AutoWIG : automatisation de l'encapsulation de librairies C++ en Python et en R

Pierre Fernique

▶ **To cite this version:**

Pierre Fernique. AutoWIG : automatisation de l'encapsulation de librairies C++ en Python et en R. 48èmes Journées de Statistique de la SFdS Montpellier, May 2016, Montpellier, France. pp.6. hal-01316276

HAL Id: hal-01316276
https://inria.hal.science/hal-01316276v1

Submitted on 16 May 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

AutoWIG: AUTOMATISATION DE L'ENCAPSULATION DE LIBRAIRIES C++ EN PYTHON ET EN R

Pierre Fernique[1]

[1] *Inria, EPI Virtual Plants, Montpellier, France; pierre.fernique@inria.fr*

Résumé. Les langages de programmation Python et R sont deux des langages les plus populaires pour le calcul scientifique. Cependant, la plupart des logiciels scientifiques incorporent des bibliothèques C ou C++. Bien qu'il existe plusieurs solutions et des outils semi-automatiques pour encapsuler des bibliothèques C++ (**RCPP**, **Boost.Python**), le processus d'encapsulation d'une grande bibliothèque C++ est long et fastidieux. Certaines solutions pour Python ont été développées dans le passé (par exemple **Py++** ou **XDress**) mais nécessitent d'écrire du code complexe pour automatiser le processus, et de compter sur des technologies qui ne sont pas entretenues. Le logiciel **AutoWIG** fait appel à la technologie **LLVM/Clang** pour l'analyse syntaxique de code C/C++ et à l'outil **Mako** pour générer l'encapsulation des bibliothèques C++ avec **Boost.Python** et **RCPP**. Nous illustrerons l'utilisation d'**AutoWIG** sur un ensemble complexe de bibliothèques C++ pour l'analyse statistique.

Mots-clés. C++, Python, R, calcul scientifique

Abstract. Python and R programming languages are two of the most popular languages in scientific computing. However, most scientific packages incorporates C and C++ libraries. While several semi-automatic solutions and tools exist to wrap C++ libraries (**RCPP**, **Boost.Python**), the process of wrapping a large C++ library is cumbersome and time consuming. Some solutions have been developed in the past (e.g. **Py++** or **XDress**) but require to write complex code to automate the process, and rely on technologies that are not maintained. **AutoWIG** relies on the **LLVM/Clang** technology for parsing C/C++ code and the **Mako** templating engine for generating **Boost.Python** wrappers. We will illustrate the usage of **AutoWIG** on a complex collection of C++ libraries for statistical analysis.

Keywords. C++, Python, R, scientific computing

1 Introduction

De nombreuses bibliothèques scientifiques sont écrites dans des langages de programmation de bas niveau tel que le C++. Les langages de programmation de ces bibliothèques imposent l'utilisation du cycle traditionnel d'édition, compilation et exécution afin de produire des logiciels performants. Cela conduit à des courts temps de traitement par

ordinateur mais un long temps de développement par des scientifiques. À l'opposé, les langages de script tels que `R` (R Core Team 2014, pour les analyses statistiques)R14 ou `Python` (Oliphant 2007, à des fins générales) fournissent un cadre interactif qui permet aux scientifiques d'explorer leurs données, de tester de nouvelles idées, de combiner différentes approches algorithmiques ainsi que d'évaluer leurs résultats à la volée. Cependant, le code exécuté dans ces langages de script a tendance à être plus lent que leur équivalent compilé. L'intérêt croissant pour le calcul scientifique combiné à l'amélioration du matériel lors de ces dernières décennies a conduit à ce que ces langages de programmation de haut niveau soient devenus très populaires dans de nombreux domaines du calcul scientifique. Néanmoins, afin de surmonter la moindre performance de ces langages, la plupart des logiciels scientifiques de haut niveau incorporent des bibliothèques compilées accessibles depuis l'interpréteur du langage de script. Pour accéder à du code compilé à partir d'un interpréteur, un programmeur doit écrire une collection de fonctions spéciales dites encapsulatrices. Le rôle de ces fonctions est de convertir les arguments et les valeurs de retour dans la représentation de données de chaque langage. Bien qu'il ne soit pas très difficile d'écrire quelques encapsulateurs, la tâche devient fastidieuse si une bibliothèque contient un grand nombre de fonctions. En outre, la tâche est beaucoup plus difficile si cette bibliothèque utilise des fonctionnalités de programmation plus avancées telles que les pointeurs, les tableaux, les classes, l'héritage, les modèles de classe, les opérateurs et la surcharge de fonctions. **Boost.Python** (Abrahams and Grosse-Kunstleve 2003) et **RCPP** (Dirk Eddelbuettel *et al.* 2011) sont considérées comme des approches classiques pour l'encapsulation en `Python` ou `R` de bibliothèques `C++`, mais ne peuvent être considérées que comme semi-automatique. En effet, si ces approches facilitent sans aucun doute la tâche d'encapsulation d'une bibliothèque, le processus d'écriture et de maintenance des encapsulateurs pour de grandes bibliothèques est encore lourd, long et pas vraiment conçu pour des bibliothèques en pleine évolution.

