéneuc. An exploratory study of the impact of code smells on software change-proneness. In *16th Working Conference on Reverse Engineering (WCRE'09)*, pages 75–84, Oct 2009.

[27] F. Khomh, M. D. Penta, Y.-G. Guéhéneuc, and G. Antoniol. An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-proneness. *Empirical Softw. Engg.*, 17(3):243–275, June 2012.

[28] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. *Journal of systems and software*, 80(7):1120–1128, 2007.

[29] M. Linares-Vásquez, S. Klock, C. McMillan, A. Sabané, D. Poshyvanyk, and Y.-G. Guéhéneuc. Domain matters: bringing further evidence of the relationships among anti-patterns, application domains, and quality-related metrics in java mobile apps. In *Proc. of the 22nd International Conference on Program Comprehension*, pages 232–243. ACM, 2014.

[30] A. Lockwood. How to Leak a Context: Handlers and Inner Classes. http://www.androiddesignpatterns.com/2013/01/inner-class-handler-memory-leak.html, 2013. [Online; accessed May-2015].

[31] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*. Citeseer, 2005.

[32] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320, 1976.

[33] R. Minelli and M. Lanza. Software Analytics for Mobile Applications–Insights and Lessons Learned. In *17th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 144–153. IEEE, 2013.

[34] N. Moha, Y. Guéhéneuc, L. Duchien, and A. Le Meur. DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, 36(1):20–36, Jan 2010.

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

[36] A. Murgia, G. Concas, S. Pinna, R. Tonelli, and I. Turnu. Empirical study of software quality evolution in open source projects using agile practices. *CoRR*, abs/0905.3287, 2009.

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

[38] D. L. Parnas. Software Aging. In *Proc. of the 16th International Conference on Software Engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[39] J. Reimann, M. Brylski, and U. Amann. A Tool-Supported Quality Smell Catalogue For Android Developers, 2014.

[40] J. Reimann, M. Seifert, and U. Aßmann. On the reuse and recommendation of model refactoring specifications. *Software & Systems Modeling*, 12(3):579–596, 2013.

[41] I. J. M. Ruiz, M. Nagappan, B. Adams, and A. E. Hassan. Understanding reuse in the android market. In *20th International Conference on Program Comprehension (ICPC)*, pages 113–122. IEEE, 2012.

[42] M. Schönefeld. Reconstructing dalvik applications. In *10th annual CanSecWest conference*, 2009.

[43] Y. Singh, A. Kaur, and R. Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software quality journal*, 18(1):3–35, 2010.

[44] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. JDeodorant: Identification and removal of type-checking bad smells. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 329–331. IEEE, 2008.

[45] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk. When and why your code starts to smell bad. In *Proc. of the 37th International Conference on Software Engineering*, page To appear. IEEE/ACM, 2015.

[46] J. W. Tukey. *Exploratory Data Analysis*. Addison-Wesley, 1977.

[47] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proc. of the conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[48] E. Van Emden and L. Moonen. Java Quality Assurance by Detecting Code Smells. In *Proc. of the 9th Working Conference on Reverse Engineering (WCRE'02)*, WCRE '02, pages 97–, Washington, DC, USA, 2002. IEEE Computer Society.

[49] D. Verloop. *Code Smells in the Mobile Applications Domain*. PhD thesis, TU Delft, Delft University of Technology, 2013.

[50] G. Xie, J. Chen, and I. Neamtiu. Towards a better understanding of software evolution: An empirical study on open source software. In *IEEE International Conference on Software Maintenance (ICSM'09).*, pages 51–60, Sept 2009.

[51] Z. Xing and E. Stroulia. Understanding phases and styles of object-oriented systems' evolution. In *Intl. Conf. Softw. Maint.*, pages 242–251. IEEE, 2004.

[52] L. Xu. *Techniques and Tools for Analyzing and Understanding Android Applications*. PhD thesis, University of California Davis, 2013.

[53] A. Yamashita and L. Moonen. Do code smells reflect important maintainability aspects? In *28th IEEE International Conference on Software Maintenance (ICSM)*, pages 306–315, Sept 2012.

[54] A. Yamashita and L. Moonen. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proc. of the 2013 International Conference on Software Engineering*, ICSE '13, pages 682–691, Piscataway, NJ, USA, 2013. IEEE Press.

[55] H. Zhang and S. Kim. Monitoring Software Quality Evolution for Defects. *Software, IEEE*, 27(4):58–64, July 2010.

[56] T. Zhu, Y. Wu, X. Peng, Z. Xing, and W. Zhao. Monitoring software quality evolution by analyzing deviation trends of modularity views. In *Proc. of 18th Working Conference on Reverse Engineering*, pages 229–238, Oct 2011.

High-level Language Support for the Control of Reconfigurations in Component-based Architectures

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier

To cite this version:

Frederico Alvares de Oliveira Jr., Eric Rutten, Lionel Seinturier. High-level Language Support for the Control of Reconfigurations in Component-based Architectures. 9th European Conference on Software Architecture (ECSA), Sep 2015, Dubrovnick, Croatia. Springer, 9278, pp.285-293, 2015, LNCS. <http://ecsa-conference.org/2015/index.php>. <hal-01160612>

HAL Id: hal-01160612
https://hal.inria.fr/hal-01160612

Submitted on 9 Sep 2015

High-level Language Support for Reconfiguration Control in Component-based Architectures

Frederico Alvares[1], Eric Rutten[1], and Lionel Seinturier[2]

[1] INRIA Grenoble, France– {frederico.alvares,eric.rutten}@inria.fr
[2] University of Lille 1 & INRIA Lille, France– lionel.seinturier@inria.fr

Abstract. Architecting in the context of variability has become a real need in todays software development. Modern software systems and their architecture must adapt dynamically to events coming from the environment (e.g., workload requested by users, changes in functionality) and the execution platform (e.g., resource availability). Component-based architectures have shown to be very suited for self-adaptation especially with their dynamical reconfiguration capabilities. However, existing solutions for reconfiguration often rely on low level, imperative, and non formal languages. This paper presents Ctrl-F, a domain-specific language whose objective is to provide high-level support for describing adaptation behaviours and policies in component-based architectures. It relies on reactive programming for formal verification and control of reconfigurations. We integrate Ctrl-F with the FraSCAti Service Component Architecture middleware platform, and apply it to the Znn.com self-adaptive case study.

1 Introduction

From tiny applications embedded in house appliances or automobiles to huge Cloud services, nowadays software-intensive systems have to fulfill a number of requirements in terms safety and Quality of Service (QoS) while facing highly dynamic environments (e.g., varying workloads and changing user requirements) and platforms (e.g., resource availability). This leads to the necessity to engineer such software systems with principles of self-adaptiveness, i.e., to equip these software systems with capabilities to cope with dynamically changes.

Component-based Architecture. Software architecture and more specifically software components have played a very important role in self-adaptiveness. Besides the usual benefits of modularity and reuse, adaptability and reconfigurability are key properties which are sought with this approach: one wants to be able to adapt the component assemblies in order to cope with new requirements and new execution conditions occurring at run-time. Component-based Architecture defines the high-level structure of software systems by describing how they are organized by the means of a composition of components [15], which are usually captured by an Architecture Description Languages (ADL). In spite of the diversity of ADLs, the architectural elements proposed in almost all of them follow the same conceptual basis [13]. A *component* is defined as the most elementary unit of processing or data and it is usually decomposed into two parts: the implementation and the *interface*. The implementation describes the internal behaviour of the component, whereas the interfaces define how the component should interact with the environment. A component can be defined as *atomic* or *composite*, i.e., composed of other components. A *connector* mediates diverse forms of interactions of inter-component communications, and *configuration* corresponds to a directed graph of components and connectors describing the application's structure. Other elements like attributes, constraints or architectural styles also appear in ADLs [13], but for brevity we omit further details on these elements.

Initial assemblies (or configurations) are usually defined with the help of ADLs, whereas adaptive behaviours are achieved by programming fine-grained actions (e.g., to add, remove, connect elements), in either general-purpose languages within reflective component-based middleware plaforms [20], or with the support of reconfiguration domain-specific languages (DSLs)[8]. This low level of abstraction may turn the definition of transitions among configurations into a very costly task, which consequently may lead to error-prone adaptive behaviours. In fact, it may be non-trivial, especially for large and complex architectures, to obtain assurances and guarantees about the result of these reconfiguration behaviours. We claim that there is a need for a language, not only for the definition of configurations in the form of component assemblies, but also for the explicit specification of the transitions among them and the policies driving when and under which conditions reconfigurations should be triggered.

This paper presents Ctrl-F, a language that extends classic ADLs with high-level constructs to express the dynamicity of component-based architectures. In addition to the usual description of assemblies (configurations), Ctrl-F also comprises a set of constructs that are dedicated for the description of: (i) behavioural aspects, that is, the order and/or conditions under which reconfigurations take place; and (ii) policies that have to be enforced all along the execution.

Heptagon/BZR. We formally define the semantics of Ctrl-F with Heptagon/BZR [10], a Reactive Language based on Finite State Automata (FSA). It allows for the definition of generalized Moore machines, with mixed data-flow equations and automata. A distinguished characteristics is that its compilation involves formal tools for Discrete Controller Synthesis (DCS): a controller is automatically generated so as to enforce that a system behaves at runtime in concordance with the specification. The Heptagon/BZR definition of Ctrl-F programs allows to benefit from: (i) guarantees on the correctness of adaptive behaviours by either verification or control (i.e., by DCS); (ii) the compilation of adaptive behaviours towards executable code in general purpose languages (e.g., Java or C). Due to space limitation, the detailed definition is reported on elsewhere [1].

In the remainder of this paper, Sections 2 presents the self-adaptation case study Znn.com [7], used all along the paper. Section 3 presents the Ctrl-F language. Section 4 provides some details on its integration with a real component platform as well as the evaluation of its applicability through *Znn.com*. Related work is discussed in Section 5 and Section 6 concludes this paper.

2 The Znn.com Example Application

Znn.com [7] is an experimental platform for self-adaptive applications, which mimics a news website. Znn.com follows a typical client-server n-tiers architecture where a load balancer redirects requests from clients to a pool of replicated servers. The number of active servers can be regulated in order to maintain a good trade-off between response time and resource utilization. Hence, the objective of Znn.com is to provide news content to its clients/visitors within a reasonable response time, while keeping costs as low as possible and/or under control (i.e., constrained by a certain budget).

At times, the pool of servers is not large enough to provide the desired QoS. For instance, in order to face workload spikes, Znn.com could be forced to degrade the content fidelity so as to require fewer resource to provide the same level of QoS. For this, Znn.com servers are able to deliver news contents with three different content fidelity: (i) high quality images, (ii) low quality images, and (iii) only text. The objectives are: (1) Keep the performance (response time) as high as possible; (2) Keep content fidelity as high as possible or above a certain threshold; (3) Keep the number of active servers as low as possible or under a certain threshold. In order to achieve them, we may tune: (1) The number of active servers; (2) The content fidelity of each server.

As a running example for our proposal, in the next section, we extend Znn.com by enabling its replication in presence of different content providers: one specialized in soccer and another one specialized in politics. These two instances of Znn.com will be sharing the same physical infrastructure. Depending on the contract signed between the service provider and his/her clients that establishes the terms of use of the service, Znn.com Service Provider can give more or less priority to a certain client. For instance, during the World Cup the content provider specialized in soccer will always have priority over the other one. Conversely, during the elections, the politics-specialized content provider is the one that has the priority.

3 Ctrl-F Language

3.1 Overview and Common Concepts

Ctrl-F is our proposal for a domain-specific language that extends classic ADLs with high-level constructs for describing reconfigurations' behaviour and policies to be enforced all along the execution of the target system.

The abstract syntax of Ctrl-F can be divided into two parts: a static one, which is related to the common architectural concepts (components, connections, configurations, etc.); and a dynamic one, which refers to reconfiguration behaviours and policies that must be enforced regardless of the configuration.

The static part of Ctrl-F shares the same concepts of many existing ADLs (e.g., Fractal [6], Acme [13]). A *component* consists of a set of *interfaces*, a set of *event ports*, a set of *attributes* and a set of *configurations. Interfaces* define how a component can interact with other components. So they are used to express a required functionality (*client interface*) that may be provided by another *component* and/or to express a provided functionally (*server interface*) that might be used by other *components*. *Event Ports* describe the events, of the given *Event Type*, a *component* is able to emit (*port out*) and/or listen to (*port in*). A *configuration* is defined as a set of *instances* of *components*, a set of *bindings* connecting *server* and *client interfaces* of those *instances* (i.e., an assembly), and/or a set of *attribute* assignments to *values*.

The dynamic part consists of a *behaviour* and a set of *policies* that can be defined for each component. A *behaviour* takes the form of orders and conditions (w.r.t. *events* and attribute *values*) under which transitions between configurations (reconfigurations) take place. The *policies* are high-level objectives/constraints, which may imply in the inhibition of some of those transitions.

The Znn.com example application of Section 2 can be modeled as a hierarchical composition of four components: *Main*, *Znn*, *LoadBalancer*, and *AppServer*. These components are instantiated according to execution conditions, the system current state (architectural composition), adaptation behaviours and policies defined within each component. Listing 1.1 shows the definition of such components with the static part of Ctrl-F.

The *Main* component (lines 1-14) encompasses two instances of *Znn*, namely *soccer* and *politics* within a single configuration (lines 7 and 8). The server interfaces of both instances (lines 9 and 10), which provides access to news services, are bound to the server interfaces of the *Main* component (lines 3 and 4) in order for them to be accessed from outside. A policy to be enforced is defined (line 13) and discussed in Section 3.3.

Component *Znn* (lines 16-33) consists of one provided interface (line 18) through which news can be requested. The component listens to events of types *oload* (overload) and *uload* (underload) (lines 20 and 21), which are emitted by other components. In addition, the component also defines two attributes: *consumption* (line 23), which is used to express the level of consumption (in terms of percentage of CPU) incurred by the component execution; and *fidelity* (line 24), which expresses the content fidelity level of the component.

Three configurations are defined for *Znn* component: *conf1*, *conf2* and *conf3*. *conf1* (lines 26-33) consists of one instance of each *LoadBalancer* and *AppServer* (lines 27 and 28); one binding to connect them (line 29), another binding to expose the server interface of the *LoadBalancer* component as a server interface of the *Znn* component (line 30), and the attribute assignments (lines 31 and 32). The attribute *fidelity* corresponds to the counterpart of instance *as1*, whereas for the *consumption* it corresponds to the sum of the consumptions of instances *as1* and *lb*. *conf2* (lines 34-39) *extends conf1* by adding one more instance of *AppServer*, binding it to the *LoadBalancer* and redefining the attribute values with respect to the just-added component instance (*as2*).

In that case, the attribute fidelity values the average of the counterparts of instances *as1* and *as2* (line 37), whereas for the consumption the same logics is applied so the consumption of the just-added instance is incorporated to the sum expression (line 38). Due to lack of space we omit the definition of configuration *conf3*. Nevertheless, it follows the same idea, that is, it extends *conf2* by adding a new instance of *AppServer*, binding it and redefining the attribute values.

Listing 1.1. Architectural Description of Components *Main*, *Znn*, *Load Balancer* and *AppServer* in Ctrl-F.

```
1  component Main {
2
3    server interface sis
4    server interface sip
5
6    configuration main {
7        soccer:Znn
8        politics:Znn
9        bind sis to soccer.si
10       bind sip to politics.si
11   }
12
13   policy {...}
14 }
15
16 component Znn {
17
18   server interface si
19
20   port in oload
21   port in uload
22
23   attribute consumption
24   attribute fidelity
25
26   configuration conf1 {
27      lb:LoadBalancer
28      as1:AppServer
29      bind lb.ci1 to as1.si
30      bind lb.si to si
31      set fidelity to as1.fidelity
32      set consumption to sum(as1.
             consumption,lb.consumption)
33   }
34   configuration conf2 extends conf1 {
35      as2:AppServer
36      bind lb.ci2 to as2.si
37      set fidelity to avg(as1.fidelity
             ,as2.fidelity)
38      set consumption to sum(as1.
             consumption,as2.consumption,
             lb.consumption)
39   }
40
41   configuration conf3 extends conf2
         {...}
42
43   behaviour {...}
44   policy {...}
45 }
46
47 component LoadBalancer {
48   server interface si
49   client interface ci1,ci2,c3
50
51   port out oload
52   port out uload
53
54   attribute consumption=0.2
55 }
56
57 component AppServer {
58   server interface si
59
60   port in oload
61   port in uload
62
63   attribute fidelity
64   attribute consumption
65
66   configuration text {
67     set fidelity to 0.25
68     set consumption to 0.2
69   }
70   configuration img-ld {
71     set fidelity to 0.5
72     set consumption to 0.6
73   }
74   configuration img-hd {...}
75
76   behaviour {...}
77   policy {...}
78 }
```

Component *LoadBalancer* (lines 47-55) consists of four interfaces: one provided (line 48), through which the news are provided; and the others required (line 49), through which the load balancer delegates each request for balancing purposes. We assume that this component is able to detect overload and underload situations (in terms of number of requests per second) and in order for this information to be useful for other components we define two event *ports* that are used to emit events of type *oload* and *uload* (lines 51 and 52). Like for component *Znn*, attribute *consumption* (line 54) specifies the level of consumption of the component (e.g., 0.2 to express 20% of CPU consumption). As there is no explicit definition of configurations, *LoadBalancer* is implicitly treated as a single-configuration component.

Lastly, the atomic component *AppServer* (lines 57-78) has only one interface (line 58) and listens to events of type *oload* and *uload* (lines 60 and 61). It has also two attributes: fidelity and consumption (lines 63 and 64), just like component

Znn. Three configurations corresponding to each level of fidelity (lines 66-69, 70-73 and 74) are defined, and the attributes are valuated according to the configuration in question, i.e., the higher the fidelity the higher the consumption.

3.2 Behaviours

A particular characteristic of Ctrl-F is the capability to comprehensively describe behaviours in component-based applications. We mean by behaviour the process in which architectural elements are changed. More precisely, it refers to the order and conditions under which configurations within a component take place.

Behaviours in Ctrl-F are defined with the aid of a high-level imperative language. It consists of a set of behavioural statements (*sub-behaviours*) that can be composed together so as to provide more complex behaviours in terms of sequences of configurations. In this context, a *configuration* is considered as an atomic behaviour, i.e., a behaviour that cannot be decomposed into other *sub-behaviours*. A reconfiguration occurs when the current configuration is terminated and the next one is started. We assume that configurations do not have the capability to directly terminate or start themselves, meaning that they are explicitly requested or ended by behaviour *statements* according to the defined events and policies. Nevertheless, as components are capable to emit events, it would not be unreasonable to define components whose objective is to emit events in order to force a desired behaviour.

Statements Table 1 summarizes the behaviour statements of the Ctrl-F behavioural language. During the execution of a given behaviour B, the *when-do* statement states that when a given event of event type e_i occurs the configuration(s) that compose(s) B should be terminated and that (those) of the corresponding behaviour B_i are started.

Table 1. Summary of Behaviour Statements.

Statement	**Description**
B **when** e_1 **do** B_1, ... , e_n **do** B_n **end**	While executing B when e_i execute B_i

case c_1 **then** B_1,
... ,
c_n **then** B_n
else B_e **end**
	Execute B_i if c_i holds, otherwise execute B_e

$B_1 \mid B_2$	Execute either B_1 or B_2
$B_1 \mid\mid B_2$	Execute B_1 and B_2 in parallel
do B **every** e	Execute B and re-execute it at every occurrence of e

The *case-then* statement is quite similar to *when-do*. The difference resides mainly in the fact that a given behaviour B_i is executed if the corresponding condition c_i holds (e.g., conditions on attribute values), which means that it does not wait for a given event to occur. In addition, if none of the conditions holds ($c_1 \wedge ... \wedge c_n = 0$), a default behaviour ($B_e$) is executed, which forces the compiler to choose at least one behaviour. The *parallel* statement states that two behaviours are executed at the same time, i.e., at a certain point, there must be two independent branches of behaviour executing in parallel. This construct is also useful in the context of atomic components like *AppServer*, where we could, for instance, define configurations composed of orthogonal attributes like fidelity and font size/color (e.g., `text || font-huge`).

The *alternative* statement allows to describe choice points among configurations or among more elaborated sequential behaviour statements. They are left free in local specifications and will be resolved in upper level assemblies, in such a way as to satisfy the stated policies, by controlling these choice points appropriately. Finally, the *do-every* statement allows for execution of a behaviour B and re-execution of it at every occurrence of an event of type e. It is noteworthy that behaviour B is preempted every time an event of type e occurs. In other words, the configuration(s) currently activated in B is (are) terminated, and the very first one(s) in B is (are) started.

Example in Znn.com We now illustrate the use of the statements we have introduced to express adaptation behaviours for components *AppServer* and *Znn* the of *Znn.com* case study. The expected behaviour for component *AppServer* is to pick one of its three configurations (*text*, *img-ld* or *img-hd*) at every occurrence of events of type *oload* or *uload*. To that end, as it can be seen in Listing 1.2, the behaviour can be decomposed in a *do-every* statement, which is, in turn, composed of an *alternative* one. It is important to mention that the decision on one or other configuration must be taken at runtime according to input variables (e.g., income events) and the stated policies, that is, there must be a control mechanism for reconfigurations that enforces those policies. We come back to this subject in Section 4.1.

Regarding component *Znn*, the expected behaviour is to start with the minimum number of *AppServer* instances (configuration *conf1*) and add one more instance, i.e., leading to configuration *conf2*, upon an event of type (*oload*). From *conf2*, one more instance must be added, upon an event of type *oload* leading to configuration *conf3*. Alternatively, upon an event of type *uload*, one instance of *AppServer* must be removed, which will lead the application back to configuration *conf1*. Similarly, from configuration *conf3*, upon a *uload* event, another instance must be removed, which leads the application to *conf2*. It is notorious that this behaviour can be easily expressed by an automaton, with three states (one per configuration) and four transitions (triggered upon the occurrence of *oload* and *uload*). However, Ctrl-F is designed to tackle the adaptation control problem in a higher level, i.e., with process-like statements over configurations.

For these reasons, we describe the behaviour with two embedded *do-every* statements, which in turn comprise each a *when-do* statement, as shown in Listing 1.3 (lines 6-14 and 8-12). We also define two auxiliary configurations: *emit-*

ter1 (line 2) and *emitter2* (line 3), which extend respectively configurations *conf2* and *conf3*, with an instance of a pre-defined component *Emitter*. This component does nothing but emit a given event (e.g., *e1* and *e2*) so as to force a loop step and thus go back to the beginning of the *when-do* statements. The main *do-every* statement (lines 6-14) performs a *when-do* statement (lines 7-13) at every occurrence of an event of type *e1*. In practice, the firing of this event allows going back to *conf1* regardless of the current configuration being executed. *conf1* is executed until the occurrence of an event of type *oload* (line 7), then the innermost *do-every* statement is executed (lines 8-12), which in turn, just like the other one, executes another *when-do* statement (lines 9-11) and repeats it at every occurrence of an event of type *e2*. Again, that structure allows the application to go back to configuration *conf2*. Configuration *conf2* is executed until an event of type either *oload* or *uload* occurs. For the former case (line 9), another *when-do* statement takes place, whereas for the latter (line 10) configuration *emitter1* is the one that takes place. Essentially, at this point, an instance of component *Emitter* is deployed along with *conf2*, since *emitter1* extends *conf2*. As a consequence, this instance fires an event of type *e1*, which forces the application to go back to *conf1*. The innermost *when-do* statement (line 9) consists in executing *conf3* until an event of type *uload* occurs, then configuration *emitter2* takes place, which makes an event of type *e2* be fired in order to force going back to *conf2*.

It is important to notice that this kind of construction allows to achieve the desired behaviour while sticking to the language design principles, that is, high-level process-like constructs and configurations. It also should be remarked that while in Listing 1.3 we present an imperative approach to forcibly increase the number of *AppServer* instances upon *uload* and *oload* events, in Listing 1.3 we leave the choice to the compiler to choose the most suitable fidelity level according to the runtime events and conditions. Although there is no straightforward guideline, an imperative approach is clearly more suitable when the solution is more sequential and delimited, whereas as the architecture gets bigger, in terms of configurations, and less sequential, then a declarative definition becomes more interesting.

Listing 1.2. AppServer's Behaviour.

```
component
     AppServer {
  ...
  behaviour {
    do
      text |
      img-ld |
      img-hd
    every
      (oload
         or uload)
  }
}
```

Listing 1.3. Znn's Behaviour.

```
1  component Znn {...
2    configuration emitter1 extends conf2 { e:Emitter }
3    configuration emitter2 extends conf3 { e:Emitter }
4
5    behaviour {
6      do
7        conf1 when oload do
8          do
9            conf2 when oload do (conf3 when uload do
                  emitter2 end),
10                       uload do emitter1
11           end
12         every e2
13       end
14     every e1
15   }
16 }
```

3.3 Policies

Policies are expressed with high-level constructs for constraints on configurations, either temporal or on attribute values. In general, they define a subset of all possible global configurations, where the system should remain invariant: this will be achieved by using the choice points in order to control the reconfigurations. An intuitive example is that two component instances in parallel branches might have each several possible configurations, and some of them to be kept exclusive. This exclusion can be enforced by choosing the appropriate configurations when starting the components.

Constraints/Optimization on Attributes This kind of constraints are predicates and/or primitives of optimization objectives (i.e., maximize or minimize) on component attributes. Listing 1.4 illustrates some constraints and optimization on component attributes. The first two policies state that the overall fidelity for component instance *soccer* should be greater or equal to 0.75, whereas that of instance *politics* should be maximized. Putting it differently, instance *soccer* must never have its content fidelity degraded, which means that it will have always priority over *politics*. The third policy states that the overall consumption should not exceed 5, which could be interpreted as a constraint on the physical resource capacity, e.g., the number of available machines or processing units.

Listing 1.4. Example of Constraint and Optimization on Attributes.

```
component Main { ...
 policy { soccer.fidelity >= 0.75 }
 policy { maximize politics.fidelity }
 policy { (soccer.consumption +
          politics.consumption) <= 5 }
}
```

Listing 1.5. Example of Temporal Constraint.

```
component AppServer { ...
 policy { img-ld succeeds text }
 policy { img-ld succeeds img-hd }
}
```

Temporal Constraints Temporal constraints are high-level constructs that take the form of predicates on the order of configurations. These constructs might be very helpful when there are many possible reconfiguration paths (by either *parallel* or *alternative* composition, for instance), in which case the manual specification of such constrained behaviour may become a very difficult task.

To specify these constraints, Ctrl-F provides four constructs, as follows:

- $conf_1$ **precedes** $conf_2$: $conf_1$ must take place right before $conf_2$. It does not mean that it is the only one, but it should be among the configurations taking place right before $conf_2$.
- $conf_1$ **succeeds** $conf_2$: $conf_1$ must take place right after $conf_2$. Like in the precedes constraint, it does not mean that it is the only one to take place right after $conf_2$.
- $conf_1$ **during** $conf_2$: $conf_1$ must take place along with $conf_2$.
- $conf_1$ **between** $(conf_2, conf_3)$: once $conf_2$ is started, $conf_1$ cannot be started and $conf_3$, in turn, cannot be started before $conf_2$ terminates.

Listing 1.5 shows an example of how to apply temporal constraints, in which it is stated that configuration *img-ld* comes right after the termination of either configuration *text* or configuration *img-ld*. In this example, this policy avoids abrupt changes on the content fidelity, such as going directly from text to image high definition or the other way around. Again, it does not mean that no other configuration could take place along with *img-ld*, but the *alternative* statement in the behaviour described in Listing 1.2 leads us to conclude that only *img-ld* must take place right after either *text* or *img-hd* has been terminated.

4 Heptagon/BZR Model and Implementation

4.1 Modeling Ctrl-F in Heptagon/BZR

As architectures get larger and more complex, conceiving behaviours that respect the stated policies becomes a hard and error-prone task. This is the main reason why we model Ctrl-F behaviours and policies with Heptagon/BZR. Indeed, the FSA-based model of Heptagon/BZR allows programs to be formally exploited and verified by model checking tools [10]. The general model of Ctrl-F behaviours is as surveyed in Figure 1. Basically, each component accommodates an automaton corresponding to its adaptive behaviour, in which states correspond to configurations and transitions to reconfigurations. So, based on a vector of input events (e.g., *oload* and *uload*, in the Znn.com example) and runtime conditions (e.g., on the attribute values), transitions may be triggered while emitting signals for stopping the current configuration and starting the new one. In the case the behaviour contains choice points, that is, *alternative* statements, we model the transition conditions to each one of the choice branches as free-variables. The resulting controller from the DCS, which takes the form of a deterministic automata, is in charge of the control on those variables such that, regardless of the input events, the stated policies are enforced. It is noteworthy that although the DCS algorithms has exponential complexity as any other model checking approach, the controller is synthesized in an off-line manner and thus with no impact on the running controlled system. The same structural translation is performed hierarchically for every sub-component, i.e., in every component instantiated within another component. Due to space limitation, we have to omit the details on the translation schemes, but the full translation of Ctrl-F behaviour statements and policies to Heptagon/BZR is available in [1].

4.2 Compilation Tool-chain

As can be seen in Figure 2, the compilation process can be split into two parts: (i) the reconfiguration logics and (ii) the behaviour/policy control and verification. The reconfiguration logics is implemented by the *ctrlf2fscript* compiler, which takes as input a Ctrl-F definition and generates as output a script containing a set procedures allowing going from one configuration to another. To this end, we rely on existing differencing/match algorithms for object-oriented models [23].

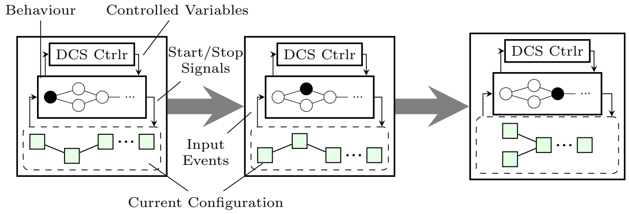


Fig. 1. Role of the Behaviour Automaton over the Transitions.

The behaviour control and verification is performed by the *ctrlf2ept* compiler, which takes as input a Ctrl-F definition and provides as output a synchronous reactive program in Heptagon/BZR. The result of the compilation of an Heptagon/BZR code is a sequential code in a general-purpose programming language (in our case Java) comprising two methods: `reset` and `step`. The former initializes the internal state of the program, whereas the latter is executed at each logical step to compute the output values based on a given vector of input values and the current state.

These methods are typically used by first executing `reset` and then by enclosing `step` in an infinite loop, in which each iteration corresponds to a reaction to an event (e.g., *oload* or *uload*), as sketched in Listing 1.6. The step method returns a set of signals corresponding to the start or stop of configurations (line 4). From these signals, we can find the appropriate script that embodies the reconfiguration actions to be executed (lines 5 and 6).

We wrap the control loop logics into three components, which are enclosed by a composite named *Manager*. Component *EventHandler* exposes a service allowing itself to be sent events (e.g., *oload* and *uload*). The method implementing this service is defined as non-blocking so the incoming events are stored in a First-In-First-Out queue. Upon the arrival of an event coming from the *Managed System* (e.g., Znn.com), component *EventHandler* invokes the step method, implemented by component *Architecture Analyzer*. The step method output is sent to component *Reconfigurator*, that encompasses a method to find the proper reconfiguration script to be executed.

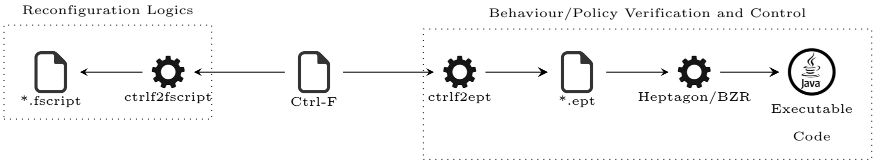


Fig. 2. Ctrl-F Compilation Chain.

Listing 1.6. Control Loop Sketch.

```
reset();
...
on event oload or uload
<...,stop_conf1,start_conf2,...>=step(oload,uload);
reconfig_script=find_script(...,stop_conf1,start_conf2,...);
execute(reconfig_script);
```

In this work, we rely on the Java-based Service Component Architecture (SCA) middleware FraSCAti [20], since it provides mechanisms for runtime reconfiguration. The FraSCAti Runtime is itself conceived relying on the SCA model, that is, it consists of a set of SCA components that can be deployed *a la carte*, according to the user's needs. For instance, in our case, the *Manager* instantiates the *frascati-fscript* component, which provides services allowing for the execution of an SCA-variant of FPath/FScript [8], a domain-specific language for introspection and dynamic reconfiguration of Fractal components. The *frascati-fscript* component relies on other components integrating the middleware, inside the FraSCAti Composite, to perform introspection and runtime reconfiguration on the managed system's components.

4.3 Adaptation Scenario

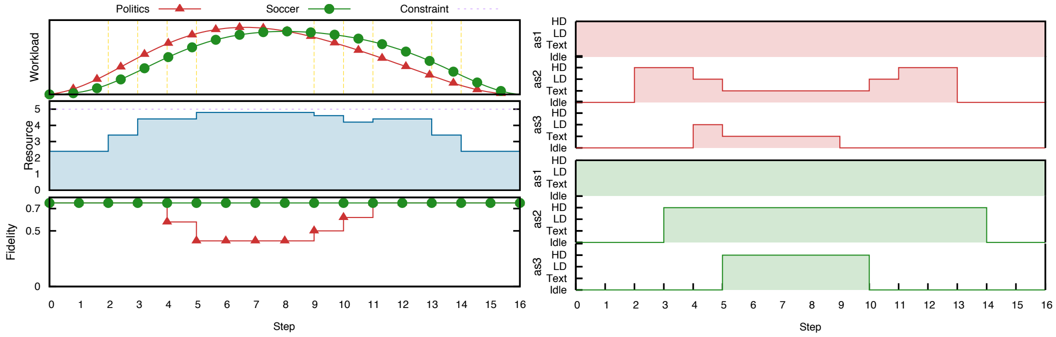


Fig. 3. Execution of the Adaptation Scenario.

We simulated the execution of the two instances of *Znn.com* application, namely *soccer* and *politics*, under the administration of the *Manager* presented in last section, to observe the control of reconfigurations taking into account a sequence of input events. The behaviours of components *AppServer* and *Znn* are stated in Listings 1.2 and 1.3, respectively, while policies are defined in Listing 1.4 and 1.5.

As it can be observed in the first chart of Figure 3, we scheduled a set of overload (*oload*) and underload (*uload*) events (vertical dashed lines), which simulate an increase followed by a decrease of the income workload for both soccer and politics instances. The other charts correspond to the overall resource consumption, the overall fidelity, and the fidelity level (i.e., configurations *text*, *img-ld* or *img-hd*) of the three instances of component *AppServer* contained in both instances of component *Znn*.

