Inria International Program
Associate Team proposal 2020-2022
Submission form

Title: CharActerisation of Program Evolution with Static Analyses.

Associate Team acronym: CAPESA

Principal investigator (Inria): Laure Gonnord, University Claude Bernard Lyon 1 , Compilation, Analysis, for Software and Hardware (CASH), Inria Rhône Alpes, LIP.

Principal investigator (Partner institution): Sébastien Mosser, Département d'informatique, UQAM (Canada).

Other participants: Ludovic Henrio (CNRS, CASH, LIP), Matthieu Moy (University Claude Bernard Lyon 1, CASH, LIP), Jean Privat (UQAM, Canada).

Key Words: Add key words with regard to: A- Research themes on digital science: A2.2. Compilation, A2.2.1. Analyse statique, A2.5.3. Génie logiciel empirique, A2.5.4. Maintenance, évolution.

B- Other research themes and application areas: B5 Industries du futur, B6 Informatique et télécommunications, B9 Société et Connaissance.

1 Scientific program

1.1 Context

Pieces of software are not anymore designed and then deployed forever in an "and they all lived happily ever after" way. They are constantly evolving according to different optics, which can be functional (e.g., new requirements) or extra-functional (e.g., performance optimization). It is not new to consider software evolution as a stepwise process [O7], and the software variability community addresses this topic since decades [O6].

However, it is interesting to note that the work done in this context always work at a single level and inside a single mechanism, e.g., code within one kind of application, model with a given granularity. In the CAPESA project, we want to investigate the commonality that exists between the different mechanisms involved in the incremental software evolution process, possibly at different levels. We will increase the state of the art by defining formal and high-level descriptions to be used to i) model and ii) enact the stepwise evolution of software. By defining a uniform model, we want to improve the understanding of evolution and define verification methods that can work at any level.

We propose to study the characteristics of code evolution *at small granularity*, by focusing on *small-steps* changes. Contrarily to existing approaches, we want to address several evolution mechanisms, but restrict ourselves to changes at a small level of granularity: small code changes in a program. The different versions of the code we will analyse should be close enough to define code evolution as a sequence of "elementary differences".

Two examples of such evolution will be particularly studied in our project: Git commits, and LLVM optimisations passes:

- Commits in the Git revision control system are snapshots of the state of a project at some point in time. A common situation is to have two developers making commits in parallel and merging them after the fact. If commits A and B are made in parallel, one would expect the result of the merge operation to reflect both the sequence $A; B$ and $B; A$, i.e. the order should not matter. While this is often the case, it is not in the presence of merge conflicts (when the same code is modified in both A and B), where neither the sequence $A; B$ nor $B; A$ is well-defined. In its current state, the notion of merge conflict is defined textually, without any semantic knowledge. This leads to both false positive (conflicts that could easily be resolved automatically) and false negative where the resulting code is incorrect but without any warning from Git (typical examples include: A introduces a new call to a function, while B removes any mention of this function). A semantic merge algorithm could increase the reliability of the result while reducing the manual work.
- LLVM uses a typical internal architecture: the optimizer uses a succession of individual transformation or analysis on the program called "passes". The order of passes may change the resulting program, at least in terms of performance. Finding the optimal order of passes is a difficult problem. The knowledge of dependencies between passes can help developers of compilers: passes can be fully independent (i.e. commutative), in which case their order does not matter, or it may be beneficial to apply one or the other.

The expected output of the CAPESA project is a concrete tool that gives a meaningful description of the code evolution between two small steps of the development: a diagnosis, to understand the impact of a given pass, or help the understanding of a unique commit or a set of commits. It is visible that the commutativity of the changes play a major role in the example and that we should define carefully abstractions such that commutativity is easy to identify or characterise.

At tool that helps characterising the impact of code changes would help in several ways: it will help the programmer have a more faithful understanding of git commits, and could help the automatic characterisation of bug solving for example. Such a tool would have very helpful in the context of large-scale analysis of Github data, as investigated in [O1] (where the author identify the weakness of automatic analysis of Github commit messages for tracking bugs).

1.2 Objectives (for the three years)

In the CAPESA projet, we propose to define and manipulate a notion of "structural semantic diff", which would enable to precisely characterise on a structural version of the code the changes induced by a small step transformation. The approach will be validated on two instances of the problem: git commits and LLVM transformations.

