Access Graph Visualization: A step towards better understanding of static access control

Olivier Gout, Gilles Ardourel, Marianne Huchard

URL: http://www.elsevier.nl/locate/entcs/volume72.html 10 pages

Access Graph Visualization : A step towards better understanding of static access control

Olivier Gout [1]

LIRMM, Université Montpellier 2
Montpellier, France

Gilles Ardourel [2]

LIRMM, Université Montpellier 2
Montpellier, France

Marianne Huchard [3]

LIRMM, Université Montpellier 2
Montpellier, France

Abstract

In object-oriented software development, design and implementation of static access control is a tricky task that has currently received few attention in the framework of development environments. In a previous work, we have defined a graph-based access control formalism and specified a suite of tools (AGATE) using this formalism as a foundation. In this paper, we investigate the implementation and the use of the visualization aspect. We describe how visualization is achieved thanks to Royere, a framework dedicated to graph visualization, and we outline results of a case study.

1 Introduction

Software development is supported by environments more and more sophisticated that support not only common design and programming tasks, but also software measurement or reverse engineering. In such tools, visualization is a crucial point which guides developers during software construction and interpretation. In this paper, we investigate implementation and use of a visualization tool which is part of a tool suite, AGATE [2], dedicated to static access

[1] Email: ogout@lirmm.fr
[2] Email: ardourel@lirmm.fr
[3] Email: huchard@lirmm.fr

control in object-oriented development. Static access control is a feature which receives very few attention in current development environments, though it is the main basis for ensuring encapsulation and modularity.[14] In current object-oriented languages, it is implemented thanks to specific mechanisms that interpret special keywords which denote the level of static protection which applies. These keywords are mainly applied to properties, namely instance attributes, class (`static`) attributes, instance methods, class (`static`) methods), but may also be used to protect classes, internal types or inheritance links. Kinds of accesses are also involved, as "read" or "write" access for attributes, "call" for methods, "use" for internal types. Well known examples of such access control mechanisms are `export` in Eiffel[15] or `public`, `protected` in Java[5] and C++[16], which are used to achieve implementation hiding or to define particular interfaces adapted to different client classes. Static access control mechanisms unfortunately are not so easy to understand and use, leading to softwares where they are either under-used, or a source of confusion. Furthermore, each programming language has specific mechanisms, making tricky to design an access control policy for a software independently of the target programming language, or to transfer a policy from a language to another during reverse engineering. In a previous work [3,4], we have proposed a new graph-based formalism, language-independent and with clear semantics, that we think adapted to static access control in the various steps of a development, for designing, characterizing, evaluating and comparing access rights. The graphical aspect of the formalism should help integration in notations like UML and support developer intuition. Nevertheless, visualization of such graphs requires elaborated layout algorithms and the definition of different views that help in understanding them. This paper describes an ongoing research on access graph visualization. We successively develop a simplified definition of access graph notation section 2, suggestions for coping with the graph complexity when displayed section 3, implementation using the visualization framework Royere[13] section 4, and a case study that highlights the role of access graph visualization in software understanding section 5. We conclude with some perspectives of this work

2 Access graphs

We present here a simplified definition of access graphs, the whole model description [4] being out of the scope of this paper. Access Graphs are labelled oriented graphs where nodes represent classes, while edges convey access information. In the general case, access information can indicate allowed accesses as well as effective accesses. In this paper, we focus on allowed accesses.

An edge (C_1, C_2) is labelled by a set of 3-tuples (m, n, ak) denoting the feasibility of a class-level access from C_1 to C_2 i.e an access such that:

- m is a method of C_1 containing an access expression e

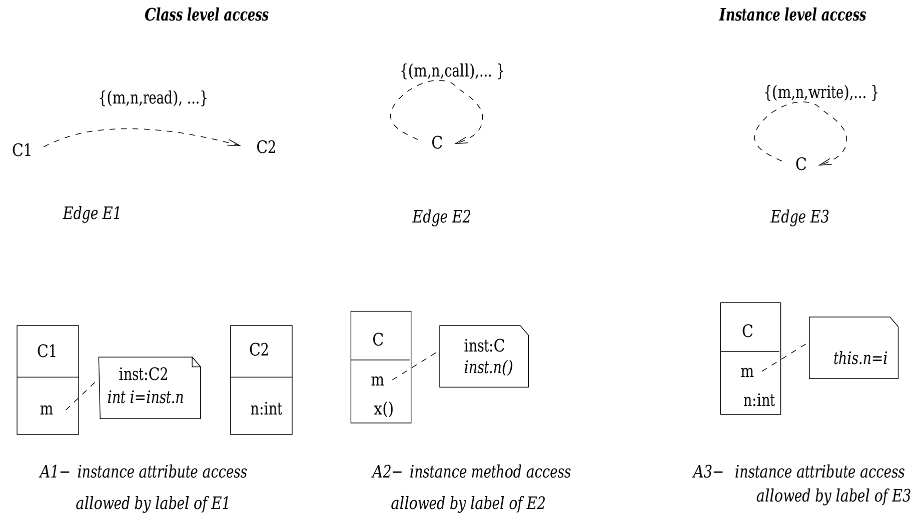

Fig. 1. A graph-based representation of allowed static accesses

- e denotes an access of kind ak to the property p of C_2
- p is accessed by applying the name n to:
  - C_2 when p is a class property
  - an object statically typed as a C_2 if p is an instance property.

We will denote a class-level access by the 5-tuple (C_1, C_2, m, n, ak). In Figure 1 $A1$ and $A2$ are class-level accesses allowed by Edges $E1$ and $E2$.

For the special case where $C_1 = C_2$, an additional set of 3-tuples represents instance-level accesses that are accesses where the name n is applied on special variables like `this`, `self` or `super`. We will denote an instance-level access by the 4-tuple (C_1, m, n, ak). In Figure 1, $A3$ is an instance-level access allowed by Edge $E3$.

The access graph is then a natural representation of the sets $\mathcal{A}_C$ of class-level accesses (5-tuples) and $\mathcal{A}_I$ of instance-level accesses (4-tuples).

We use the following simplification that Factorizes Accessing Methods (FAM) : If all methods in a class C_1 have the same access rights, then class-level accesses can be denoted by (C_1, C_2, n, ak) and instance-level accesses by (C_1, n, ak).

3 Towards visualization

In order to represent clearly the information contained in the access graphs, we use several techniques that simplify the graph or highlight some properties, which are detailed below.

3.1 Factorization

We factorize access rights shared by several classes on a meta-node representing them. This concentrates information and makes the graph more readable.

This tec hniqueis mainly used in the specific and common case where the meta-node represents all the classes. Export to the `ANY` class in Eiffel and `public` mechanisms in Jav a and C++ generate concrete cases where such ANY factorization is useful. More precisely, we add to the set of classes $\mathcal{C}$ the meta-node ANY that represent all classes. Then we define the new set $\mathcal{RA}_C$ of class-level accesses that can be factorized. $\mathcal{RA}_{\mathcal{C}} = \{(C_1, C_2, m, n, ak) s.t \forall C' \in \mathcal{C}, m' \in Methods(C'), (C', C_2, m', n, ak) \in \mathcal{A}_C\}$, where $Methods(C')$ denotes the sets of methods of C'. Next, the set of factorization edges $\mathcal{FA}_C$ is defined as: $\{(ANY, C_2, n, ak) s.t \exists (C_1, C_2, m, n, ak) \in \mathcal{RA}_C\}$. Finally, the new set of class-level accesses $\mathcal{A}'_C$ is $\mathcal{A}_C \setminus \mathcal{RA}_C \cup \mathcal{FA}_C$.

3.2 Metrics

According to J. Hogan [10] there are three main classes of metrics distinguished by their context: first is reusability, second is productivity and last are complexity, cohesion and coupling. Metrics concerning access control should belong to the first and third class, because they're expressing constraints on a system structure (by enforcing modularity) and evolution. Because the nodes of a an access graph represent classes, we studied metrics that concern classes, and more precisely those about cohesion [7,12] and coupling[6,9].

Our conclusion is that Access control is seldom taken into account in metrics. M. Lanza's approach [11] which defines metrics counting n umber of private, protected and public properties, is the relevant but language specific.

Since the access graphs are language independent, so should be metrics concerning access graphs. We propose sev erallanguage independent metrics for access control that we use in our tool:

- **NAA** (Number of Allowed Accesses) for a class C_1 is the number of 5-tuple (C_2, C_1, m, n, ak) in $\mathcal{A}_{\mathcal{C}}$.
- **AAR** (Allowed Accesses Ratio) for a class C_1 is NAA divided by the number of properties of C_1.