Il devient donc nécessaire d'automatiser le processus d'encapsulation de bibliothèques disposant d'un grand nombre de classes et de fonctions et évoluant rapidement. Le goulot d'étranglement dans la construction d'une approche automatique pour l'encapsulation en `Python` ou `R` de bibliothèques `C++` est la nécessité d'effectuer l'analyse syntaxique du code de cette bibliothèque. Une fois cette tâche réalisée, on obtient une abstraction du code qui permet son introspection. L'introspection du code est la capacité d'examiner ses composants, de savoir ce qu'ils représentent et quelles sont leurs relations avec les autres composants du même code. Cette introspection peut donc être utilisée entre autres pour automatiser la génération d'encapsulateurs. Dans le passé, certaines solutions telles que **Py++** (Yakovenko 2011) et **XDress** (Scopatz 2013) ont été développées pour automatiser l'encapsulation en `Python` de grandes bibliothèques `C++`. Ces outils nécessitent d'écrire *a priori* des scripts, potentiellement complexes, qui sont ensuite interprétés *a posteriori* pour modifier l'abstraction du code afin de générer les encapsulateurs. De telles approches nécessitent un haut niveau d'expertise dans ces logiciels et limitent la capacité de superviser ou de déboguer le processus d'encapsulation. Ainsi, le coût d'encapsulation avec ces

approches, bien qu'étant automatique, est encore considéré par de nombreux développeurs comme prohibitif. Le but d'**AutoWIG** est de remédier à ces lacunes en proposant une approche interactive d'encapsulation de bibliothèques `C++` à la fois en `R` et `Python`. En particulier, l'interface `Python` proposée permet de fournir un logiciel facile à utiliser, offrant le bénéfice de l'introspection de code pour des bibliothèques `C++` et fournissant ainsi différentes stratégies d'encapsulation.

2 Choix de conception

Le logiciel **AutoWIG** aspire à automatiser le processus d'encapsulation de bibliothèques `C++`. La construction d'un tel système implique certaines exigences et fonctionnalités:

- Analyse syntaxique du code `C++`, afin d'extraire les différents éléments `C++` disponibles, suivie d'une encapsulation automatique de ces éléments en `R` ou `Python`. Ces éléments du langage sont les espaces de noms, les énumérations, les variables, les fonctions, les classes et les alias.

- Documentation automatique des composants encapsulés en utilisant leur documentation `C++`. L'écriture et la vérification de la documentation sont des tâches fastidieuses. La minimisation de la redondance de documentation entre les composants encapsulés et les composants `C++` permet d'économiser beaucoup de travail et de s'assurer que la documentation `Python` ou `R` soit toujours à jour.

- Adaptation des patrons de conception `C++` à ceux de `Python` et de `R`. Certains éléments syntaxiques sont spécifiques au `C++` et doivent impérativement être adaptés en `Python` et `R` afin d'obtenir des progiciels fonctionnels.

- Gestion conforme de la mémoire dans l'interpréteur. Comme `R` et `Python` gèrent l'allocation de mémoire de manière automatique, les concepts de pointeur ou de référence n'ont pas de sens en `R` et `Python`. Cependant, beaucoup de bibliothèques `C++` exposent soit des pointeurs, des pointeurs intelligents ou des références partagées. Une attention particulière est donc nécessaire pour faire face aux problèmes de gestion de mémoire pouvant être engendrés par l'utilisation de références et de pointeurs.

- Gestion des erreurs `C++`. Il est important de veiller à ce que les exceptions levées par le code `C++` soient interceptées et traduites dans les langages `Python` ou `R`. La traduction doit conserver la valeur informative de ces exceptions.

