Practical Data Exchange for Reverse Engineering Frameworks: Some Requirements, Some Experience, Some Headaches

Michael W. Godfrey
Software Architecture Group (SWAG)
Department of Computer Science, University of Waterloo
Waterloo, Ontario, CANADA, N2L 3G1
email: migod@acm.org

Abstract

Reverse engineering systems hold great promise in aiding developers regain control over long-lived software projects whose architecture has been allowed to "drift". However, it is well known that these systems have relative strengths and weaknesses, and to date relatively little work has been done on integrating various subtools within other reverse engineering systems. The design of a common interchange format for data used by reverse engineering tools is therefore of critical importance.

In this position paper, we describe some of our previous work with TAXFORM (Tuple Attribute eXchange FORMat) [2,6], and in integrating various "fact extractors" into the PBS reverse engineering system. For example, we have recently created translation mechanisms that enable the Acacia system's C and C++ extractors to be used within PBS, and we have used these mechanisms to create software architecture models of two large software systems: the Mozilla web browser (2.2 MLOC of C++ and C) and the VIM text editor (150 KLOC of C) [6]. We also describe our requirements for an exchange format for reverse engineering tools and some problems that must be resolved.

Keywords: reverse engineering, fact extractors, data exchange, TAXFORM

Introduction and motivation

Reverse engineering tools hold great promise in helping developers regain control over legacy systems whose architecture has been allowed to drift. A reverse engineering tool typically consists of (at least) a "fact extractor", storage and manipulation engines for the "facts", and a visualization tool. However, it is well known that different fact extractors have different relative strengths and weaknesses; languages and dialects modelled, level of detail extracted, robustness (particularly with malformed code or "cross-compilation" extractions without proper libraries), and ease of fact manipulation all vary significantly between tools.

We have some experience in adapting fact extractors from other tools for use within the PBS system as part of the TAXFORM project [2]. Most recently, we have created automated mechanisms for translating output from the Acacia system's [4] C and C++ extractors for use within PBS [5,11]; we used these translators to create software architecture models for two large software systems [6]: the Mozilla web browser (over 2.2 million LOC of C++ and C), and the VIM text editor (over 150,000 LOC of C) [12].

Our particular goal in attending this workshop is to push for a *concrete* agreement on data schemas for procedural and object-oriented program components for use within reverse engineering tools. We are concerned that a "perfect" solution that pleases everyone and works at all levels of detail for all kinds of systems will never come to fruition, based on the outcomes of previous discussions on this topic at various conferences and workshops.

Syntactic and semantic issues

In our work on TAXFORM, we have used the TA language to represent information about program entities and their relationships; we have done so as we have extensive experience using TA, because our own tool (PBS) understands it, and because the relational calculator language `grok` allows for the creation of simple transformations of information stored in TA. However, we feel that the particular syntax to define an exchange format is a small issue, and other notations are attractive. For example, XML and XMI have much wider tool support than does TA. We consider the semantic model (design of the schemas) to be the most important issue, and do not further discuss syntax.

"TAXFORM utopia"

Ideally, we would like to foster an agreement on the modelling of program entity and relationship schemas, high level (architectural) schemas, and visualization schemas as shown below in Fig. 1 [2]. However, we recognize that this level of agreement between researchers is unlikely to occur in the near future.

Realistically, we would like to see two resolutions come out of this workshop:

1) an agreement on abstract schemas for representing top-level procedural and object-oriented programming language entities and their interrelationships, and
2) a commitment from reverse engineering tool creators to support output in the exchange format using these schemas.