Even if these metrics take into account access control in a language independant way, they do not exploit the expressiviy of the access graph model. Metrics should distinguish class-level access from instance-level access (see Section 2), class properties from instance properties and different access kinds.

We propose the **WAA** and **WAAR** metrics that are weighted b y a function $weight$ $(Level \times AccessKind \times PropertyKind \times Target) \Rightarrow \mathbb{R}$ where $Level$ is either class-level or instance-level, $PropertyKind$ is either Attribute or Method and $Target$ is Intern if Level is instance level or Level is class and the access concerns a property of the same class as the accessing one; $Target$ is Extern otherwise.

We can obtain an order from the following considerations:

- class-level accesses to properties from another class (Extern) should require higher rights than class-level accesses to properties from the same class.

- accesses to attributes should require higher rights than accesses to methods
- write accesses should require higher rights than read accesses
- class-level accesses should require higher rights than instance-level accesses

Now that *weight* is determined we can define more precise metrics on access graphs using $AccessWeight : \mathcal{A} \Rightarrow \mathbb{R}$ where $AccessWeight((C_2, C_1, n, ak)) = weight(Class, ak, PropertyKind(n)$,Intern if $C_1 = C_2$ Extern otherwise).
$AccessWeight((C_1, n, ak)) = weight(Instance, ak, PropertyKind(n), Intern)$.

- **WAA** (Weighted Allowed Accesses) for a class C_1 is the sum of the valuation of all tuples (C_2, C_1, n, ak) in $\mathcal{A}$.
- **WAAR** (Weighted Allowed Accesses Ratio) for a class C_1 is WAA divided by the number of properties of C_1.

4 Implementation with Royere

Our objective is to extend the suite tool AGATE [2] which general purpose is to help in design, understanding and managing static access control. AGATE currently offers services like automatic extraction of access graphs from code (currently implemented for Java and Eiffel), adaptation of any access graph to the rules of a specific language, code generation or check of high-level rules. Visualization clearly is decisive for the success of such tools.

We chose the open source Royere[13][4] as it provides several relevant features for access graph visualization:

- layout algorithms included admit the size of our graphs (50-1000 nodes),
- metrics and filtering can be integrated,
- the format GraphXML allows to exchange graphs with other visualization frameworks like Tulip[5],
- node labels can be edited.

Figure 2 outlines the architecture of AGATE (left) and its connection (interface) with Royere (right). The format used for store and exchange access and inheritance graphs in AGATE is XML following the *AccessGraph DTD*, while Royere uses another DTD, namely *GraphXML DTD*. Three new tools have been added for visualization:

- the "Inheritance Graph Converter" takes an inheritance graph (IG) as input and translates it into GraphXML,
- the "Access Graph Converter" calculates metrics and filters an access graph (AG) and encodes the result in a GraphXML file,
- the "Inheritance Graph Position Mapper" uses an inheritance graph and an access graph of the same set of classes where coordinates have been set by