3 Méthodologie d'encapsulation

Une fonctionnalité importante d'**AutoWIG** est son interactivité. Dans le contexte particulier de l'encapsulation, une approche interactive permet aux développeurs d'explorer l'abstraction de leur code, de tester de nouvelles stratégies d'encapsulation et d'évaluer directement leurs résultats. Dans de tels cas, l'utilisateur doit suivre la méthodologie suivante,

`Analyse:` Dans une bibliothèque `C++`, les fichiers en-tête contiennent toutes les déclarations des composants `C++`. Dans cette tâche, **AutoWIG** effectue une analyse syntaxique et sémantique de ces fichiers pour obtenir une abstraction du code `C++`. Cette abstraction est une base de données organisée sous forme d'un graphe. Dans ce graphe, chaque nœud représente une entité `C++` (espaces de noms, énumération, variable, fonction, classe, alias...) et une arête entre deux nœuds représente une relation syntaxique ou sémantique entre ces entités (héritage, typage, dépendance...). Les entrées obligatoires de cette tâche d'analyse sont les fichiers en-tête et des options de compilation pertinentes pour mener à bien l'analyse du code `C++`.

`Contrôle:` Une fois que le code `C++` à été analysé, la base de données peut être utilisée pour inspecter de manière interactive le code `C++`. Cette tâche est particulièrement utile pour le contrôle de la sortie de cette tâche. Par défaut, **AutoWIG** dispose d'un ensemble de règles pour déterminer quels sont les composants `C++` à encapsuler, pour sélectionner la bonne politique de gestion de la mémoire, pour identifier les classes spéciales représentant des exceptions ou des pointeurs intelligents, ainsi que pour adapter les patrons de conception `C++` en `Python` et `R`. Ces règles produisent une encapsulation réflexive des bibliothèques `C++` et conforme si des directives précises ont été suivies. Si un développeur ne veut pas produire une encapsulation réflexive de sa bibliothèque `C++` ou si les directives recommandées ne sont pas respectées dans la bibliothèque, il est possible de modifier les paramètres de contrôle pour assurer une encapsulation pertinente de sa bibliothèque `C++`.

`Génération:` La dernière étape du processus d'encapsulation consiste à générer des fonctions encapsulatrices pour chaque nœud, ainsi qu'à produire du code interprété (`Python` ou `R`) pour adapter la syntaxe bas-niveau de la bibliothèque au style des langages cibles. Cette génération est basée sur des règles utilisant l'introspection du code permise par la base de données organisée par un graphe (à savoir type des variables, les entrées et sorties des fonctions, les classes héritées...). Les sorties de cette étape de génération sont des fichiers contenant les fonctions encapsulatrices qui doivent être compilées, ainsi que du code interprété.

Si une méthode interactive est très pratique pour les premières approches utilisant **AutoWIG**, une fois que les stratégies d'encapsulation ont été choisies, le passage en

mode automatique est du plus grand intérêt. Notez que l'utilisation de la console **IPython** (Perez and Granger 2007) et de sa fonction magique `%history` permet d'enregistrer les commandes des étapes successives dans un fichier `Python` qui peut être ensuite exécuté en mode automatique.

Dans la plupart des cas, afin de faciliter le déploiement des bibliothèques, en particulier pour des bibliothèques multiplateforme, des outils de construction de logiciels tels que **CMake** (Martin and Hoffman 2010) ou **SCons** (Knight 2005) sont utilisés par les développeurs. Étant donné que ces outils fournissent la plupart des informations nécessaires pour l'analyse syntaxique du code par **AutoWIG** (à savoir les fichiers entête et les options de compilation), il est pratique et souhaitable d'imbriquer le processus d'encapsulation d'**AutoWIG** avec le processus de déploiement ces outils de construction de logiciels. Ceci est rendu possible dans **SCons** par la définition d'un constructeur d'encapsulation.

Bibliographie

[1] R Core Team (2014), A Language and Environment for Statistical Computing, URL http://www.R-project.org
[2] Travis E Oliphant (2007), Python for scientific computing, Computing in Science & Engineering, 9(3):10–20
[3] David Abrahams and Ralf W Grosse-Kunstleve (2003), Building hybrid systems with boost.python, CC Plus Plus Users Journal, 21(7):29–36
[4] Dirk Eddelbuettel, Romain François, J Allaire, John Chambers, Douglas Bates, and Kevin Ushey (2011), Rcpp: Seamless R and C++ integration, Journal of Statistical Software, 40(8):1–18
[5] Roman Yakovenko (2011), Py++, URL https://sourceforge.net/projects/pygccxml
[6] Anthony Scopatz (2013), XDress - Type, But Verify, URL http://conference.scipy.org/scipy2013/presentation_detail.php?id=164
[7] Fernando Perez and Brian E Granger (2007), IPython: a system for interactive scientific computing, Computing in Science & Engineering, 9(3):21–29
[8] Ken Martin and Bill Hoffman (2010), Mastering CMake, Kitware
[9] Steven Knight (2005), Building software with SCons, Computing in Science & Engineering, 7 (1):79–8