While the idea of studying difference between two programs structurally (typically on an abstract syntax tree instead of the textual form) is not new (see e.g. [0]), our objectives go far beyond the computation of such "diff": we need to give a useful meaning to such diff in a context different from the programs on which they were initially computed. For example, a clever merging algorithm in Git could merge commits A and B, given a common ancestor C, by computing the diff between C and A, and them *re-applying it* on top of B. In the context of LLVM passes, we need not only to compute diffs between a program before and after a pass (partially solved by the tool `llvm-diff`), but also to study the interaction of this diff with other diffs corresponding to other compiler passes.

The challenge here is not only to semantically characterise evolution, but also to be able to produce algorithms and semi-algorithms to provide useful feedbacks to the user (being a final user or a software developer). The problem of equivalence or bisimulation between programs is already known and frequently studied but from a strict behavioural equivalence perspective (e.g. [O3]), however here we will have to capture and define a "weak" or "relative" notion of equivalence or similarity, since we want to capture small changes.

We propose to make a two-level analysis: a structural analysis on graphs, trees will decide if there is a relative similarity ("weak isomorphism") between the code before and after transformation, and where "interesting" changes might have been made, and additional semantic knowledge will be used to show "relative equivalence" and provide useful feedback.

Such a formalism would be a first step toward a more ambitious characterisation that would also be more hierarchical. For example in [O1] the authors identify the wrong characterisation of a bug that occurred in a test module (automatic classification tool did not make any difference between a bug in the framework and a bug in the test-suite testing the framework).

We claim that the approach we promote particularly suits our needs, since good practices with Git is to do small step commits (however we will reason on the composition of small steps in the context of the project); and we will firstly study LLVM passes that do not strongly modify code structure.

1.3 Work-program (for the first year)

In Figure 1, we depict a code evolution taking the form of a compiler optimisation, namely a *loop code motion.* An expression or a statement in a while loop can be `hoisted` before the loop if it is invariant (and other conditions that we ignore here). In this example, it is straightforward to see that a "textual diff", as provided by the Unix diff tool, would give us a non structured result, depicted in the right column, from which it seems nearly impossible to recognize the transformation pattern.

In this project we propose to characterise the transformation in terms of "structural diff", for which we depict an informal instance in Figure 2. We claim that such information will help

the understanding of small step evolutions.

```
int i,x,y;
...
for (i = 0; i< 5; i++) {
  x = y + z;
  a[i] =  10 * i + x*x);
}

int i,x,y,t;
x = y + z
t = x * x;
for (i = 0; i< 5; i++) {
  a[i] =  10 * i + t);
}
```

```
<    int i,x,y,z;
---
>    int i,x,y,z,t;
8a9,10
>    x = y + z;
>    t = x*x;
11,12c13
<            x = y + z;
<            a[i] = 10 * i + x*x;
---
>            a[i] = 10 * i + t;
```

Figure 1: Program before and after code motion, and their textual diff

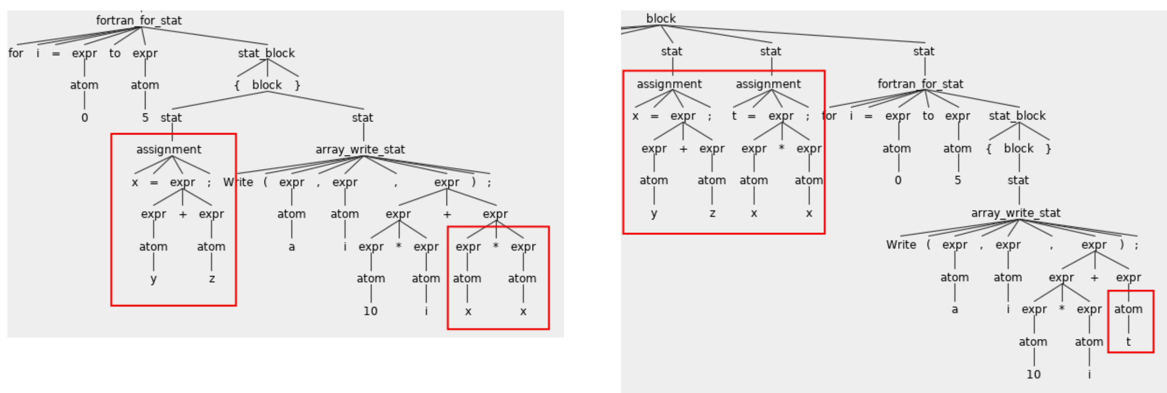

Figure 2: AST version of the program before and after code motion (parts) Some instructions inside the "for_stat" have been moved before, and a structural diff would capture this move and characterise it as a "loop motion".