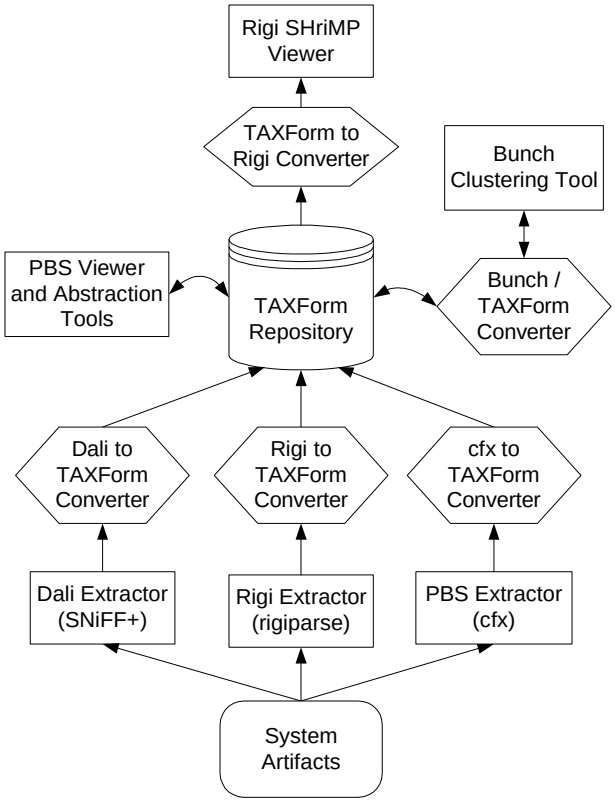

Figure 1. Desired architecture of a reverse engineering too that uses TAXFORM.

Detailed requirements

Our particular interest is in modelling attributes and relationships of top-level programming language entities (*e.g.*, functions, variables, methods, constants, typedefs) and their containers (*e.g.*, files, modules, classes, packages), rather than lower-level facts such as ASG-type information at the level of loops and if-statements. We recognize that this is not true for many workshop participants, but we feel that our particular goals are more modest, less controversial, and eminently achievable.

At a minimum, we would like the following information to be modelled:

Top-level programming language entities:

- functions, variables, constants, type definitions (procedural languages)
- methods, class member data, static methods and member data (object-oriented languages)

Entity containers:

- files, modules, classes, packages

Entity attributes:

- Name, unique identifier (UID — see next section)
- UID of container, UID of containing file (if container is not a file)
- Signature/data type
- Line number information (see below)
- Declared scope/visibility, static or not, final or not
- Definition or declaration (see below)

Entity container attributes:

- Name, UID
- Relative path (if a file), version identifier (if provided)
- UID of container (if not a file), UID of containing file (if not a file)

Relationships:

- Function calls, variable uses
- Line number information (see below)
- Container use/inclusion (by other containers)
- Inheritance (various kinds)
- "Friendship", various template relationships

Relationship attributes:

- Line number information (see below)

Problems

As we have discussed elsewhere [2], there are a number of issues that must be resolved in creating an exchange format. While most are not technically difficult to resolve, they all require that some agreement be reached. These issues include:

1) *Unique identification of entities* — Many programming languages allow the same name to be used by different entities as long as either their scopes are distinct or the entities are of different kinds. However, a fact extractor must be able to distinguish easily between like-named entities to resolve relationships.[1] One approach, as used by the older Acacia extractor `cia`, is to generate a unique identifier by some simple means, such as a counter. This has the disadvantage of not being portable across systems or between extractions. A second approach is to use a "name mangling" or hashing technique based on a composition of entity attributes that are enough to uniquely identify each entity. This approach works well, but the UIDs are then usually long or cryptic and it requires that a particular convention be agreed upon.
2) *Resolution of entities* — Suppose within the definition of a function named `f` we see a call to another function named `g`. Which `g` is being called here and how do we tell? Some reverse engineering tools, such as the Datrix fact extractor [7], do not resolve relationships to the entity definitions themselves, using a "name-only" approach to extraction. Other extractors resolve relationships to the *declaration* of the entities, and others to the *static definition* of the entities. Object-oriented languages and the use of function pointers

[1] Some languages, such as Java, insist that entity names use scoping information as part of the entity name to disambiguate *e.g.*, `System.out.println`(). Object-oriented languages have the additional problem of disambiguating calls to overloaded functions and operators.

demand resolution to the *dynamic definition* to get the "full story", although of course this cannot be accomplished by static analysis alone.

3) *Line number ranges* — Many entity definitions and relationships take place entirely within one line of source code. Other definitions and relationships, such as function definitions, take place across one contiguous block of source code. Still others, such as C++ namespace definitions or multiple references to the same global variable within a function definition, may span multiple non-contiguous lines or blocks of code. Therefore, to model these relationships "properly", line number information must be represented as a list. However, many fact extractors make the simplifying assumption that relationships exist within a single line of code and the entity definitions are comprised of a single block of code.