As the workload of *politics* increases, an event of type *oload* occurs at step 2. That triggers the reconfiguration of that instance from *conf1* to *conf2*, that is, one more instance of *AppServer* is added within the *Znn*instance *politics*. We can observe also the progression in terms of resource consumption, as a consequence of this configuration. The same happens with *soccer* at step 3, and is repeated with *politics* and *soccer* again at steps 4 and 5. The difference, in this case, is that at step 4, the *politics* instance must reconfigure (to *conf3*) so as to cope with the current workload while keeping the overall consumption under control. In other words, it forces the *AppServer* instances *as2* and *as3* to degrade their fidelity level from *img-hd* to *img-ld*. It should be highlighted that although at least one of the *AppServer* instances (*as2* or *as3*) could be at that time at maximum fidelity level, the knowledge on the possible future configurations guarantees the maximum overall fidelity for instance *soccer* to the detriment of a degraded fidelity for instance *politics*, while respecting the temporal constraints expressed in Listing 1.5. Hence, at step 5, when the last *oload* event arrives, the fidelity level of *soccer* instance is preserved by gradually decreasing that of *politics*, that is, both instances *as2* and *as3* belonging to the *politics* instance are put in configuration *text*, but without jumping directly from from *img-hd*. At step 9, the first *uload* occurs as a consequence of the workload decrease. It triggers a reconfiguration in the *politics* instance as it goes from *conf3* to *conf2*, that is, it releases one instance of *AppServer* (*as3*). The same happens with *soccer* at step 10, which makes room on the resources and therefore allows *politics* to bring back the fidelity level of its *as2* to *img-ld*, and to the maximum level again at step 11. This is repeated at steps 13 and 14 for instances *politics* and *soccer* respectively, bringing their *consumptions* at the same levels as in the beginning.

The adaptation scenario is very useful to understand the dynamics behind an *Manager* that is derived from a synchronous reactive programming, which is in turn, obtained from Ctrl-F. Moreover, the scenario illustrates, in a pedagogical way, how controllers obtained by DCS are capable to control reconfigurations based not only on the current events and current/past configurations (states), but also on the possible future behaviours, that is, how controllers avoid branches that may lead to configurations violating the stated policies.

5 Related Work

In the literature, there is a large and growing body of work on runtime reconfiguration of software components. Our approach focuses on the language support for enabling self-adaptation in component-based architectures while relying on reactive systems and the underlying formal control tools for ensuring adaptation policies. This section summarizes the related work, more detailed elsewhere [1].

Classically, runtime adaption in software architectures is achieved by first relying on ADLs such as Acme [13] or Fractal [6] for an initial description of the software structure and architecture, then by specifying fine-grained reconfiguration actions with dedicated languages like Plastik [3] or FPath/FScript [8], or simply by defining Event-Condition-Actions (ECA) rules to lead the system to the desired state. A harmful consequence is that the space of reachable configuration states is only known as side effect of those reconfiguration actions, which makes it difficult to ensure correct adaptive behaviours. Moreover, a drawback of ECA rules is that, contrary to Ctrl-F, they cannot describe sequences of configurations. Even though, ECA rules can be expressed in Ctrl-F with a set of *when-do* (for the E part) and *case* (for the C and A parts) statements in parallel.

Rainbow [12] is an autonomic framework that comes with Stitch, a domain-specific language allowing for the description of self-adaptation of Acme-described applications. It features system-level actions grouped into *tactics*, which in turn, are aggregated within a tree-like strategy path. We can draw an analogy between tactics and the set of actions triggered upon a reconfiguration; as well as strategies and behaviours in the Ctrl-F language. Nonetheless, alternative and parallel, as well as event-based constructs make Ctrl-F more expressive. Furthermore, Ctrl-F's formal model enables to ensure correct adaptation behaviours.

A body of work [2][21][22][22][5][17][4] focus on how to plan a set of actions that safely lead component-based systems to a target configuration. These approaches are complementary to ours in the sense that our focus is on the choice of a new configuration and its control. Once a new configuration chosen, we rely on existing mechanisms to determine the plan of action actually leading the system from the current to the next configuration.

In [18], feature models are used to express variability in software systems. At runtime, a resolution mechanism is used for determining which features should be present so as to constitute configuration. In the same direction, Pascual et al. [19] propose an approach for optimal resolution of architectural variability specified in the Common Variability Language (CVL) [14]. A drawback of those approaches is that in the adaptation logics specified with feature models or CVL, there is no way to define stateful adaptation behaviours, i.e., sequences of reconfigurations. The resolution is performed based on the current state and/or constraints on the feature model. While in our approach, in the reactive model based on FSA, decisions are taken also based on the history of configurations which allows us to define more interesting adaptation behaviours and policies.

W.r.t. formal methods, Kouchnarenko and Weber [16] propose the use of temporal logics to integrate temporal requirements to adaptation policies. While in this approach, enforcement and reflection are performed at runtime in order to ensure correct behaviour, we rely on discrete controller synthesis.

As in our approach, in [11], authors also rely on Heptagon/BZR to model adaptive behaviours of Fractal components. However, there is no high-level description (e.g., ADL) like Ctrl-F, and reconfigurations are controlled at the level of fine-grained reconfiguration actions, which can be considered time-consuming and difficult to scale. Delaval et al. [9] propose to have modular controllers that can be coordinated so as to work together in a coherent manner. The approach is complementary to ours in the sense that it does not provide high-level language support for describing those managers, although the authors provide interesting intuitions on a methodology to do so.

6 Conclusion

This paper presented Ctrl-F, a high-level domain-specific language that allows for the description of adaptation behaviours and policies of component-based architectures. A distinguished feature of Ctrl-F is its formalization with the synchronous reactive language Heptagon/BZR, which allows to benefit, among other things, from formal tools for verification, control, and automatic generation of executable code. In order to show the language expressiveness, we applied it to Znn.com, a self-adaptive case study, and we integrated it with FraSCAti, a Service Component Architecture middleware.

For future work, we intent to address issues of modularity and coordination of controllers, as well as their distribution. The reactive language and models we rely on have recent results that can be exploited, and can lead to deploy controllers taking into account the physical location of components.

References

1. Alvares, F., Rutten, E., Seinturier, L.: Behavioural Model-based Control for Autonomic Software Components. In: Proc. 12th Int. Conf. Autonomic Computing (ICAC'15). Grenoble, France. (extended version available as a Research Report: `https://hal.inria.fr/hal-01103548`) (Jul 2015)
2. Arshad, N., Heimbigner, D.: A Comparison of Planning Based Models for Component Reconfiguration. Research Report CU-CS-995-05, U. Colorado (Sep 2005)
3. Batista, T., Joolia, A., Coulson, G.: Managing dynamic reconfiguration in component-based systems. In: Proc. 2nd European Conference on Software Architecture. pp. 1–17. EWSA'05 (2005)
4. Becker, S., Dziwok, S., Gerking, C., Heinzemann, C., Schäfer, W., Meyer, M., Pohlmann, U.: The mechatronicuml method: Model-driven software engineering of self-adaptive mechatronic systems. In: Companion Proceedings of the 36th International Conference on Software Engineering. pp. 614–615. ICSE Companion 2014, ACM, New York, NY, USA (2014), `http://doi.acm.org/10.1145/2591062.2591142`
5. Boyer, F., Gruber, O., Pous, D.: Robust reconfigurations of component assemblies. In: Proc. 2013 Int. Conf. on Software Engineering. pp. 13–22. ICSE '13 (2013)
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: An open component model and its support in java. In: Proc. International Symp. on Component-based Software Engineering (CBSE). Edinburgh, Scotland (May 2003)
7. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the rainbow self-adaptive system. In: Software Engineering for Adaptive and Self-Managing Systems, 2009. SEAMS '09. ICSE Workshop on. pp. 132–141 (May 2009)
8. David, P.C., Ledoux, T., Léger, M., Coupaye, T.: FPath & FScript: Language support for navigation and reliable reconfiguration of Fractal architectures. Annals of Telecommunications: Special Issue on Software Components (2008)

9. Delaval, G., Gueye, S.M.K., Rutten, E., De Palma, N.: Modular coordination of multiple autonomic managers. In: Proc. 17th Int. ACM Symp. on Component-based Software Engineering. pp. 3–12. CBSE '14 (2014)
10. Delaval, G., Marchand, H., Rutten, E.: Contracts for modular discrete controller synthesis. In: ACM International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2010). Stockholm, Sweden (Apr 2010)
11. Delaval, G., Rutten, E.: Reactive model-based control of reconfiguration in the fractal component-based model. In: 13th International Symposium on Component Based Software Engineering (CBSE 2010). Prague, Czech Republic (Jun 2010)
12. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (Oct 2004)
13. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
14. Haugen, O., Wasowski, A., Czarnecki, K.: Cvl: Common variability language. In: Proceedings of the 17th International Software Product Line Conference. pp. 277–277. SPLC '13, ACM, New York, NY, USA (2013)
15. Jacobson, I., Griss, M., Jonsson, P.: Software reuse: architecture process and organization for business success. ACM Press books, ACM Press (1997)
16. Kouchnarenko, O., Weber, J.F.: Adapting component-based systems at runtime via policies with temporal patterns. In: 10th Int. Symp. Formal Aspects of Component Software. LNCS, vol. 8348, pp. 234–253. Nanchang, China (2014)
17. Luckey, M., Nagel, B., Gerth, C., Engels, G.: Adapt cases: Extending use cases for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. pp. 30–39. SEAMS '11, ACM, New York, NY, USA (2011), http://doi.acm.org/10.1145/1988008.1988014
18. Morin, B., Barais, O., Nain, G., Jezequel, J.M.: Taming dynamically adaptive systems using models and aspects. In: Proc. 31st Int. Conf. on Software Engineering. pp. 122–132. ICSE '09, IEEE (2009)
19. Pascual, G.G., Pinto, M., Fuentes, L.: Run-time support to manage architectural variability specified with cvl. In: Proc. 7th European Conf. on Software Architecture. pp. 282–298. ECSA'13 (2013)
20. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.B.: A component-based middleware platform for reconfigurable service-oriented architectures. Software: Practice and Experience 42(5), 559–583 (2012)
21. da Silva, C.E., de Lemos, R.: Dynamic plans for integration testing of self-adaptive software systems. In: Proc. 6th Int. Symp. on Software Engineering for Adaptive and Self-Managing Systems. pp. 148–157. SEAMS '11 (2011)
22. Tichy, M., Klöpper, B.: Planning self-adaption with graph transformations. In: Proc. 4th Int. Conf. on Applications of Graph Transformations with Industrial Relevance. pp. 137–152. AGTIVE'11 (2012)
23. Xing, Z., Stroulia, E.: Umldiff: An algorithm for object-oriented design differencing. In: Proc. 20th IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 54–65. ASE '05 (2005)

Session du groupe de travail IDM

Ingénierie Dirigée par les Modèles

Omniscient Debugging and Concurrent Execution of Heterogeneous Domain-Specific Models

Benoit Combemale

Inria and Univ. Rennes 1, France
benoit.combemale@inria.fr

1 Context and Motivations

In the software and systems modeling community, research on domain-specific modeling languages (DSMLs) is focused on providing technologies for developing languages and tools that allow domain experts to develop system solutions efficiently. Unfortunately, the current lack of support for explicitly relating concepts expressed in different DSMLs makes it very difficult for software and system engineers to reason about information spread across models describing different system aspects [2].

As a particular challenge, we investigate in this paper relationships between, possibly heterogeneous, behavioral models to support their concurrent execution. This is achieved first by following a modular executable metamodeling approach [3], which include an explicit model of concurrency (MoC) [4], domain-specific actions (DSA) [6], and a well-defined protocol between them (incl., mapping, feedback and callback) [7]. Then, the protocol is used for inferring a relevant behavioral language interface supporting the definition of coordination patterns to be applied on the conforming executable models [8].

All the tooling of the approach is gathered within the GEMOC studio. The GEMOC studio provides a language workbench, seamlessly integrated on top of the Eclipse Modeling Framework (EMF), for designing and composing executable domain-specific modeling languages (xDSML). xDSMLs are then automatically deployed into a modeling workbench providing model execution (incl., graphical animation and omniscient debugging) and behavioral coordination of, possibly heterogeneous, models.

2 Approach Overview

2.1 Sirius Animator: Engineering xDSMLs for model executability, animation and debugging

Domain-specific models are used in the development processes to reason and assess specific properties over the system under development as early as possible. This usually leads to a better and cheaper design as more alternatives can be explored. While many models only represent structural aspects of systems, a large amount express behavioral aspects of the same systems. Behavioral models are used in various areas (e.g., enterprise architecture, software and systems engineering, scientific modeling. . .), with very different underlying formalisms (e.g., business processes, orchestrations, functional chains, scenarios, protocoles, activity or state diagram, etc.).

To ensure that a behavioral model is correct with regard to its intended behavior, early dynamic validation and verification (V&V) techniques are required, such as simulation, debugging, model checking or runtime verification. In particular, debugging is a common dynamic facility to observe and control an execution in order to better understand a behavior or to look for the cause of a defect. Standard (stepwise) debugging only provides facilities to pause and step forward during an execution, hence requiring developers to restart from the beginning to give a second look at a state of interest. Omniscient debugging extends stepwise debugging by relying on execution traces to enable free traversal of the states reached by a model, thereby allowing designers to "go back in time."

Debugging, and all dynamic V&V techniques in general, require models to be executable, which can be achieved by defining the execution semantics of modeling languages (i.e. DSLs) used to describe them. The execution semantics of a modeling language can be implemented either as a compiler or as an interpreter to be applied on models conforming to the modeling language [1].

Despite the specificities of each modeling language captured into the syntax and the execution semantics, it is possible to identify a common set of debugging facilities for all modeling languages: control of the execution (pause, resume, stop), representation of the current state during the execution (i.e., model animation), breakpoint definition, step into/over/out and step forward/backward. To drastically reduce the development cost of domain-specific debuggers, Sirius Animator[1] provides a tool-supported approach to complement a modeling language with an execution semantics, and automatically get an advanced and extensible environment for model simulation, animation and debugging.

Sirius Animator provides a modular approach to complement an Ecore metamodel with:

- a Java-based interpreter, and
- a Sirius-based representation of the execution state.

Both are automatically deployed, together with a generated efficient execution trace metamodel, into an advanced omniscient debugging environment providing:

- an integration with the Eclipse debug UI (incl., the binding of the execution state with the variable view),
- a generic execution engine with execution control facilities (pause, resume, stop),
- a generic control panel (incl. breakpoints, step into/over/out and step forward/backward), and
- a generic timeline.

[1] Cf. https://github.com/SiriusLab/ModelDebugging, and more information at http://gemoc.org/breathe-life-into-your-designer

2.2 MoCCML: Reifying the concurrency concern into xDSML specifications

The emergence of modern concurrent systems (e.g., Cyber-Physical Systems and Internet of Things) and highly-parallel platforms (e.g., many-core, GPGPU and distributed platforms) call for Domain-Specific Modeling Languages (DSMLs) where concurrency is of paramount importance. Such DSMLs are intended to propose constructs with rich concurrency semantics, which allow system designers to precisely define and analyze system behaviors. However, implementing the execution semantics of such DSMLs is a particularly difficult task. Most of the time the concurrency model remains implicit and ad-hoc, embedded in the underlying execution environment.

The lack of an explicit concurrency model prevents: the precise definition, the variation and the complete understanding of the DSML's semantics, the effective usage of concurrency-aware analysis techniques, and the exploitation of the concurrency model during the system refinement (e.g., during its allocation on a specific platform).

To address this issue, we have defined a *concurrency-aware executable metamodeling approach*, which supports a modular definition of the execution semantics, including the concurrency model, the semantic rules (aka. domain-specific actions, DSA), and a well-defined and expressive communication protocol between them [3]. The protocol supports both *the mapping* of the concurrency model to the semantic rules, and *the feedback*, possibly with data, from the semantic rules to the concurrency model [7]. The concurrent executable metamodeling approach also comes with a dedicated meta-language, namely MoCCML, to define the concurrency model and the protocol, and an execution environment to simulate and analyze behavioral models [4].

MoCCML is a declarative meta-language specifying constraints between the events of a model of concurrency. At any moment during a run, an event that does not violate the constraints can occur. The constraints are grouped in libraries that specify concurrency constraints. These constraints can also be of a different kinds, for instance to express a deadline, a minimal throughput or a hardware deployment. They are eventually instantiated to define the execution model of a specific model. The execution model is a symbolic representation of all the acceptable schedules for a particular model. To enable the automatic generation of the execution model, the concurrency model is weaved into the context of specific concepts from the abstract syntax of a DSML. This contextualization is defined by a mapping between the elements of the abstract syntax and the constraints of the concurrency model. The mapping defined in MoCCML is based on the notion of domain-specific event (DSE). The separation of the mapping from the concurrency model makes the concurrency model independent of the DSML so that it can be reused, and semantic variation points can be managed for a single abstract syntax. From such description, for any instance of the abstract syntax it is possible to automatically generate a dedicated execution model.

In our approach, the execution model is acting as the configuration of a generic execution engine that can be used for simulation or analysis of any model conforming to the abstract syntax of the DSML.

MoCCML is defined by a metamodel (i.e., the abstract syntax) associated to a formal structural operational semantics. MoCCML[2] comes with a model editor combining textual and graphical notations, as well as analysis tools based on the formal semantics for simulation and exhaustive exploration.

2.3 B-COoL: Coordination of xDSMLs for concurrent execution of heterogeneous models

To capture the specification of coordination patterns between languages, a behavioral interface is required at the language level. A language behavioral interface must abstract the behavioral semantics of a language, thus providing only the information required to coordinate it, i.e., a partial representation of concurrency and time-related aspects. Furthermore, to avoid altering the coordinated language semantics, the specification of coordination patterns between languages should be non intrusive, i.e., it should keep separated the coordination and the computation concerns. In the previous subsection, elements of event structures are reified at the language level to propose a behavioral interface based on sets of *event types* and *contraints*. Event types (named DSE for Domain Specific Event) are defined in the context of a metaclass of the abstract syntax, and abstract the relevant semantic (domain-specific) actions. Jointly with the DSE, related constraints give a symbolic (intentional) representation of an event structure. With such an interface, the concurrency and time-related aspects of the language behavioral semantics are explicitly exposed and the coordination is event-driven and non intrusive.

Then, for each model conforming to the language, the model behavioral interface is a specification, in intention, of an event structure whose events (named MSE for Model Specific Event) are instances of the DSE defined in the language interface. While DSEs are attached to a metaclass, MSEs are linked to one of its instances. The causality and conflict relations of the event structure are a model-specific unfolding of the constraints specified in the language behavioral interface. Just like event structures were initially introduced to unfold the execution of Petri nets, we use them here to unfold the execution of models.

We propose to use DSE as "coordination points" to drive the execution of languages. These events are used as control points in two complementary ways: to observe what happens inside the model, and to control what is allowed to happen or not. When required by the coordination, constraints are used to forbid or delay some event occurrences. Forbidding occurrences reduces what can be done by individual models. When several executions are allowed (i.e., nondeterminism), it gives some freedom to individual semantics for making their own choices. All this put together makes the DSE suitable to drive coordinated simulations

[2] Cf. https://github.com/gemoc/concurrency

without being intrusive in the models. Coordination patterns are captured as constraints at the language level on the DSE.

BCOoL is a dedicated (meta)language to explicitly capture the knowledge about system integration [8]. With BCOoL, an integrator can explicitly capture coordination patterns at the language level. Specific *operators* are provided to build the coordination patterns and specify how the DSE of different language behavioral interfaces are combined and interact. From the BCOoL specification, an executable and formal coordination model can be generated by instantiating all the constraints on all instances of DSE. Therefore, the generated coordination model implements the coordination patterns defined at the language level.

The design of BCOoL is inspired by current structural composition languages, relying on the *matching* and *merging* phases of syntactic model elements. A matching rule specifies what elements from different models are selected. A merging rule specifies how the selected model elements are composed. In these approaches the specification is at the language level, but the application is between models. Similarly, a BCOoL operator relies on a *correspondence matching* and a *coordination rule*. The correspondence matching identifies what elements from the behavioral interfaces (i.e., what instances of DSE) must be selected. The merging phase is replaced by a coordination rule. While in the structural case the merging operates on the syntax, the coordination rule operates on elements of the semantics (i.e., instances of DSE). Thus, coordination rules specify the, possibly timed, synchronizations and causality relationships between the instances of DSE selected during the matching.

BCOoL[3] comes with a textual editor to support the specification of coordination patterns between xDSMLs, as well as a generative approach to instantiate a set of selected coordination patterns for a given set of models.

3 The GEMOC Studio

The GEMOC Studio is an Eclipse package, seamlessly integrated with the Eclipse Modelign Framework, that contains components supporting the GEMOC methodology for building and composing executable Domain-Specific Modeling Languages (DSMLs). It includes two workbenches: the *GEMOC Language Workbench* and the *GEMOC Modeling Workbench*. The language workbench is intended to be used by language designers (aka domain experts), it allows to complement Ecore metamodels to build and compose executable DSMLs. The Modeling Workbench is intended to be used by domain designers, it allows to create and execute heterogeneous models conforming to executable DSMLs.

The GEMOC Studio results in various integrated tools that belong into either the language workbench or the modeling workbench. The language workbench put together the following tools seamlessly integrated to the Eclipse Modeling Framework (EMF)[4]:

[3] Cf. https://github.com/gemoc/coordination

[4] Cf. https://eclipse.org/modeling/emf

- *Melange* (http://melange-lang.org), a tool-supported meta-language to modularly define executable modeling languages with execution functions and data, and to extend (EMF-based) existing modeling languages [5].
- *MoCCML*, a tool-supported meta-language dedicated to the specification of a Model of Concurrency and Communication (MoCC) and its mapping to a specific abstract syntax of a modeling language [4].
- *GEL*, a tool-supported meta-language dedicated to the specification of the protocol between the execution functions and the MoCC to support feedback of the runtime data and to support the callback of other expected execution functions [7].
- *BCOoL* (http://timesquare.inria.fr/BCOoL), a tool-supported meta-language dedicated to the specification of language coordination patterns, to automatically coordinates the execution of, possibly heterogeneous, models [8].
- *Sirius Animator*, an extension to the model graphical syntax designer Sirius (http://www.eclipse.org/sirius) to create graphical animators for executable modeling languages[5].

The different concerns of an executable modeling language (as defined with the tools of the language workbench) are automatically deployed into the modeling workbench that provides the following tools:

- *A Java-based execution engine* (parameterized with the specification of the execution functions), possibly coupled with TimeSquare[6] (parameterized with the MoCC), to support the concurrent execution and analysis of any conforming models.
- *A model animator* parameterized by the graphical representation defined with Sirius Animator to animate executable models.
- *A generic control panel* mapped to the Eclipse debug UI to control the execution of a running model (pause, resume, and step forward/backward) and define breakpoints.
- *A generic trace manager*, which allows system designers to visualize, save, replay, and investigate different execution traces of their models.
- *A generic event manager*, which provides a user interface for injecting external stimuli in the form of events during the simulation (e.g., to simulate the environment).
- *An heterogeneous coordination engine* (parametrized with the specification of the coordination in BCOoL), which provides runtime support to simulate heterogeneous executable models.

The GEMOC studio is open-source and domain-independent. The studio is available at http://gemoc.org/studio.

[5] For more details on Sirius Animator, we refer the reader to http://siriuslab.github.io/talks/BreatheLifeInYourDesigner/slides

[6] Cf. http://timesquare.inria.fr

4 Conclusion

The GEMOC studio provides the necessary tools to complement an EMF-based modeling language with a modular and formal behavioral semantics, possibly including an explicit model of concurrency. Such an executable domain-specific modeling language (xDSML) is automatically deployed in the modeling framework to support model execution, graphical animation, omniscient debugging and concurrency analysis. The model of concurrency is also the support of the definition of coordination patterns between different xDSMLs. Such coordination patterns are used to automatically generate the coordination between a set of specific conforming models.

The major scientific breakthroughs addressed by this work are i) the cross-fertilization of the algorithm theory and the concurrency theory to reify the concurrency concern in the specification of an operational semantics of a DSML, ii) the characterization of an event-based behavioral language interface, and iii) the reification of the coordination concerns at the language level.

The tool-supported appraoch presented in this paper open various perspectives regarding model executability, including the coordination of discrete and continuous models, the co-simulation (incl., FMI), the definition of adaptable concurrency model (e.g., design space exploration, optimizing compilers, code adaptation, code obfuscation), as well as the support of live and collaborative modeling.

Acknowledgement. This work is supported by the ANR INS Project GEMOC (ANR-12-INSE-0011) and The GEMOC Initiative (cf. http://gemoc.org/ins).

References

1. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X.: Essay on Semantics Definition in MDE. An Instrumented Approach for Model Verification. Journal of Software 4(9), 943–958 (Nov 2009)
2. Combemale, B., Deantoni, J., Baudry, B., France, R., Jézéquel, J.M., Gray, J.: Globalizing Modeling Languages. Computer pp. 68–71 (Jun 2014)
3. Combemale, B., Deantoni, J., Vara Larsen, M., Mallet, F., Barais, O., Baudry, B., France, R.: Reifying Concurrency for Executable Metamodeling. In: 6th International Conference on Software Language Engineering (SLE). Lecture Notes in Computer Science, Springer-Verlag (2013)
4. Deantoni, J., Issa Diallo, P., Teodorov, C., Champeau, J., Combemale, B.: Towards a Meta-Language for the Concurrency Concern in DSLs. In: Design, Automation and Test in Europe Conference and Exhibition (DATE) (Mar 2015)
5. Degueule, T., Combemale, B., Blouin, A., Barais, O., Jézéquel, J.M.: Melange: A Meta-language for Modular and Reusable Development of DSLs. In: 8th International Conference on Software Language Engineering (SLE) (Oct 2015)
6. Jézéquel, J.M., Combemale, B., Barais, O., Monperrus, M., Fouquet, F.: Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. Software and Systems Modeling 14(2), 905–920 (2015)

7. Latombe, F., Crégut, X., Combemale, B., Deantoni, J., Pantel, M.: Weaving Concurrency in eXecutable Domain-Specific Modeling Languages. In: 8th ACM SIGPLAN International Conference on Software Language Engineering (SLE). ACM, Pittsburg, United States (2015)
8. Vara Larsen, M.E., Deantoni, J., Combemale, B., Mallet, F.: A Behavioral Coordination Operator Language (BCOoL). In: International Conference on Model Driven Engineering Languages and Systems (MODELS). p. 462. No. 18, ACM (Sep 2015)

Model-Driven Process Engineering for flexible yet sound process modeling, execution and verification

Reda Bendraou

Sorbonne Universités, UPMC Univ Paris 06, UMR 7606, F-75005, Paris, France

Reda.bendraou@Lip6.fr

By capturing team's best practices, task ordering, flows of artifacts, agent coordination and communications, process models succeeded to become an important asset which ensures repeatability and quality in building software. In the context of large-scale industrial projects, they are more than vital and represent the means to foster the respect of project's deadlines and reliability criteria.

For many years, process models have been used only for training and communication purposes. However, they are used more and more as inputs for PSEEs (Process-centered Software Engineering Environments) to be enacted. The enactment of a process model consists of improving the coordination between the project's developers by automatically affecting the activities to the appropriate roles, routing the artifacts from one activity into another and ensuring the monitoring of activity's deadlines. The role of the PSEE is also to enforce the fact that the developers respect the execution order of the process's activities and to make sure that they deliver the accurate outcomes. This is how a company can guarantee the repeatability of its development processes and thus, gains in maturity and reliability.

However, this is based on three strong assumptions: **(i) the first, is that the process model is accurate and perfectly captures the right activities, milestones, artifacts and roles**. This also means that it represents the best possible enactment of a process for a given project's context. **(ii)** The second assumption is that **the process's agents are strictly following the process model and that they don't take any personal initiative to perform the process differently**. **(iii)** The third assumption is that **the process model is sound and free of any inconsistencies**. The notion of soundness here relates to the fact that there is no blocking state, unreachable activities, etc. while inconsistencies can be more related to some of the process's business or organizational constraints that might be violated with one of the possible enactment scenarios of the process.

Regarding the first assumption, for companies that have adopted process modeling and depending on the context, process models represent the outcome of many years of experience, best practices and process iterations. In some domains such as health care or military procedures, these models are intensively tested and evaluated before they are used in real production projects. If some improvements are identified during process iterations, they usually have to undergo a test and validation phases beforehand to be integrated to the process model. However, in the software domain this assumption is not always satisfied. Development processes depend on too many human factors and resources which make them highly unpredictable. Due to unexpected project constraints, occasionally, process models may have to evolve or be adapted during the process enactment. Examples of such constraints are: canceling the execution of some process's activities, switching the order of their execution or extending their deadlines.

The second assumption is also related to the process domain. In critical processes, it is unthinkable that the agent deviates from the process model. This would have critical impacts on the safety of the process. However, in software processes, this is different. Empirical studies showed that most agents do not follow an assigned process 98% of the time. This comes from the fact that developers may be confronted with unanticipated situations which are not represented in the process model, future and unexpected project constraints or their ability to accomplish the activity according to their experience and intuitions.

Finally, regarding the third assumption, investigations on some process repositories demonstrated that between 10% [Mendling 09] to 50% [Vanhatalo 07] of process models are flawed and contain inconsistencies. If you consider some critical businesses, having unsound process models could be very harmful for the quality of the delivered services and products. The SOA (Service Oriented Architecture) for instance heavily relies on representing the company's valuable processes in terms of BPMN (Business Process Modeling Notation) process models. These BPMN models are then used to automatically derive the web services' orchestration (calls) through a BPEL code (Business Process Execution Language). Having flawed BMPN process models would negatively impact this orchestration.

These three strong, and evidently wrong, assumptions were exactly the reasons why we initiated our work on the detection and handling of deviations in process enactment as well as our work on process verification. They were also in a way, what led to the fact that developers and some process actors lose faith in process execution engines and monitoring tools. For the two last decades the proposed solutions were either too much restrictive forbidding any kind of deviations or too permissive and lacked the automatic support for decision making initiatives. Probably one sure yet obvious think I confirmed after 12 years working on the topic of processes, is that no one can ever force humans to follow a process! Once this fact is recognized, the only possible option is to compromise with the different factors intervening in the realization of a process and to bring ***flexibility*** in the way you want to achieve process's goals. One could compare this to the notion of *optimistic transactions* in concurrent programming i.e. *I am optimistic, I believe that everything will go as planned, but if any problem arises, I should be informed so I can act accordingly*. Flexibility was one of our main driving goals in the different contributions that will be presented in this document. However, as this will be demonstrated too, this was never achieved at the expense of **formality** and **reliability** aspects.

Our second driving goal was to work for a **larger adoption of process technologies** by the industry. At this aim, every approach we proposed these last 12 years is based on mainstream **standards** and frameworks and comes with an intuitive, user-friendly tool that can be integrated to mainstream modeling tools and development environments. One again, this was always based on **solid and formal basis** which remains a critical goal for us.

The presentation will go through the different contributions we made to face the above mentioned issues/assumptions. It essentially presents our work on detecting and handling deviations during process enactment [Almeida 10, 11, 13] and our work on process verification[Laurent 13, 14a, 14b]. *Flexibility* and *a larger adoption* were the driving goals but at the same time, these goals where challenged by various issues and research objectives. **A detailed motivation of the issues and challenges we tried to solve** under each topic is giving in the presentation. Each topic's research objectives are justified, after which the solutions we proposed are presented. Obviously, this would never be as exhaustive as the material that one can find in our published papers which remain the master source for more details.

It is worth noticing that these contributions are **interdisciplinary** and revolve around **three main axes**: *Process Engineering, Model-Driven Engineering and Formal methods*.

References

[Almeida 10] M.A. Almeida da Silva, R. Bendraou, X. Blanc and M-P. Gervais, "Early Deviation Detection in Modeling activities of MDE Processes", in proceedings of MoDELS 2010, LNCS, Springer, October 3-8, 2010.

[Almeida 11] M.A. Almeida da Silva, X. Blanc and R. Bendraou, "Deviation Management during Process Execution", in proceedings of the 26th IEEE/ACM Int. Conf. On Automated Software Engineering (ASE 2011), November 2011,

[Almeida 13] Marcos Aurélio Almeida da Silva, Xavier Blanc, Reda Bendraou, Marie-Pierre Gervais. "Experiments on the impact of deviations to process execution" in the ISI Journal (Ingénierie des Systèmes d'Information), 18(3) : 295-119 (2013)

[Laurent 13] Laurent, Y., Bendraou, R., Gervais, M.P. : Generation of Process using Multi-Objective Genetic Algorithm. In : Proceedings of the 2013 International Conference on Software and Systems Process, ACM (2013), 161-165

[Laurent 14a] Y. Laurent, R. Bendraou, S. Baarir, M.‑P. Gervais : "Formalization of fUML: An Application to Process Verification", CAiSE 2014, Springer, p347-363

[Laurent 14b]Y. Laurent, R. Bendraou, S. Baarir, M.-P. Gervais : Alloy4SPV : a Formal Framework for Software Process Verification, in ECMFA, Lecture Notes in Computer Science, pp.83-100, (Springer) (2014)

[Mendling 09] Jan Mendling. Empirical studies in process model verification. In Transactions on Petri Nets and Other Models of Concurrency II, pages 208–224. Springer, 2009. 3, 4

[Vanhatalo 07]J.Vanhatalo, H. Völzer, and F.Leymann. Faster and more focused control-flow analysis for business process models through sese decomposition. In ICSOC 2007, pages 43–55. Springer, 2007. 3, 33, 143

An Overview of OMG Specifications for Executable UML Modeling.

Arnaud Cuccuru, Jérémie Tatibouet, Sahar Guermazi, Sébastien Revol and Sébastien Gérard
CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
P.C. 174, Gif-sur-Yvette, 91191, France
<firstname.lastname>@cea.fr

The purpose of this presentation is to give an introduction on a series of OMG standard specifications dealing with executable UML modeling, as well as to demonstrate concrete usages of these specifications.

In the first part of the talk, the following specifications are presented: fUML [1], PSCS [2], PSSM [3], and Alf [4]. fUML (Semantics of a Foundational Subset for Executable UML Models) identifies a subset of the UML metamodel, and provides a formal execution model for it. Any model defined with this subset can be interpreted by the execution model. The subset mostly comprises UML Classes for structural modeling, and UML Activities for behavioral modeling. While voluntarily minimalistic, this subset is expressive enough to describe systems composed of communicating concurrent entities, with reactive behaviors. The fUML execution model is designed following the Visitor design pattern. Each (executable) element of the fUML abstract syntax is associated with a semantic visitor that describes its execution semantics.

PSCS (Precise Semantics of UML Composite Structures) extends fUML with the ability to model structured classes, which can have ports, and an internal structure based on interconnected parts. These capabilities provide the basis for component-oriented modeling in UML. The fUML execution model is extended as well, in order to account for key aspects of composite structures. It includes aspects related to the life cycle of composite objects (instantiation/destruction of composite structures and their constituents) as well as the semantics of communications (propagation of requests and messages along ports and connectors).

PSSM (Precise Semantics of UML State Machines) is an ongoing standardization effort, which aims at further extending fUML and PSCS with State Machine modeling. State machines are one of the most used behavioral formalism in UML, and will significantly help in broadening the scope of application domains for these series of OMG specifications on executable UML modeling. At the time this presentation is given, an initial submission has already been voted (March 2016), and a revised submission is scheduled for September 2016.

To conclude the first part of the talk, we introduce Alf, the Action Language for fUML. The Alf specification focuses on syntactic aspects of executable UML modeling, by providing a concrete and concise textual notation for fUML. Alf basically enables to program directly in UML. As such, it greatly improves the productivity of executable model designers.

In the second part of the talk, we highlight the added value of these specifications with tooling and usage examples taken from actual CEA LIST R&D activities, which leverage their formal execution semantics for specific uses and/or application domains: Performing tradeoff analysis with discrete event simulation, controlling and observing systems with models at runtime, and evaluating executable UML models in a Cyber-Physical System co-simulation environment (by relying on the FMI – Functional Mockup Interface – co-simulation standard).