Informally, a "semantic diff" would capture the *structural changes* of the AST as well as semantic information (loops, statements) that would be enough to capture the whole "loop code motion" information.

Although not clearly trivial, this first example is one of the simplest example of code transformation since the transformation is done with *constant semantics*. In the general case, a code evolution (for instance captured by a git commit) do not have this nice property. We thus propose the following methodology for the first year:

1. First work on code evolution that implies two semantically equivalent codes with minor structural changes; as an example, work on classical compiler passes to detect minor structural changes; then extend this work with semantic information ("while/repeat loops can have the same semantics").

2. Secondly, extend work on non equivalent code, beginning with code evolution that imply classical code refactoring [O8].

3. Then try to extend the method for “weakly similar code” for which we will have to define a notion of similarity and give structural and semantically based algorithms at least to give a partial idea of code change.

Evaluation of our method and methodology The methodology of the evaluation of our “semantic diffs” is itself a subject and part of the first year work program. Indeed:

- If we find a “major semantic diff” in a sequence of git commits, either it would mean that there was a really major modification of the code, or our characterisation is not the right one. This is the kind of questions that the software engineering experts (Canadian side) are able to deal with: search for a big enough set of examples in order to validate the pertinence of the approach. UQAM has a strong experience in long-term development (e.g., the Nit language development started a decade ago), as well as large-scale experiments (e.g., using the ComputeCanada cluster to parallelize large benchmarks).
- For compiler optimisations such as code motion, we need to validate the fact our methodology is capable to give a small “similarity score”. For this purpose we will use the test infrastructure of LLVM to validate the method on a middle-sized benchmark. The expertise of the french team on LLVM for which they already developed analysis passes [P6, P8] will be a strong advantage for this activity.

2 Data Management Plan

The project will not include any personal data, nor any confidential data. Datasets are small enough to be versionned with the source code of the tools (we plan to use `gitlab.inria.fr` for this).

With respect to the software merge axis of this project, we plan to collect a dataset of git merge commits that will complement the one published by Menezes et al[O4]. The collected merge examples will be carefully selected with respect to their software license and made available to the research community as a git repository.

3 Added value

Both teams will benefit from this cooperation. The Canadian team will benefit from the French expertise in compiler engineering (Laure Gonnord, Matthieu Moy), mathematical foundations (Laure Gonnord), and semantics (Ludovic Henrio). The French team will benefit from the Canadian expertise on software engineering (Sébastien Mosser) and long term language development (Jean Privat).

In addition to the technical objectives, this cooperation has the goal of enhancing the ties that are already established between Inria and UQAM and more broadly between institutions of France and the Canadian Quebec region. We expect these exchanges to also contribute to the respective international visibility of Inria and UQAM.

The teams involved in this cooperation also share the goal of educating experts in both compiler technology and software engineering. Thus, this cooperation will give graduate students the opportunity to enhance their education. Paul Iannetta will have the opportunity to do a long stay in Montreal as part of their Ph.D.

This project will produce innovations in both the practical and theoretical fields.

From a theoretical point of view, this project will produce new techniques to characterize code evolution and evaluate formally the difference between two programs that result from

software evolution. Given that this work joins expertise from different, yet related fields, namely compilation technology, theoretical and experimental software engineering, and static analysis, we believe that these new theories will be solid, and useful to other researchers. The novelty of the result also results from the joint work of the different communities. From a practical point of view, we will produce tools that help the programmer evaluate the difference between two software version with semantics and structural arguments, we will also produce tools that evaluate the impact of a compilation pass on the intermediary code. In practice, all the tools and technologies developed during this cooperation will be made open access, so that they can be used by other researchers and compiler enthusiasts.