[4] http://gvf.sourceforge.net
[5] http://www.tulip-software.com

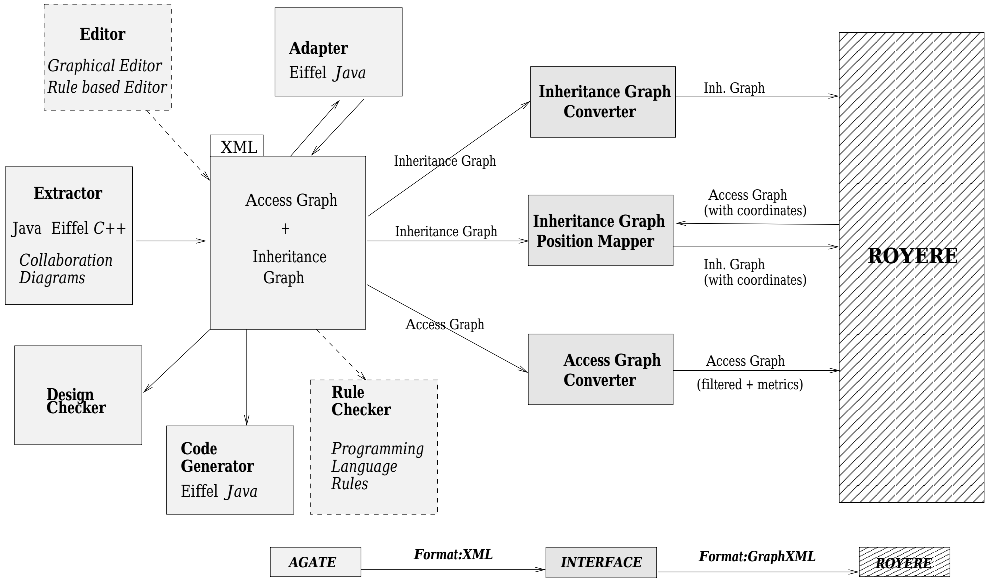

Fig. 2. Royere Interface

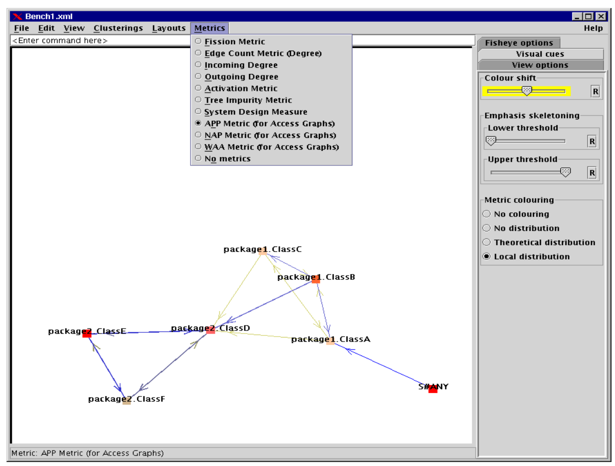

Fig. 3. Royere Interface

Royere. It produces a GraphXML file encoding the inheritance graph with coordinates that come from the access graph.

Filtering is done outside Royere for efficiency reasons.

5 Case study

Access graph visualization has been applied to several software developed in Eiffel or Java (Royere 518 classes, Mars-Sim[6] 205 classes, MegaMek[7] 182 classes, Agate 69 classes). We detail here the analysis of *ResynAssistant* [8], a tool suite dedicated to graph-based modelling molecule. This Java soft-

[6] http://mars-sim.sourceforge.net
[7] http://megamek.sourceforge.net

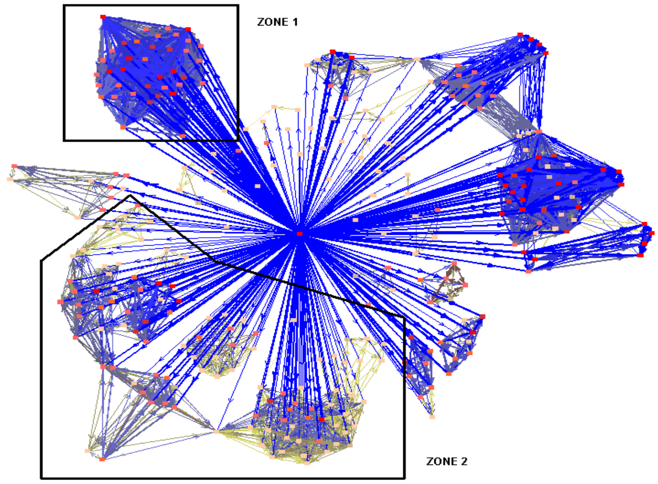

Fig. 4. ResynAssistant. Global view

ware has several interesting features from a reengineering point of view: it is medium-sized being composed of 291 classes, it is frequently modified or extended and several programmers have successively been in charge of the development.

5.1 Global view

Figure 4 shows a global view of the access graph generated from *ResynAssistant* classes. Edge an Node color is determined by **WAA**. Color ranges from yellow for low values to blue for high values. The layout algorithm is *GEM* [8]. The central node is the meta-node *ANY* added by the factorization described in 3.1. Some node groups, like *ZONE 1* correspond to packages. Packages may be connected by small sets of classes. As most of the nodes are connected to *ANY*, revealing that most of the classes possess at least one `public` property, removing the node *ANY* facilitates interpretation by clarifying the graph.

5.2 Selected views of ResynAssistant

Figure 5, in which the removal of *ANY* has been done, shows the set of packages labelled *ZONE2* in Figure 4. Most packages can now be easily identified and named, and two interesting areas might catch an attentive eye.