[1] Semantics of a foundational subset for executable UML models (fUML) http://www.omg.org/spec/FUML/

[2] Precise semantics of UML composite structures (PSCS) http://www.omg.org/spec/PSCS/

[3] Precise semantics of UML state machines RFP (PSSM) http://www.omg.org/cgi-bin/doc.cgi?ad/2015-3-2

[4] Action language for foundational UML (Alf) http://www.omg.org/spec/ALF/

Session du groupe de travail IE

Ingénierie des Exigences

Utilisation des ontologies pour l'ingénierie des exigences

Driss Sadoun[1]
1 : INALCO, Driss.Sadoun@inalco.fr.

L'ingénierie des exigences (IE) consiste à comprendre ce que l'on a l'intention de concevoir avant de le concevoir. Elle repose sur la capture de connaissances pouvant provenir de diverses sources et parties prenantes ayant leurs propres intérêts et points de vue. L'objectif principal est de permettre une compréhension commune, par les différents intervenants du système à concevoir, afin de guider et de faciliter sa réalisation. En général, cette compréhension est représentée sous la forme de documents écrits, ce qui correspond à la spécification des exigences.

La spécification d'exigences est un travail collaboratif. Les exigences doivent être simples et spécifiées dans un langage commun, qu'experts et non experts seront aptes à comprendre. Elles sont donc généralement écrites en langue naturelle (LN). En revanche, le manque de rigueur et l'ambiguïté inhérente aux textes rédigés en LN peut mener à des différences d'interprétation entre les parties prenantes et mener à des problèmes de conception. Ainsi, il n'est pas aisé de spécifier un ensemble complet et cohérent d'exigences. Les exigences et leur qualité ont alors tendance à évoluer au fil du processus de développement et de la compréhension du système à concevoir. L'amélioration des spécifications d'exigences passe entre autres par les phases d'analyse et de vérification. Pour cela, elles doivent être représentées de telle sorte à pouvoir être traitées par des méthodes de vérification formelle.

Les spécifications formelles qui sont plus précises et concises que les spécifications LN constituent un meilleur point de départ pour les étapes d'analyse, de vérification et de conception. Il est plus aisé d'éviter les erreurs de conception en passant de spécifications formelles à une implémentation [Andrews et Gibbins, 1988]. Cependant, la complexité de leur syntaxe et, plus encore, leurs fondements mathématiques, les rendent difficilement compréhensibles pour une personne non experte. La validation des exigences dans leur forme finale requiert alors le plus souvent qu'elles soient décrites en langue naturelle, de sorte à être contractuelles.

Lors de la conception d'un système, spécifications LN et spécifications formelles sont donc complémentaires. Mener à terme un projet d'ingénierie impose l'emploi itératif de ces deux formalismes dans un processus évolutif. Gérer les transitions entre ces représentations est la clé du succès de toute tâche d'ingénierie [Kitapci et Boehm, 2006]. La difficulté principale d'une telle transition réside dans l'ampleur du fossé entre spécifications LN et spécifications formelles. Cette question a été abordée par l'emploi de formalismes intermédiaires [Bajwa *et al.*, 2012, Guissé *et al.*, 2012, Njonko et El Abed, 2012, Sadoun, 2014] permettant de réduire ce fossé. La problématique est alors de choisir une représentation suffisamment formelle et précise pour permettre un passage automatisé vers des spécifications formelles et suffisamment expressive pour représenter la sémantique de connaissances formulées en langage naturel.

Au fil du temps, les langues naturelles ont évolué davantage pour répondre au besoin de communication qu'au besoin de représentation [Russell *et al.*, 1996]. De sorte que les connaissances nécessaires à l'interprétation d'un texte en LN ne sont pas forcément toutes explicitées. Les connaissances dont l'inférence est supposée possible sont le plus souvent laissées implicites. Ces connaissances implicites peuvent être inférées à partir de connaissances partagées par le(s) rédacteur(s) et ses lecteurs.

Ces dernières années, les ontologies se sont imposées comme l'un des modèles de représentation les plus simples et efficaces pour la description d'un domaine d'application. En effet, elles permettent d'expliciter et de formaliser les concepts clés d'un domaine ainsi que leurs propriétés, définissant ainsi le vocabulaire conceptuel du domaine. Chaque élément de ce vocabulaire doit posséder une interprétation unique, issue d'une compréhension commune et partagée. Ce cadre unificateur permet de passer de jargons individuels à une terminologie commune, atténuant ainsi les malentendus au sujet des termes et permettant de réduire la confusion terminologique et conceptuelle.

Le potentiel des ontologies comme moyen de représenter les connaissances d'un domaine, d'améliorer la compréhension commune, de préciser et d'organiser les connaissances, amène une quantité croissante de travaux à s'intéresser à leur utilisation dans le domaine de l'ingénierie des exigences (IE). Notamment, pour la spécification [Souag., 2012, Al Balushi *et al.*, 2013], l'interopérabilité [Boukhari *et al.*, 2012], la validation [Körner et Brumm, 2010, Chniti *et al.*, 2012] ou l'amélioration [Castañeda *et al.*, 2012, Körner et Brumm, 2010] des d'exigences.

Les fondements logiques des ontologies permettent de décrire de manière formelle et donc interprétable par les machine, le comportement et les contraintes d'un système [Sadoun *et al.*, 2011]. Il est alors possible à l'aide de techniques de traitement automatique de la langue (TAL) d'exploiter les connaissances formalisées au sein d'une ontologie pour guider l'extraction automatique de nouvelles connaissances dans des spécifications d'exigences en langue naturelle [Sadoun *et al.*, 2013b, Sadoun *et al.*, 2013a]. Ces nouvelles connaissances peuvent alors s'ajouter aux connaissances de l'ontologie pour représenter une instanciation particulière, celle de la spécification d'exigences analysée. En outre, les mécanismes d'inférence associés aux ontologies permettent non seulement de déduire des connaissances implicites ou erronées, mais aussi de vérifier la complétude, la cohérence et la conformité des connaissances modélisées [Sadoun *et al.*, 2012]. Par ailleurs, le formalisme logique des ontologies autorise une transformation simple et intuitive de la représentation issue de l'analyse des spécification d'exigence LN vers un langage de spécification formel [Sadoun *et al.*, 2014], permettant l'application de méthodes de vérification formelle. Enfin, l'ontologie, en tant que modèle de représentation conçu pour capturer la sémantique du langage naturel, permet de garder une trace du lien entre les deux niveaux de spécifications et ainsi de faciliter la navigation entre spécifications en langue naturelle et spécifications formelles [Sadoun, 2014].

Références

[Al Balushi *et al.*, 2013] Al Balushi, T. H., Sampaio, P. R. F. et Loucopoulos, P. (2013). Eliciting and prioritizing quality requirements supported by ontologies : a case study using the elicito framework and tool. *Expert Systems*, 30(2):129–151.

[Andrews et Gibbins, 1988] Andrews, D. et Gibbins, P. (1988). *An Introduction to Formal Methods of Software Development.* An Introduction to Formal Methods of Software Development. McGraw-Hill Education.

[Bajwa *et al.*, 2012] Bajwa, I. S., Bordbar, B., Lee, M. et Anastasakis, K. (2012). Nl2alloy : A tool to generate alloy from nl constraints. *Journal of Digital Information Management*, 10(6).

[Boukhari *et al.*, 2012] Boukhari, I., Bellatreche, L. et Jean, S. (2012). An ontological pivot model to interoperate heterogeneous user requirements. *In Proceedings of the 5th international conference on Leveraging Applications of Formal Methods, Verification and Validation : applications and case studies - Volume Part II*, ISoLA'12, pages 344–358.

[Castañeda *et al.*, 2012] Castañeda, V., Ballejos, L. C. et Caliusco, M. L. (2012). Improving the quality of software requirements specifications with semantic web technologies. *In WER.*

[Chniti *et al.*, 2012] Chniti, A., Albert, P. et Charlet, J. (2012). Gestion de la cohérence des règles métier éditées à partir d'ontologies owl. *In Actes de IC2011*, pages 589–606.

[Guissé *et al.*, 2012] Guissé, A., Lévy, F. et Nazarenko, A. (2012). From regulatory texts to brms : how to guide the acquisition of business rules ? *In Proceedings of the 6th international conference on Rules on the Web : research and applications (RuleML'12)*, pages 77–91.

[Kitapci et Boehm, 2006] Kitapci, H. et Boehm, B. W. (2006). Using a hybrid method for formalizing informal stakeholder requirements inputs. *In Proceedings of the Fourth Internationa Workshop on Comparative Evaluation in Requirements Engineering (CERE '06)*, pages 48–59.

[Körner et Brumm, 2010] Körner, S. J. et Brumm, T. (2010). Natural language specification improvement with ontologies. *International Journal of Semantic Computing (IJSC)*, 3(4):445–470.

[Njonko et El Abed, 2012] Njonko, P. et El Abed, W. (2012). From natural language business requirements to executable models via sbvr. *In Systems and Informatics (ICSAI), 2012 International Conference on.*

[Russell *et al.*, 1996] Russell, S. J., Norvig, P., Candy, J. F., Malik, J. M. et Edwards, D. D. (1996). *Artificial intelligence : a modern approach.* Prentice-Hall, Inc.

[Sadoun, 2014] Sadoun, D. (2014). *Des spécifications en langage naturel aux spécifications formelles via une ontologie comme modèle pivot.* Thèse de doctorat, Univ. Paris 11.

[Sadoun *et al.*, 2011] Sadoun, D., Dubois, C., Ghamri-Doudane, Y. et Grau, B. (2011). An ontology for the conceptualization of an intelligent environment and its operation. *In 10th Mexican International Conference on Artificial Intelligence (MICAI)*, pages 16–22.

[Sadoun *et al.*, 2012] Sadoun, D., Dubois, C., Ghamri-Doudane, Y. et Grau, B. (2012). Formalisation en owl pour vérifier les spécifications d'un environnement intelligent. *In Actes de la conférence RFIA.*

[Sadoun *et al.*, 2013a] Sadoun, D., Dubois, C., Ghamri-Doudane, Y. et Grau, B. (2013a). From natural language requirements to formal specification using an ontology. *In 25th International Conference on Tools with Artificial Intelligence (ICTAI).*

[Sadoun *et al.*, 2013b] Sadoun, D., Dubois, C., Ghamri-Doudane, Y. et Grau, B. (2013b). Peuplement d'une ontologie guidé par l'identification d'instances de propriété. *In Actes de la 10e conférence TIA.*

[Sadoun *et al.*, 2014] Sadoun, D., Dubois, C., Ghamri-Doudane, Y. et Grau, B. (2014). Formal rule representation and verification from natural language requirements using an ontology. *In RuleML 2014, Prague, Czech Republic*, pages 226–235.

[Souag., 2012] Souag., A. (2012). Vers une nouvelle génération de définition des exigences de sécurité fondée sur l'utilisation des ontologies. *In INFORSID*, pages 583–590.

Vérification Formelle d'Observateurs Orientée Contexte

Ciprian Teodorov, Philippe Dhaussy
ciprian.teodorov@ensta-bretagne.fr,
philippe.dhaussy@ensta-bretagne.fr

Lab-STICC, UMR CNRS 6285, ENSTA Bretagne, Brest.

Malgré la performance croissante des outils de $model - checking$, leur utilisation reste difficile en contexte industriel. Leur intégration dans un processus d'ingénierie industriel est encore trop faible comparativement à l'énorme besoin de fiabilité dans les systèmes critiques. Cette contradiction trouve en partie ses causes dans la difficulté réelle de mettre en œuvre des concepts théoriques dans un contexte industriel. Une première difficulté liée à l'utilisation de ces techniques de vérification provient du problème bien identifié de l'explosion combinatoire du nombre de comportements des modèles, induite par la complexité interne du logiciel qui doit être vérifié. Cela est particulièrement vrai dans le cas des systèmes embarqués temps réel, qui interagissent avec des environnements impliquant un grand nombre d'entités. Une autre difficulté est liée à l'expression formelle des propriétés nécessaire à leur vérification. Traditionnellement, cette expression s'effectue à l'aide de formalismes tels que les logiques temporelles. Bien que ces logiques aient une grande expressivité, elles ne sont pas faciles à utiliser par des ingénieurs dans des contextes industriels.

L'approche développée dans nos travaux a pour vocation d'appréhender les difficultés mentionnées précédemment. Elle repose sur deux idées conjointes : d'une part, réduire les comportements du système à valider lors du model-checking de manière à repousser plus loin les limites de l'explosion combinatoire et, d'autre part, aider l'utilisateur à spécifier les propriétés formelles à vérifier. Pour cela, nous avons proposé un langage nommé CDL (*Context Description Language*) [DBR11,DBRL12] pour la spécification de contextes et de propriétés formelles. Ce langage est associé à l'outil OBP[1] qui permet d'explorer des modèles au format Fiacre [FGP$^+$08] et prend en charge les descriptions CDL. Les modèles Fiacre peuvent provenir de chaînes de transformation de modèles décrits dans des formalismes de plus haut niveau tel que UML ou SysML et largement utilisés dans les équipes de développement. De nombreuses expérimentations de cet outil ont été menées en collaboration avec des partenaires industriels [CLP14,TDR16].

Pour repousser plus loin les limites de l'explosion combinatoire, nous cherchons, dans notre travail, à traiter deux sources de la complexité : la complexité

[1] Outil accessible librement avec documentation sous http://www.obpcdl.org.

externe (i.e., celle de l'environnement) et la complexité interne (i.e., celle du système). Plus précisément, cette approche cherche à réduire l'espace des comportements possibles en considérant un modèle explicite de l'environnement du système. En d'autres termes, il s'agit de "fermer" le système avec l'ensemble des cas d'utilisation de l'environnement auxquels il est sensé répondre, et uniquement ceux-ci. La réduction est basée sur une description formelle de ces cas d'utilisation, que nous nommons contextes, avec lesquels le système interagit. L'objectif est ainsi de guider le model-checker à concentrer ses efforts non plus sur l'exploration de l'espace complet des comportements, qui peut être gigantesque, mais sur une restriction pertinente de ce dernier pour la vérification de propriétés spécifiques. La vérification de propriétés devient alors possible sur l'espace d'états ainsi réduit. Cette méthode se base ainsi sur la connaissance qu'ont les concepteurs du système et leur expertise qui permet de spécifier de manière explicite l'environnement du système

Afin de rendre plus accessible l'expression de propriétés formelles, le langage CDL permet de définir les propriétés formelles basées sur des patrons de propriétés étendus à partir des propositions de [DAC99,SACO02,KC05]. Lors de la définition du langage CDL, nous avons inclus la possiblité pour l'utilisateur d'exprimer les propriétés à l'aide de patrons conjointement à la description des contextes. Les patrons qui ont été identifiés dans une première approche permettent d'exprimer des propriétés de réponse (Response), de prérequis (Precedence), d'absence (Absence), d'existence (Existence). Les propriétés font référence à des évènements détectables tels que des envois ou réceptions de signaux, des changements d'état de processus ou de variables. Ces formes de base peuvent ensuite être enrichies par des options (*Pre-arity, Post-arity, Immediacy, Precedency, Nullity, Repeatability*) à l'aide d'annotations. Dans CDL, nous avons enrichi les patrons avec la possibilité de manipuler des ensembles d'évènements, ordonnés ou non, comme proposé par [JMM$^+$99]. Les opérateurs AN et ALL précisent respectivement si un évènement ou tous les évènements, ordonnés ($Ordered$) ou non ($Combined$), d'un ensemble d'évènements sont concernés par la propriété.

Une sémantique formelle des patrons CDL a été définie en COQ [SSMP09] et permet une transformation vers des automates observateurs acceptable par l'outil OBP. Un observateur est construit de manière à encoder une propriété logique [ABBL98] et a pour rôle d'observer, partiellement et de manière non intrusive, les évènements significatifs survenant dans le modèle du système à valider. Lors de la simulation, il est composé, de manière synchrone, au modèle à observer. Les automates observateurs contiennent des états spéciaux appelés états de rejet ($reject$). Lors de l'exploration exhaustive du résultat de cette composition, s'il existe une exécution faisant basculer l'observateur dans l'état de rejet, la propriété encodée par l'observateur sera considérée fausse. Le diagnostic consiste alors en une analyse d'accessibilité des états de rejet sur le graphe d'exécution qui résulte de la composition du modèle à valider et de l'observateur. Dans ce cas, si aucun état de rejet n'est atteint, la propriété est vérifiée. Les observateurs permettent ainsi d'exprimer des propriétés de sûreté, d'accessibilité et de vivacité bornée.

Les perspectives de recherche sont nombreuses. Elles concernent la technique de réduction des graphes, la formalisation de la composition d'observateurs et l'aide au diagnostics basé sur les données de retour des explorations de modèles. L'exposé proposé rendra compte des résultats déjà obtenus avec cette outillage sur des cas inspirés du domaine industriel. Les publications concernant nos recherches sont accessibles sur le site http://www.obpcdl.org.

References

[ABBL98] L. Aceto, P. Bouyer, A. Burgueño, and K. G. Larsen. The power of reachability testing for timed automata. In *Proc. 18th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'98), Chennai, India, Dec. 1998*, volume 1530, pages 245–256. Springer, 1998.

[CLP14] Teodorov Ciprian, Leroux Luka, and Dhaussy Philippe. Context-aware verification of a cruise-control system. In *h International Conference on Model and Data Engineering (MEDI)*, Larnaca, Cyprus, September 2014.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *21st Int. Conf. on Software Engineering*, pages 411–420. IEEE Computer Society Press, 1999.

[DBR11] Philippe Dhaussy, Frédéric Boniol, and Jean-Charles Roger. Reducing state explosion with context modeling for model-checking. In *13th IEEE International High Assurance Systems Engineering Symposium, Hase*, Boca Raton, USA, 2011.

[DBRL12] Philippe Dhaussy, Frédéric Boniol, Jean-Charles Roger, and Luka Leroux. Improving model checking with context modelling. *Advances in Software Engineering*, ID 547157:13 pages, 2012.

[FGP$^+$08] Patrick Farail, Pierre Gaufillet, Florent Peres, Jean-Paul Bodeveix, Mamoun Filali, Bernard Berthomieu, Saad Rodrigo, Francois Vernadat, Hubert Garavel, and Frédéric Lang. FIACRE: an intermediate language for model verification in the TOPCASED environment. In *European Congress on Embedded Real-Time Software (ERTS)*, Toulouse, january 2008. SEE.

[JMM$^+$99] Wil Janssen, Radu Mateescu, Sjouke Mauw, Peter Fennema, and Petra Van Der Stappen. Model checking for managers. In *SPIN*, pages 92–107, 1999.

[KC05] S. Konrad and B. Cheng. Real-time specification patterns. In *27th Int. Conf. on Software Engineering (ICSE05), St Louis, MO, USA*, 2005.

[SACO02] R. Smith, G.S. Avrunin, L. Clarke, and L. Osterweil. Propel: An approach supporting property elucidation. In *24st Int. Conf. on Software Engineering(ICSE02), St Louis, MO, USA*, pages 11–21. ACM Press, 2002.

[SSMP09] Jagadish Suryadevara, Cristina Cerschi Seceleanu, Frédéric Mallet, and Paul Pettersson. The coq proof assistant references manual version 8.2. *The Coq Development Team*, February 2009.

[TDR16] Ciprian Teodorov, Philippe Dhaussy, and Luka Le Roux. Environment-driven reachability for timed systems : Safety verification of an aircraft landing gear system. In *(to be published) Int. Software Tools for Technology Transfer (STTT)*. Springer-Verlag, 2016.

Ciprian Teodorov est enseignant-chercheur au laboratoire Lab-STICC, UMR CNRS 6285, à l'ENSTA-Bretagne. Ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles pour l'industrialisation d'outils de preuve pour les systèmes embarqués et systèmes sur puce.

Avant intégrer l'équipe MOCS du Lab-STICC à l'ENSTA-Bretagne, il a été ingénieur CAO à Dolphin Integration en charge principalement de l'intégration d'une nouvelle infrastructure de simulation VHDL dans le simulateur mixte SMASH. En 2011 il a obtenu le titre de docteur en informatique de l'Université de Bretagne Occidentale pour ses travaux sur une approche dirigée par les modèles pour la synthèse physique de circuits émergents nanométriques.

Philippe Dhaussy est professeur à l'ENSTA-Bretagne et membre de l'équipe MOCS au Lab-STICC, UMR CNRS 6285. Son expertise et ses recherches sont orientées vers l'Ingénierie Dirigées par les Modèles et les techniques de validation formelle pour le développement de logiciels embarqués et temps réel. Après un parcours industriel (1980-1991), il a rejoint l'ENSTA-Bretagne et a contribué à la création du groupe de recherche en modélisation de systèmes et logiciels embarqués, intégré au laboratoire Lab-STICC. Il a obtenu le titre de docteur (1994) à Telecom Bretagne et son habilitation à diriger des recherches en 2014.

Vers des systèmes logiciels auto-adaptatifs qui permettent la re-configuration lors de l'exécution

Raúl Mazo

Centre de Recherche en Informatique (CRI), Université Panthéon - Sorbonne, Paris 1
90 rue de Tolbiac, 75013 Paris – France
{raul.mazo}@univ-paris1.fr

Abstract. Ce document présente le résumé de mon exposé lors des journées du GDR-GPL 2016. En particulier, ce résumé présente les enjeux, questions de recherche, fondements théoriques et résultats de la recherche que les chercheurs du Centre de Recherche en Informatique réalisent autour des systèmes logiciels auto-adaptatifs. De plus, il est accompagné de quelques références sur nos travaux relatifs à la présentation et quelques lectures recommandées pour en savoir plus.

1 Introduction

Les lignes de produits logiciels et une nouvelle génération de middleware et d'architectures orientées services, nées récemment, ont aidé à définir, mettre en œuvre et exécuter des systèmes capables d'adapter leur comportement en fonction de changements propres à eux ou de changements qui interviennent dans leur environnement et dans leurs conditions d'exécution.

Ces systèmes permettent de faire des adaptations dynamiques face aux changements environnementaux. Il s'agit d'une méthode bien plus radicale que l'utilisation de simples adaptations contrôlées par des paramètres [1].

L'émergence d'un tel mécanisme d'auto-adaptation a de profondes implications dans le développement de logiciels. En particulier, il permet (i) le développement de logiciels pouvant fonctionner sur un large éventail de contextes sans avoir à énumérer comment le système doit se comporter dans chaque environnement; et (ii) le développement de logiciels pouvant être conçus et pouvant fonctionner en présence d'incertitudes [2] quant aux contextes qui peuvent survenir lors de l'exécution.

En effet, grâce à la capacité d'auto-adaptation, les décisions de conception ne sont plus faites uniquement au moment de la conception mais aussi au moment de l'exécution. Cela est possible uniquement si le système est capable de s'auto-surveiller (c'est-à-dire de surveiller ses propres états et ses exigences correspondantes) et de surveiller son environnement au moment de l'exécution. Donc, en utilisant les données de surveillance de l'état interne du système, du contexte et des exigences du domaine, chaque système peut déclencher ses propres adaptations si nécessaire. Toutefois, ces adaptations doivent être conduites par la nécessité de satisfaire de manière intelligente les exigences du système.

Il existe actuellement un large fossé conceptuel entre ce que les systèmes auto-adaptatifs peuvent faire et ce qu'ils devraient faire lorsque les «changements» surviennent. Par exemple, comment un système peut-il tolérer le changement (par optimisation, par re-configuration, par mutation)? Quels genres de changements doit-on considérer (les changements des critères de qualité, de nouvelles exigences qui émergent, de nouveaux éléments de contexte, les changements d'état dans le système ou dans le contexte)? A quel moment doivent-ils être pris en compte (au moment de la conception, de l'exécution)?

Anderson et al. [3] caractérisent les changement donnant lieu à des adaptations comme : prévu (*pris en charge*), prévisible (*prévu pour*) et imprévu (*non prévu pour*). L'adaptation dynamique, qui permet de satisfaire de nouvelles exigences, répond au *changement imprévu*. Quant aux systèmes entièrement autonomes, ils sont possibles, à ma connaissance, uniquement dans le domaine de la science-fiction, pour le moment.

Les middleware adaptatifs et les architectures orientées services sont désormais en mesure de répondre aux *changements prévus*. En revanche, la conception de systèmes capables de faire face aux *changements prévisibles* mais *non prévus* au moment de la conception reste un vrai défi. Lorsqu'il sera relevé, la conception de systèmes auto-adaptatifs sera nettement améliorée.

Cependant, même si l'adaptation consiste à optimiser un ensemble d'exigences fixes (comme cela est le cas pour les changements *prévus* et *prévisibles*), les techniques existantes pour raisonner sur les exigences ne sont pas suffisantes. En effet, il faut spécifier non seulement le comportement du système lorsque l'environnement est dans un état stable, mais aussi le comportement adaptatif (c'est-à-dire dans quelles circonstances le système doit s'adapter).

Pire encore, si, comme cela est souvent le cas, la connaissance de l'environnement est incomplète au moment de la conception, le comportement du système doit être spécifié non seulement pour les contextes environnementaux que le système risque de rencontrer, selon les prévisions de l'analyste, mais aussi pour les contextes environnementaux que le système pourrait rencontrer mais qui n'auraient pas été considérés par l'analyste.

L'utilisation de lignes de produits logiciels peut être considérée comme une stratégie pour faire face à l'incertitude et à l'adaptation au changement venant des environnements d'exécution des systèmes dérivés, des systèmes dérivés eux mêmes, des exigences des clients et des exigences du marché de la ligne de produits.

Contrairement aux systèmes auto-adaptatifs autonomes, la dérivation de nouvelles configurations à partir des lignes de produits implique généralement la satisfaction de nouveaux objectifs plutôt que la façon dont un ensemble d'objectifs fixes sont réalisés. En outre, certains des objectifs des lignes de produits ne sont pas définis lorsque la ligne est conçue, mais sont ajoutés par la suite. Dans ce cas, la mise en œuvre de ces nouvelles exigences devrait être liée à l'exécution. C'est ce que nous appelons une ligne de produits dynamique.

2 Quelques références sur mes travaux relatifs à la présentation

Dans les articles suivants, l'équipe du CRI étudie de nouvelles idées pour améliorer la façon dont les chercheurs et les industriels utilisent les techniques d'ingénierie des lignes de produits pour mettre en œuvre la re-configuration dynamique des systèmes (auto) adaptatifs.

Sprovieri D., Diaz D., Hinkelmann K., Mazo R. Run-time planning of case-based business processes. In the X International Conference on Research Challenges in Information Science (RCIS), IEEE Press, Grenoble-France (2016)

Kirsch-Pinheiro M., Mazo R., Souveyet C., Sprovieri D. Requirements Analysis for Context-oriented Systems. In the VII International Conference on Ambient Systems, Networks and Technologies (ANT), Madrid-España (2016)

Muñoz-Fernández J., Tamura G., Mazo R., Salinesi C. Towards a Requirements Specification Multi-View Framework for Self-Adaptive Systems. In CLEI electronic journal, Volume 18, Number 2, Paper 5 (2015)

Dounas L., Mazo R., Salinesi C., El-Beqqali O. Continuous Monitoring of Adaptive e-learning Systems Requirements. In the XII ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), Marrakech-Morocco (2015)

Alférez G., Pelechano V., Mazo R., Salinesi C., Diaz D. Dynamic Adaptation of Service Compositions with Variability Models. The Journal of Systems & Software, Volume 91, pp. 24-47 (2014)

Sawyer P., Mazo R., Diaz D., Salinesi C., Hughes D. Constraint Programming as a Means to Manage Configurations in Self-Adaptive Systems. Special Issue in IEEE Computer Journal "Dynamic Software Product Lines", ISSN 0018-9162, vol. 45, Number 10, pp. 56-63 (2012)

Mazo R., Salinesi C., Diaz D., Djebbi O., Lora-Michiels A. Constraints: the Heart of Domain and Application Engineering in the Product Lines Engineering Strategy. International Journal of Information System Modeling and Design IJISMD. pp. 33-68. ISSN 1947-8186, eISSN 1947-819 (2012)

3 Autres lectures pour en savoir plus

E. Letier and A. van Lamsweerde. Reasoning about partial goal satisfaction for requirements and design engineering. Proc. of Twelfth ACM SIGSOFT International Symposium on Foundations of Software Engineering (2004)

W. Robinson. A requirements monitoring framework for enterprise systems. Requirements Engineering Journal, 11(1) (2005)

S. A. DeLoach and M. Miller. A goal model for adaptive complex systems. International Journal of Computational Intelligence: Theory and Practice, 5(2) (2010)

L. Baresi and L. Pasquale. Fuzzy goals for requirements-driven adaptation. Proc. Eighteenth International IEEE Requirements Engineering Conference (RE'10), Sydney, Australia (2010)

A. Lapouchnian. Exploiting requirements variability for software customization and adaptation. Ph.D. dissertation, University of Toronto (2011)

Amir Molzam Sharifloo, Andreas Metzger, Clément Quinton, Luciano Baresi and Klaus Pohl. Learning and Evolution in Dynamic Software Product Lines. In Proceedings of the 11th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS. To appear (2016)

References

[1] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng.: Composing adaptive software. IEEE Computer, 37(7) (2004)

[2] K. Welsh, P. Sawyer.: Understanding the Scope of Uncertainty in Dynamically Adaptive Systems. Proc. Sixteenth International Working Conference on Requirements Engineering: Foundations of Software Quality (REFSQ), Essen-Germany (2010)

[3] J. Andersson, R. Lemos, S. Malek, and D. Weyns.: Modeling dimensions of self-adaptive software systems," in Software Engineering for Self-Adaptive Systems. Springer-Verlag (2009)

Session du groupe de travail LaMHA

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

8th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2015)
Pisa, Italia
July 2-3, 2015

Multi-ML: Programming Multi-BSP Algorithms in ML

V. Allombert · F. Gava · J. Tesson

Abstract The MULTI-BSP model is an extension of the well known BSP bridging model. It brings a tree based view of nested components to represent hierarchical architectures. In the past, we designed BSML for programming BSP algorithms in ML. We now propose the MULTI-ML language as an extension of BSML for programming MULTI-BSP algorithms in ML.

Keywords BSP · MULTI-BSP · ML.

1 Introduction

Context of the work. Nowadays, parallel programming is the norm in many areas but it remains a *hard task*. And the more complex the parallel architectures become, the harder the task of programming them *efficiently* is. As we moved from *unstructured* sequential code cluttered with goto statement toward *structured* code, there has been a move in parallel programming community to leave unstructured parallelism, with pairwise communications, in favour of *global communication scheme* [2,6,26] and of structured *abstract* models like BSP [2,33] or of higher-level concepts like *algorithmic skeletons* [15].

Programming in the context of a *bridging model*, such as BSP, allows to simplify the task of the programmer, to ease the reasoning on *cost* and to ensure a better *portability* from one system to another [2,22,33]. However, designing a programming language [17,21] for such a model requires to chose a *trade-off* between the possibility to control parallel aspects necessary for *predictable* efficiency (but which make programs more difficult to write, to prove and to port) and the *abstraction* of such features which are necessary to make programming easier —but which may hampers efficiency and *performance* prediction.

With *flat homogeneous architectures*, like clusters of mono-processors, BSP has been proved to be an effective target model for the design of efficient algorithms and languages [35]: while its structured nature allows to avoid *deadlocks* and *non-determinism* with little care and to reason on program *correctness* [13,36,37] and cost, it is general enough to express many algorithms [2]. But *modern* parallel architecture have now *multiple layers* of parallelism. For example, supercomputers are made of thousands of *interconnected nodes*, each one carrying *several multicores* processors. Communications between distant nodes cannot be as fast as communications among the cores of a given proces-

LACL, University of Paris-East, Créteil, France
E-mail: victor.allombert@lacl.fr · E-mail: {frederic.gava,julien.tesson}@univ-paris-est.fr

sor; Communications between cores, by accessing shared processor cache are faster than communications between processors through RAM.

Contribution of this paper. Those architectures specifics led to a *new bridging model*, MULTI-BSP [34], where the hierarchical nature of parallel architectures is reflected by a *tree-shaped* model describing the dependencies between memories. The MULTI-BSP model gives a more precise picture of the cost of computations on modern architectures. While the model is more complex to grasp than the BSP one, it keeps structured characteristics that prevents deadlock and non-determinism. We propose a language, MULTI-ML, which aim at providing a mean to program MULTI-BSP algorithms so as our past BSML [14] is a mean to program BSP ones. MULTI-ML combines the high degree of *abstraction* of ML (without poor performances because often, ML programs are as efficient as C ones) with the *scalable* and *predictable* performances of MULTI-BSP.

Outline. The remainder of this paper is structured as follows. We first give in section 2 an overview of previous works: the BSP model at Section 2.1 and then the BSML language at Section 2.2 following with the MULTI-BSP model at Section 2.3. Our language MULTI-ML is presented at Section 3. Its formal semantics and implementation, together with examples and benchmarks are given at Section 4. Section 5 discusses some related work and finally, Section 6 concludes the paper and gives a brief outlook on future work.

2 Previous Work

In this section, we present shortly the BSP and MULTI-BSP models and how to program BSP algorithms using the BSML language. We also give an informal semantics of BSML primitives and simple examples of BSML programs. We assume the reader is familiar with ML programming.

2.1 The BSP Model of Computation

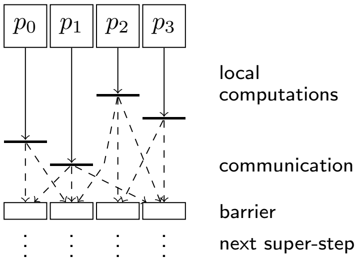


Fig. 1 A BSP super-step.

In the BSP model [2,33], a computer is a set of **p** uniform processor-memory pairs and a communication network. A BSP program is executed as a sequence of *super-steps* (Fig. 1), each one divided into three successive disjoint phases: (1) each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) the network delivers the requested data; (3) a global synchronisation barrier occurs, making the transferred data available for the next super-step. This *structured* model enforces a strict *separation* of communication and computation: during a super-step, no communication between the processors is allowed but only transfer requests; only at the *barrier* information is actually exchanged. Note that a BSP library can send data during the computation phase of a super-step, but this is hidden to programmers.

The *performance* of the BSP computer is characterised by 4 *parameters*: (1) the local processing speed **r**; (2) the number of processor **p**; (3) the time **L** required for a barrier; (4) and the time **g** for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word. The network can deliver an h-relation (every processor receives/sends at most h words) in time $\mathbf{g} \times h$. To accurately *estimate* the execution time of a BSP program, these 4 parameters could be easily benchmarked [2]. The execution time (cost) of a super-step s is the sum of the maximal of the local processing time, the data delivery and the global synchronisation times. The total cost (execution time) of a BSP program is the sum of its super-steps's costs.

2.2 BSP Programming in ML

BSML [14] uses a *small set of primitives* and is currently implemented as a library (`http://bsmllib.free.fr`) for the ML programming language OCAML (`http://caml.org`). An important feature of BSML is its *confluent* semantics: whatever the order of execution of the processors, the final value will be the same. Confluence is convenient for *debugging* since it allows to get an *interactive loop* (toplevel). That also eases programming since the parallelisation can be done *incrementally* from a ML program. Last but not least, it is possible to reason about BSML programs using the COQ (`http://https://coq.inria.fr/`) interactive theorem prover [13,36] and to extract actual parallel programs from proofs.

A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its ML type is `'a` **par**. A vector expresses that each of the **p** processors *embeds* a value of any type `'a`. The processors are *labelled* with *ids* from 0 to $\mathbf{p}-1$. The nesting of vectors is not allowed. We use the following notation to describe a vector: $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1}\rangle$. We distinguish a vector from an usual array because the different values, that will be called *local*, are blind from each other; it is only possible to access the local value v_i in two cases: locally, on processor i (using a specific syntax), or after some communications.