Case studies

We have created translation mechanisms for adapting output from the Acacia C and C++ extractors for use within the PBS system. We have successfully used these translators in the creation of software architecture models of several software systems, including the Mozilla web browser (2.2 MLOC of C++ and C), the VIM text editor (150 KLOC of C), the GNU `tar` utility (25 KLOC of C) and the `ctags` utility (10 KLOC of C).[2]

We found that the (Acacia) extraction and subsequent translation into TA of VIM's "facts" and those of smaller systems to be slightly faster than a typical full compile of the systems, and slightly slower than simple extraction using the native PBS extractor `cfx`. We found that the overall quality of the facts extracted by Acacia was higher than that of `cfx`, which has since led to improvements in `cfx`.[3]

However, the extraction and translation of Mozilla was much slower. Mozilla compiled on our Linux system[4] in about 35 minutes, while the Acacia extraction took three and a half hours and the translation into TA took another three hours. We note that the amount of information extracted was very large: almost one million "facts" taking up over 133 megabytes of diskspace (uncompressed). The software architecture model derived from this can be found on the web [8].

Credits

Ivan Bowman did much of the preliminary work on TAXFORM [2], and performed several experimental extractions using other extractors (including Rigi). Bowman also created source code and byte code fact extractors for the Java programming language [3].

The current author (Godfrey) wrote the original translation scripts for the Acacia C extractors to work within the PBS system, and performed a comparative analysis of output from the Acacia and PBS C extractors using the VIM system [12] as a guinea pig [6].

Eric Lee rewrote Godfrey's scripts to work with Acacia's C++ extractor and wrote additional scripts to allow the PBS visualization tool to model software architecture views of systems written in object-oriented languages. Lee also performed the extraction of facts from Mozilla [6,8,9].

[2] We describe our experiences with Mozilla and VIM in more detail elsewhere [6].

[3] We also found a few serious problems with the Acacia extraction output; for example, the extractor generated multiple UIDs for some entities, which introduced errors in the relationship resolutions. We subsequently adopted out own "name mangling" convention that solved this problem for us [6].

[4] Our Linux system was a dual processor Pentium III 450 MHz with 512 megabytes of RAM running Redhat 6.1.

References

[1] Armstrong, Matt and Chris Trudeau. Evaluating Architectural Extractors. In *Proc. of the 1998 Working Conference on Reverse Engineering* (WCRE'98), Honolulu HI, October 1998.

[2] Bowman, I. T., Michael W. Godfrey, and Richard C. Holt. Connecting Architecture Reconstruction Frameworks. In *Proc. of Symposium on Constructing Software Engineering Tools* (CoSET'99), May 1999. Also published in *Journal of Information and Software Technology*, 42(2), February 2000.

[3] Bowman, I. T., Michael W. Godfrey, and Ric Holt. Extracting Source Models from Java Programs: Parse, Disassemble, or Profile? In preparation, preprint available from `http://plg.uwaterloo.ca/~migod/papers/`.

[4] Chen, Y-F, Emden R. Gansner, and Eleftherios Koutsofios. A C++ Data Model Supporting Reachability Analysis and Dead Code Detection. In *IEEE Trans. on Software Engineering*, 24(9), September 1998.

[5] Finnigan, P., R. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. Müller, J. Mylopoulos, S. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. In *IBM Systems Journal*, 36(4), November 1997.

[6] Godfrey M. W., and Eric H. S. Lee. Secrets from the Monster: Extracting Mozilla's Software Architecture. In *Proc. of Second Symposium on Constructing Software Engineering Tools* (CoSET'00), Limerick, Ireland, June 2000.

[7] Lagüe, B., C. Leduc, A. Le Bon, E. Merlo, and M. Dagenais. An Analysis Framework for Understanding Layered Software Architectures. In *Proc. of the 1998 Intl. Workshop on Program Comprehension* (IWPC '98), Ischia, Italy, June 1998.

[8] Lee, E. H. S. The Software Bookshelf for Mozilla. Website, `http://swag.uwaterloo.ca/~ehslee/pbs`.

[9] Lee E. H. S. "Mozilla: Its Extracted Software Architecture". In preparation.

[10] Müller, H. A. Criteria for Success of an Exchange Format. Workshop meeting minutes, CASCON'98. 30 November 1998, available at: `http://plg.uwaterloo.ca/~holt/sw.eng/exch.format/criteria_muller.html/`.

[11] PBS Homepage. Website, `http://www-turing.cs.toronto.edu/pbs/`.

[12] VIM Homepage. Website, `http://www.vim.org/`.

[13] Woods, S., Liam O'Brien, Tao Lin, Keith Gallagher, Alex Quilici. An Architecture for Interoperable Program Understanding Tools. In *Proc. of the 1998 Intl. Workshop on Program Comprehension* (IWPC '98), Ischia, Italy, June 1998.