First, the *symetry* package has a strong blue coloration that suggest a higher **WAAR** value. We focus on this package in Figure 6. The code source of this package reveals a lot of *default* (package level) properties in some of its classes. These properties, and especially the attributes, increase the **WAAR** value because they can be accessed by all the classes in the package. This high number of *default* properties is due to a misconception of the programmer, who decided to change it after seeing the graph.

A second zone seems rather surprising because despite of its aspect it

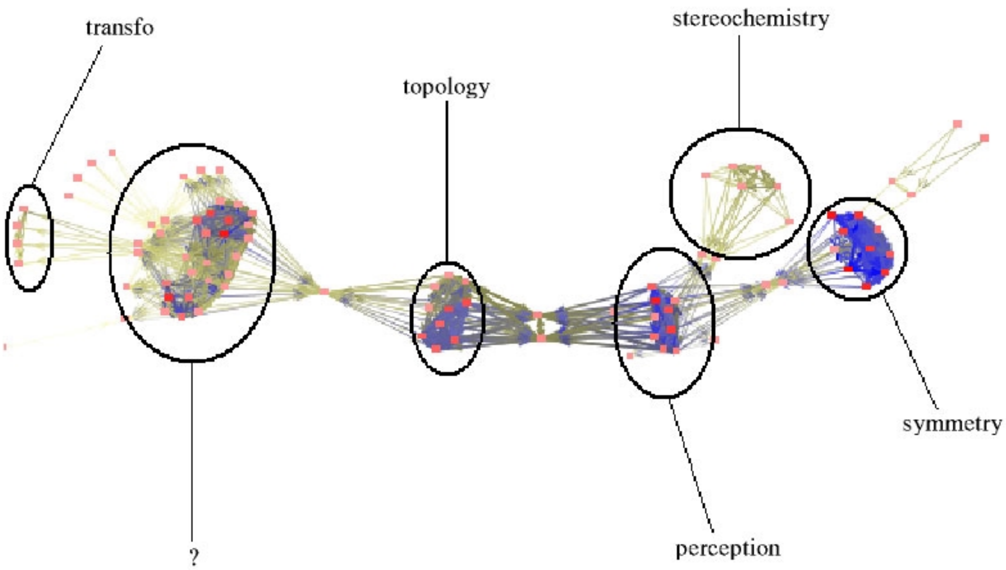

Fig. 5. ResynAssistant. Selected views

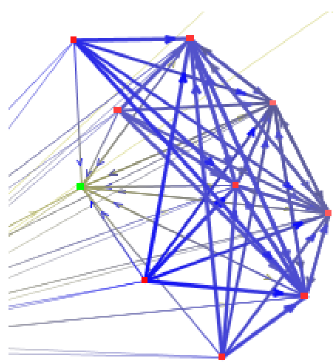

Fig. 6. Package symetry

couldn't be identified to only one package. Figure 7 shows a more detailed view of this zone.

Actually the zone is composed by three packages tightly coupled: *connaissances_chimiques* (chemical knowledge), *concepts* and *graph*. This situation is better explained by having some knowledge about Java access control mechanisms. In Java, the only way of allowing access from outside the declaring package without giving access to all the classes is to have protected properties inherited in a subclass declared outside the package of the base class. By using the *Position Mapper* we generate the inheritance graph and position its nodes like the access graphs nodes. Figure 7 shows this rather complex inheritance graph spanning over the three packages. Such an intrication of accesses be-