Since a BSML program deals with a whole parallel machine and individual processors at the same time, a *distinction* between the 3 *levels of execution* that take place will be needed: (1) **Replicated** execution r is the default; Code that does not involve BSML primitive is run by the parallel machine as it would be by a single processor; Replicated code is executed at the same time by every processor, and leads to the same result everywhere; (2) **Local** execution l is what happens inside parallel vectors, on each of their components; The processor uses its local data to do computation that may be different from the other's; (3) **Global** execution g concerns the set of all processors together as for BSML communication primitives. The distinction between local and replicated is strict: the replicated code cannot depend on local information. If it were to happen, it would lead to replicated inconsistency.

Parallel vectors are handled through the use of different *communication primitives*. Their implementation relies either on MPI or on TCP/IP. Fig. 2 shows their use. Informally, they work as follows: let $\ll$`e`$\gg$ be the vector holding `e` everywhere (on each processor), the $\ll\ \gg$ indicates that we enter a

Primitive	Level	Type	Informal semantics
≪e≫	g	'a **par** (if e:'a)	$\langle \mathsf{e}, \ldots, \mathsf{e} \rangle$
pid	g	int **par**	A predefined vector: i on processor i
`$v$`	l	'a (if v: 'a **par**)	v_i on processor i, assumes $\mathsf{v} \equiv \langle \mathsf{v}_0, \ldots, \mathsf{v}_{\mathbf{p}-1} \rangle$
proj	g	'a **par**→ (int→ 'a)	$\langle x_0, \ldots, x_{p-1} \rangle \mapsto (\textbf{fun } i \rightarrow x_i)$
put	g	(int→ 'a)**par**→ (int→ 'a)**par**	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\textbf{fun } i \rightarrow f_i\ 0), \ldots, (\textbf{fun } i \rightarrow f_i\ (\mathbf{p}-1)) \rangle$

Fig. 2 Summary of the BSML primitives.

vector and switch to the local level. Replicated values are available inside the vectors. Now, only within a vector, to access to local information, we add the syntax `$x$` to read the vector x and get the local value it contains. The *ids* can be accessed with the predefined vector **pid**. For example, using the toplevel for a *simulated* BSP machine with 3 processors:

```
# let vec1 = ≪ "HLPP" ≫ in
            ≪ $vec1$^",␣proc␣"^(string_of_int $pid$) ≫ ;;
— : string par = <"HLPP,␣proc␣0", "HLPP,␣proc␣1", "HLPP,␣proc␣2">
```

The # symbol is the prompt that invites the user to enter an expression to be evaluated. Then, the toplevel gives the evaluated value with its type. Thanks to BSML confluence, it is ensured that the results of the toplevel or of the parallel implementation are identical.

The **proj** primitive is the only way to *extract* local values from a vector. Given a vector, it returns a function such that applied to the *pid* of a processor, the function returns the value of the vector at this processor. **proj** performs communication to make local results available globally and ends the current super-step. For example, if we want to convert a vector into a list, we write:

```
# let list_of_par vec = List.map (proj vec) procs;;
— : val list_of_par : 'a par → 'a list = <fun>
# list_of_par ≪ $pid$ ≫ ;;
— : int list = [0; 1; 2]
```

where procs is the list of *ids* [0; 1; $\cdots$;**p**-1].

The **put** primitive is another communication primitive. It allows any local value to be *transferred* to any other processor. It is also synchronous, and ends the current super-step. The parameter of **put** is a vector that, at each processor, holds a function of type (int→ 'a) returning the data to be sent to processor j when applied to j. The result of **put** is another vector of functions: At a processor j the function, when applied to i, yields the value *received from* processor i by processor j. For example, a total_exchange could be written:

```
# let total_exchange vec =
   | let msg = put ≪ fun dst → $vec$≫ in
   |  | ≪ List.map $msg$ procs ≫ ;;
— : val total_exchange : 'a par → 'a list par = <fun>
# total_exchange ≪ $pid$ ≫ ;;
— : int list par = <[0;1;2], [0;1;2], [0;1;2]>
```

where the BSP cost is $(\mathbf{p}\text{–}1) \times s \times \mathbf{g} + \mathbf{L}$ where s is the size of the biggest sent value.

2.3 The MULTI-BSP Model for Hierarchical Architectures

The MULTI-BSP model [34] is another *bridging model* as the original BSP, but adapted to *clusters of multicores*. The MULTI-BSP model introduces a vision where a *hierarchical architecture* is a *tree* structure of *nested components* (*sub-machines*) where the lowest stage (*leaf*) are processors and every other stage

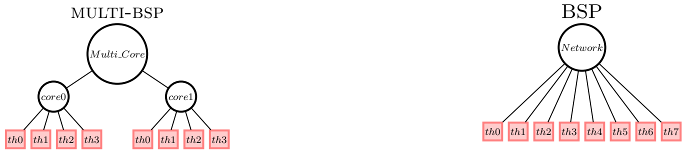


Fig. 4 The difference between the MULTI-BSP and BSP models for a multicore architecture.

(*node*) contains memory. Fig. 4 illustrates the difference between both models for multicores. There exist other hierarchical models [38], such as D-BSP [1] or H-BSP [7], but MULTI-BSP describes them in a simpler way. An instance of MULTI-BSP is defined by **d** the depth of a tree and 4 parameters for each *stage* i:

- $\mathbf{p_i}$ is the number of components inside the i stage. We consider $\mathbf{p}_1$ as a basic computing unit where a step on a word is considered as the unit of time.
- $\mathbf{g_i}$ is the *bandwidth* between stages i and $i+1$: the ratio of the number of operations to the number of words that can be transmitted in a second (illustrated in Fig. 3).
- $\mathbf{L_i}$ is the *synchronisation cost* of all components of $i-1$ stage, but *no* synchronisation across above branches in the tree. Every components can execute codes but they have to synchronise in favour of data exchange. Thus, MULTI-BSP does not allow subgroup synchronisation as the D-BSP does: at a stage i there is only a synchronisation of the sub-components, a synchronisation of each of the computational units that manage the stage $i-1$.

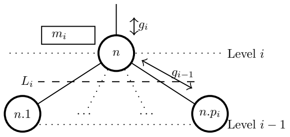


Fig. 3 MULTI-BSP parameters.

- $\mathbf{m_i}$ is the amount of memory available at stage i.

A node executes some codes on its nested components (*aka* "*children*"), then waits for results, do the communication and synchronised the sub-machine.

Considering C_j^i as the communication cost of a super-step i communicating with level j, $C_j^i = h_i \times \mathbf{g}_j + \mathbf{L}_j$. With h_i the message at step i, $\mathbf{g}_j$ communication bandwidth with level j and $\mathbf{L}_j$ the synchronisation cost. We can *recursively* express the cost of a MULTI-BSP algorithm as follow: $\sum_{i=0}^{N-1} w_i + \sum_{j=0}^{d-1} \sum_{i=0}^{M_j-1} C_j^i$ where N is the number of computational super-steps, w_i is the cost of a single computation step and M_j is the number of communication phases at level j.

3 Design of the Multi-ML Language

MULTI-ML is based on the idea of executing a BSML-like code on every stage of the MULTI-BSP architecture, that is on every sub-machine. For this, We add a *specific syntax* to ML in order to code special functions, called *multi-functions*, that recursively go through the MULTI-BSP tree. At each stage, a multi-function allows the execution of any BSML code. We first present the execution model that follows this idea; then, we present the specific syntax and finally, we give the limitations when using some advanced OCAML features.

3.1 Execution Model

A MULTI-ML tree is formed by *nodes* and *leaves* as proposed in MULTI-BSP with the difference that a node is not only a memory but has the ability to *manage* (*coordinate*) the values exchanged by its sub-machines. However, as common architectures do not have dedicated processors for each memory, one (or more, implementation dependent) selected computation unit has the responsibility to perform this task which is little in practice. Because leaves are their own computing units, our approach coincides with MULTI-BSP if we consider that all the computations and memory accesses, at every nodes, are performed by one (or more) leaf: *replicated codes* (outside vectors) that takes place in nodes will be costlier than in leaves. This is the reason why computations on nodes are reserved to the simple task of *coordination.* The MULTI-ML approach is also a bit more relaxed than MULTI-BSP regarding synchronisation. We allow asynchronous codes in the sub-machines when only lower memories accesses are used, unlike MULTI-BSP who forces a synchronisation. Of course, we do synchronise if a communication primitive is used. As suggested in [34], we also allow flat communications between nodes and leaves without using an upper level.

The type *′a tree*, symbolised by ≀*t*≀, stands for a MULTI-ML tree. Every node and leaf contains a value of type *′a* in its own memory. It is important to notice that the values contained in a tree are accessed by the corresponding node (or leaf) only. It is impossible to access the values of another component without using explicit communications. In MULTI-ML codes, we differentiate 4 *strictly separated execution levels*: (1) the level *m* (MULTI-BSP) is the upper one (*outside* trees) and is appropriate to call multi-functions and managing the trees; codes at this level are executed by all the computation units in a SPMD fashion; (2) the level *b* (BSP) is use inside multi-functions and is dedicated to execute BSML codes on nodes; (3) level *l* (local) corresponds to the codes that are run inside vectors; (4) level *s* stands for standard OCAML codes finally executed by the leaves. It is to notice that it is impossible to exchange vectors or trees and, like in BSML, the *nesting of parallelism* (of vectors/trees) is forbidden.

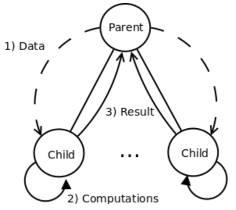


Fig. 5 Code propagation.

The main idea of MULTI-ML is to structure parallel codes to control all the stage of a tree: we generate the parallelism by allowing a node to call recursively a code on each of its sub-machines (children). When leaves are reached, they will execute their own codes and produce values, accessible by the top node using a vector. The data are distributed on the stages (toward leaves) and results are gathered on nodes toward the root node as shown in Fig. 5.

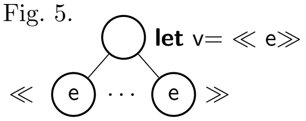


Fig. 6 Vector distribution.

Let us consider a code where, on a node, the following code is executed: ≪e≫. As shown in Fig. 6, the node creates a vector containing, for each sub-machine *i*, the expression e. As the code is run asynchronously, the execution of the node code will continue until reaching a barrier.

3.2 The MULTI-ML Language

Fig. 9 shows the MULTI-ML primitives (without recall the BSML ones); their authorised level of execution and their informal semantics.

Primitive	Level	Type	Informal semantics
§e§	m	'a tree	Build ≀e≀, a tree of e
$t$	s	'a	In a §e§ code, t_n on node/leaf n of the tree ≀t≀
v (if v: 'a tree)	b	'a	v_n on node n of tree ≀v≀,
$v$	l	'a	In the ith component of a vector, $v_{n.i}$ on node/leaf n of the tree ≀v≀
gid	m	id	The predefined tree of nodes and leaves ids
≪...f...≫	l	'a	In a component of a vector, recursive call of the multi-function
#x#	l	'a	In a component of a vector, reading the value x at upper stage (id)
mkpar f	b	'a **par**	$\langle v_0, \ldots, v_{p_n} \rangle$, where $\forall i$, f i $= v_i$, at id n of the tree
finally v_1 v_2	b,s	'a	Return value v_1 to upper stage (id) and keep v_2 in the tree
this	b,l,s	'a option	Current value of the tree if exists, None otherwise

Fig. 9 Summary of the MULTI-ML primitives.

n denotes the id of a node/leaf, *i.e.* its position in the tree encoded by the top-down path of positions in node's vectors. For example, 0 stands for the root, 0.0 for its first child, *etc.* For the ith component of a vector at node n, the id is $n.i$. Fig. 7 illustrates this naming. We now describe in details those primitives and let-multi functions.

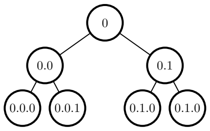


Fig. 7 Node identifiers.

The let-multi Construction. The goal is to define a multi-function *i.e.* a recursive function over the MULTI-BSP tree. Except for the tree's root, when the code ends on a stage i, the values are made available to the upper stage $i-1$ in a vector. The let-multi construction offers a syntax to declare codes for two levels, one for the nodes (level b) and one for the leaves (level s):

```
let multi f [args] =
  where node = ... (* BSML code *)
  where leaf = ... (* OCAML code *)
```

[args] are the arguments of the newly define multi-function f. In the *leaf block* (*i.e.* level s), we find usual ML computations and most of the calculation need to take place here. In the *node block* (*i.e.* level b), we find the code that will be executed at every stage of the tree, but on the leaves. Typically, a node is charged to *propagate* the computations and the data using vectors (level l); to *manage* the sub-machine computations; and finally, *gather* the results using the **proj** — extraction of values from a vector. To go one step deeper on the tree, the node code must *recursively call* the multi-function inside a vector. This call *must* be done inside a vector in order to spread the computation all over the tree in the deeper stages. It is also to notice that a multi-function can only be called at m level of the code and values at this level are available throughout the multi-function execution. Fig. 8 shows, as an example of how data moves through the MULTI-BSP tree, a simple program summing of the elements of a list.

```
1 let multi par_fold l =
2   | where node =
3   |   | let v=mkpar (fun i→split i l) in
4   |   | let res=≪par_fold $v$≫  in
5   |   |   | sum (flatten res)
6   | where leaf =
7   |   | List.fold_left (fun x y→x+y) 0 l
8 (* flatten:'a par→'a list *)
9 (* sum: int list→int *)
```

Fig. 8 Sum MULTI-ML example.

It works as follows: We define the multi-function (line 1); Lines $2-5$ give the code for the nodes and lines $6-7$ give the code for the leaves; The list is scattered across each component i of the vector (line 3); We recursively call the multi-function on the sublists (line 4); We finally gather the results in line 5 (sum, a BSML code, performs a **proj** to sum the $\mathbf{p}_n$ projected integers).

Tree Construction. It is not fundamentally different of the above multi-functions: Unlike generating a single usual ML value only, the "let multi" builds a *tree* ≀t≀ of type *'a tree*. For this, a new constructor determines the values that are stored (*keep*) at each id and those that are ascended (*up*) to the upper stage —until now, the value returned by nodes and leaves was implicitly considered as the value given to the upper stage. This is the role of **finally** which works as follows: **finally** ~**up**:v1 ~**keep**:v2 to up the value v1 and store to the tree the value v2 (having first replaced the previous stored value in the tree). The primitive **this** returns the last value stored using **finally** or None if it has never been called at this point. It is useful to update the tree.

```
 1 let par_scan_list op li =
 2    let multi m_scan flag l =
 3       where node =
 4          if flag then
 5             let spl=mkpar (fun i→split i l) in
 6             let deep=bsp_scan op ≪m_scan true $spl$≫ in
 7             let v=last ≪if $pid$≠0
 8                              then (m_scan false [$deep$])
 9                              else noSome $this$≫  in
10             finally ~up:v ~keep:[v]
11          else
12             let v=last ≪m_scan false #l#≫  in
13                finally ~up:v ~keep:[v]
14       where leaf =
15          let final,l'= if flag then (seq_scan op l)
16             else (map_and_last (op (List.hd l)) this) in
17          finally ~up:final ~keep:l'
18  in m_scan true li
19 (* bsp_scan:'a par→('a→'a→'a)→'a par ⇒ BSML scan *)
20 (* last:'a par→'a ⇒ gives the last element of a vector *)
21 (* seq_scan:'a list→('a→'a→'a)→'a*'a list
22    ⇒ computes the scan and also returns the last element *)
23 (* map_and_last:'a list→('a→'a)→'a*'a list
24     ⇒ do a map and also returns the last element *)
```

Fig. 10 Scan MULTI-ML example.

Fig. 10 shows the modifications that has to be added to sum_list in order to obtain a tree containing the list of partial sums on every leaves and the maximal sub-sums on each node —thus the final sum on the root node. The code works as follows: We use a generic operator and a list to be distributed (line 1); then we traverse twice (distinguished with a boolean flag) the MULTI-BSP machine, first to split the list and then to compute the partial sums; for the nodes, we split the list (line 5) and then we do a *recursive call* over the scattered lists to continue the splitting at lower levels and then recover the partial sums of children nodes (line 6), BSP scan is used on this partial results to transfer and compose partial results from sibling nodes left to right (bsp_scan could be any BSML scan code, direct or logarithmic); finally, we do a recursive call again in a vector (lines $7-9$) to complete the partial sums with the communicated values and for this, we transmit down a list containing only the last value for each branch, keep it in the tree and give it to the upper level (line 10); when reaching the leaves the first time, we compute the partial sums (line 15) and, each time a value (communicated by other branches) comes down to a leaf, we add it to its own partial sums (line 16). Note that using two multi-functions, one to first split the list and another one to compute the partial sums, is surely easier, but using a one-shot multi-function, we exhibit more features of MULTI-ML.

Variables Accesses. There are three different ways to access to variables in addition to the usual ML access. First, v stands for reading the value of a vector "v" inside a vector (l level, as in BSML) but also for reading a previously defined tree "v": if the vector is constructed at id n, then at component i, v stands for reading the value of "v" at id $n.i$ —stage $i-1$. Within the tree construction §e§, v stands for reading the value of "v" at id n. For short, v stands for reading one of the component of "v".

The second way is to read a value *inside* a vector which had been declared *outside*. As explained above, the values available on a node are not implicitly available on the child nodes. It is thus necessary to copy them from a node to its sub-machines. For example, the code **let** x=1 **in let** vect=≪x+1≫ is incorrect because "x" is not available on children. It is imperative to use ≪#x#+1≫ to copy the value of x from the b level into the vector (l level).

Outside a vector (g level), when the code is computing at id n, accessing to a tree t is *implicitly* reading its value at id n. Finally, **gid** is the predefined tree of nodes/leaves ids. Within the component i of a vector constructed on node n (l level), $**gid**$ stands for the id $n.i$ whereas inside a §e§ code, it stands for the identifier of the node evaluating e; at evaluation level g, **gid** stands for n (if executed at id n) and at evaluation level m, it is the tree of node identifiers.

A Convenient Tree Construction. For *building* a tree *without using* a multi-function (which induce communications), we add the §e§ syntax. It allows to execute e on every nodes and leaves. One can read the values of a previously defined tree ≀t≀ using the t access in the code of e. In this way, using the predefined tree **gid**, we can execute *different* codes on each components of a tree without any need (and possibility) of communication between the stages.

A New Primitive. For *performances* reason, we chose to add the new primitive mkpar. Indeed, in BSML, a replicated code is duplicated on every processors so it is *not* necessary to take care of data transfer in code like: **let** lst=[...] **in** ≪split $**pid**$ lst≫ where lst is a large list and split a splitting function. With the MULTI-ML model, data are not distributed everywhere and we have to transfer data explicitly. One can write ≪split $**pid**$ #lst#≫ but it is not useful to copy the whole list on every children in order to extract a sub list and throw the rest. This is the reason why the **mkpar** compute first p_n values and then *distribute* them to the sub-machines thus building a vector. This method is more expensive for the node n in computation time, but it reduces drastically the amount of data transfers.

3.3 Current Limitations

Exceptions and Partially Evaluated Trees. Exceptional situations are handled in OCAML with a system of *exceptions*. In parallel, this is at least as relevant: if one processor triggers an exception during a computation, BSML [14] as well as MULTI-ML have to deal with it, like OCAML would, and prevent a crash.

The problem is when an exception is raised *locally* (l level) on (at least) one processor but other processors continue to follow the execution stream, until they are stopped by the need of synchronisation. This happens when an exception *escapes* the scope of a parallel vector. Then, a crash can occur: a processor misses the global barrier. To prevent this situation, in [14], we introduce a *specific handler* of local exceptions that have not been caught locally. The structure **trypar**...**withpar** catches any exception and handles it as usual in OCAML. To do this, a barrier occurs and exceptions are communicated to all processors in order to allow a global decision to be taken. Furthermore, any access to a vector that is in an incoherent state will raise again the exception.

For MULTI-ML, if an exception is no correctly handled in a stage, it must be propagated to the upper stage at the next barrier —as in BSML. If an exception

$$\boxed{\text{Core-ML inductive rules } \Downarrow_\delta}$$

$$\frac{}{\mathcal{E} \vdash \mathbf{cst} \Downarrow_\delta \mathbf{cst}} \quad \frac{}{\mathcal{E} \vdash \mathbf{op} \Downarrow_\delta \mathbf{op}} \quad \frac{\{x \mapsto v\} \in \|\mathcal{E}\|_\delta}{\mathcal{E} \vdash x \Downarrow_\delta v} \quad \frac{}{\mathcal{E} \vdash (\mathbf{fun}\, x \to e) \Downarrow_\delta \overline{(\mathbf{fun}\, x \to e)[\mathcal{E}]}}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \overline{(\mathbf{fun}\ x\ \to\ e)[\mathcal{E}']} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v \quad \mathcal{E}' \oplus_\delta \{x \mapsto v\} \vdash e \Downarrow_\delta v'}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow_\delta v'} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta v_1 \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v_2}{\mathcal{E} \vdash (e_1, e_2) \Downarrow_\delta (v_1, v_2)}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{op} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v \quad \overline{(\mathbf{op}\ v)} \equiv v'}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow_\delta v'} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta v \quad \mathcal{E} \oplus_\delta \{x \mapsto v\} \vdash e_2 \Downarrow_\delta v'}{\mathcal{E} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow_\delta v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{true} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v_2}{\mathcal{E} \vdash \mathbf{if}\, e_1 \mathbf{then}\, e_2 \mathbf{else}\, e_3 \Downarrow_\delta v_2} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{false} \quad \mathcal{E} \vdash e_3 \Downarrow_\delta v_3}{\mathcal{E} \vdash \mathbf{if}\, e_1 \mathbf{then}\, e_2 \mathbf{else}\, e_3 \Downarrow_\delta v_3} \quad \frac{\mathcal{E} \oplus_\delta \{f \mapsto (\mathbf{rec}\ f \to e)\} \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash (\mathbf{rec}\ f \to e) \Downarrow_\delta v}$$

$$\boxed{\text{BSML-like primitives inductive rules } \Downarrow_n^b}$$

$$\frac{\forall i \in \{1, \ldots, \mathbf{p}_n\} \quad \mathcal{E} \vdash e \Downarrow_{n.i}^l v_i}{\mathcal{E} \vdash < e > \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle}{\mathcal{E} \vdash \mathbf{proj}\ e \Downarrow_n^b \overline{(\mathbf{fun}\ i \to v_i)}} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b \langle \overline{f_1}, \ldots, \overline{f_{\mathbf{p}_n}} \rangle}{\mathcal{E} \vdash \mathbf{put}\ e \Downarrow_n^b \langle \overline{f'_1}, \ldots, \overline{f'_{\mathbf{p}_n}} \rangle}$$

$$\frac{\{x \mapsto \langle v_1, \ldots, v_i, \ldots, v_{\mathbf{p}_n} \rangle\} \in \|\mathcal{E}\|_n}{\mathcal{E} \vdash \$x\$ \Downarrow_{n.i}^l v_i} \quad \frac{\{x \mapsto v\} \in \|\mathcal{E}\|_n}{\mathcal{E} \vdash \#x\# \Downarrow_{n.i}^l v} \quad \frac{}{\mathcal{E} \vdash \$\mathbf{pid}\$ \Downarrow_{n.i}^l i}$$

$$\frac{\|\mathcal{E}\|_n \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash e \Downarrow_n^{b,l} v} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b f \quad \forall i \in \{1, \ldots, \mathbf{p}_n\} \quad \mathcal{E} \vdash (f\ i) \Downarrow_n^b v_i \text{ with } f \equiv \overline{(\mathbf{fun}\ x \to e')[\mathcal{E}']}}{\mathcal{E} \vdash \mathbf{mkpar}\ e \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle}$$

$$\boxed{\text{Multi functions inductive rules } \Downarrow^m}$$

$$\frac{}{\mathcal{E} \vdash (\mathbf{multi}\, f\, x \to e_1 \to e_2) \Downarrow^m \overline{(\mathbf{multi}\, f\, x \to e_1 \to e_2)[\mathcal{E}]}} \quad \frac{}{\mathcal{E} \vdash \mathbf{gid} \Downarrow_n^{b,l} n}$$

In what follows $g \equiv \overline{(\mathbf{multi}\ f\ x \to e'_1 \to e'_2)[\mathcal{E}']}$

$$\frac{\mathcal{E} \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash e \Downarrow^m v} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow^m g \quad \mathcal{E} \vdash e_2 \Downarrow^m v \quad \mathcal{E}' \oplus_0 \{x \mapsto v\} \oplus_0 \{f \to g\} \vdash e'_1 \Downarrow_0^b v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow^m v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_{n.i}^l g \quad \mathcal{E} \vdash e_2 \Downarrow_{n.i}^l v \quad \mathcal{E}' \oplus_{n.i} \{x \mapsto v\} \oplus_{n.i} \{f \to g\} \vdash e'_1 \Downarrow_{n.i}^b v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow_{n.i}^l v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_{n.i}^l g \quad \mathcal{E} \vdash e_2 \Downarrow_{n.i}^l v \quad \mathcal{E}' \oplus_{n.i} \{x \mapsto v\} \oplus_{n.i} \{f \to g\} \vdash e'_2 \Downarrow_{n.i}^s v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow_{n.i}^l v'}$$

Fig. 11 Operational big-step semantics rules of core-MULTI-ML.

is no handled in a multi-function, it must be thrown at the global level m as a standard OCAML exception. An exception thrown in a node of a tree leads this node in an incoherent state until the exception has been caught in a upper level. Any access to this tree must raise again the exception. This handling has not been yet implemented for MULTI-ML but the first author works on it.

An application case is *partially evaluated trees*. Take for example the following code: ≪**if** random() **then** f **else** 0≫ where f is a multi-function. A part of the tree will never be instantiated. If a partially evaluated tree is accessed during a code execution an exception could be immediately throw.

Type System. The main limitation of our prototype is the lack of a type system. Currently, *nesting* of BSML vectors/trees are not checked. A type system for BSML exists [14] but has not been implemented yet. We are convinced that adding multi-functions will not fundamentally change the type system: it's mainly a matter of adding just a new level of execution.

Other ML *Features.* We have not yet studied all the interactions of all the OCAML features with the multi-function (as well in BSML). Mainly: objects, first-order modules and GADT. We let them for future works.

4 Semantics, Implementation and Examples

We present a formal semantics of MULTI-ML as well as two implementations. To get the *assurance* that both implementations are *coherent*, using the semantics, we first prove that MULTI-ML is confluent. A semantics is useful as a *specification* of the language so as to simplify the design of the implementations. We give some examples and benchmarks to illustrates the usefulness of MULTI-ML. Our prototype is freely available at `http://lacl.fr/allombert/multi-ml.html`.

4.1 Operational Semantics

We give a *big-step* semantics of a core-language —without tree creation to simplify the presentation. The syntax (Fig. 12) extends that of popular core-ML.

$$
\begin{array}{l}
e ::= \ /\text{* core-ML *}/ \\
\quad \mid x \mid \mathbf{cst} \mid \mathbf{op} \mid (e,e) \mid \mathbf{let}\ x = e\ \mathbf{in}\ e \mid (e\ e) \\
\quad \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid (\mathbf{fun}\ x \to e) \mid (\mathbf{rec}\ f \to e) \\
\quad /\text{* BSML-like primitives *}/ \\
\quad \mid \$x\$ \mid \#x\# \mid {<}e{>} \ \mid \mathbf{pid} \\
\quad \mid \mathbf{mkpar}\ e \mid \mathbf{gid} \mid \mathbf{proj}\ e \mid \mathbf{put}\ e \\
\quad /\text{* multi-fun, without tree construction *}/ \\
\quad \mid (\mathbf{multi}\ f\ x \to e \to e)
\end{array}
$$

Fig. 12 Syntax of core-MULTI-ML.

Programs contain variables, constants (integers, *etc.*), operators ($\leq$, $+$, *etc.*), pairing, **let**, **if**, **fun** statements as usual in ML, **rec** for recursive calls, the BSML primitives ($<e>$, **put**, **proj**), **mkpar**, access $\$x\$$ to the local value of a vector x, local copy $\#x\#$ of a parent's variable x, the vector of **pid** component and **gid** the tree of ids. Finally, we define let-multi as particular functions with codes for nodes and leaves.

The semantics is a big-step one with *environments* that represent the *memories*. We have a tree of memories (one per node and leaf) and we note them $\mathcal{E}$. $\|\mathcal{E}\|_n$ denotes the memory of $\mathcal{E}$ at n where n is the id of the node/leaf. $\{x \mapsto v\}$ denotes a binding of a variable x to a value v in the memory; $\in$ denotes a membership test and $\oplus$ denotes an update. Those operators have the subscript n that denotes the application in a specific memory.

$\mathcal{E} \vdash e \Downarrow v$ denotes the *evaluation* of e in the environment $\mathcal{E}$ to the value v. A value is a constant, an operator or a functional *closure* (noted $(\mathbf{fun}\, x \to e)[\mathcal{E}]$) that is a function with its own environment of execution. The rules of evaluation are define by *induction* and fully given in Fig. 11. Note that to simplify the reading of the rules, we count vector pids from 1 to $\mathbf{p}_n$ and not from 0 to $\mathbf{p}_n{-}1$.

Even if the semantics contains many rules, there is no surprise and it has to be read naturally. As explain before, there are 4 different levels of execution: (1) level m for the execution on all computation units; (2) BSP level b for BSML codes. These rules are subscripted with the id n of the sub-machine like for memories; (3) local level l for the codes inside a vector; (4) and finally level s on the leaves. In this way, the evaluation $\Downarrow$ is upscripted by the level. Note that a code at level l becomes at level b if a recursive call of a multi-function occurs.

The first rules are for the core-ML part and are as intended: a value returns a value; an application needs a closure (or an operator), a value and then the core of the closure is used with the properly modified environment; pairing builds two values; a "let" adds a new binding. They are used to evaluate any expression that do only contains ML code (no MULTI-ML primitives)

The rules for the BSML primitives work as follows: creating a new vector for the machine of id n is triggering $\mathbf{p}_n$ local evaluations, each with $n.i$ as subscript since we are going one step deeper, in the ith component; **proj** and **put** rules build the functions of exchanges; $\$x\$$ rule read at $n.i$ the the ith value of the vector $x$ created by node n; $\#x\#$ rule read the value $x$ at the node n from its child; $pid rule returns i on child $n.i$; **mkpar** rule creates the vector but first the node creates the values to be sent down.

For the multi-functions, we have a rule to create the function and a rule to initialise the deeper computations. In this way, at level m, we start to recurse down in the machine from id 0 with the appropriate environment and level g. Then, inside the component i of a vector of sub-machine n, the recursive call of the multi-function generates an evaluation on $n.i$. Except if we reach a leaf, then the last rule says that the code is evaluated on leaf $n.i$ with level s. We can then prove by induction the following lemma:

Lemma 1 (Confluence) $\forall \mathcal{E}$ *if* $\mathcal{E} \vdash e \Downarrow^m v_1$ *and* $\mathcal{E} \vdash e \Downarrow^m v_2$ *then* $v_1 \equiv v_2$

Co-inductive rules [25] $\Downarrow_\infty$ (for diverging programs) can be easily infered from the above rules. For sake of conciseness, we only present some typical examples:

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \textbf{true} \quad \mathcal{E} \vdash e_2 \Downarrow_\infty}{\mathcal{E} \vdash \textbf{if}\, e_1 \textbf{then}\, e_2 \textbf{else}\, e_3 \Downarrow_\infty} \qquad \frac{\exists i \in \{1, \ldots, \mathbf{p}_t\} \quad \mathcal{E} \vdash e \Downarrow_\infty}{\mathcal{E} \vdash < e > \Downarrow_\infty} \qquad \frac{\mathcal{E} \vdash e \Downarrow_\infty}{\mathcal{E} \vdash \textbf{proj}\ e \Downarrow_\infty}$$

We can then prove by co-induction the following lemma:

Lemma 2 (Mutually exclusive) $\forall \mathcal{E}$ *if* $\mathcal{E} \vdash e \Downarrow^m v$ *then* $\neg(\mathcal{E} \vdash e \Downarrow^m_\infty)$

Results do not depend of the order of evaluation of the processors nor of the BSP sub-machines. All strategy work and return the same values, especially a sequential simulation and a distributed implementation. The former is fine for debugging whereas the latter is for benchmarking. We now present both.

4.2 Sequential Simulation and Distributed Implementation

Sequential Simulation. We propose a sequential simulator that works as the OCAML toplevel. Given an architecture as a configuration file, the toplevel allows simulating MULTI-ML codes on a single core machine and printing the results. Currently, all the main features of OCAML are available, without typing. To be executed, the MULTI-ML code is converted into a sequential code using a modified OCAML parser. The simulator creates a tree structure to represent the whole MULTI-BSP architecture. Vectors are represented as arrays of data. A global hashtable is used to simulate the memory available at each levels as suggested by the semantic. Fig. 13 shows the result when using the toplevel for a simulated MULTI-BSP machine composed of 2 cores with respectively 2 threads.

```
#let multi f n =
 | where node =
 |  | let _=≪f $pid$≫ in
 |  |  | finally ˜up:() ˜keep:(gid^"=>"^n)
 | where leaf=finally ˜up:() ˜keep:(gid^"=>"^n);;
— : val f : int→string tree = <multi−fun>
#(f 0)
o "0→ 0"
|
o "0.0→ 1"
|  |  → "0.0.0→ 3"
|  |  → "0.0.1→ 4"
o "0.1→ 2"
|  |  → "0.1.0→ 5"
|  |  → "0.1.1→ 6"
```

Fig. 13 Example of the Toplevel.

Distributed Implementation. To be portable, our implementation is written to use various communication libraries. We have thus a modular approach and our implementation depends of a generic functors that requires the architecture configuration and some particular communication routines, mainly, asynchronous broadcasting and gathering values for a group of processors, total echange, and builds groups of processes. Our first implementation of this module is currently based on MPI. We create one MPI process for every nodes and leaves of the architecture. Those processes are distributed over the physical cores and threads in order to balance the charge. As the code executed by nodes is, most of the time, a simple task of parallel management, this extra job is thus distributed over the leaves to reduce its impact.

Our implementation is based on a daemon running on each MPI processes. Each daemon have 3 MPI communicators to communicate with their father (upper node), their sons (leafs) and their brothers (processes at the same sub-level).These daemon are waiting for a task given by their father. When a task is received, it is executed by the daemon, and then, they returns to the waiting state until they receive a ”kill” signal, corresponding to the end of the program.

As the code is a SPMD one, every processes know the entire code and they just need to receive a signal to execute a task. To do so, and to avoid serialising codes that are inside functional values (the closures) and know by all the nodes due to a SPMD execution, when transmitting down values and thus creating parallel vectors, we identify the vectors by two kinds of identifiers: (1) a static identifier is generated globally for every vectors and reference their computations through the execution; (2) when a node need to create a parallel vector, it generates a dynamic identifier that represent the data on its leaves. Then, when a node execute some code using parallel values inside a vector, it just sends the static identifier (that references the code to execute) with the dynamic identifier (to substitute the distributed value) to its sons which can then execute the correct tasks. The main advantage of this method it to avoid serialising unnecessary codes when creating vectors, and thus reduces the size of the data exchanged by the processes. But, associating a value by a dynamic identifier can lead to a memory over-consumption, for example in loops. When the life cycle of a vector is terminated, we manually clean the memory by removing all the obsolete identifier and calling the garbage collector.