3.1 Previous Associate Teams

Laure Gonnord was involved in the **PROSPIEL** (Profiling and specialization for locality) associate team (PI Collange, Inria Rennes PACAP), in 2015-2017. The foreign PI was Fernando Pereira, who belongs to the University of Mineas Gerais, in Brasil.

4 Impact

The number of developers using Git on a daily basis is tremendous. GitHub only claims 31 millions of unique users collaborating on more than 96 millions of code repositories in its *State of the Octoverse* 2019 report. Providing a better understanding of how source code evolve using version control such as git, and potentially integrating the result if this research inside the tool (Matthieu Moy has already contributed 287 commits to Git source code) will have an immediate impact on software developers. We are also discussing with young startups aush as Mergify that try to ease source code merging process. With respect to the LLVM part of the project, having a better understanding of how compiler passes interact with each others might help companies working on critical systems, as well as the definition of certified or at least most trustable production compilers.

5 Intellectual Property Right Management

5.1 Background

We do not need any patented knowledge nor process for this project.

5.2 Protective measures

The result of our research will be published as public research report whenever necessary, and we will submit them to national and international conferences. We follow the classical rules for communication and storage (signed emails, institutional resources for communication, encrypted disks).

6 Ethical Issues

We do not manipulate personal data nor films, and related work is research papers that are cited in the classical way. Our object of interests are open-source programs.

7 References

7.1 Joint publications of the partners

[J1] Laure Gonnord and Sébastien Mosser. "Du code aux modèles, des modèles au code: enseigner les langages dédiés (DSL)". In: *Conférence en Ingénierie du Logiciel (CIEL'18)*. Grenoble, France, June 2018. URL: https://hal.archives-ouvertes.fr/hal-01816239.

[J2] Laure Gonnord and Sébastien Mosser. "Practicing Domain-Specific Languages: From Code to Models". In: *14th Educators Symposium at MODELS 2018*. Copenaghen, Denmark, Oct. 2018, pp. 1–8. URL: https://hal.archives-ouvertes.fr/hal-01865448.

7.2 Main publications of the participants relevant to the project

[P1] Benjamin Benni, Sébastien Mosser, Naouel Moha, and Michel Riveill. "A delta-oriented approach to support the safe reuse of black-box code rewriters". In: *Journal of Software: Evolution and Process* 31.8 (2019). URL: https://doi.org/10.1002/smr.2208.

[P2] Kiko Fernandez-Reyes, Dave Clarke, Ludovic Henrio, Einar Broch Johnsen, and Tobias Wrigstad. "Godot: All the Benefits of Implicit and Explicit Futures". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. Ed. by Alastair F. Donaldson. Vol. 134. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, 2:1–2:28. ISBN: 978-3-95977-111-5. URL: http://drops.dagstuhl.de/opus/volltexte/2019/10794.

[P3] Sami Lazreg, Maxime Cordy, Philippe Collet, Patrick Heymans, and Sébastien Mosser. "Multifaceted automated analyses for variability-intensive embedded systems". In: *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. Ed. by Joanne M. Atlee, Tevfik Bultan, and Jon Whittle. IEEE, 2019, pp. 854–865. ISBN: 978-1-7281-0869-8. URL: https://doi.org/10.1109/ICSE.2019.00092.

[P4] Geoffrey Hecht, Hafedh Mili, Ghizlane El-Boussaidi, Anis Boubaker, Manel Abdellatif, Yann-Gaël Guéhéneuc, Anas Shatnawi, Jean Privat, and Naouel Moha. "Codifying Hidden Dependencies in Legacy J2EE Applications". In: *25th Asia-Pacific Software Engineering Conference, APSEC 2018, Nara, Japan, December 4-7, 2018*. IEEE, 2018, pp. 305–314. ISBN: 978-1-7281-1970-0. URL: https://doi.org/10.1109/APSEC.2018.00045.

[P5] Frank De Boer, Vlad Serbanescu, Reiner Hähnle, Ludovic Henrio, Justine Rochas, Crystal Chang Din, Einar Broch Johnsen, Marjan Sirjani, Ehsan Khamespanah, Kiko Fernandez-Reyes, and Albert Mingkun Yang. "A Survey of Active Object Languages". In: *ACM Comput. Surv.* 50.5 (Oct. 2017), 76:1–76:39. ISSN: 0360-0300. URL: http://doi.acm.org/10.1145/3122848.