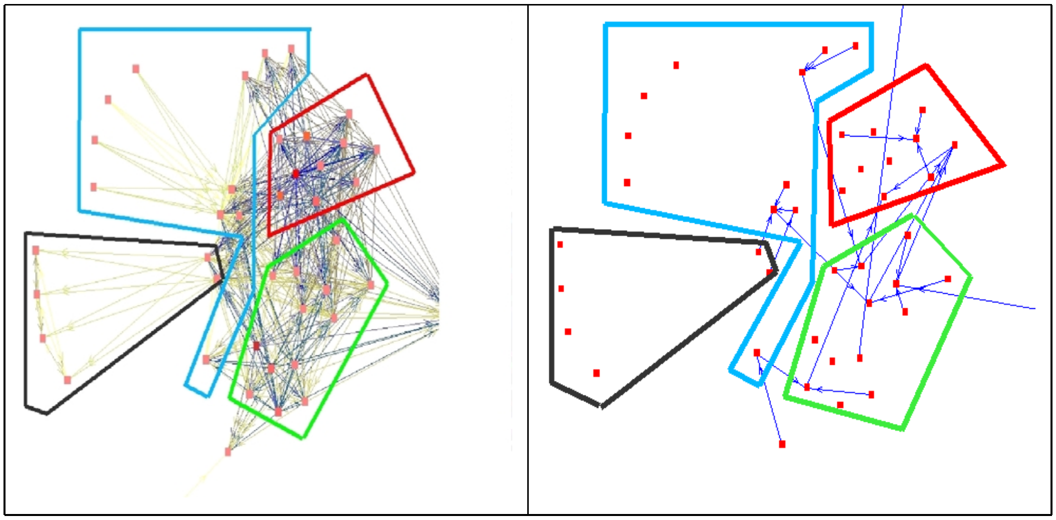

Fig. 7. The entangled packages and their inheritance graph

tween these packages does not favor easy understanding, maintainability and secure extension.

6 Conclusion

In this paper we presented a visualization tool for access graphs, a graph-based representation of access control in Object-Oriented Programming. We are currently applying this visualization to find defects in the use of access control in software. Our first analyses with this tool show that visualization should be improved in different ways. Different layout algrithms could be used or implemented, including algorithms that combines hierarchical display of the inheritance graph and clustering guided by accesses. New filtering techniques should be investigated. Facilities to compare call graphs[1] and access graphs would show distance between allowed and effective accesses as well as measure expressive power provided by a given programming language. We expect from this work a new perspective on access control which would enhance its understanding and improve the design of access control policies in object-oriented software.

References

[1] F. Allen. Interprocedural Data Flow Analysis. In IEEE, editor, *Proceedings of IFIP Congress*, pages 398–402, Amsterdam, 1974. North Holland Publishing Company.

[2] G. Ardourel and M. Huchard. AGATE, Access Graph bAsed Tools for handling Encapsulation . In *Proceedings of the 16th IEEE International Conference Automated Software Engineering (ASE) 26-29 Nov. 2001 San Diego California*, pages 311–314.

[3] G. Ardourel and M. Huchard. Synthèse et modélisation des accès dans les langages à classes : vers une formalisation des systèmes. In *Actes de la conférence langages et modeles a objets 2001 (LMO2001), Le Croisic, L'OBJET*.

[4] G. Ardourel and M. Huchard. Access graphs: Another view on static access control for a better understanding and use. *Journal of Object Technology*, 2002. Accepted for publication.

[5] K. Arnold and J. Gosling. *The Java Programming Language Second Edition*. Addison Wesley, 1998.

[6] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *IEEE Transactionson Sofware Engineering*, 20(6):476–493, 1994.

[7] N. Fenton and S. Pfleeger. *Software Metrics: A rigorous and practical approach*. International Thomson Computer Press, London, 1997.

[8] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In R. Tamassia and I. G. Tollis, editors, *Proc. DIMACS Int. Work. Graph Drawing, GD*, number 894, pages 388–403, Berlin, Germany, 10–12 1994. Springer-Verlag.

[9] M. Hitz and B. Montazeri. Chidamber and kemerer's metrics suite: A measurement theory perspective. *Software Engineering*, 22(4):267–271, 1996.

[10] J. Hogan. An analysis of OO software metrics. Technical Report CS-RR-324, Coventry, UK, 1997.

[11] M. Lanza. *Combining Metrics and Graphs for Object Oriented Reverse Engineering*. Master thesis, 1999.

[12] W. Li and S. Henry. Object-oriented metrics that predict maintenability. *Journal of Systemes and software*, 23:111–122, 1993.

[13] M. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software Practice and Experience*, 31(8):739–756, 2001.

[14] B. Meyer. *Object-oriented Software Construction*. Englewood Cliffs NJ: Prentice Hall, 1988.

[15] B. Meyer. *Eiffel, The Language*. Prentice Hall - Object-Oriented Series, 1992.

[16] B. Stroustrup. *The C++ programming language, Third Edition*. Addison–Wesley, 1997.

Acknowledgements

The authors wish to thank Sandra Berasaluce, Fabien Jourdan and Yannick Tognetti for their help on Royere and ResynAssistant.