Shared memory. We propose an implementation to avoid some unnecessary copies of the transmitted data. Indeed, the OCAML memory is garbage collected and, to be safe, only a copy of the data can me transmitted. Using the standard POSIX “mmap” routine and some IPC functionalities (to synchronise the processes), the child (as daemons) can read asynchronously the transmitted serialised value in the mapping of the virtual address space of the father and synchronise with the father only, as the MULTI-BSP model suggest. As architectures can have different types of memory (distributed or shared), it is possible to mix executions schemes. Since the OCAML memory is garbage collected (currently with a global thread lock), we sadly cannot use pthreads as done in [35] to share values without performing first a copy. We are currently working on using some tricks to overcome this limitation but we leave it as future work.

4.3 Benchmarks

In this section we present the benchmark of a simple scan with integer addition and a naive implementation of the sieve of Eratosthenes. A scan can be used to perform the sieve of Eratosthenes using a particular operator which implies more computations and communications than a simple list summing.

The MULTI-BSP cost of the scan algorithm is as follows:

$$\sum_{i\in[0\dots\mathbf{d}[} V_{sp}(i) + \mathcal{O}(|l_{\mathbf{d}}|) + \sum_{i\in[0\dots\mathbf{d}[} C_i.$$

Where $\sum_{i\in[0\dots\mathbf{d}[} V_{sp}(i)$ is the total cost of splitting the list until the leaves (at depth $\mathbf{d}-1$); $\mathcal{O}(|l_{\mathbf{d}}|)$ is the time needed to compute the local sums in the leaves; and $\sum_{i\in[0\dots\mathbf{d}[} C_i$ corresponds to the cost of diffusing partial sum back to leaves and to add these values to values held by leaves. This diffusion is done once per node. $V_{sp}(i)$ is the work done at level i to split the locally held chunk l_i and scatter it among children nodes. Splitting l_i in $\mathbf{p}_i$ chunks costs $\mathcal{O}(|l_i|)$ where $|l_i|$, the size of l_i, is $n * \prod_{j\in[0\dots i[} \frac{1}{\mathbf{p_j}}$ where n is the size of the initial list holds by the root node. Scattering it among children nodes costs $\mathbf{p_i} * \mathbf{g_{i-1}} + \frac{n_i}{\mathbf{p_i}} + \mathbf{L_i}$. The sequential list scan cost at leaves is $\mathcal{O}(|l_d|) = \mathcal{O}(n * \prod_{i\in[0\dots\mathbf{d}[} \frac{1}{\mathbf{p_i}})$.

The cost C_i at level i is the cost of a BSP scan, of a diffusion of the computed values until leaves and the sequential cost of a map on list held by leaves. Let s be the size of the partial sum, the cost of BSP scan at level i is $s * \mathbf{p_i} * \mathbf{g_{i-1}} + \mathbf{L_i}$, the diffusion cost is $\sum_{j\in]i\dots\mathbf{d}]} \mathbf{g_j} * s + l_j$ and the final map is cost $\mathcal{O}(s_{\mathbf{d}})$. The size s may be difficult to evaluate: for a sum of integers it will simply be the size of an integer, but for Eratosthenes sieve, the size of exchanged lists varies depending on which data are held by the node.

```
let scan_direct op vv =
 let mkmsg pid v dst =
   if dst<pid then None else Some v in
 let procs_lists =
   ≪ fun pid → from_to 0 pid≫  in
 let receivedmsgs =
   put(apply(mkpar mkmsg) vv) in
 let values_lists =
   ≪ List.map ((compose noSome)
          $receivedmsgs$) $procs_lists$ ≫
in
 ≪ (fun (h::t)→ List.fold_left op h t )
      $values_lists$ ≫
```

Fig. 14 BSML direct scan code.

The sieve of Eratosthenes generates a list of primary numbers below a given integer n. From the list of all elements lesser than n, it iteratively removes elements that are a multiple of the smaller element of the list that as not been yet considered. We generate only the integers that are not multiple of the 4 first prime numbers, then we iterate $\sqrt{n}$ time (as it is known to be the maximum number of needed iteration). On our architectures, the direct and logarithmic scans are as efficient. Fig. 14 gives the BSML code of the direct scan. This code build the list of elements to communicate to its neighbours and exchange the values using the **put** primitive. Then every processes maps the received values on their own data. We used the following functions: elim:int list→ int→ int list which deletes from a list all the integers multiple of the given parameter; final_elim:int list→ int list→ int list iterates elim using elements from the first list to delete elements in the second; seq_generate:int → int→ int list which returns the list of integers between 2 bounds; and select:int → int list→ int list which gives the $\sqrt{n}$th first prime numbers of a list.

For this naive example, we use a generic scan computation with final_elim as the ⊕ operator. In our computation, we also did extend the scan so that the sent values are first modified by a given function (select) to just sent the $\sqrt{n}$th first prime numbers. The BSP methods is thus simple: each processor i

$p_0 = 4$	$g_0 = \infty$	$L_0 = 149000$	$m_0 = 0$	$p_0 = 8$	$g_0 = \infty$	$L_0 = 195400$	$m_0 = 0$
$p_1 = 2$	$g_1 = 89$	$L_1 = 1100$	$m_1 = 64Gb$	$p_1 = 2$	$g_1 = 14$	$L_1 = 472$	$m_1 = 16Gb$
$p_2 = 16$	$g_2 = 6$	$L_2 = 1800$	$m_2 = 20Mb$	$p_2 = 4$	$g_2 = 6$	$L_2 = 800$	$m_2 = 6Mb$
$p_3 = 0$	$g_3 = 3$	$L_3 = 0$	$m_3 = 0$	$p_2 = 0$	$g_2 = 5$	$L_2 = 0$	$m_2 = 0$

Fig. 15 Multi-BSP parameters of Mirev3 (left) and Mirev2 (right).

holds the integers between $i \times \frac{n}{\mathbf{p}} + 1$ and $(i+1) \times \frac{n}{\mathbf{p}}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is applied. We then eliminate on processor i the integers that are multiple of integers received from processors of lower identifier.

Benchmark were done on two parallel architectures named Mirev2 and Mirev3. Here are the main specifications of these machines:

- Mirev2: 8 nodes with 2 quad-core (AMD 2376) at $2.3Ghz$ with $16Gb$ of memory per node and a $1Gb/s$ network.
- Mirev3: 4 nodes with 2 hyper-threaded octo-core (16 threads) (Intel XEON $E5{-}2650$) at $2.6Ghz$ with $64Gb$ of memory per node and a $10Gb/s$ network.

The MULTI-BSP and BSP model can be used to estimate the cost of an algorithm. Thus, we estimated the cost of transferring values and the sequential cost of summing lists of integer. Then, we used the BSP parameters given in Fig. 15 in order to predict and compare the execution time of scan. The **g** and **L** BSP parameters are given in Fig.19, one can notice that until 64 cores **g** evolves linearly, but after that, too many cores access the network, producing a bottleneck and severely hindering performances.

We measured the time to compute the sieve without the time to generate the initial list of values. The experiments have been done on Mirev2 and Mirev3 using BSML (MPI version) and MULTI-ML implementations over lists of increasing size on an increasing number of processors. The processes have been assigned to machines in order to scatter as much as possible the computation among the machines, *i.e.* a 16 process run will uses one core on each processor of the 8 machines of Mirev3. Fig. 16 and 17 shows the results of our experimentations. We can see that the efficiency on small list is poor but as the list grows, MULTI-ML exceeds BSML. This difference is due to the fact that BSML communicates through the network at every super steps; while MULTI-ML focusing on communications through local memories and finally communicates through the distributed level.

Fig. 18 gives the computation time of the simple scan using a summing operator. We can see that MULTI-ML introduce a small overhead due to the level management; however it is as efficient as BSML and concord to the estimated execution times.

As the experiment shows, MULTI-ML out-performs BSML when the MULTI-BSP model allows to confine heavy communications in a node before communicating with other nodes. For the sum of integers, where the MULTI-BSP model cannot bring improvements, we can see that the additional managing cost of MULTI-ML is negligible.

	100_000		500_000		1_000_000		3_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	22.4	105.0	125.3	430.7	...	...
16	0.5	0.8	13.3	50.3	68.1	331.5	1200.0	...
32	0.3	0.5	2.6	18.9	11.3	122.2	173.2	...
48	0.5	0.4	1.75	14.5	5.5	88.4	69.3	...
64	0.3	0.3	1.3	8.7	4.1	56.1	51.1	749.9
96	0.3	0.38	1.6	6.3	3.9	30.8	38.1	576.1
128	0.5	0.45	2.1	5.2	4.7	24.3	30.6	443.7

Fig. 16 Execution time of Eratosthenes using MULTI-ML and BSML on Mirev3.

	100_000		500_000		1_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	1.5	1.7	64.5	106.1	402.9	1538.1
16	0.45	0.93	16.0	49.3	91.4	631.7
32	0.14	0.45	4.1	18.7	21.1	219.7
48	0.13	0.40	2.6	11.0	10.8	123.5
64	0.11	0.34	1.89	7.5	8.2	80.5

Fig. 17 Execution time of Eratosthenes using MULTI-ML and BSML on Mirev2.

	5_000_000			
	MULTI-ML	BSML	*Pred*_MULTI-ML	*Pred*_BSML
8	2.91	2.8	3.44	1.83
16	1.42	1.4	1.72	0.92
32	0.92	0.73	0.43	0.46
48	0.84	0.75	0.28	0.31
64	0.83	0.74	0.21	0.23

Fig. 18 Execution time and predictions of scan (sum of integers) on Mirev3.

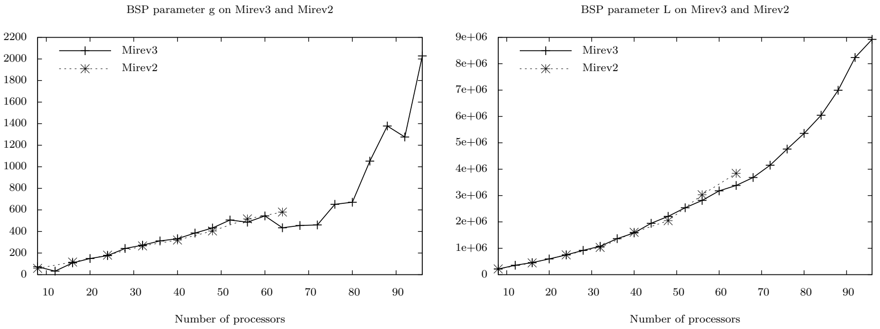


Fig. 19 **g** and **L** BSP parameters of Mirev2 and Mirev3 in flops.

5 Related Work

There are a lot parallel languages or parallel extensions of a sequential language (functional, iterative, object-oriented, *etc.*). It would be too long to list all of them. We chose to point out those that were the most important to our mind. Notice that, except in [26], there is a lack of comparisons between parallel languages. It is difficult to compare them since many parameters have to be taken into account: efficiency, scalability, expressiveness, *etc.*

5.1 Programming Languages and Libraries for BSP like Computing

Historically, the first library for BSP computing was the BSPLIB [21] for the C language; it has been extended in the PUB library [4] by adding subgroup synchronisations, high performance operations and migration of threads. For the GPU architectures, a BSP library was provided in [23] with mainly DRMA primitives close to the BSPLIB's ones.

For JAVA, different libraries exist. The first one is [17]. There is also BSP-CORE [35] (also for C++). A library with scheduling and migration of BSP threads has been designed in [29] —the scheduling is implicit but the migration can be explicit. The library HAMA [32] is implemented using a "MAPREDUCE-like" framework. We can also highlight the work of NESTSTEP [24] which is C/JAVA library for BSP computing, which authorises nested computations in case of a cluster of multi-core but without any safety.

The BSML primitives were adapted for C++ [18]: the BSP++ library provides nested computation in the case of a cluster of multi-cores (MPI+OPEN-MP). But it is the responsibility of the programmer to avoid harmful nested parallelism. BSML also inspired BSP-PYTHON [22] and BSP-HASKELL [28].

5.2 Functional Parallel Programming

A survey to parallel functional programming can be found in [20]. It has been used as a basis for the following classification with some updates.

Data-parallel Languages. The first functional one was NESL [3]. This language allows to create particular arrays and nested computations within these arrays. The abstract machine is responsible for the distribution of the data over the available processors. For ML, there is MANTICORE [12], an extension of NESL with the dynamic creation of asynchronous threads and send/received primitives. For GPU architectures, an extension of OCAML, using a special syntax for the kernels has been developed in [5].

SAC (Single Assignment C) [16] is a lazy language (with a syntax close to C) for array processing. Some higher-order operations on multi-dimensional arrays are provided and the compiler is responsible for generating an efficient parallel code. A data-parallel extension of HASKELL has been done in [10] where the language allows to create data arrays that are distributed across the processors. And some specific operations permit to manipulate them.

The main drawback of all these languages is that cost analysis is hard to do since the system is responsible for the data distribution.

Explicit process creation. We found two extensions of HASKELL in this category: EDEN and GPH [31]. Both use a small set of syntactic constructs for explicit process creation. Their fine-grain parallelism, while providing enough control to implement parallel algorithms efficiently, frees the programmer from the tedious task of managing low-level details of communications —which uses lazy shared data. Processes are automatically managed by sophisticated runtime systems for shared memory machines or distributed ones.

As above, cost analysis is hard to do and, sometimes, the runtime fails to distribute correctly the data [31]; it introduces too much communication and thus a lack of scalability. Another distributed language is HUME [19]. The main advantage of this language is that it is provided with a cost analysis of the programs for real-time purpose but with limitations of the expressiveness.

Algorithmic skeletons. Skeletons are patterns of parallel computations [15]. They can be seen as high-order functions that provide parallelism. They thus fall into the category of functional extensions. They are many skeleton libraries [15]. For OCAML, the most known work is the one of [11].

Distributed functional languages. In front of parallel functional languages, there are many concurrent extensions of functional languages such as ERLANG, CLEAN or JOCAML [27]. The latter is a concurrent extension of OCAML, which added explicit synchronisations of processes using specific patterns.

ALICE-ML [30] adds what is called a “future” for communicating values. A future is a placeholder for an undetermined result of a concurrent computation. When the computation delivers a result, the associated future is eliminated by globally replacing it by the result value. The language also contains “promises” that are explicit handles of futures. SCALA is a functional extension of JAVA which provides concurrency, using the actor model: mostly, creation of agents that can migrate across resources. Two others extensions of OCAML are [8] and [9]. The former uses SPMD primitives with a kind of futures. The latter allows migration of threads that communicate using particular shared data.

All these languages have the same drawbacks: they are not deadlock and race condition free; furthermore, they do not provide any cost model.

6 Conclusion and Future Work

6.1 Summary of the Contribution

The paper presents a language call MULTI-ML to program MULTI-BSP algorithms. It extends our previous BSML that has been designed for programming BSP algorithms. They both have the following advantages: *confluent* operational semantics; equivalence of the results for both toplevel and distributed implementation; cost model and efficiency.

The MULTI-BSP model extends the BSP one as a *hierarchical tree* of *nested* BSP machines. MULTI-ML extends BSML with a special syntax for define *special recursive functions* over this tree of nested machines, each of them programmed using BSML. In a tree, nodes contain codes to *manage* the submachines whereas leaves perform the *largest parts* of the computation. In this work, we focus on the informal presentation of MULTI-ML, an operational semantics of a core-language and benchmarks of simple examples with a comparison with *predicted performances* associated with the MULTI-BSP cost model. We also compare MULTI-ML codes with BSML ones as well as the performances of both languages on a typical cluster of hyper-threaded multi-cores. As predicted, MULTI-ML codes run *faster* when the cores share the network: there is *no bottleneck*; And the multi-core synchronisations are cheaper.

Compared to BSML, MULTI-ML have several drawbacks. First, the codes, the semantics and the implementation are a bit more complex. Second, the cost model associated to the program is more difficult to grasp: designing MULTI-BSP algorithms and programming them in MULTI-ML is more difficult than using BSML only. But, from our experience, we can say that it is not so hard.

6.2 Future Work

In a close future, we plan to axiomatise the MULTI-ML primitives inside COQ, as we did for BSML in [13,36], in order to prove the *correctness* of MULTI-BSP algorithms. We also consider to formally prove that the implementations follow the formal semantics. We plan also to benchmark bigger examples. We think of model-checking problems and algebraic computations that better follow high-level languages than intensive float operations can do.

But the most important work to do is the *implementation of a type system* for MULTI-ML to ensure a true safety of the codes: forbid nesting of vectors, forbid data-races if imperatives features, such as handling exceptions [14], are used. In the long term, the type system could be used to optimise the compiler. Indeed, currently, even in the case of a share-memory architecture, only serialised values are exchanged between nodes. We consider implementing a dedicated *concurrent garbage collector*.

References

1. Beran, M.: Decomposable BSP Computers. In: Theory and Practice of Informatics (SOFSEM), *LNCS*, vol. 1725, pp. 349–359. Springer (1999)
2. Bisseling, R.H.: Parallel Scientific Computation. A Structured Approach Using BSP and MPI. Oxford University Press (2004)
3. Blelloch, G.: NESL. Encyclopedia of Parallel Computing, pp. 1278–1283. Springer (2011)
4. Bonorden, O., Judoiink, B., von Otte, I., Rieping, O.: The Paderborn University BSP Library. Parallel Computing **29**(2), 187–207 (2003)
5. Bourgoin, M., Chailloux, E., Lamotte, J.L.: SPOC: GPGPU Programming through Stream Processing with OCAML. Parallel Processing Letters **22**(2), 1–12 (2012)
6. Cappello, F., Guermouche, A., Snir, M.: On Communication Determinism in HPC Applications. In: Computer Communications and Networks (ICCCN), pp. 1–8. IEEE (2010)
7. Cha, H., Lee, D.: H-BSP: A Hierarchical BSP Computation Model. Journal of Supercomputing **18**(2), 179–200 (2001)
8. Chailloux, E., Foisy, C.: A Portable Implementation for Objective Caml Flight. Parallel Processing Letters **13**(3), 425–436 (2003)
9. Chailloux, E., Ravet, V., Verlaguet, J.: HIRONDML: Fair threads migrations for Objective Caml. Parallel Processing Letters **18**(1), 55–69 (2008)
10. Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G., Marlow, S.: Data Parallel HASKELL. In: Declarative Aspects of Multicore Prog. (DAMP), pp. 10–18. ACM (2007)

11. Cosmo, R.D., Li, Z., Pelagatti, S., Weis, P.: Skeletal Parallel Programming with OCAMLP3L 2.0. Parallel Processing Letters **18**(1), 149–164 (2008)
12. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly-threaded Parallelism in MANTICORE. SIGPLAN Not. **43**(9), 119–130 (2008)
13. Gava, F.: Formal Proofs of Functional BSP Programs. PPL **13**(3), 365–376 (2003)
14. Gesbert, L., Gava, F., Loulergue, F., Dabrowski, F.: Bulk Synchronous Parallel ML with Exceptions. Future Generation Computer Systems **26**, 486–490 (2010)
15. González-Vélez, H., Leyton, M.: A Survey of Algorithmic Skeleton Frameworks. Software, Practrice & Experience **40**(12), 1135–1160 (2010)
16. Grelck, C., Scholz, S.B.: Classes and Objects as Basis for I/O in SAC. In: Implementation of Functional Languages (IFL), pp. 30–44 (1995)
17. Gu, Y., Le, B.S., Wentong, C.: JBSP: A BSP Programming Library in JAVA. Journal of Parallel and Distributed Computing **61**(8), 1126–1142 (2001)
18. Hamidouche, K., Falcou, J., Etiemble, D.: A Framework for an Automatic Hybrid MPI +OPEN-MP Code Generation. In: SpringSim (HPC), pp. 48–55. ACM (2011)
19. Hammond, K.: The Dynamic Properties of HUME: A Functionally-based Concurrent Language with Bounded Time and Space Behaviour. In: Implementation of Functional Languages (IFL), *LNCS*, vol. 2011, pp. 122–139. Springer (2000)
20. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer (2000)
21. Hill, J.M.D., McColl, B., et *al*: BSPLIB: The BSP Programming Library. Parallel Computing **24**, 1947–1980 (1998)
22. Hinsen, K., Langtangen, H.P., Skavhaug, O., Ødegård, Å.: Using BSP and PYTHON to Simplify Parallel Programming. Future Generation Comp. Syst. **22**(1-2), 123–157 (2006)
23. Hou, Q., Zhou, K., Guo, B.: BSPGP: Bulk-Synchronous GPU Programming. ACM Trans. Graph. **27**(3), pp. 1–30 (2008)
24. Keßler, C.W.: NESTSTEP: Nested Parallelism and Virtual Shared Memory for the BSP Model. The Journal of Supercomputing **17**(3), 245–262 (2000)
25. Leroy, X., Grall, H.: Coinductive Big-step Operational Semantics. Inf. Comput. **207**(2), 284–304 (2009)
26. Loidl, H.W., et *al*: Comparing Parallel Functional Languages: Programming and Performance. Higher Order and Symb. Comp. **16**(3), 203–251 (2003)
27. Mandel, L., Maranget, L.: Programming in JOCAML. In: European Symposium on Programming (ESOP), no. 4960 in LNCS, pp. 108–111. Springer (2008)
28. Miller, Q.: BSP in a Lazy Functional Context. In: Trends in Functional Programming, vol. 3. Intellect Books (2002)
29. da Rosa Righi, R., et *al*: MIGBSP: A Novel Migration Model for BSP Processes Re-scheduling. In: HPC and Communications (HPCC), pp. 585–590. IEEE (2009)
30. Rossberg, A.: Typed Open Programming – a Higher-order, Typed Approach to Dynamic Modularity and Distribution. Ph.D. thesis, Universität des Saarlandes (2007)
31. Scaife, N., Michaelson, G., Horiguchi, S.: Empirical Parallel Performance Prediction From Semantics-Based Profiling. Scalable Computing: Prac. and Exp. **7**(3) (2006)
32. Seo, S., Yoon, et *al*: HAMA: An Efficient Matrix Computation with the MapReduce Framework. In: Cloud Computing (CloudCom), pp. 721–726. IEEE (2010)
33. Valiant, L.G.: A Bridging Model for Parallel Computation. Comm. of the ACM **33**(8), 103–111 (1990)
34. Valiant, L.G.: A Bridging Model for Multi-core Computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
35. Yzelman, A.N., Bisseling, R.H.: An Object-oriented BSP Library for Multicore Programming. Concurrency and Computation: Practice and Experience **24**(5), 533–553 (2012)
36. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct BSP Programs. In: PDCAT, pp. 334–340. IEEE (2010)
37. Fortin, J., Gava, F.: BSP-WHY: a Tool for Deductive Verification of BSP Algorithms with Subgroup ynchronization. Journal of Parallel Programming. To appear. 2015.
38. Li, C., Hains, G.: SGL: Towards a Bridging Model for Heterogeneous Hierarchical Platforms IJHPCN. **7**(2), pp. 139–151 (2012)

Organisation des structures de données : abstractions et impact sur les performances

Sylvain Jubertie, Joël Falcou, Ian Masliah

Laboratoire de Recherche en Informatique
Université Paris-Sud 11
Faculté des Sciences d'Orsay - Bat 650
91405 Orsay Cedex
prenom.nom@lri.fr

Abstract. L'optimisation de codes de calculs (simulations scientifiques, traitements d'images, etc) nécessite souvent la réorganisation des structures de données utilisées afin de pouvoir exploiter au mieux les processeurs et accélérateurs actuels. Les tableaux de structures (AoS : Array of Structures) sont ainsi souvent convertis en structures de tableaux (SoA : Structure of Arrays). Dans les codes de calculs effectuant les mêmes opérations sur les éléments de chaque structure, cette transformation permet d'exploiter les unités vectorielles des processeurs et accélérateurs. Cependant, cette structure de données (SoA) n'est pas communément communément utilisée par les développeurs, ce qui peut affecter la maintenabilité et l'évolutivité du code.
L'approche proposée consiste en une couche d'abstraction indépendante de l'organisation des données en mémoire de manière à combiner les avantages des deux représentations. Le développeur conserve sa vision des données organisées en structures, tandis que les données peuvent être stockées dans une autre représentation adaptée à l'architecture. Cette abstraction est réalisée en C++ à l'aide de différentes classes permettant au développeur de définir ses structures de données. L'interface proposée est similaire à celle des conteneurs de la bibliothèque standard STL. Les résultats sur différents algorithmes montrent que cette couche d'abstraction n'introduit pas de surcoût en terme de performance par rapport à une version écrite explicitement.

1 Introduction

L'écriture et l'optimisation de codes de calculs se heurte au paradoxe suivant : d'une part, il faut simplifier leur programmation à l'aide de langages de haut-niveau afin de se focaliser sur le problème à résoudre, d'autre part, il faut considérer des optimisations de bas-niveau qui nécessitent une connaissance approfondie des architectures et une écriture particulière du code et des structures de données. Par exemple, dans le cadre d'un traitement d'images, il est plus simple de se représenter une image comme un tableau de pixels dotés de trois composantes, rouge, verte et bleue, et de définir des traitements sur cette structure, que de considérer cette image comme une structure de trois tableaux, chacun stockant une composante différente. Cependant, cette dernière représentation est la plus adaptée lorsqu'il s'agit d'exploiter les unités vectorielles des processeurs et accélérateurs, permettant ainsi d'obtenir des facteurs d'accélérations conséquents.

Les unités vectorielles sont présentes dans tous les processeurs modernes et permettent d'appliquer une opération sur un ensemble de données, appelé vecteur, en une seule instruction, contrairement aux unités scalaires traditionnelles qui appliquent une opération à une unique donnée à la fois. Les unités vectorielles exploitent donc le parallélisme de données suivant le paradigme SIMD (Single Instruction on Multiple Data). Différentes implantations sont disponibles suivant l'architecture du processeur ou de l'accélérateur considéré. Ces implantations diffèrent dans la largeur du vecteur et dans les instructions disponibles. Les processeurs basés sur l'architecture ARM, présents par exemple dans les smartphones, disposent d'une unité vectorielle appelée NEON capable de traiter un vecteur de 128 bits en une instruction, soit par exemple 4 entiers 32 bits, ou 4 nombres flottants en simple précision simultanément. Les processeurs basés sur l'architecture x86 peuvent disposer d'une unité vectorielle appelée AVX permettant de traiter des vecteurs de 256 bits, soit 8 nombres flottants en simple précision ou 4 en double précision. Les accélérateurs de type GPU sont également constitués d'unités vectorielles plus larges. Les unités des GPU Nvidia CUDA permettent ainsi de traiter 32 nombre flottants en une instruction.

Ne pas exploiter les unités vectorielles limite donc la performance exploitable à une fraction de la puissance maximale disponible. Cela est plus particulièrement critique pour les codes limités par la puissance de calcul et pas par la bande passante. De nombreux travaux illustrent les gains obtenus par la transformation des structures de données afin d'exploiter les unités vectoriels.

2 Réalisation de l'abstraction

Notre approche propose plusieurs manières d'organiser les structures de données présentant la même interface. L'utilisateur peut ainsi utiliser une structure de type AoS pour écrire son code puis basculer vers une autre représentation sans avoir à modifier son code.

2.1 Définitions des structures

AoS. Une structure de type AoS est définie comme un vecteur de tuples :

```
template< typename... Types >
class aos: private std::vector< std::tuple< Types... > >
```

Les types des attributs de la structure peuvent être de longueur arbitraire par l'utilisation des paramètres variadiques des templates.

Pour définir une image dotée de 3 composantes, l'utilisateur peut déclarer le type suivant :

```
using imageRGB = dod::aos< uint8_t , uint8_t , uint8_t >;
```

SoA. Une structure de type AoS est définie comme un tuple de vecteurs :

```
template< typename... Types >
class soa: private std::tuple< std::vector< Types >... >
```

Pour définir une image dotée de 3 composantes, l'utilisateur peut déclarer le type suivant :

```
using imageRGB = dod::soa< uint8_t, uint8_t, uint8_t >;
```

2.2 Interface

Les structures décrites précédemment proposent une interface proche de celle du vecteur de la STL. Néanmoins, l'opérateur [] ne peut être surchargé pour prendre deux paramètres. C'est donc l'opérateur () qui est utilisé. L'accès à l'attribut n de l'élément à la position i de la structure st s'écrit donc :`st( n, i );`.

La conversion d'une image couleur en niveau de gris peut donc s'écrire de la manière suivante :

```
void grayscale( imageRGB const & in,
                imageL & out,
                size_t const rows,
                std::size_t const cols )
{
  for( std::size_t i = 0 ; i < rows * cols ; ++i )
  {
    unsigned int l = 307 * in( R, i )
                   + 604 * in( G, i )
                   + 113 * in( B, i );
    out[ i ] = l >> 10;
  }
}
```

Dans l'exemple ci-dessus, les indices des composantes peuvent être remplacés par des identifiants plus explicites. Il suffit pour cela d'ajouter les définitions suivantes :

```
std::integral_constant< std::size_t, 0 > R;
std::integral_constant< std::size_t, 1 > G;
std::integral_constant< std::size_t, 2 > B;
```

La verbosité de ce code peut être masquée par la définition d'une macro simplifiant cette écriture.

3 Expérimentation et résultats

Notre approche a été appliquée sur un exemple de base permettant de faire varier le nombre et le type des attributs avec un calcul simple, ainsi que sur des algorithmes de calcul vectoriel et une simulation N-Body. Les compilateurs GCC et ICC ont été utilisés dans différentes versions pour vérifier le comportement et l'impact du compilateur. Les options d'optimisation et d'auto-vectorisation sont activées dans tous les cas.

3.1 Coût et impact de l'approche

Les différents tests montrent que par rapport à un code écrit explicitement en SoA, AoS ou hybride, notre approche n'entraîne pas de surcoût à l'exécution. L'analyse des codes assembleurs générés révèle que ceux-ci sont quasiment identiques pour les versions respectives modulo le renommage des registres. Seul le calcul des adresses peut parfois varier, notamment dans le cas de la normalisation de vecteur, mais l'impact est négligeable sur les performances par rapport à la variabilité des résultats entre plusieurs exécutions.

	GCC	GCC fastmath	ICC
nbody-std-aos	518	362	401
nbody-abs-aos	517	283	414
nbody-std-soa	570	414	78
nbody-abs-soa	568	414	78
normalize-std-aos	1488	662	1545
normalize-abs-aos	1366	666	1374
normalize-std-soa	1321	343	386
normalize-abs-soa	1348	368	1373

Fig. 1. Comparaison des versions standards (std) et avec abstraction (abs) des algorithmes N-Body et normalize. (temps en ms)

Il convient de noter que suivant les tests l'option -fast-math de GCC est activée car ICC active parfois cette option automatiquement. Les deux dernières lignes du tableau 3.1 montre que le compilateur Intel n'active pas cette option dans la version utilisant notre approche. Cela permet de vérifier si les gains obtenus sont dûs à cette option ou à la vectorisation.

Dans le cas de l'algorithme N-Body, il est intéressant de noter que le compilateur Intel déclenche la vectorisation automatique du code pour la version SoA, avec ou sans l'utilisation de notre approche. C'est un bon point pour notre approche car cela ne casse pas le processus de vectorisation du compilateur. De plus, le code étant le même entre la version AoS et SoA, le gain apporté par le compilateur Intel ne nécessite pas d'effort de programmation supplémentaire.

Dans le cas de l'algorithme de normalisation de vecteur, tous les codes sont automatiquement vectorisés par les compilateurs, les différences de performances sont dues uniquement à l'organisation des données et à l'activation de l'option -fast-math. Le passage à une structure de type SoA n'apporte pas de gain notable, sauf pour la version compilée avec l'option -fast-math. Dans la version AoS, le compilateur génère des instructions vectorielles pour réorganiser les données et vectoriser les calculs. Ces instructions ne représentent que quelques cycles sur les dizaines de cycles utilisés pour calculer l'inverse de la racine du carré des composantes des vecteurs. Par contre l'option -fast-math active le remplacement de ce calcul par l'instruction _mm256_rsqrt_ps dont la latence est de seulement 5 cycles (architecture Intel Haswell). La transformation devient donc plus coûteuse par rapport au calcul dans ce mode ce qui explique le doublement des performance entre les versions AoS et SoA.

3.2 Comparaison entre représentations

La comparaison entre les différentes versions de l'algorithme N-Body montre, dans le cas de l'utilisation du compilateur GCC, que les performances sont en retrait lors du passage à une structure de type SoA. En effet, l'utilisation de plusieurs tableaux nécessite le stockage d'autant de pointeurs pour la manipulation des données, ce qui accroît l'occupation des registres généraux disponibles en nombre limité. Des expérimentations sur notre programme de test montre que ces registres sont rapidement saturés et que le compilateur est obligé de générer des opérations sur la pile pour manipuler les pointeurs supplémentaires. Suivant le contexte : nombre d'itérateurs manipulés (boucles), pointeurs, ce phénomène peut apparaître dès l'utilisation de 5 à 6 tableaux. L'impact sur les performances est particulièrement important sur les codes dont le facteur limitant les performances est la bande passante. Dans le cas de l'algorithme N-Body, qui est limité par la puissance de calcul, ce phénomène est largement effacé par le gain apporté par la vectorisation comme le montre les résultats du compilateur Intel.

4 Conclusion

L'utilisation d'une approche permettant d'abstraire l'organisation des structures de données permet à la fois au développeur de se concentrer sur son application, tout en permettant la transformation des structures de données afin de pouvoir exploiter les unités vectorielles. Dans certains cas, le passage d'une structure de type AoS à SoA permet la vectorisation automatique du code par le compilateur. L'exemple de l'algorithme N-Body montre que les performances sont multipliées par cinq dans ce cas. Toutefois, dans d'autres cas, le compilateur arrive à vectoriser des structures de données de type AoS ce qui peut être bénéfique si la limitation est la puissance de calcul. Dans de nombreux cas également, la vectorisation reste à la charge du développeur.

Ces travaux constituent une première étape d'un projet visant à abstraire l'organisation et la transformation des structures de données au développeur. L'implantation d'une nouvelle structure de données hybride, aussi appelée AoSoA (Array of Structure of Array), combinant les avantages de la représentation AoS et SoA est en cours. La possibilité de convertir la représentation automatiquement, suivant la représentation fournie et l'architecture considérée est envisagée. En effet, de nombreuses bibliothèques fournissent des structures de données de type AoS, ce qui induit un surcoût. Il est envisageable d'effectuer la conversion de structure à la volée lors du premier calcul sur les données.

From BSP regular expressions to BSP automata

Thibaut Tachon[1,2], Gaetan Hains[1], Frederic Loulergue[2], and Chong Li[1]

[1] DPSL-DAL, Central Software Institute, Huawei Technologies, France
[2] LIFO, Université d'Orléans, France
thibaut.tachon@huawei.com

Abstract. The theory of finite automata has been recently adapted to bulk-synchronous parallel computing by introducing BSP words, BSP automata and BSP regular expression. We propose here an algorithm that, from a BSP regular expression, generates a BSP automaton recognizing the same language.

Keywords: BSP, automata, regular expressions

1 Introduction

Bulk synchronous parallel (BSP) is a model of parallel computing introduced by Valiant [5]. By adapting automata theory [4] to BSP, Hains [3] proposed Bulk-Synchronous Parallel Automata (BSPA) to describe BSP computation, and Bulk-Synchronous Parallel Regular Expressions (BSPRE) to describe BSP language. However, there was no automatical way to couple BSPA and BSPRE. We summarize here the (implemented) tranformation from BSPRE to BSPA.

2 BSP Word and Language

The set of symbols is denoted by Σ, and p denote the number of processors.