[P6] Maroua Maalej, Vitor Paisante, Pedro Ramos, Laure Gonnord, and Fernando Pereira. "Pointer Disambiguation via Strict Inequalities". In: *Code Generation and Optimisation*. Austin, United States, Feb. 2017. URL: https://hal.archives-ouvertes.fr/hal-01387031 (cit. on p. 5).

[P7] Anas Shatnawi, Hafedh Mili, Ghizlane El-Boussaidi, Anis Boubaker, Yann-Gaël Guéhéneuc, Naouel Moha, Jean Privat, and Manel Abdellatif. "Analyzing program dependencies in Java EE applications". In: *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. Ed. by Jesús M. González-Barahona, Abram Hindle, and Lin Tan. IEEE Computer Society, 2017, pp. 64–74. ISBN: 978-1-5386-1544-7. URL: https://doi.org/10.1109/MSR.2017.6.

[P8] Henrique Nazaré, Izabela Maffra, Willer Santos, Leonardo Barbosa, Laure Gonnord, and Fernando Magno Quintão Pereira. "Validation of memory accesses through symbolic analyses". In: *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. OOPSLA '14. ACM, 2014, pp. 791–809 (cit. on p. 5).

[P9] Julien Henry, David Monniaux, and Matthieu Moy. "Succinct Representations for Abstract Interpretation". In: *Static analysis symposium (SAS)*. Lecture notes in Computer Science 7460. Deauville, France: Springer, Sept. 2012, pp. 283–299. URL: https://hal.archives-ouvertes.fr/hal-00709833.

[P10] Sébastien Mosser, Mireille Blay-Fornarino, and Laurence Duchien. "A Commutative Model Composition Operator to Support Software Adaptation". In: *Modelling Foundations and Applications: 8th European Conference, ECMFA 2012, Kgs. Lyngby, Denmark, July 2-5, 2012. Proceedings*. Ed. by Antonio Vallecillo, Juha-Pekka Tolvanen, Ekkart Kindler, Harald Störrle, and Dimitris Kolovos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 4–19. ISBN: 978-3-642-31491-9. URL: https://doi.org/10.1007/978-3-642-31491-9_3.

7.3 Other references

[O1] Emery D. Berger, Celeste Hollenbeck, Petr Maj, Olga Vitek, and Jan Vitek. "On the Impact of Programming Languages on Code Quality". In: *CoRR* abs/1901.10220 (2019). arXiv: 1901.10220. URL: http://arxiv.org/abs/1901.10220 (cit. on p. 3).

[O2] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[O3] Shuvendu K. Lahiri, Andrzej Murawski, Ofer Strichman, and Mattias Ulbrich. "Program Equivalence (Dagstuhl Seminar 18151)". In: *Dagstuhl Reports* 8.4 (2018). Ed. by Shuvendu K. Lahiri, Andrzej Murawski, Ofer Strichman, and Mattias Ulbrich, pp. 1–19. ISSN: 2192-5283. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9758 (cit. on p. 3).

[O4] G. G. L. Menezes, L. G. P. Murta, M. O. Barros, and A. Van Der Hoek. "On the Nature of Merge Conflicts: a Study of 2,731 Open Source Java Projects Hosted by GitHub". In: *IEEE Transactions on Software Engineering* (2018), pp. 1–1 (cit. on p. 5).

[O5] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. "Provably Correct Peephole Optimizations with Alive". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: ACM, 2015, pp. 22–32. ISBN: 978-1-4503-3468-6. URL: http://doi.acm.org/10.1145/2737924.2737965.

[O6] Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. "Delta-oriented programming of software product lines". In: *International Conference on Software Product Lines*. Springer. 2010, pp. 77–91 (cit. on p. 2).

[O7] Don S. Batory. "Using modern mathematics as an FOSD modeling language". In: *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, TN, USA, October 19-23, 2008, Proceedings*. Ed. by Yannis Smaragdakis and Jeremy G. Siek. ACM, 2008, pp. 35–44. ISBN: 978-1-60558-267-2. URL: https://doi.org/10.1145/1449913.1449921 (cit. on p. 2).

[O8] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999. ISBN: 0-201-48567-2 (cit. on p. 4).