Definition 1. *Elements of $(\Sigma^*)^p$ is called word-vectors. A BSP word over Σ is a sequence of word-vectors, i.e., a sequence of $((\Sigma^*)^p)^*$. A BSP language over Σ is a set of BSP words over Σ.*

3 BSP Automata (BSPA)

Definition 2. *A BSP automaton $\vec{A}$ is an hexa-tuple*

$$(\{Q^i\}_{i\in[p]}, \Sigma, \{\delta^i\}_{i\in[p]}, \{q_0^i\}_{i\in[p]}, \{F^i\}_{i\in[p]}, \Delta)$$

such that for every i, $(Q^i, \Sigma, \delta^i, q_0^i, F^i)$ is a finite automaton[3], and $\Delta : \vec{Q} \to \vec{Q}$ is called the synchronization function where $\vec{Q} = (Q^0 \times ... \times Q^{(p-1)})$ is the set of global states.

In other words a BSP automaton is a vector of sequential automata A^i over the same alphabet Σ, together with a synchronization function (Δ) that maps state-vectors to state-vectors.

[3] Q^i is the finite set of states, δ the transition function, $q^i \in Q^i$ the initial state, and $F^i \subseteq Q^i$ the non-empty set of accepting states.

4 BSP Regular Expressions (BSPRE)

We define the grammar of BSPRE (left column of the right-side table) and language of BSPRE (right column of the right-side table), where $r^{i\ (=1 \dots p-1)}$ is a (scalar) regular expression. The type of the language is:
$L : \mathit{BSPRE} \rightarrow \mathcal{P}(((\Sigma^*)^p)^*)$.

Grammar: R	Language: $L(R)$
$\emptyset$	$\{\ \}$
ϵ	$\{\epsilon\}$
$< r^0, \dots, r^{p-1} >$	$L(r^0) \times \dots \times L(r^{p-1})$
$R_1; R_2$	$L(R_1)L(R_2)$
R^*	$L(R)^*$
$R_1 + R_2$	$L(R_1) \cup L(R_2)$

5 From BSPRE to BSPA

Types of a BSPRE, regular expression, automata, BSPA are, resctively, $\alpha\ \mathit{bspre}$, $\alpha\ \mathit{expr}$, $\alpha\ \mathit{autom}$, $\alpha\ \mathit{bspa}$, where α is the type of alphabet (set of symbols). The followings describe an algorithm that, from a BSP regular expression, generates a BSP automaton recognizing the same language.

1. *Sequentialization* : $\Sigma\ \mathit{bspre} \rightarrow ((\Sigma \times [p]) \cup \{;\})\ \mathit{expr}$
   Removing vectors by giving each symbol its removed-vector position.
2. *Distribution* : $((\Sigma \times [p]) \cup \{;\})\ \mathit{expr} \rightarrow ((\Sigma \cup \{;\})\ \mathit{expr})^p$
   Constructing a vector of regular expression by puting each symbol to the associated component and replicating semicolons in all vector components.
3. *Transformation* : $((\Sigma \cup \{;\})\ \mathit{expr})^p \rightarrow ((\Sigma \cup \{;\})\ \mathit{autom})^p$
   Giving vector of NFA, each of them accepts its component of vector. The algorithm used for transformation was invented by Bruggemann-Klein [1] which is an improvement of Glushkov [2].
4. *Synchronization* : $((\Sigma \cup \{;\})\ \mathit{autom})^p \rightarrow \Sigma\ \mathit{bspa}$
   Semicolon-labelled transitions that were the same before the replication are glued together into an hyperedge (Δ). The hyperedge connects all its transitions' NFA into one BSPA.

6 Conclusion

We have presented an algorithm to get a BSPA from a BSPRE. In the future, we will perform more complexity studies, in order to analyze the third step – transformation – of our algorithm that may be the bottleneck: quadratic in size of inputed BSPRE.

References

1. Anne Brüggemann-Klein. Regular expressions into finite automata. *Theoretical Computer Science*, 120(2):197 – 213, 1993.
2. Pascal Caron and Djelloul Ziadi. Characterization of glushkov automata. *Theoretical Computer Science*, 233(12):75 – 90, 2000.
3. G. Hains. Enumerated BSP automata. Technical report. DAL-TR-2016-1.
4. S. C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, pages 3–41, 1956.
5. Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

Session du groupe de travail MFDL

Méthodes Formelles dans le Développement Logiciel

Un processus de développement Event-B pour des applications distribuées

Badr Siala, Mohamed Tahar Bhiri, Jean-Paul Bodeveix et Mamoun Filali
Université de Sfax
IRIT CNRS UPS Université de Toulouse

Résumé

Nous présentons une méthodologie basée sur le raffinement manuel et automatique pour le développement d'applications distribuées correctes par construction. À partir d'un modèle Event-B, le processus étudié définit des étapes successives pour décomposer et ordonnancer les calculs associés aux événements et distribuer le code sur des composants. La spécification de ces deux étapes est faite au travers de deux langages dédiés. Enfin, une implantation distribuée en BIP est générée. La correction du processus repose sur la correction des raffinements et de la traduction vers le code cible BIP.

1 Introduction

Les systèmes distribués demeurent difficiles à concevoir, construire et faire évoluer. Ceci est lié à la concurrence et au non déterminisme. Plusieurs formalismes tels que les algèbres de processus, ASM et TLA$^+$ sont utilisés pour modéliser et raisonner sur la correction des systèmes distribués. Un spécifieur utilisant ces formalismes est censé modéliser des mécanismes de bas niveau tels que canaux de communication et ordonnanceurs.

Dans cet article, nous proposons un support outillé d'aide à la conception système en utilisant une méthodologie basée sur des raffinements prouvés. Les systèmes considérés sont vus comme un ensemble d'acteurs en interaction. Les premiers raffinements fournissent une vue centralisée du système. Ils sont construits en prenant en compte progressivement les exigences du système. Ces raffinements sont exprimés à l'aide de machines abstraites décrites en Event-B [2]. Ensuite, nous proposons des schémas de raffinements destinés à prendre en compte la nature distribuée du système étudié. Ces schémas de raffinements sont guidés par l'utilsateur et les machines Event-B associées sont automatiquement générées. En conséquence, on obtient un ensemble de machines qui interagissent, dont la composition est prouvée correcte et conforme avec le niveau abstrait. Le système peut ensuite être exécuté sur une plateforme distribuée via une traduction en BIP [3]. Remarquons que notre objectif n'est pas d'automatiser complètement le processus de distribution, mais de l'assister. Tout en restant modeste, la différence est similaire à celle entre un vérificateur de modèle où la preuve d'un jugement est automatique et un assistant de preuve où l'utilisateur doit composer des stratégies de base afin de résoudre son but. De même qu'un assistant de preuve aide à construire la preuve d'un but, notre objectif est d'aider à l'élaboration par raffinements d'un modèle distribué.

Event-B et BIP admettent une sémantique basée sur les systèmes de transitions étiquetées. Ceci favorise leur couplage. Event-B est utilisée pour la spécification et la décomposition formelles des systèmes distribués. BIP est utilisée pour leur implantation et leur déploiement sur une plateforme à mémoire répartie. Le passage d'Event-B vers BIP s'appuie sur des raffinements manuels et automatiques. Les raffinements manuels horizontaux permettent l'introduction progressive de propriétés du futur système. Les raffinements manuels verticaux permettent l'obtention de modèles Event-B traduisibles vers BIP : déterminisation d'actions et concrétisation des données. Quant aux raffinements automatiques, nous en avons élaboré deux sortes : l'une est appelée *fragmentation* (voir section 3) et l'autre *distribution* (voir section 4). Ces deux sortes de raffinement sont guidées par le spécifieur via deux langages dédiés (DSL). Cette démarche peut être comparée aux travaux portant sur la génération automatique de code source à partir de spécifications formelles. [4] propose un générateur de code efficace séquentiel en utilisant un sous-ensemble B0 impératif de B. Le raffinement automatique de machines B est également possible grâce à l'outil Bart [5]. Cependant, il ne concerne pas les modèles Event-B et est destiné aux raffinements de modèles B vers le sous-ensemble B0. Concernant Event-B, plusieurs générateurs de code source ont été proposés [6, 8]. Il est possible de générer du code Ada parallèle. Cette dernière cible est cependant plus restrictive que BIP puisqu'elle n'offre que de la synchronisation binaire.

Le processus de développement correct par construction de systèmes distribués préconisé combine les raffinements manuels et automatiques. Il se termine par la génération de code BIP. Dans la suite, après une introduction aux formalismes Event-B et BIP, nous présentons les deux transformations par raffinement (la fragmentation et la distribution), puis la génération de code BIP. Ces étapes seront illustrées sur l'exemple de l'hôtel[1].

Le passage d'Event-B vers BIP se fait en trois étapes : fragmentation, distribution et génération de code (voir Figure 1).

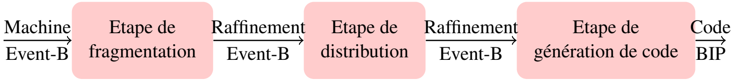


FIGURE 1 – Etapes du processus

[1] Le texte complet est accessible via le lien https://dl.dropboxusercontent.com/u/98832434/hotelrefinements.html.

2 Event-B et BIP

2.1 Event-B

Event-B permet de décrire le comportement d'un système par une chaine de raffinements de machines abstraites. Une machine contient des événements, éventuellement paramétrés, faisant évoluer l'état de la machine tout en préservant un invariant. Le non-déterminisme s'exprime dans le corps des événements grâce aux actions (:∈ et :|) ou via leurs paramètres introduits dans la clause `ANY` et contraints par les propriétés spécifiées dans la clause `WHERE`. Dans une optique de décomposition formelle d'une spécification centralisée en plusieurs composants Event-B, les contraintes d'un paramètre d'un événement partagé doivent être résolues localement. Nous adressons ce problème ultérieurement.

Récemment, deux opérations ont été introduites dans Event-B : la composition et la décomposition [7]. Elles ont pour but d'introduire la notion de composant dans la démarche de raffinement. Ces deux opérations sont reliées par la propriété suivante : la composition des sous-composants produits par décomposition d'un modèle doit raffiner ce modèle. Deux variantes ont été identifiées : par variables partagées et par événements partagés. La première est adaptée aux systèmes à mémoire partagée, tandis que la deuxième est plutôt adaptée aux systèmes distribués. Dans ce travail, nous nous intéressons à la composition/décomposition par événements partagés. La plateforme Rodin offre un outil interactif [7] sous forme d'un plugin permettant la composition/décomposition par événements partagés. Dans la section 4, nous situerons notre plugin de distribution vis-à-vis de ce plugin.

2.2 BIP

Le modèle de composant BIP comporte trois couches : *Behavior*, *Interaction* et *Priority*. La couche *Behavior* permet de décrire les aspects **comportementaux** des composants atomiques tandis que les couches *Interaction* et *Priority* décrivent les aspects **architecturaux** d'un système. Les contraintes de synchronisation entre les composants BIP sont exprimées par des interactions regroupées au sein de la construction `connector` de BIP tandis que les contraintes d'ordonnancement des interactions sont exprimées grâce au concept Priority de BIP. En BIP, un composant atomique englobe des données, des ports et un comportement. Les données (`data`) sont typées. Les ports (`port`) donnent accès à des données et constituent l'**interface** du composant. Le comportement est un ensemble de transitions définies par un port, une garde et une fonction de mise à jour des données.

3 La fragmentation

Cette étape, dite de fragmentation, prend en entrée un modèle Event-B quelconque (voir figure 1). Elle a pour but de réduire le non-déterminisme lié au calcul des paramètres locaux d'un événement. L'ordre de calcul des paramètres est décrit par le spécifieur à l'aide d'un DSL (voir listing 1). En se basant sur cette description et le modèle Event-B abstrait, l'étape de fragmentation génère un modèle Event-B. Ce raffinement automatique s'appuie sur des règles simples assurant l'obtention d'un raffinement : introduction d'un nouvel événement convergent, raffinement d'un événement par plusieurs (one to many), introduction d'une nouvelle variable, renforcement de garde, renforcement d'invariant et instanciation d'un paramètre local d'un événement par une variable d'état.

Plugin de fragmentation. La transformation de fragmentation est implantée par un plugin Rodin. Elle génère un raffinement de la machine en question à partir d'une *spécification de la fragmentation*. Celle-ci comporte des déclarations donnant pour un paramètre p d'un événement ev les paramètres p_i dont il dépend et les gardes g_i le spécifiant. Une valeur initiale v est requise pour des besoins de typage.

event ev **when** p_1 ... p_n **parameter** p **init** v **with** g_1 ... g_m

Exemple d'illustration. Dans l'exemple de l'hôtel, l'événement `register` a trois paramètres : `g,r,c`. Nous spécifions que le paramètre client `g` doit être calculé en premier. Une chambre r est alors choisie et la carte d'accès c est enfin générée. La fragmentation de l'événement `register` est ainsi spécifiée :

```
splitting hotel_splitted refines hotel
events
  event register
    parameter g init g0 with tg // tg ne dépend pas de r,c
    when g parameter r init r0 with tr g1 // déclenché après le calcul de g
    when g r parameter c init c0 with tc g2 g3 // déclenché après g,r
end
```

Listing 1 – Spécification de la fragmentation

La fragmentation produit une machine dont le schéma général est le suivant :

```
machine generated refines input_machine
variables
 ev_p  ev_p_computed// témoin et statut pour param. p de l'événement ev
invariants
  @ev_gi ev_p_computed = TRUE ⇒ gi // ou p est remplacé par ev_p
variant  // compteur des paramètres restant à calculer
  {FALSE ↦ 1, TRUE ↦ 0}(ev_p_computed) + ...
events
  event INITIALISATION extends INITIALISATION
  then
    @ev_p ev_p := v
    @ev_p_comp ev_p_computed := FALSE
  end
  convergent event compute_ev_p // calcule le  paramètre p de ev
  any p where
   @gi gi // garde spécifiant  p
   @pi ev_pi_computed = TRUE //paramètres dont p dépend ont été calculés
   @p ev_p_computed = FALSE // p reste à calculer
  then
    @a ev_p := p
    @computed ev_p_computed := TRUE // décroissance du variant
  end
  event ev refines ev
  when
    @p_comp ev_p_computed = TRUE
  with
    @p p = ev_p //le paramètre p de l'événement hérité est raffiné en ev_p
  then
    // remplacer p par ev_p dans les actions de l'événement raffiné
  end
end
```

Listing 2 – Machine générée pour le raffinement de fragmentation

En fait, cette machine implante les contraintes d'ordonnancement par l'introduction d'une variable booléenne, *computed state*, par paramètre. L'invariant de la machine est étendu par les propriétés des variables introduites. Si une variable a été calculée (`ev_p_computed = TRUE`), sa spécification, donnée par sa garde `gi`, est alors satisfaite. Lorsque tous les paramètres d'un événement ont été calculés, l'événement en question peut être activé. Enfin, la progression du calcul des paramètres requis est assurée par un variant défini comme le nombre de paramètres restant à calculer.

4 La distribution

Après l'étape de fragmentation (voir figure 1), l'étape de gestion de la distribution prend en compte les particularités de l'architecture cible. Il s'agit ici de composants BIP synchronisés par des connecteurs n-aires et supposés ne réaliser que des transferts de données (pas de traitement interne dans un connecteur). À l'instar de l'étape de fragmentation, l'étape de distribution prend en entrée un modèle abstrait et une spécification de distribution. Celle-ci est décrite par le spécifieur à l'aide d'un DSL. Elle introduit la configuration retenue : les noms des sous-composants. De même, elle répartit les variables et éventuellement les gardes sur les sous-composants. Les variables référencées par une garde doivent être localisées sur un même sous-composant. Autrement, des copies *faiblement cohérentes* de ses variables seront automatiquement ajoutées. Elles sont mises à jour par des événements convergents ordonnancés avant l'événement accédant aux copies. De même, une action est réalisée par le composant (supposé unique) sur lequel les variables modifiées sont localisées. Les variables lues par une action peuvent être distantes. Les valeurs de ces variables seront transmises lors de la synchronisation. Notons que la gestion des copies faiblement cohérentes et des variables distantes sont spécifiques à notre proposition. Ceci peut être vu comme une extension du plugin de *décomposition par événement partagé* [7]. Pour y parvenir, nous introduisons une phase de pré-traitement.

La phase de pré-traitement. Cette phase prend en entrée une machine abstraite, des identifiants de sous-composants et la répartition de ces variables sur des sous-composants. Elle produit un raffinement introduisant des copies des variables distantes lues par les gardes, les événements anticipés mettant à jour ces variables, et des paramètres recevant les valeurs des variables distantes lues par les actions. Nous supposons dans ce qui suit que les variables v_i sont localisées sur les composants C_i. Les événements convergents sont partagés par les origines (C_i) et destinations (C_j) des variables lues par les gardes. Les raffinements des événements hérités sont partagés par les origines (C_i) des copies (sur C_j) et par les composants (C_k). Ces derniers comportent les variables directement lues par les actions.

```
machine generated refines input_machine
variables
  vi // variables héritées, sur Ci
  Cj_vi // copie sur Cj de vi (utilisée par une garde sur Cj)
  vi_fresh // true si vi a été copié, sur Ci
invariants
  @Cj_vi_f vi_fresh = TRUE ⇒ Cj_vi = vi // la copie est à jour
variant
  {FALSE ↦ 1, TRUE ↦ 0}(vi_fresh) + ...
events
  convergent event share_vi // partagé par Ci et Cj
  any local_vi where
    @g vi_fresh = FALSE // sur Ci
    @l local_vi = vi // sur Ci
  then
    @to_Cj Cj_vi := local_vi // sur Cj
    @done vi_fresh := TRUE // sur Ci
  end
  event ev refines ev // partagé par Ci,Cj,Ck
  any local_vk where
    @vj_access local_vk = vk // sur Ck, accès direct par les actions
     @vi_fresh vi_fresh = TRUE // sur Ci, la copie sur Cj est à jour
    @g [vi := Cj_vi]g // sur Cj, garde héritée avec accès aux copies
  then
    @a vj := [vi := Cj_vi;vk := local_vk]e // sur Cj, synchro avec Ck
  end
end
```

Listing 3 – Pré-traitement

La phase de projection. Cette phase construit une machine pour chaque sous-composant, et reprend le plugin de décomposition par événement partagé [7]. Les sites de projection des gardes et actions sont indiqués dans le listing 3. Notons que les invariants faisant référence à des variables distantes sont naturellement écartés.

5 La génération de code BIP

L'étape de génération de code BIP prend en entrée les composants Event-B issus de l'étape de distribution. Elle génère les éléments suivants :

Types de port. Pour chaque événement de chaque sous-composant nous générons un type de port. Les variables associées à un type de port donné sont issues des variables référencées par l'événement correspondant.

Types de connecteur. Pour chaque événement faisant référence à des variables de plusieurs composants, nous générons un type de connecteur.

Squelette de sous-composants. Pour chaque sous-composant, nous générons un composant atomique BIP. Il comporte les variables du sous-composant ainsi que les variables distantes appartenant à d'autres sous-composants, les instances des types de ports associés à ce sous-composant et, pour chaque événement, une transition synchronisée sur l'instance du port correspondant.

Le composant composite. Il regroupe une instance de chaque sous-composant et connecteur. Chaque instance de connecteur admet une instance de port appartenant aux instances de sous-composants définies précédemment.

6 Conclusion

Nous avons présenté une méthode de développement de systèmes distribués corrects par construction. À partir d'une machine Event-B représentant un modèle centralisé, nous appliquons deux types de raffinements automatiques (la fragmentation et la distribution) dont les paramètres sont déclarés par deux langages dédiés. Enfin, un modèle BIP exécutable sur une architecture répartie est produit. La correction de cette méthode repose sur la correction des étapes de raffinement et de la traduction finale en code BIP.

Nous envisageons maintenant d'améliorer l'outillage de ce processus. Les transformations de modèles Event-B sont réalisées via xtext et xtend [1]. L'étape suivante sera la finalisation du générateur BIP.

Références

[1] Language engineering for everyone ! `https://eclipse.org/Xtext`. Acc : 16-01-06.

[2] J.-R. Abrial. *Modeling in Event-B : System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] A. Basu, S. Bensalem, M. Bozga, J. Combaz, M. Jaber, T.-H. Nguyen, and J. Sifakis. Rigorous component-based system design using the bip framework. *IEEE Software*, 28(3) :41–48, 2011.

[4] D. Bert, S. Boulmé, M.-L. Potet, A. Requet, and L. Voisin. Adaptable translator of B specifications to embedded c programs. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME*, volume 2805 of *LNCS*, pages 94–113. Springer, 2003.

[5] Clearsy. Bart (b automatic refinement tool). `http://tools.clearsy.com/wp-content/uploads/sites/8/resources/BART_GUI_User_Manual.pdf`.

[6] A. Edmunds and M. Butler. Tasking Event-B : An extension to Event-B for generating concurrent code. Event Dates : 2nd April 2011, February 2011.

[7] R. Silva and M. Butler. Shared event composition/decomposition in Event-B. In *FMCO Formal Methods for Components and Objects*, November 2010. Event Dates : 29 November - 1 December 2010.

[8] N. K. Singh. Eb2all : An automatic code generation tool. In *Using Event-B for Critical Device Software Systems*, pages 105–141. Springer London, 2013.

Projet ANR BINSEC
analyse formelle de code binaire pour la sécurité

Sébastien Bardin (coordinateur)
prenom.nom@cea.fr
Airbus Group, CEA LIST, IRISA, LORIA, Université Grenoble Alpes

BINSEC : BINary-code analysis for SECurity

– Projet ANR INS, appel 2012
– Axes 1 (sécurité) et 2 (génie logiciel)
– Projet de recherche fondamentale sur 4 ans (2013-2017)
– Sujet : Techniques formelles d'analyse de sécurité au niveau binaire

Contexte, objectifs et défis. L'objectif général du projet BINSEC est de combler le fossé actuel entre le succès des méthodes formelles pour la sûreté logicielle (niveau source ou modèle) et les besoins des analyses de sécurité niveau binaire (analyse de vulnérabilité, analyse de malware), pour l'instant peu outillées (approches syntaxiques). Pour les malware, nous nous concentrons sur les problèmes de camouflage et d'obscurcissement (*obfuscation*). Pour les vulnérabilités, nous nous concentrons sur la découverte de bugs et sur la distinction entre bugs bénins et bugs exploitables.
Les défis principaux viennent de la structure même du code binaire (données et contrôle bas niveau), de la complexité des architectures modernes, du manque de compréhension claire de certaines propriétés considérées (obscurcissement, exploitabilité), de la présence d'un attaquant et enfin de la volonté de considérer des programmes non critiques (robustesse et passage à l'échelle).

Bien qu'a priori distincts, les domaines que nous attaquons partagent des problèmes communs, par exemple le manque de sémantique claire et unifiée, la reconstruction précise du flot de contrôle (même avec obscurcissement), le besoin de raisonnement bas niveau précis, etc. Le projet est ainsi architecturé autour de quelques problématiques générales (sémantique, analyses de base) sur lesquelles se basent les travaux applicatifs. Les résultats sont en partie intégrés dans la plate-forme open-source BINSEC.

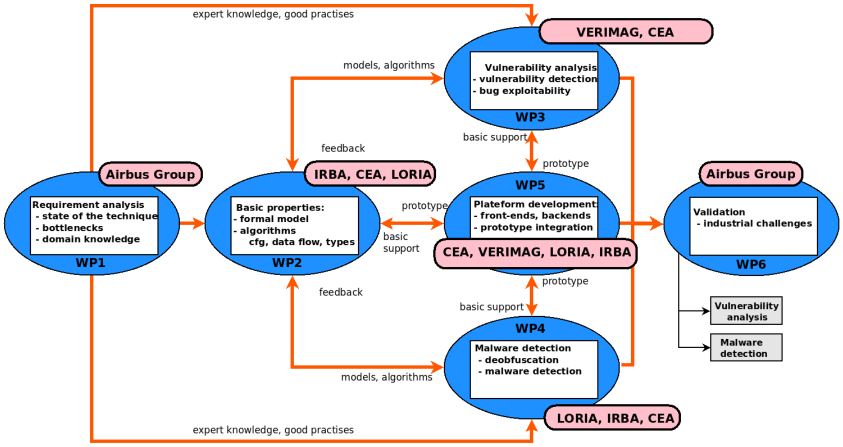


Figure 1 – Aperçu du projet

Quelques résultats. Les résultats du projet jusqu'à ce jour incluent : une représentation intermédiaire formelle et concise pour l'analyse de code binaire [5], basée entre autre sur un nouveau modèle mémoire à régions "bas niveau" [1, 2] raffinant le modèle usuel "à la CompCert" [7] ; une technique originale et un prototype de détection de "use-after-free" sur code exécutable [6] ; une technique et un prototype de désassemblage de code obscurci par auto-modification et chevauchement d'instruction [3] ; une plate-forme open-source d'analyse de code binaire [5] proposant des moteurs originaux d'analyse statique (reconstruction sûre de graphe de contrôle) et d'exécution symbolique [4] (exploration partielle précise). La plate-forme sera disponible à partir de juin 2016 (`http://binsec.gforge.inria.fr`) .

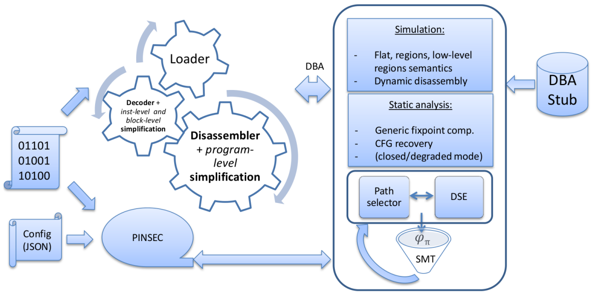


FIGURE 2 – Plate-forme Binsec

Participants. Sébastien Bardin, Frédéric Besson, Sandrine Blazy, Guillaume Bonfante, Richard Bonichon, Robin David, Adel Djoudi, Benjamin Farinier, Josselin Feist, Colas Le Guernic, Jean-Yves Marion, Laurent Mounier, Marie-Laure Potet, Than Dihnh Ta, Franck Védrine, Pierre Wilke, Sara Zennou.

Références

[1] Blazy S., Besson F., Wilke P. : A Precise and Abstract Memory Model for C using Symbolic Values. In : APLAS 2014. Springer, Heidelberg (2014)

[2] Besson F., Blazy S., Wilke P. : A Concrete Memory Model for CompCert. In : ITP 2015. Springer, Heidelberg (2015)

[3] G. Bonfante, J. Fernandez, JY. Marion, B. Rouxel, F. Sabatier, A. Thierry : Co-Disasm : Concatic disassembly of self-modifying binaries with overlapping. In : CCS 2015. ACM (2015)

[4] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In : SANER '16. IEEE, 2016

[5] A. Djoudi and S. Bardin. BINSEC : Binary code analysis with low-level regions. In : TACAS '15. Springer, 2015

[6] J. Feist, L. Mounier and M.L. Potet : Statically detecting Use-After-Free on Binary Code. Journal of Computer Virology and Hacking Techniques, 2014.

[7] Leroy, X., Appel, A.W., Blazy, S., Stewart, G. : The CompCert memory model. In : Program Logics for Certified Compilers. Cambridge University Press (2014)

Combiner des diagrammes d'état étendus et la méthode B pour faciliter la validation de systèmes industriels

Thomas Fayolle
LACL, Université Paris Est
GRIL, Université de Sherbrooke
Ikos Consulting, Levallois Perret

11 mars 2016

1 Introduction

Dans le cadre industriel, les systèmes sont développés par des ingénieurs systèmes spécialisés en partenariat avec des ingénieurs métier. Lorsque le système est critique, l'utilisation de méthodes formelles est fortement recomandée pour son développement. Les méthodes formelles utilisent des notations mathématiques parfois complexes, qui ne facilitent pas le dialogue entre ceux qui connaissent le fonctionnement du système et ceux qui connaissent le langage pour l'exprimer. Pour faciliter ce dialogue, une solution peut être d'utiliser un langage de spécification graphique.

Les ASTD [2](Algebraic State Transition Diagram) sont un langage de spécification qui combine les diagrammes d'état [5] et des opérateurs d'algèbre de processus [4]. Ils permettent de décrire graphiquement le comportement d'un système. Ils peuvent donc apporter une solution pour la validation de spécifications. Cependant, ils ne permettent de modéliser que l'ordonnancement des opérations. Pour décrire l'effet de ces opérations sur les données du système, nous proposons de les combiner au langage B.

2 Méthodologie

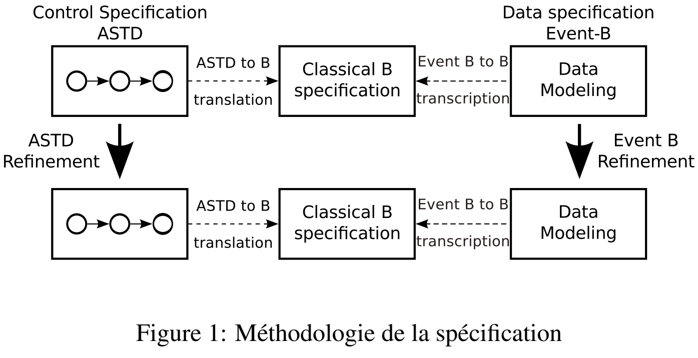


Figure 1: Méthodologie de la spécification

Notre méthode de spécification combine le langage ASTD et le langage Event-B. Cette méthode est résumée à la figure 1. Le système est modélisé en deux parties. Son comportement dynamique est décrit dans la première partie en utilisant les ASTD (spécification à gauche sur la figure 1). La deuxième partie en Event-B décrit les modifications apportées au modèle de données (spécification à droite sur la figure 1). Elle contient un événement pour chaque label de la spécification ASTD. Lorsque la transition est effectuée, le modèle de données est modifié en suivant la spécification de l'événement Event-B. La spécification en B classique (partie centrale de la figure 1) permet d'assurer la cohérence du système.

2.1 La spécification ASTD

La spécification ASTD modélise l'ordonnancement des opérations en utilisant des notations graphiques des diagrammes d'état et des opérateurs des algèbres de processus. La spécification graphique facilite la validation tout en restant formelle. L'ajout des opérateurs des algèbres de processus permet de modéliser des systèmes dans lesquels plusieurs processus fonctionnent en parallèle, comme c'est le cas dans beaucoup de systèmes industriels.

2.2 La spécification Event-B

La spécification ASTD modélise l'ordre des opérations, tandis que la spécification Event-B modélise les effets de chaque opération sur le modèle de données. Elle contient un événement pour chaque label de transition de la spécification ASTD. Les invariants Event-B traduisent les propriétés statiques, notamment les propriétés de sûreté. Les outils de la plateforme RODIN [1] permettent de générer et de prouver les obligations de preuve associées à la préservation des invariants. Au contraire du raffinement de la méthode B, le raffinement de la méthode Event-B permet l'ajout de nouveaux événements. Pour coller au raffinement des ASTD, c'est ce langage qui a été choisi pour la spécification de la partie donnée.

2.3 Spécification B

Pour un niveau de spécification, la modélisation en B classique contient deux machines. La première est une traduction de la spécification ASTD, la seconde est une transcription de la spécification Event-B.

Un outil effectue la traduction automatique de la spécification ASTD. Cette traduction est effectuée suivant les règles définies dans [6]. Le mécanisme de la traduction peut être résumé de la manière suivante. Les états des ASTD sont encodés par une variable d'état B. Une opération B est écrite pour chaque label de la spécification ASTD. La précondition de cette opération vérifie que les variables d'état sont bien dans l'état initial de la transition. La postcondition met à jour les variables d'état de manière à ce qu'elles correspondent à l'état final de la transition.

De plus, nous souhaitons que les variables du modèle de données soient modifiées lors de chaque transition. Puisqu'il n'est pas possible d'appeler un événement Event-B dans une spécification en B classique, la spécification Event-B est traduite en B. Les variables d'état et les invariants de typage sont conservés. Les événement sont réécrits en opération, la garde devient une précondition et la postcondition reste identique.

[1] http://event-b.org

Regrouper les deux spécifications en une seule nous permet de vérifier la cohérence globale du système (c'est-à-dire un niveau de spécification horizontal sur la figure 1). En effet, les propriétés statiques de la spécification Event-B ne sont vérifiées que si les événements sont exécutés quand leurs gardes sont vraies. La génération des obligations de preuves de la méthode B oblige à vérifier que les préconditions des opérations appelées sont vraies au moment de l'appel. La preuve de ces obligations garantit que les événements du modèle de données sont toujours exécutés quand leur garde est vraie. Pour prouver ces obligations, il est nécessaire d'ajouter des invariants qui lient les variables du modèle de données aux états de l'ASTD.

2.4 Raffinement

Le raffinement du système est effectué en parallèle sur les deux modèles. La spécification de l'ordonnancement des opérations est effectuée en suivant la définition du raffinement pour les ASTD. Cette définition (donnée dans [3]) permet de garantir la préservation des traces. Le raffinement du modèle de données suit la définition du raffinement Event-B.

3 Conclusion

La méthode présentée dans cet article permet de modéliser graphiquement un système critique ce qui facilite la validation du système dans un domaine industriel. La méthode a été testée sur un exemple dans le cadre de la modélisation d'un système ferroviaire [1].

La modélisation de ce cas a permis de voir l'efficacité de la méthode. Cependant, la preuve de la cohérence globale du système s'est montrée fastidieuse. Des travaux sont en cours pour simplifier les règles automatiques de traduction et définir des règles et des outils de raffinement pour les ASTD.

References

[1] T. Fayolle. Specifying a Train System Using ASTD and the B Method. Technical report, 2014. http://www.lacl.fr/~tfayolle.

[2] M. Frappier, F. Gervais, R. Laleau, B. Fraikin, and R. Saint-Denis. Extending Statecharts with Process Algebra Operators. *Inovation in System Software Engeneering*, Volume 4, Number 3:285–292, 2008.

[3] M. Frappier, F. Gervais, R. Laleau, and J. Milhau. Refinement patterns for ASTDs. *Formal Aspects of Computing*, pages 1–23, 2012.

[4] M. Frappier and R. St-Denis. EB3 : an entity-based black-box specification method for information systems. *Software and Systems Modeling*, 2(2):134–149, July 2003.

[5] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8:231–274, 1987.

[6] J. Milhau. *A formal integration of access control policies into information systems*. Theses, Université Paris-Est ; Université de Sherbrooke, December 2011.

Vers un développement formel non incrémental

Thi-Kim-Dung Pham[1,3], Catherine Dubois[2], Nicole Levy[1]

1. Conservatoire National des Arts et Métiers, lab. Cedric, Paris

2. ENSIIE, lab. Samovar, Évry

3. University of Engineering and Technology, Vietnam National University, Hanoi

Résumé

Modularité, généricité, héritage sont des mécanismes qui facilitent le développement et la vérification formelle de logiciels corrects par construction en permettant de réutiliser des spécifications, du code et/ou des preuves. Cependant les lignes de produits exploitent d'autres techniques de réutilisation ou de modification graduelle. Les méthodes formelles permettant la production de code correct par construction (en B ou FoCaLiZe par exemple) ne sont pas bien adaptées à la variabilité telle qu'elle apparaît dans les lignes de produits. Nous proposons d'approcher ce problème par la définition d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits permettant de spécifier, implanter et prouver. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Cet article illustre par l'exemple une des constructions offertes par GFML ainsi que sa traduction en FoCaLiZe.

1 Introduction

Nous aimerions construire des programmes corrects par construction, c'est-à-dire qui répondent au cahier des charges, sont cohérents et complets. Il est effectivement possible de le faire à l'aide de certaines méthodes qui utilisent des langages formels tels que B [1] ou FoCaLiZe [7]. Il faut alors définir les propriétés que le programme doit vérifier et utiliser des outils de preuve pour prouver que l'on a bien suivi toutes les étapes de la méthode.

Un des principaux problèmes de ces méthodes réside dans la difficulté à les appliquer. On pourrait espérer que les méthodes de réutilisation facilitent le développement. Il existe plusieurs manières de réutiliser un programme. Par exemple, les méthodes orientées objets qui proposent de compléter ou de modifier, par héritage, une classe d'objets. Le langage FoCaLiZe possède l'héritage et permet ainsi de réutiliser des spécifications, des programmes et des preuves.

Mais ce n'est pas suffisant. On aimerait réutiliser une fonctionnalité d'un programme mais pour l'utiliser dans un contexte différent. Il existe des mécanismes permettant de décrire les propriétés d'un contexte, indispensables pour un programme, sous la forme de paramètres de ce programme. En FoCaLiZe, les composants s'appellent des espèces (*species*) et peuvent être paramétrés par des espèces représentant des objets ou des propriétés du contexte requis.

Mais ce n'est pas encore suffisant. L'utilisation de la paramétrisation se révèle assez difficile et restrictive.

En fait, pour réutiliser un programme (ou une partie d'un programme), il faut savoir comment il a été construit et connaître les décisions prises lors de son développement.

Le problème qui se pose est l'expression des savoirs-faire et leur réutilisation. La réalité, c'est qu'il est bien difficile de décrire ou de formaliser ce savoir-faire. Mais c'est un des avantages des approches lignes de produits [4] qui permettent de décrire la succession des décisions à prendre et d'associer à chaque décision une solution concrète sous la forme d'un morceau de code (*asset*). En général, les décisions prises sont constructives : il s'agit d'ajouter une information, une fonctionnalité, une propriété, etc. Dans ce cas, la composition des morceaux peut être définie simplement.

Mais qu'en est-il lorsque la décision prise à une étape est de modifier l'étape précédente et non seulement de rajouter quelque chose ? Comment formaliser des méthodes de réutilisation non incrémentales ?

Il existe des approches telle que celle de Thomas Thüm [13, 12] qui utilise les lignes de produits pour développer des programmes Java selon une approche par contrats à l'aide de pré et post-conditions. Il utilise le langage JML pour écrire les contrats et les traduit ensuite en Coq pour les vérifier. C'est de là qu'est venue l'idée d'introduire un langage dédié pour décrire la construction des modifications à apporter au code.

Notre proposition est de faciliter de telles étapes de développement, permettant de modifier une étape précédente, c'est-à-dire de modifier le code préalablement défini, et prouvé. L'idée est d'introduire un langage d'expression de ces modifications et un compilateur réalisant la modification à opérer et générant le code. Le langage que nous proposons s'appelle GFML. Couplée à une approche lignes de produits, notre démarche permet de décrire des savoirs-faire qui ne sont pas purement incrémentaux mais peuvent passer par des étapes de remise en cause de développements préalables, tout en bénéficiant de l'approche de développement de code correct par construction.

Nous avons choisi d'illustrer notre démarche dans le cadre proposé par FoCaLiZe. En effet, il permet de définir à la fois spécifications, code et preuves, donc tous les artefacts associés à une caractéristique d'une ligne de produits. D'autre part, FoCaLiZe admet pour seul mécanisme de raffinement, l'enrichissement par héritage, ce qui laisse beaucoup de liberté.

La suite de cet article est structurée de la manière suivante. Dans un premier temps nous allons étudier les constructions existantes en FoCaLiZe qui permettent de suivre un développement formel incrémental. Nous signalerons aussi les limites de ce modèle. Nous montrons à l'aide d'un exemple de compte bancaire un exemple de modification souhaitée, à savoir le renforcement d'une fonctionnalité tout en conservant au maximum les preuves déjà faites. Cette modification peut être exprimée en FoCaLiZe à l'aide de l'utilisation conjointe de l'héritage et de la paramétrisation. Nous présentons ensuite l'expression de cette modification dans notre langage GFML et sa traduction en FoCaLiZe. Un avantage de cette approche est le fait qu'elle s'adapte très bien aux lignes de produits puisque la composition des morceaux associés à chaque étape entraîne une suite de modifications et non seulement des enrichissements. L'approche générale est présentée dans [8], nous détaillons ici, sur un exemple, une des constructions offertes par GFML et sa compilation en FoCaLiZe.

2 Le modèle FoCaLiZe

L'environnement FoCaLiZe (anciennement FoCal) [7, 6] permet le développement de programmes certifiés pas à pas, de la spécification à l'implantation. Cet environnement propose un langage également nommé FoCaLiZe et des outils d'analyse de code, de preuve (Zenon) et des compilateurs vers Ocaml, Coq et Dedukti [5]. Le langage FoCaLiZe permet d'écrire des propriétés en logique du premier ordre, des signatures et des fonctions dans un style fonctionnel et enfin des preuves dans un style déclaratif. Ce langage offre également des mécanismes inspirés par la programmation orientée objets, comme l'héritage, la liaison retardée et la redéfinition afin de faciliter la modularisation et la réutilisation. FoCaLiZe permet d'hériter de spécifications, de code mais aussi de preuves [9]. Un mécanisme de paramétrisation vient compléter l'ensemble.

Les constructions de FoCaLiZe [3] permettent d'ajouter à une espèce, une signature, une définition, une propriété, une preuve, de définir une représentation concrète pour les objets manipulés, de redéfinir une fonction. Dans ce dernier cas, le calcul des dépendances fait par le compilateur détermine si les preuves déjà effectuées concernant la fonction redéfinie doivent être refaites ou non. Ces primitives assurent une construction incrémentale des spécifications, des programmes et des preuves. Dans le modèle actuel de FoCaLiZe, par héritage, les spécifications d'une fonction redéfinie sont héritées, elles peuvent être complétées par de nouvelles propriétés mais elles ne peuvent pas être *redéfinies* (en restreignant par exemple la précondition comme dans la section suivante), voire supprimées comme on le désirerait par exemple pour développer les fonctionnalités du logiciel en mode dégradé. La redéfinition mise en œuvre dans FoCaLiZe impose également de conserver l'arité d'une fonction, il n'y a donc pas de possibilité de surcharge. Enfin, la représentation d'un type (mot-clé `representation` dans les exemples de la section 3), une fois définie dans un composant C, ne peut être redéfinie dans les composants qui héritent de C. Ceci conduit à un choix tardif de cette représentation. Mais dans le cadre d'un développement non incrémental, on peut être amené à modifier la représentation choisie, par exemple en ajoutant des attributs à la manière des langages orientés objets. Une étude similaire a été proposée dans le cadre de la programmation orientée *feature* et de la programmation par contrats par Thüm et al. dans [11]. Relativement à la classification proposée dans cet article, le modèle actuel de FoCaLiZe, en cas d'héritage, met en œuvre le raffinement de spécifications par sous-typage. Avec l'exemple ci-dessous, nous nous rapprochons du raffinement dit explicite et du raffinement par surcharge de contrats.

3 Raffinement d'une propriété

Nous utilisons dans la suite l'exemple (inspiré de [13]) de la ligne de produits concernant la gestion de comptes bancaires représentée graphiquement (par son diagramme de caractéristiques ou *features*) à la figure 1. La feature racine, BankAccount (ou BA), définit le fonctionnement de base d'un compte : calculer le solde d'un compte, débiter et créditer un compte. Le débit n'est autorisé que si le compte reste créditeur d'une certaine somme (ici `over`). A ces fonctionnalités élémentaires peuvent être ajoutées, de manière optionnelle, les fonctionnalités dénotées par les features DailyLimit (DL), LowLimit (LL) et Currency. DL introduit une limitation pour les retraits. Dans la suite, seuls BA et DL sont utilisés.

A chaque feature de l'arbre est associé un fichier FoCaLiZe contenant des spécifications, du code et des preuves. Les artefacts associés au feature racine, BA, définissent le noyau minimal. Le fichier FoCaLiZe associé contient trois espèces données ci-dessous, `BA_spec0`, `BA_spec` et `BA_impl`, les deux premières contiennent la spécification et la dernière, l'implantation des fonctions ainsi que les preuves des propriétés énoncées dans les autres espèces. Comme nous le verrons plus tard, la séparation de la spécification en deux espèces facilitera la définition des autres features.

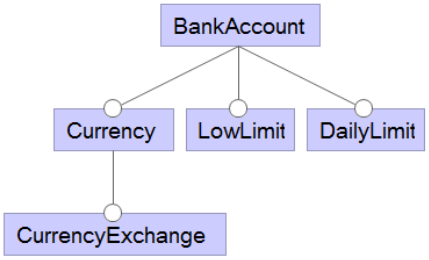


FIGURE 1 – Ligne de produits des comptes bancaires

```
species BA_spec0 =
signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;
property ba_over : all  b: Self,  balance(b) >= over;
end ;;

species BA_spec =
inherit BA_spec0;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;
end ;;

species  BA_impl =
inherit BA_spec;
representation = int;
let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;
proof of ba_update = by definition of update, balance ;
proof of ba_over = assumed ;
end ;;
```

Dans BA_spec0 sont introduites les signatures de deux fonctions, balance et update et une constante over. La fonction balance permet de consulter le solde du compte passé en argument, update permet de créditer/débiter le compte passé en argument, le résultat est le nouveau compte. La propriété/spécification ba_over est une propriété invariante, elle indique que tout compte doit avoir un solde supérieur ou égal à over. Cette propriété doit être vérifiée dans tout produit dérivé de cette ligne de produits. La deuxième espèce hérite des signatures et des propriétés de la première espèce et ajoute une nouvelle propriété spécifiant la mise à jour d'un compte, en particulier les conditions de succès de cette opération. Enfin, l'espèce BA_impl réalise la spécification. Elle précise la représentation des comptes bancaires. Ici, un compte est représenté par son solde, soit un entier. Et donc l'accès au solde d'un compte (la fonction balance) est l'identité. La preuve de la propriété ba_update énoncée précédemment est faite très simplement par le prouveur automatique Zenon [2]. Il suffit de lui indiquer d'utiliser les définitions des fonctions update et balance. La preuve de la propriété ba_over est, quant à elle, admise. En fait, elle ne peut être faite directement mais elle doit être remplacée par la preuve que cette propriété est préservée par chacune des fonctions (dont le résultat a le type Self) de l'espèce [10].

La caractéristique DL permet de définir les fonctionnalités des comptes bancaires avec une limite de retrait. Ainsi on ajoute une nouvelle constante limit_withdraw. Cette extension demande une redéfinition de la fonction update, ce que permet le langage FoCaLiZe. En cas de retrait (argument négatif), la fonction va vérifier que l'on est bien en deçà de la limite de retrait. Cependant la propriété ba_update n'est plus vraie dans le contexte de DL. Nous avons ici une propriété plus restreinte. Nous définissons deux propriétés par renforcement des pré-conditions de ba_update, ce qui revient à spécifier séparément les actions créditer et débiter. Nous appelons ce schéma, un raffinement de propriété : il se limite à la redéfinition d'une propriété en ajoutant une prémisse. Du fait de cette modification de comportement, on peut définir la spécification de DL en héritant de BA_spec0 mais on ne peut hériter de BA_spec car la propriété ba_update doit être adaptée. C'est la raison pour laquelle nous avions scindé en deux espèces la spécification de BA.

L'idée est de réutiliser le code existant et les preuves existantes. Le super tel qu'on le trouve en Java (ou original en programmation par aspects) n'existe pas en FoCaLiZe, il va donc falloir le simuler en utilisant héritage et paramétrisation.

La caractéristique DL est donc définie elle aussi en trois espèces : la première introduit la limite de retrait et hérite des spécifications réutilisables de BA, i.e. l'espèce BA_spec0, la deuxième définit les propriétés obtenues par raffinement et enfin la dernière contient les définitions des fonctions et les preuves des propriétés.

```
species DL_spec0 =
inherit BA_spec;
signature limit_withdraw : nat;
end ;;

species DL_spec =
inherit DL_spec0;
property ba_update_ref1 : all b:Self, all a: int,
  a >= 0 -> balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;

property ba_update_ref2 : all b:Self, all a: int,
  a < 0 -> (0 - a) <= limit_withdraw ->
    balance(b) + a >= over ->
      balance(update(b, a)) = balance(b) + a;
end ;;

species DL_impl (M is BA_impl) =
inherit DL_spec;
representation = M;
let balance(b) = M!balance (b) ;
let over = M!over ;

proof of ba_over =
  by definition of balance, over property M!ba_over ;
let update(b,a) =
  if a < 0 then
    if (0 - a) <= limit_withdraw then M!update (b, a)  else b
  else M!update(b, a);

proof of ba_update_ref1 =
  by definition of update, balance, over
     property M!ba_update int_ge_le_eq  ;

proof of ba_update_ref2 =
  by definition of update, balance, over
     property M!ba_update ;
end ;;
```

L'espèce `DL_impl` a été écrite de manière à réutiliser autant que possible le code et les preuves déjà faites dans le cadre de la caractéristique BA.

4 De GFML à FoCaLiZe

L'approche que nous proposons consiste à décrire à l'aide du langage GFML, les modifications à réaliser, c'est-à-dire les seules informations qui différencient DL de BA. Le module GFML correspondant à DL, décrit ci-dessous, permet de générer automatiquement les espèces de DL, soient `DL_spec0`, `DL_spec` et `DL_impl`. On remarquera que les preuves sont faites dans le format accepté par FoCaLiZe (cette partie est en fait recopiée textuellement par le compilateur GFML vers FoCaLiZe). On peut également noter que la syntaxe de GFML est largement inspirée de celle de FoCaLiZe.

```
fmodule DL from BA

signature limit_withdraw : nat;

property ba_update_ref1 refines BA!ba_update
extends premise (a >= 0);
property ba_update_ref2 refines BA!ba_update
extends a < 0 ∧ -a <= limit_withdraw;

let update(b,a) =
if a < 0
  then if -a <= limit_withdraw then BA!update (b, a) else b
  else BA!update(b, a);

proof of ba_update_ref1 =
{* focalizeproof =  by definition of update, balance, over
     property M!ba_update, int_ge_le_eq  ; *}
proof of ba_update_ref2 =
{* focalizeproof =  by definition of update, balance, over
     property M!ba_update ; *}
end ;;
```

La caractéristique racine BA est également décrite à l'aide d'un module GFML, détaillé ci-dessous, qui rassemble les signatures, définitions et preuves. La compilation de ce module en FoCaLiZe produit exactement les trois espèces `BA_spec0`, `BA_spec` et `BA_impl`.

```
fmodule BA

signature update : Self -> int -> Self;
signature balance : Self -> int;
signature over : int;
```

```
invariant property ba_over : all  b: Self,  balance(b) >= over;
property ba_update : all b:Self, all a: int,
  balance(b) + a >= over ->
    balance(update(b, a)) = balance(b) + a;

representation = int;

let balance(b) = b;
let update(b,a) = if b + a >= over then b + a else b;

proof of ba_update =
  {* focalizeproof = by definition of update, balance ;*}
proof of ba_over = assumed ;
end ;;
```

5 Conclusion

Dans cet article, nous avons présenté une approche de la construction non incrémentale de logiciels à l'aide d'un langage formel, GFML, proche de la variabilité mise en œuvre dans les lignes de produits et permettant de spécifier, implanter et prouver les étapes de développement par ajout ou modification. Ce langage est compilé vers un formalisme existant, ici FoCaLiZe. Nous avons illustré notre approche par l'exemple en ne présentant qu'une des constructions offertes par GFML et avons explicité sa traduction en FoCaLiZe.

GFML est un langage formel dont la sémantique est donnée par sa traduction en FoCaLiZe. Le compilateur de GFML vers FoCaLiZe est en cours de réalisation, en même temps que la définition de différents opérateurs de développement liés à l'approche lignes de produits. En plus de l'exemple donné d'ajout d'une pré-condition, citons l'ajout ou la modification d'un paramètre d'une fonction qui va avoir un impact partout où est appelée cette fonction. Les modifications calculées ont un impact sur les preuves déjà réalisées. L'opérateur décrit le travail à réaliser pour obtenir une nouvelle version prouvée du logiciel. Ainsi, l'ambition de GFML est de proposer aux concepteurs de lignes de produits, un langage permettant de décrire les modifications à apporter à une première version d'une caractéristique pour prendre en compte une nouvelle idée. Les constructions de GFML représentent des décisions et le compilateur les prend en compte pour générer le code correspondant.

Remerciements. Nous remercions François Pessaux pour son aide lors du développement du traducteur.

Références

[1] J. Abrial. On constructing large software systems. In J. van Leeuwen, editor, *Algorithms, Software, Architecture - Information Processing '92, Volume 1, Proceedings of the IFIP 12th World Computer Congress, Madrid, Spain*, volume A-12 of *IFIP Transactions*, pages 103–112. North-Holland, 1992.

[2] R. Bonichon, D. Delahaye, and D. Doligez. Zenon : An extensible automated theorem prover producing checkable proofs. In N. Dershowitz and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007,*

Proceedings, volume 4790 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 2007.

[3] S. Boulmé. *Spécification d'un environnement dédié à la programmation certifiée de bibliothèques de Calcul Formel*. Thèse de doctorat, Université Paris 6, 2000.

[4] P. Clements and L. Northrop. *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, 2001.

[5] Dedukti. `http://dedukti.gforge.inria.fr/`.

[6] C. Dubois, T. Hardin, and V. Donzeau-Gouge. Building certified components within FOCAL. In H. Loidl, editor, *Revised Selected Papers from the Fifth Symposium on Trends in Functional Programming, TFP 2004*, volume 5 of *Trends in Functional Programming*, pages 33–48, 2006.

[7] FoCaLiZe. `http://focalize.inria.fr/`.

[8] T.-K.-Z. Pham, C. Dubois, and N. Levy. Towards correct-by-construction product variants of a software product line : Gfml, a formal language for feature modules. In Proceedings 6th Workshop on *Formal Methods and Analysis in SPL Engineering,* London, UK, 11 April 2015, volume 182 of *EPTCS*, pages 44–55, 2015.

[9] V. Prevosto and D. Doligez. Algorithms and proofs inheritance in the FOC language. *J. Autom. Reasoning*, 29(3-4) :337–363, 2002.

[10] R. Rioboo. Invariants for the focal language. *Ann. Math. Artif. Intell.*, 56(3-4) :273–296, 2009.

[11] T. Thüm, I. Schaefer, M. Kuhlemann, S. Apel, and G. Saake. Applying design by contract to feature-oriented programming. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 255–269. Springer, 2012.

[12] T. Thüm. Product-line verification with feature-oriented contracts. In M. Pezzè and M. Harman, editors, *Int'l Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 374–377. ACM, 2013.

[13] T. Thüm, I. Schaefer, M. Kuhlemann, and S. Apel. Proof composition for deductive verification of software product lines. In *Proc. Int'l Workshop Variability-intensive Systems Testing, Validation and Verification (VAST'11)*, pages 270–277. IEEE Computer Society, 2011.

Session du groupe de travail MTV^2

Méthodes de test pour la validation et la vérification

Quelle confiance peut-on établir dans un système intelligent ?

Lydie du Bousquet[1,2] , Masahide Nakamura[3]
[1]Univ. Grenoble Alpes, LIG, F-38000 Grenoble, France
[2]CNRS, LIG, F-38000 Grenoble, France
[3]Kobe University, Japon
e-mail : Lydie.du-Bousquet@imag.fr, masa-n@cs.kobe-u.ac.jp

Résumé

De plus en plus, nous sommes confrontés à des systèmes intelligents (“*smart systems*”). Ce sont des systèmes supposés apprendre de leur environnement pour apporter des réponses les plus pertinentes possibles pour leurs utilisateurs. Une caractéristique de ces systèmes est que leurs comportements évoluent avec le temps. Dans le projet CNRS-PICS *Safe-Smartness*, collaboration entre le LIG et l’université de Kobe, nous étudions comment exprimer les propriétés d’oracle quant à la qualité d’un système intelligent.

Introduction Les systèmes intelligents (*smart systems*) sont caractérisés par leur capacité à prédire ou à adapter leurs comportements à partir d’une analyse d’un ensemble de données. Ce type d’application se retrouve dans différents domaines (domotique, médical, transport, ...). Selon la *Royal Academy of Engineering*, on peut classer les systèmes intelligents en trois catégories [1]. Ainsi, selon la façon dont les données sont collectées et utilisées, ces systèmes permettent :

- d’aider les concepteurs à produire une nouvelle version de l’application plus efficace,
- de calculer et de présenter des informations à l’opérateur humain pour qu’il puisse prendre des décisions pertinentes par rapport à la situation,
- de choisir les actions à effectuer en fonction de la situation et de les exécuter, sans intervention humaine.

En ce qui concerne les deux dernières catégories, la question de la qualité du système prend toute son importance. Un tel système ne doit pas induire ses utilisateurs en erreur et/ou ne doit pas exécuter des actions inappropriées. L’objet du projet CNRS-PICS *Safe-Smartness* est d’établir des méthodes de validation pour ce type de systèmes. Dans la suite, illustrons les difficultés sur la base d’un exemple.

Exemple Soit un système de climatisation programmable “intelligent”. L’utilisateur peut programmer une heure `h` et une température `t`. Le système doit alors faire

[1] http ://www.raeng.org.uk/publications/reports/smart-infrastructure-the-future

en sorte que la pièce considérée soit à la température `t` à l'heure `h`, tout en minimisant l'énergie nécessaire pour y parvenir. Pour cela, le système doit tenir compte de l'inertie thermique, qui est propre à chaque pièce. Le système est donc doté d'une partie "intelligente", qui enregistre un certain nombre de facteurs et "apprend" la vitesse à laquelle la pièce se tempère en fonction de ces facteurs. Après le temps d'apprentissage, le système doit choisir au mieux à quel moment démarrer pour atteindre la température `t` à l'heure fixée.

Propriétés attendues du système Un des premiers éléments à vérifier est donc que la pièce soit à la température `t` à l'heure `h`. Comme cela dépend de la pièce, il faut envisager de le faire à chaque exécution (*monitoring*). Pour autant, la non-satisfaction de cette propriété ne signifie pas forcément que le système est incorrect. Deux situations peuvent se produire. (a) L'utilisateur peut fixer un objectif impossible à atteindre ; par exemple demander à 7h55 que la pièce soit à 21 ˚C à 8h alors que la température actuelle est de 10 ˚C. (b) L'objectif peut aussi être atteignable en théorie, mais l'environnement réel ne pas le permettre ; par exemple, si une fenêtre est laissée ouverte en plein hiver.

Dans ces conditions, le système doit être en mesure de fournir une notification si l'objectif est jugé non-atteignable (e.g. 7h55 pour 8h et 11 ˚C d'écart). Il sera alors possible de valider si la notification est exacte (i.e. pertinente). La présence de faux-négatifs (notification "impossible" mais la température est atteinte dans les délais) et de faux-positifs (notification "possible" mais la température n'est finalement pas atteinte dans les délais) est un indicateur de qualité du système .

Le système doit aussi être en mesure d'indiquer si les caractéristiques de l'environnement ont changé (e.g. inertie différente si fenêtre ouverte). Cela est nécessaire pour faire la différence entre un dysfonctionnement du système et une anomalie ponctuelle d'usage.

Par ailleurs, on attend du système qu'il minimise l'énergie nécessaire pour atteindre l'objectif. Ici, on peut par exemple s'assurer que le système ne démarre pas trop tôt, c'est-à-dire qu'à partir du moment où le climatiseur commence à chauffer (resp. à refroidir) la pièce, il ne passe pas plusieurs fois en attente.

Mais ces propriétés ne permettent pas de s'assurer directement que le système *apprend* correctement. L'apprentissage est un processus progressif. Pour décider de la qualité de l'apprentissage, il faut observer les exécutions successives et s'assurer que l'objectif est atteint de plus en plus souvent, et/ou de mieux en mieux (ici, en consommant de moins en moins d'énergie). La difficulté ici est qu'il s'agit d'une tendance, qui peut se dégrader de temps à autre, en particulier si l'utilisateur change ses habitudes. Le fait que la tendance se dégrade pendant un temps ne signifie donc pas forcément que le système est incorrect, mais peut caractériser le fait qu'il entre dans une nouvelle étape d'adaptation.

Les outils habituels d'expression et d'évaluation de propriétés ne sont pas adaptés pour nos besoins. C'est pourquoi, l'objectif du projet *Safe-Smartness* est de proposer des moyens alternatifs pour exprimer et valider la qualité de l'apprentissage.

Génération systématique de scénarios d'attaques contre des systèmes industriels

Maxime Puys, Marie-Laure Potet, and Jean-Louis Roch

VERIMAG, Univ. Grenoble Alpes / Grenoble-INP, France
prénom.nom@imag.fr[⋆]

Résumé Les systèmes industriels (SCADA) sont la cible d'attaques informatiques depuis Stuxnet [4] en 2010. De part leur interaction avec le mode physique, leur protection est devenue une priorité pour les agences gouvernementales. Dans cet article, nous proposons une approche de modélisation d'attaquants dans un système industriel incluant la production automatique de scénarios d'attaques. Cette approche se focalise sur les capacités de l'attaquant et ses objectifs en fonction des protocoles de communication auxquels il fait face. La description de l'approche est illustrée à l'aide d'un exemple.

1 Introduction

Les systèmes industriels aussi appelés SCADA (Supervisory Control And Data Acquisition) sont la cible de nombreuses attaques informatiques depuis Stuxnet [4] in 2010. De part leur interaction avec le monde physique, ces systèmes peuvent représenter une réelle menace pour leur environnement. Suite à une récente augmentation de la fréquence des attaques, la protection des installations industrielles est devenue une priorité des agences gouvernementales. Ces systèmes diffèrent également de l'informatique de gestion du fait de leur très longue durée de vie, de leur difficulté à appliquer des correctifs et des protocoles souvent spécifiques utilisés.

État de l'art : La modélisation et la génération de scénarios d'attaques sont essentielles à la sécurité des systèmes industriels. En 2013, la norme IEC 62443-3-3 [2] détaille de façon très précise une méthode d'analyse de la sécurité informatique des installations industrielles. En 2015, Conchon *et al.* [1] proposent une approche se basant sur EBIOS. Toujours en 2015, Kriaa *et al.* [3] décrivent S-CUBE, une approche de modélisation de systèmes industriels faisant le pas entre la sécurité et la sûreté de fonctionnement. Leur article analyse un grand nombre d'approches de production de scénarios d'attaques et conclut notamment qu'aucune n'est automatisée ni facilement automatisable.

[⋆]. Ce travail a été partiellement financé par le LabEx PERSYVAL-Lab (ANR–11-LABX-0025) et le projet Programme Investissement d'Avenir FSN AAP Sécurité Numérique n^o 3 ARAMIS (P3342-146798).

Contributions : Nous proposons dans cet article une approche de modélisation d'attaquants basée à la fois sur l'infrastructure et les possibilités d'attaques dans un système industriel. Cette approche se base sur les composants d'une infrastructure et les canaux de communication entre ses composants [6]. Elle vise la production automatique de scénarios d'attaques. Nous nous focalisons sur l'attaquant en modélisant sa position dans l'infrastructure, ses objectifs d'attaques et les protocoles qu'il peut utiliser en tenant compte de leurs propriétés de sécurité. Enfin, à l'aide d'une approche systématique, nous générons l'ensemble des scénarios d'attaques pour lesquels un attaquant est capable de réaliser l'un de ses objectifs face à un protocole donné.

Plan : La section 2 détaille la modélisation des attaquants et la production des scénarios d'attaques à l'aide d'un un exemple. Ensuite, la section 3 décrit comment nous souhaitons inclure cette approche dans une approche *Model-Based Testing* plus globale. Enfin la section 4 conclut.

2 Approche de génération des scénarios d'attaques

Dans cette section, nous détaillons le formalisme utilisé dans notre modélisation des attaquants et comment exploiter cette modélisation pour générer des scénarios d'attaques. Cette approche est double. Elle propose dans un premier temps d'étudier les objectifs des attaquants et comment ils pourraient les réaliser (approche descendante) avant de les mettre dans un second temps face aux protections apportées par les protocoles de communication (approche ascendante).

2.1 Approche descendante

Nous commençons par définir l'ensemble des attaquants $\mathcal{A}$ et l'ensemble des objectifs d'attaques $\mathcal{O}$. Ces ensembles sont liés par la relation $\mathcal{R}_{Obj}$ entre un attaquant a et un objectif o tel que $\mathcal{R}_{Obj} \subseteq \mathcal{A} \times \mathcal{O}$ si a cherche à atteindre o fait partie de notre analyse de risque. Il va donc de soit que cette relation, fournie par le concepteur du modèle doit tenir compte de la position des attaquants dans l'architecture globale du système (ex. : un attaquant ne peut pas avoir pour objectif la modification d'un message s'il n'y a jamais accès).

——————————— Exemple : ———————————

Nous considérons l'infrastructure de communication en figure 1 où les attaquants (en couleur) sont $\mathcal{A} = \{Client_A, Routeur_A\}$ (un client et un routeur compromis) et les objectifs d'attaques (tirés de recommandations gouvernementales) considérés, sont $\mathcal{O} = \{VolId, ContAuth, Alte, Alte_C\}$ avec :

— $VolId$ = Vol d'identifiants,

- $ContAuth$ = Contournement d'authentification,
- $Alte$ = Altération d'un message,
- $Alte_C$ = Altération ciblée d'un message.

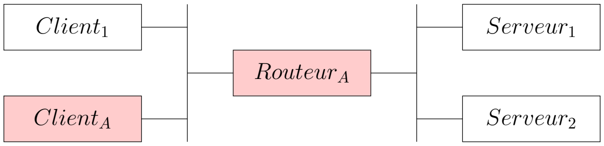


FIGURE 1: Exemple d'infrastructure

Les attaquants $\mathcal{A}$ et les objectifs $\mathcal{O}$ sont liés par la relation $\mathcal{R}_{Obj}$ définie en Table 1, où un ✓ signifie que l'objectif o est retenu pour l'attaquant a.

$\mathcal{R}_{Obj}$	$VolId$	$ContAuth$	$Alte$	$Alte_C$
$Client_A$		✓		
$Routeur_A$	✓	✓	✓	✓

TABLE 1: Exemple d'objectifs retenus pour chaque attaquant

Nous définissons ensuite l'ensemble des vecteurs d'attaques $\mathcal{V}$ et $Real \subseteq \mathcal{O} \times \mathbb{P}(\mathcal{V})$, la fonction entre un objectif et l'ensemble des parties de $\mathcal{V}$ (ie. les combinaisons de vecteurs d'attaques). Ainsi, $Real(o)$ décrit la réalisation d'un objectif o à l'aide de vecteurs de $\mathcal{V}$. Un vecteur peut être vu comme les techniques, tactiques ou procédés pouvant servir à la réalisation d'une attaque [1].

Exemple :

Nous considérons ici les vecteurs $\mathcal{V} = \{Lire, Usurp, Mod, Rej\}$ avec :

- $Lire$ = Lecture (et compréhension) d'un message
- $Usurp$ = Usurpation d'une identité
- Mod = Modification d'un message
- Rej = Rejeu d'un message

La fonction $Real$ décrivant comment réaliser les objectifs d'attaque à l'aide des vecteurs d'attaques peut par exemple être :

- $Real(VolId) = \{\{Lire\}\}$
- $Real(ContAuth) = \{\{Usurp\}, \{Rej\}\}$
- $Real(Alte) = \{\{Mod\}\}$
- $Real(Alte_C) = \{\{Lire, Mod\}\}$

En particulier, $ContAuth$ peut être réalisé en usurpant une identité **ou** en rejouant un message d'authentification (ex. : l'envoi d'un mot de passe). Tandis que $Alte_C$ nécessite **à la fois** la capacité de modifier un message et d'en comprendre le contenu.

Ainsi, nous sommes en mesure de représenter les attaquants, leurs objectifs et comment ces objectifs peuvent être réalisés au moyen de vecteurs d'attaques. La section suivante décrit comment une analyse des propriétés de sécurité offertes par les protocoles vérifie si ces objectifs sont atteignables.

2.2 Approche ascendante

Dans un second temps, nous définissons l'ensemble des configurations des protocoles $\mathcal{P}$ considérés pour l'analyse. Dans la suite de cet article nous considérerons chaque configuration comme un protocole différent. $Vect \subseteq \mathcal{P} \times \mathbb{P}(\mathcal{V})$ est l'ensemble des vecteurs d'attaques de $\mathcal{V}$ accessibles à l'attaquant pour chaque protocole (ie. leurs faiblesses exploitables).

Exemple :

Pour les protocoles $\mathcal{C} = \{$MODBUS, FTP, FTP$_{Auth}$, OPC-UA$_{None}$, OPC-UA$_{Sign}$, OPC-UA$_{SignEnc}$ $\}$, les capacités d'attaques $Vect$ sont définies en table 2, où un ✓ signifie que le vecteur d'attaque est accessible à l'attaquant pour ce protocole.

$Vect$	$Lire$	$Usurp$	Mod	Rej
MODBUS	✓	✓	✓	✓
FTP	✓	✓	✓	✓
FTP$_{Auth}$	✓		✓	✓
OPC-UA$_{None}$	✓	✓	✓	✓
OPC-UA$_{Sign}$	✓			
OPC-UA$_{SignEnc}$				

TABLE 2: Exemple de capacités d'attaques retenus pour chaque protocole

Les protocoles MODBUS, FTP et OPC-UA$_{None}$ ne garantissent aucune sécurité et permettent donc tous les vecteurs d'attaques. Le protocole FTP$_{Auth}$ ajoute une authentification à l'aide d'un mot de passe empêchant l'usurpation d'identité. Les protocoles OPC-UA$_{Sign}$ et OPC-UA$_{SignEnc}$ apportent des signatures cryptographiques et de l'estampillage aux messages, empêchant ainsi leur usurpation, modification ou rejeu. Enfin OPC-UA$_{SignEnc}$ garantit également la confidentialité des communications.

Il est alors possible de déterminer si un objectif o est réalisable à l'aide de l'ensemble des vecteurs d'attaques pour un protocole p en vérifiant si : $\exists e \in \mathcal{R}_{Real}(o) \mid e \subseteq Vect(p)$. Alors, l'ensemble des scénarios d'attaques $\mathcal{S}_{a,p}$ pour un attaquant a et un protocole p est alors défini par :

$$\mathcal{S}_{a,p} = \{(o, e) \mid o \in \mathcal{O} \wedge e \subseteq \mathcal{R}_{Real}(o) \wedge e \subseteq Vect(p) \wedge (a, o) \in \mathcal{R}_{Obj}\}$$

L'ensemble des scénarios d'attaques à considérer dans le cadre d'une campagne de test est alors l'ensemble des $\mathcal{S}_{a,p}, \forall a \in \mathcal{A}$ et pour tous les protocoles de $\mathcal{P}$ que pourraient utiliser chaque attaquants.

Exemple :

Dans notre exemple, si l'on suppose que $Client_A$ communique via FTP$_{Auth}$:

$$\mathcal{S}_{Client_A,\text{FTP}_{Auth}} = \{(ContAuth, Rej)\}$$

Cela s'explique par le fait que seul l'objectif $ContAuth$ est considéré pour $Client_A$ (table 1) et qu'il est réalisable avec au moins l'un des vecteurs d'attaques $Usurp$ ou Rej dont le dernier est offert par FTP$_{Auth}$. De même pour $Routeur_A$ qui est face à la fois à OPC-UA$_{None}$ et FTP$_{Auth}$:

$$\mathcal{S}_{Routeur_A,\text{OPC-UA}_{None}} = \{(VolId, Lire), (ContAuth, Usurp), (ContAuth, Rej), (Alte, Mod), (Alte_C, (Lire, Mod))\}$$

$$\mathcal{S}_{Routeur_A,\text{FTP}_{Auth}} = \{(VolId, Lire), (ContAuth, Rej), (Alte, Mod), (Alte_C, (Lire, Mod))\}$$

3 Méthodologie globale

Notre objectif est d'utiliser cette phase d'analyse dans une approche globale allant de la modélisation du système à la production automatique des paquets réseau implémentant et testant les attaques identifiées. Nous proposons donc une approche *Model-Based Testing* dans l'objectif de vérifier si les attaques trouvées par l'approche sont effectivement jouables sur une plate-forme, voire de quantifier leur plausibilité. La figure 2 illustre la méthodologie que nous voulons développer. L'approche part d'une architecture représentant les composants du systèmes, les canaux de communication et les protocoles (similaire à la figure 1) ; croisée avec des propriétés de sécurité spécifiant les objectifs des attaquants. Ensuite, l'analyse présentée en section 2 permet d'obtenir les vecteurs d'attaques à utiliser par des attaquants pour violer les propriétés de sécurité. Ces vecteurs sont ensuite concrétisés en paquets réseaux à l'aide d'une bibliothèque décrivant comment implémenter les vecteurs pour chaque protocole (ex. : comment modifier un paquet OPC-UA). Enfin, ces paquets sont instanciés, soit de manière aléatoire, soit en fonction de la logique applicative de la plate-forme. Cette approche pourrait se généraliser aux systèmes d'information, mais elle exploite néanmoins des propriétés souvent présentes dans les systèmes industriels telles que l'absence de réseaux dynamiques, simplifiant la représentation de l'infrastructure.

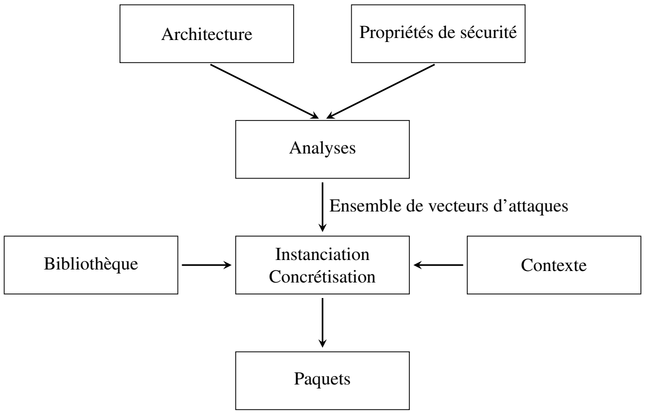


FIGURE 2: Méthodologie globale

4 Conclusion

En conclusion, nous proposons une approche de modélisation d'attaquants basée à la fois sur l'infrastructure d'un système industriel et des possibilités d'attaques contre celui-ci. Des scénarios d'attaques montrent comment un attaquant peut exploiter les faiblesses d'un protocole de communication pour satisfaire ses objectifs. Nous nous focalisons ici sur les possibilités de forger des attaques. A terme, nous souhaitons produire des attaques sur le fonctionnement du système, tenant donc compte du contenu des messages. L'analyse en elle-même pourrait être améliorée en considérant des attaques en plusieurs étapes. Par exemple, nous avons énoncé en section 2.2 que la configuration FTP$_{Auth}$ ne permet pas le vecteur d'attaque $Usurp$ car elle utilise une authentification par mot de passe. Cependant, le vecteur d'attaque $Lire$ pourrait révéler ce mot de passe et ainsi donner l'accès en un second temps au vecteur $Usurp$. Ces attaques en plusieurs étapes, tenant alors compte de l'ordre des actions à exécuter, pourraient être décrites sous forme d'arbre d'attaques [5] (représentation classique pour modéliser des attaquants). Un autre axe pourrait être la prise en compte de plusieurs attaquants coopérant pour des objectifs communs.

Références

1. Sylvain Conchon and Jean Caire. Expression des besoins et identification des objectifs de résilience. *C&esar'15*, 2015.
2. ISA-62443-3-3. Security for industrial automation and control systems, part 3-3 : System security requirements and security levels, 2013.

3. S Kriaa, M Bouissou, and Y Laarouchi. A model based approach for SCADA safety and security joint modelling : S-Cube. In *IET System Safety and Cyber Security*. IET Digital Library, 2015.

4. Ralph Langner. Stuxnet : Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3) :49–51, 2011.

5. Bruce Schneier. Attack trees. *Dr. Dobb's journal*, 24(12) :21–29, 1999.

6. Theodore J Williams. *A Reference Model for Computer Integrated Manufacturing (CIM) : A Description from the Viewpoint of Industrial Automation : Prepared by CIM Reference Model Committee International Purdue Workshop on Industrial Computer Systems*. Instrument Society of America, 1991.

MBeeTle - un outil pour la génération de tests à-la-volée à l'aide de modèles*

Julien Lorrain[1]
julien.lorrain@femto-st.fr

Frédéric Dadeau[1]
frederic.dadeau@femto-st.fr

Elizabeta Fourneret[2]
elizabeta.fourneret@smartesting.com

Bruno Legeard[1,2]
bruno.legeard@femto-st.fr

[1]Institut FEMTO-ST, Besançon, France
[2]Smartesting Solutions & Services, Besançon, France

Résumé

Le Model-Based Testing est une activité permettant, à partir de la modélisation du système sous test (SUT), de générer des tests pour la validation du SUT par rapport aux spécifications de ce système. Cet article présente MBeeTle, un outil de génération de test à-la-volée pour le MBT, ainsi que ses principes et ces stratégies de génération. Il permet la recherche des anomalies profondes et la validation du modèle de test en générant et exécutant des pas de test sur la base d'une approche aléatoire. MBeeTle est présenté avec une expérimentation sur la une norme d'interfaces utilisateurs et logicielles pour les systèmes d'exploitation (POSIX).

1 Introduction et contexte

Dans ce papier nous présentons MBeeTle, un outil de génération de test à-la-volée [1] basé sur le Model-Based Testing (MBT) [2] pour l'exploration des systèmes sous test. Le MBT est une activité permettant de générer des tests abstraits à partir de la modélisation du SUT qui nécessite l'utilisation d'une couche d'adaptation afin de les concrétiser pour pouvoir les exécuter sur le SUT. La catégorie principale de génération de test est la génération dite hors-ligne, dont le principe est de générer des scénarios de test abstrait. Pour pouvoir les exécuter automatiquement sur le SUT, il faut créer le lien entre les données abstraites du modèle et le SUT. C'est ici que la couche d'adaptation est nécessaire pour associer à chaque donnée abstraite les informations de concrétisation permettant leur exécution sur le SUT. Une autre catégorie de génération, implémentée dans l'outil MBeeTle, est la génération à-la-volée. Le principe est d'attendre le résultat du premier pas de test avant de générer les pas suivants. Il est donc nécessaire de disposer de la couche d'adaptation pour mettre en œuvre cette méthode. Cette approche permet de générer des tests fonctionnels longs (*i.e.* plusieurs centaines de pas de test) basés sur les comportements modélisés du SUT.

*Cet outil a en partie été réalisé lors des projets ANR ASTRID OSEP et ANR ASTRID maturation MBT_SEC.

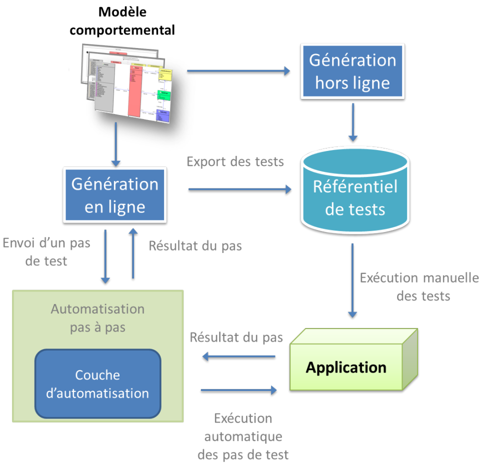


FIGURE 1 – Approche à-la-volée

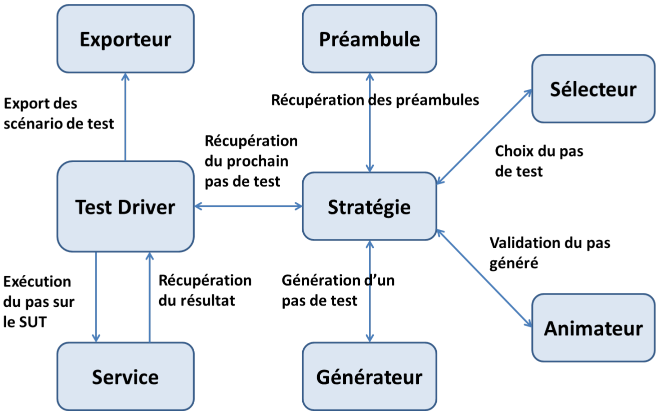

Exporteur
Préambule
Sélecteur
Export des scénario de test
Récupération des préambules
Récupération du prochain pas de test
Choix du pas de test
Test Driver
Stratégie
Validation du pas généré
Exécution du pas sur le SUT
Récupération du résultat
Génération d'un pas de test
Animateur
Service
Générateur

FIGURE 2 – Schéma de fonctionnement

MBeeTle est un outil de génération de tests à-la-volée qui génère puis exécute directement chaque pas de test sur le SUT. Il se base sur l'utilisation de modèles réalisés à l'aide de sous-ensembles des langages UML et OCL [3]. Il s'intègre dans une approche hors-ligne existante, représenté dans la figure 1 et réutilise des briques logicielles de l'outil Smartesting[1] CertifyIt [4] (un générateur de tests fonctionnels hors-ligne), notamment son animateur pour valider sur le modèle les différents pas de test générés par les algorithmes de MBeeTle.

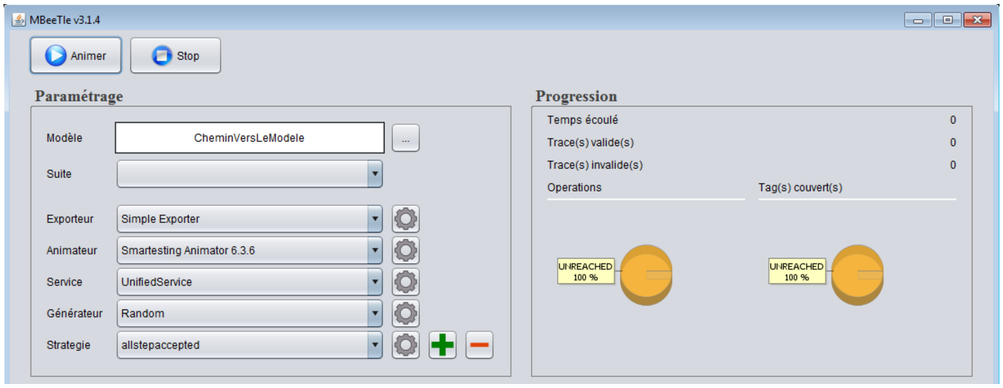
2 L'outil MBeeTle

Le principe de MBeeTle est de générer des scénarios de test plus ou moins longs (plusieurs centaines de pas) à l'aide d'une combinaison d'algorithmes de génération de pas de test, de préambules et d'un ou plusieurs critères de sélection de tests nommés sélecteurs. Cette combinaison est appelé une *stratégie de génération*.

MBeeTle est composé de plusieurs modules, décrit dans la figure 2, ayant chacun un objectif dans la création d'un scénario de test à-la-volée. Le premier, le Test Driver, pilote et lie les autres modules. Les générateurs permettent, grâce à un tirage aléatoire, de générer des pas de test en se servant des informations du modèle. Les pas générés sont cohérents,*i.e.* utilisent des valeurs existantes dans le modèle, mais pas forcément valides par rapport à l'état courant du modèle. Un pas de test est valide si son animation sur le modèle ne rentre pas en contradiction avec les contraintes et/ou les pré-conditions présentes dans le modèle. C'est le rôle de l'animateur : si le pas est valide, il est envoyé au sélecteur. Si le sélecteur choisit le pas alors la stratégie le valide et le retourne au TestDriver qui, au travers du module de service, l'exécute et récupère le résultat de cette exécution. Cette suite d'actions est alors recommencée jusqu'à remplir les conditions d'arrêt de la campagne de génération à-la-volée.

L'interface utilisateur présenté figure 3 permet de configurer entièrement l'outil et d'afficher des informations de la campagne courante, comme le nombre de scénarios générés et la couverture des opérations et des comportements du modèle. Les deux algorithmes de génération de tests utilisés dans MBeeTle sont basés sur l'aléatoire. Le premier est purement pseudo-aléatoire et ne tient pas compte de ce qui a été généré précédemment. Afin de facili-

[1] http ://www.smartesting.com/



Paramétrage

Modèle CheminVersLeModele

Suite

Exporteur Simple Exporter

Animateur Smartesting Animator 6.3.6

Service UnifiedService

Générateur Random

Strategie allstepaccepted

Progression

Temps écoulé 0

Trace(s) valide(s) 0

Trace(s) invalide(s) 0

Operations

Tag(s) couvert(s)

Figure 3 – L'interface de l'outil MBeeTle

ter le passage des comportements complexes, le second implémente une pondération sur les opérations et sur les paramètres.

Trois sélecteurs sont disponibles dans MBeeTle. Le premier est simple et utilise n'importe quel pas de test qui lui est soumis. Le second se base sur les comportements modélisés et optimise leur couverture en favorisant les pas de test qui augmentent la couverture des comportements. Le dernier explore les états du système dans des contextes plus divers que l'approche habituelle. Il se base sur les cas non-passants/passants, *i.e.* les pas qui provoquent un état d'erreur du système et les autres, le but étant de générer des séquences inhabituelles d'opérations qui peuvent provoquer de nouvelles anomalies.

Avant de lancer la génération l'outil peut importer des tests existants. Les bénéfices de cette ré-utilisation de tests sont multiples : augmentation de la diversité des contextes d'activation des comportements déjà testés en évitant les enchainements de comportements déjà couverts, diminution du temps de génération et passage des goulots d'étranglements du modèle (une difficulté inhérente aux approches aléatoire) et du coup augmentation de l'espace d'exploration. L'outil permet d'exporter les résultats sous différents formats.

3 Expérimentations

Le périmètre du standard POSIX [2] modélise la gestion de fichiers, d'utilisateurs et de droits pour les systèmes de type UNIX. Le modèle est composé de 19 classes, avec 24 opérations pour 2 442 lignes d'OCL et chaque interface de l'API est accessible à tout moment. Les tests hors-ligne ont été générés en 24 heures pour couvrir 88% du modèle. Nous avons choisi de faire des lancements/exécutions avec MBeeTle respectant un volume de 20 000 pas correspondant à celui des tests fonctionnels, tout en découpant les lancements en trois configurations types : 1000 scénarios de 20 pas de test, 250 scénarios de 80 pas de test et 100 scénarios de 200 pas de test. Pour chaque stratégie de génération on évalue la couverture des comportements, et le temps de génération de tests et d'exécution des pas de test. Par ailleurs nous avons généré, avec l'outil Pitest [3], 1290 mutants qui nous ont servit pour une analyse mutationelle. La génération hors-ligne tue 548 mutants avec 1484 tests, la génération à la volée en tue en moyenne 546 et la somme des deux approches en tue 630.

[2] https ://standards.ieee.org/findstds/standard/1003.1-2008.html

[3] http ://pitest.org/

De cette expérimentation ressort que MBeeTle permet de couvrir rapidement plus de la moitié des comportements du modèle (environ 16 minutes contre 3 heures pour la version hors-ligne). En réutilisation des tests hors-ligne générés, on augmente significativement la couverture du modèle. Par ailleurs MBeeTle est capable de générer rapidement un volume de pas de test conséquent (ici 20000 pas en 13,6 secondes en moyenne). De plus l'utilisation de cette approche a permis de détecter des mutants différents de ceux détectés par l'approche hors-ligne.

Par ailleurs, l'application de l'approche à-la-volée sur cette expérimentation montre que les artefacts de modélisation hors-ligne sont réutilisables avec un faible coût de main d'œuvre pour adapter le banc de test pour une utilisation pas à pas. Il en ressort également que MBeeTle peut être utilisé pour mettre au point le modèle en générant rapidement des scénarios de test.

4 Conclusion et perspectives

Nous avons présenté MBeeTle en association avec l'approche hors-ligne sur un modèle de taille réelle. Dans ce contexte il permet de compléter cette approche avec un faible coût d'adaptation et une réutilisation complète des ressources et des outils mis en place. Il permet d'augmenter l'espace d'exploration pour activer des comportements dans d'autres contextes. Par ailleurs sur un autre cas d'étude, une implémentation de la norme PKCS#11 [4], il a permis de montrer des erreurs dues a des fuites mémoires qui n'ont pas été trouvés par les outils classiques grâce à une utilisation intensive de l'outil [5].

Pour la suite, nous prévoyons d'adapter cette approche à-la-volée pour en faire une approche de tests en-ligne où nous utiliserons le retour du SUT pour calculer le pas de test pour calculer le prochain. De part son aspect modulaire et ses stratégies paramétrables, des expérimentations pour de la génération en-ligne (utilisation du retour d'exécution des pas de test pour décider du prochain pas de test à générer) [5] sont envisageables. Par ailleurs pour améliorer son efficacité, de nouvelles stratégies seront ajoutées pour palier aux difficultés rencontrées, par exemple sur les modèles à goulots d'étranglements.

Références

[1] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho, "Using on-the-fly verification techniques for the generation of test suites," in *Computer Aided Verification*, pp. 348–359, Springer, 1996.

[2] M. Utting and B. Legeard, *Practical model-based testing : a tools approach.* Morgan Kaufmann, 2010.

[3] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise UML for model-based testing," in *Proceedings of the 3rd International Workshop on Advances in Model Based Testing*, pp. 95–104, 2007.

[4] F. Bouquet, C. Grandpierre, B. Legeard, and F. Peureux, "A test generation solution to automate software testing," in *Proceedings of the 3rd International Workshop on Automation of Software Test*, AST '08, (New York, NY, USA), pp. 45–48, ACM, 2008.

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pp. 75–84, IEEE, 2007.

[4] https ://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-11-cryptographic-token-interface-standard.htm

[5] http ://ucaat.etsi.org/2015/presentations/FEMTOST_FOURNERET.pdf

BINSEC : Plate-forme d'analyse de code binaire [*]

Adel Djoudi [1]   Robin David [1,3]   Thanh Dinh Ta [2]   Josselin Feist [2]

[1] CEA, LIST, 91191 Gif-sur-Yvette France, e-mail : `prenom.nom@cea.fr`
[2] Vérimag, Grenoble, France, e-mail : `prenom.nom@imag.fr`
[3] Université de Lorraine, CNRS and Inria, LORIA, France, e-mail : `prenom.nom@loria.fr`

Binsec est une plate-forme pour l'analyse formelle de code binaire, disponible en LGPL (`http://binsec.gforge.inria.fr`). La plate-forme repose sur une représentation intermédiaire (IR), les DBA [1] [3, 5], bien adaptée aux analyses formelles (peu d'opérateurs, pas d'effets de bord).

La plate-forme propose quatre modules principaux : (1) un front-end incluant plusieurs méthodes de désassemblage syntaxique et une simplification de la représentation intermédiaire générée [5], (2) un simulateur de programmes DBA enrichi d'un modèle mémoire à régions bas-niveau [2], (3) un module d'analyse statique par interprétation abstraite, et (4) un module d'exécution symbolique dynamique (DSE) [4].

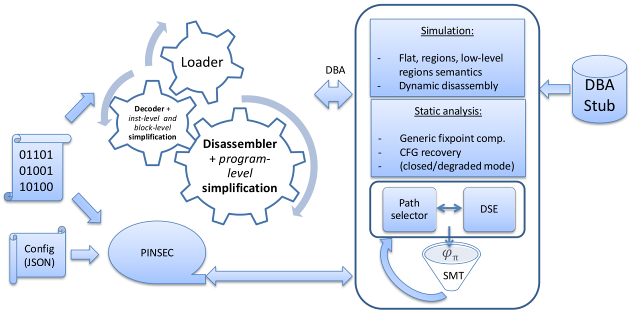


Figure 1 – Plate-forme Binsec

Front-end et simplification des DBA. Chaque instruction machine est traduite vers un bloc d'instructions DBA sémantiquement équivalent. Les analyses sémantiques se basent ensuite sur ce code DBA. Binsec fournit actuellement un parseur pour le format de binaire `ELF` (`PE` en cours) et un décodeur pour l'architecture `x86-32` (support de 380 instructions de base, ainsi que 100 instructions SIMD hors opérations flottantes). Grâce à un mécanisme de stubs formels, il est également possible de remplacer ou d'insérer un bloc DBA à une adresse spécifiée. Ceci est utile si on veut abstraire une partie du code (typiquement, une fonction de librairie standard) ou si le code n'est pas disponible à une adresse donnée (analyse de parties de programmes avant l'édition de liens). Des algorithmes basiques de reconstruction du graphe de flot de contrôle (*linear sweep*, *recursive traversal*) sont implémentés avec la possibilité d'interagir avec les autre modules (notamment exécution symbolique et analyse statique) pour découvrir les cibles des sauts dynamiques. Un moteur de simplifications permet d'éliminer certaines opérations inutiles, typiquement les mises à jour systématiques de flags (75% de suppression dans nos expériences [5]).

Simulation. Le module de simulation supporte les nouvelles extensions des DBA [5], Il permet de simuler l'exécution des programmes désassemblés et la génération aléatoire

[*] La plate-forme BINSEC est un effort commun CEA - IRISA - LORIA - Université Grenoble Alpes, financé par l'ANR (bourse ANR-12-INSE-0002). Les résultats décrits ici ont été présentés à TACAS 2015 [5] et SANER 2016 [4]. Ils ont été encadrés par Sébastien Bardin, Jean-Yves Marion, Laurent Mounier et Marie-Laure Potet.

[1] Dynamic Bit-vector Automata

de tests (*fuzzing*). Trois modèles mémoire peuvent être sélectionnés : le modèle plat (la mémoire comme tableau d'octets), le modèle à régions standard – à la CompCert [6], ou un modèle à régions bas-niveau [2]. L'adoption du modèle à région bas-niveau est motivée par sa capacité à permettre à la fois de profiter de l'abstraction offerte par le modèle à régions standard et de donner une sémantique définie aux programmes bas niveau.

Analyse statique. L'analyse statique permet d'inférer, à partir du code d'un programme et sans l'exécuter, des propriétés vraies pour toutes les exécutions de ce programme. L'analyse statique de programmes binaires permet d'analyser des exécutables dont on ne dispose pas du code source, mais aussi de compléter les analyses appliquées au niveau source avec des informations plus précises, car proches de la plateforme cible. Le module d'analyse statique de BINSEC offre un calcul générique de point fixe abstrait, paramétré par un domaine abstrait, afin de prototyper rapidement des analyses niveau binaire. L'implémentation actuelle permet une interaction avec le désassemblage syntaxique pour la résolution sûre des cibles des sauts dynamiques, ainsi qu'un mode d'analyse non-sûre (contrôlé par l'utilisateur) permettant par exemple de limiter les cibles des sauts dynamiques. L'interface est pour le moment limitée à des domaines non-relationnels mais en cours d'extension pour profiter des domaines relationnels.

Exécution symbolique. BINSEC/SE [4] est un moteur d'exécution symbolique dynamique, architecturé autour des trois composants principaux : le traceur, le noyau d'exécution symbolique et le sélecteur de chemins. Le traceur, appelé `PINSEC`, est articulé autour du framework Pin et permet de générer une trace pour des exécutables Linux et Windows. Un système interactif permet d'échanger des messages avec le noyau DSE afin de faciliter l'instrumentation. Le noyau d'exécution symbolique permet de produire le prédicat de chemin (ensemble de contraintes sur les entrées du programme) d'une trace à partir de la représentation intermédiaire des DBA, puis d'en tester la satisfiabilité via un solveur SMT. Enfin, le sélecteur de chemins permet de spécifier l'ordre dans lequel explorer les chemins du programme, suivant la méthode décrite dans [1].

Expérimentations. Le moteur d'exécution symbolique a été testé sur des programmes de type coreutils (Unix) et malware (Windows), ainsi que sur des challenges de type "crackme". Le moteur d'analyse statique a été testé sur des programmes issus des benchmarks Juliet/samate du NIST.

Références

[1] Bardin, S., Baufreton, P., Cornuet, N., Herrmann, P., Labbé, S. : Binary-level Testing of Embedded Programs. In : QSIC 2013. IEEE, Los Alamitos (2013)

[2] Blazy S., Besson F., Wilke P. : A Precise and Abstract Memory Model for C using Symbolic Values. In : APLAS 2014. Springer, Heidelberg (2014)

[3] Bardin S. , Herrmann P., Leroux J., Ly O., Tabary R., Vincent A. : The BINCOA Framwork for Binary Code Analysis. In : CAV 2011. Springer, Heidelberg (2011)

[4] R. David, S. Bardin, T. Thanh Dinh, J. Feist, L. Mounier, M.-L. Potet, and J.-Y. Marion. BINSEC/SE : A dynamic symbolic execution toolkit for binary-level analysis. In : SANER '16. IEEE, 2016

[5] A. Djoudi and S. Bardin. BINSEC : Binary code analysis with low-level regions. In : TACAS '15. Springer, 2015

[6] Leroy, X., Appel, A.W., Blazy, S., Stewart, G. : The CompCert memory model. In : Program Logics for Certified Compilers. Cambridge University Press (2014)

Table ronde : Enseignement de l'informatique dans le primaire et le secondaire

Table ronde : Enseignement de l'informatique dans le primaire et le secondaire

Animateur

Martin Quinson (IRISA, ENS Rennes)

Résumé de la table ronde
Les nouveaux programmes scolaires font apparaître l'introduction de l'informatique dans le primaire et le secondaire à la rentrée 2016. Au delà de la simple utilisation de l'outil informatique, ceux-ci envisagent une introduction plus spécifique du code informatique et de la programmation (algorithmique, représentation de l'information, fonctionnement d'un ordinateur, etc.) qui soulève un certain nombre de questions : que faire apprendre aux élèves en fonction de leur niveau ? comment former les enseignants ? quelles sont les impacts sur l'enseignement supérieur et sur la société ? L'objectif de cette table ronde est de mettre en avant les initiatives existantes pour introduire cette discipline dans les cursus scolaires et pour accompagner les formateurs dans cette démarche. Au travers de questions préparées et de discussions avec la salle, cette table ronde vise à impliquer la communauté du GDR GPL dans cette évolution majeure des formations en informatique.

Biographie :
Martin Quinson est professeur à l'ENS Rennes depuis 2015. Avant cela, il était depuis 2005 maître de conférences à Telecom Nancy, école de l'Université de Lorraine. Il enseigne ainsi depuis plus de 10 ans les bases de la programmation à des débutants, en partie grâce à une plate-forme pédagogique complète qu'il a co-développé. En matière de médiation scientifique, Martin Quinson est co-auteur d'activités d'initiation à l'informatique sans ordinateur pour des animations en classe ou en périscolaire. Il a également participé à la rédaction d'un guide pédagogique de « La Main à la Pâte » pour l'informatique en primaire à paraître prochainement. Martin Quinson mène en parallèle des recherches sur l'étude de systèmes informatiques répartis à large échelle, de type cloud ou HPC.

Participants

Sylvie Boldo est chercheuse à l'Inria depuis 2005 sur la vérification formelle de programmes numériques. Elle est impliquée en vulgarisation scientifique : comité d'organisation du castor informatique, blog `binaire.blog.lemonde.fr`, revue `interstices.info`, formation des professeurs, exposés en collège...

Christian Buso est enseignant de mathématiques depuis 1984, actuellement au lycée Haag de Besançon. Intéressé très tôt par les possibilités de l'outil informatique pour cet enseignement, il a participé pendant 15 ans à la formation continue des enseignants sur l'usage des TICE. Il a enseigné l'algorithmique en seconde et première scientifique.

Philippe Marquet est enseignant-chercheur en informatique à l'université de Lille, travaille dans le domaine des systèmes embarqués et du parallélisme au sein d'une équipe Inria. Philippe Marquet est vice-président de la SIF, Société informatique de France, en charge de l'enseignement et de la formation. Il est particulièrement intéressé par une nécessaire éducation à l'informatique pour toutes et tous, que ce soit par le biais de la médiation scientifique en informatique, ou de l'introduction d'enseignements de l'informatique.

Benjamin Wack est professeur agrégé de mathématiques. Après plusieurs années en collège et en lycée, il enseigne l'informatique dans l'UFR IM2AG à l'Université Grenoble Alpes et coordonne le groupe Algorithmique de l'IREM de Grenoble. Il a participé à la rédaction de manuels scolaires pour la spécialité ISN en terminale S et pour l'informatique de tronc commun en classes préparatoires.

Prix de thèse et Accessit du GDR Génie de la Programmation et du Logiciel

Prix du GDR GPL : From qualitative to quantitative program analysis : permissive enforcement of secure information flow

Auteur : Mounir Assaf (LSL - CEA LIST, CIDRE - Irisa/Inria/CentraleSupélec & Stevens Institute of Technology)

Résumé :

Résumé : Les mécanismes de contrôle de flux d'information permettent d'analyser des programmes manipulant de l'information sensible, afin de prévenir les fuites d'information.

Les contributions de cette thèse incluent des techniques d'analyse de programmes pour le contrôle de flux d'information tant qualitatif que quantitatif. Les techniques d'analyse qualitatives permettent la détection et la prévention des fuites d'information. Les techniques quantitatives vont au-delà de la simple détection des fuites d'information, puisqu'elles permettent d'estimer ces fuites afin de décider si elles sont négligeables.

Nous formalisons un moniteur hybride de flux d'information, combinant analyse statique et dynamique, pour un langage supportant des pointeurs et de l'aliasing. Ce moniteur permet de mettre en œuvre une *propriété de non-interférence*, garantissant l'absence de fuites d'information sensible. Nous proposons aussi une transformation de programmes, qui permet de tisser la spécification du moniteur au sein du programme cible. Cette transformation de programme permet de mettre on œuvre la propriété de non-interférence soit par analyse dynamique en exécutant le programme transformé, ou par analyse statique en analysant le programme transformé grâce à des analyseurs statiques existants, sans avoir à les modifier.

Certains programmes, de par leur fonctionnement intrinsèque, font fuiter une quantité – jugée négligeable – d'information sensible. Ces programme ne peuvent donc se conformer à des propriétés telles que la non-interférence. Nous proposons une propriété de sécurité quantitative qui permet de relaxer la propriété de non-interférence, tout en assurant les même garanties de sécurité. Cette propriété de *secret relatif* est basée sur une mesure existante de quantification des flux d'information : la *min-capacité*. Cette mesure permet de quantifier les fuites d'information vis-à-vis de la probabilité qu'un attaquant puisse deviner le secret.

Nous proposons aussi des techniques d'analyse statique permettant de prouver qu'un programme respecte la propriété de *secret relatif*. *L'abstraction des cardinaux*, une technique d'analyse par interprétation abstraite, permet de sur-approximer la mesure de min-capacité pour des programmes acceptant des entrées publiques et confidentielles, mais n'affichant le résultat de leurs calculs qu'à la fin de leur exécution. *L'abstraction des arbres d'observation*, quant à elle, supporte des programmes pouvant afficher le résultat de leurs calculs au fur et à mesure de leur exécution. Cette dernière analyse est paramétrée par l'abstraction des cardinaux. Elle s'appuie sur la combinatoire analytique afin de décrire les observations d'un attaquant et quantifier les fuites d'information pour prouver qu'un programme respecte la propriété de secret relatif.

Biographie :
Mounir Assaf est chercheur postdoctoral au Stevens Institute of Technology, où il travaille avec David Naumann sur la rationalisation d'analyses de sécurité par interprétation abstraite et le développements de nouvelles techniques en adoptant le point de vue calculatoire de cette théorie. Il a obtenu un doctorat en informatique de l'université de Rennes 1 en 2015, sous la direction de Julien Signoles (LSL, CEA LIST), Éric Totel et Frédéric Tronel (CIDRE, Irisa/Inria/CentraleSupélec). Sa thèse porte sur l'analyse de programme pour la détection et la quantification des fuites d'information sensible.

Accessit : Méthodes formelles pour le respect de la vie privée par construction

Auteur : Thibaud Antignac (Privatics, laboratoire CITI, Inria Grenoble – Rhône-Alpes, INSA Lyon)

Résumé :

Le respect de la vie privée par construction est de plus en plus mentionné comme une étape essentielle vers une meilleure protection de la vie privée. Les nouvelles technologies de l'information et de la communication donnent naissance à de nouveaux modèles d'affaires et de services. Ces services reposent souvent sur l'exploitation de données personnelles à des fins de personnalisation. Alors que les exigences de respect de la vie privée sont de plus en plus sous tension, il apparaît que les technologies elles-mêmes devraient être utilisées pour proposer des solutions davantage satisfaisantes. Les technologies améliorant le respect de la vie privée ont fait l'objet de recherches approfondies et diverses techniques ont été développées telles que des anonymiseurs ou des mécanismes de chiffrement évolués.

Cependant, le respect de la vie privée par construction va plus loin que les technologies améliorant simplement son respect. En effet, les exigences en terme de protection des données à caractère personnel doivent être prises en compte au plus tôt lors du développement d'un système car elles peuvent avoir un impact important sur l'ensemble de l'architecture de la solution. Cette approche peut donc être résumée comme « prévenir plutôt que guérir ».

Des principes généraux ont été proposés pour définir des critères réglementaires de respect de la vie privée. Ils impliquent des notions telles que la minimisation des données, le contrôle par le sujet des données personnelles, la transparence des traitements ou encore la redevabilité. Ces principes ne sont cependant pas suffisamment précis pour être directement traduits en fonctionnalités techniques. De plus, aucune méthode n'a été proposée jusqu'ici pour aider à la conception et à la vérification de systèmes respectueux de la vie privée.

Cette thèse propose une démarche de spécification, de conception et de vérification au niveau architectural. Cette démarche aide les concepteurs à explorer l'espace de conception d'un système de manière systématique. Elle est complétée par un cadre formel prenant en compte les exigences de confidentialité et d'intégrité des données. Enfin, un outil d'aide à la conception permet aux concepteurs non-experts de vérifier formellement les architectures. Une étude de cas illustre l'ensemble de la démarche et montre comment ces différentes contributions se complètent pour être utilisées en pratique.

Biographie :

Thibaud Antignac est post-doctorant à la Chalmers University of Technology à Göteborg, en Suède, depuis mars 2015. Il travaille avec David Sands et Gerardo Schneider sur des approches basées sur le langage et le contrôle du flot d'information appliquées au respect de la vie privée. Il a obtenu un doctorat de l'INSA Lyon en 2015 et a mené ses recherches au sein de l'équipe Privatics d'Inria Grenoble – Rhône-Alpes au laboratoire CITI sous la direction de Daniel Le Métayer. Sa thèse porte sur les méthodes formelles pour le respect de la vie privée par construction.

Ce document contient les actes des Huitièmes journées nationales du Groupement De Recherche CNRS du Génie de la Programmation et du Logiciel (GDR GPL) s'étant déroulées à FEMTO-ST - Université de Bourgogne Franche-Comté du 8 au 10 juin 2016.

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.