

Datrix™ Source Code Model and its Interchange Format: Lessons Learned and Considerations for Future Work

Sébastien Lapierre
sebastien.lapierre@bell.ca

Bruno Laguë
bruno.lague@bell.ca

Charles Leduc
charles.leduc@bell.ca

Bell Canada, Quality Engineering and Research
1050 Beaver Hall Hill, Montréal, Canada, H2Z-1S4

ABSTRACT

The Datrix team within Bell Canada has been offering its source code analysis tools to the research community for a number of years. These tools perform a number of analyses that revolve around a central model (*Datrix-ASG*) developed by the Datrix team, and use an interchange format similar to *TA*, which we call *Datrix-TA*. This paper intends to communicate the modeling choices that were made when creating this information model, and the lessons learned over a few years of usage.

Keywords

Datrix, ASG, TA, source code analysis, interchange format.

1 INTRODUCTION

The name Datrix identifies a project, a set of tools and a team of engineers within Bell Canada. Our mandate within Bell is to perform source code analysis of software projects developed or maintained either internally or by any of our suppliers. The goal of such an analysis is to assess the maintainability of these software products from a source code perspective. A number of elements can be considered when assessing a project's maintainability. As of today, the Datrix code analysis environment allows to study areas such as:

- appropriate software reuse/out-of-control software cloning
- evolution of the system, from a function, clone and class perspective
- clean software modularization and interfaces between sub-systems (a facet of architectural analysis)
- coding conventions, styles, structure and clarity

Other analyses are expected to come out of our R&D efforts. Our team within Bell Canada is directly involved in research areas such as flexible parsing [6], code duplication/cloning identification [7], software evolution analysis [8], architectural analysis [9, 10]. In addition, we are supporting two important R&D contracts with universities. These last projects are part of a Canadian consortium for research in software engineering, the CSER [2]. One of the many advantages of being a CSER member is the possibility to exchange tools and data between researchers. Central to this potential for collaboration is the availability of a means to build an abstract model of a system, and to be able to transfer this model across teams and tools. It should also allow the development of any kind of static source code analysis tool that can be based entirely on this model.

A parsing phase is inevitably the starting point of any source code analysis. One of our main contributions in the consortium so far has been to offer other CSER members our Java and C++ parsers. This article will discuss a number of points of interest concerning the model and interchange format of the information outputted by these parsers.

In the following sections, we will discuss:

- Datrix-ASG: modeling decisions made, and why
- Datrix-TA: interchange format chosen and why
- Observed advantages of using Datrix-ASG/TA
- Simple improvements needed to Datrix-ASG/TA
- what's coming - open questions for future design issues & preliminary design/answers

2 MODELING DECISIONS IN DATRIX-ASG

Our model is based on the concept of an ASG, which stands for *Abstract Semantic Graph*. Here is a short description of an ASG taken from [1]:

"The role of a parser is usually limited to producing an *abstract syntax tree* (AST). The AST is then processed to extract semantic information such as identifier' scope, variable' type, etc. This information is added to the AST as node attributes, edges or any other kind of annotations, resulting in an *abstract semantic graph* (ASG)."

In choosing the model for our ASG, our main objectives-goals were to:

- be able to represent all the relevant basic facts (lexical, syntactical, and semantic) from the source code for further analyses, such that those analyses would not require to return to the source code to retrieve some missing information; in other words, all the basic facts about the analyzed software (ex.: functions, assignments, classes, etc.) should be captured within the model;
- be able to commonly represent similar languages whenever possible (C-C++-Java-Pascal-Ada...) so that further analysis could be possible on the same model, no matter the language analyzed (this is important for us because the systems built by our suppliers are written in various languages, many being proprietary);
- enable analysis of software, not compilation or object code creation.

With these goals in mind, the ASG model was conceived based on standard AST terminology, with strong influences from other projects such as *gcc* and *Gen++* [11]. The Datrix-ASG is composed of nodes and edges representing elements of the source code analyzed. The model will not be described here in details but such details can be found in [1].

3 INTERCHANGE FORMAT SELECTED: DATRIX-TA

In choosing the interchange format to exchange our ASG, our main objectives-goals were to:

- express the ASG into a linearized format. After all, it is the goal of a fact extractor to linearize something which was first complex; otherwise, the best compact and structured representation of the source code would probably be the code itself;
- put the barrier as low as possible for anyone who wishes to develop a parser that can read the ASG;
- attribute no semantics to the ordering of nodes and edges within the output;
- enable easy understanding by humans, without documentation;
- enable one to easily add links between elements.

The interchange format chosen is a TA-like format (see description of TA in [3]). Here is an excerpt from [4] that gives an idea about the TA format:

"Inspired by RSF, TA retains the multigraph (or equivalently, the entity-relationship) meta-metamodel as well as the basic 3-tuple of space-separated test strings to record facts about a system. However it extends and refines RSF in the following ways. Firstly, it defines a distinct sub-language to specify node and arc attributes, and secondly, it provides the means to include metamodel information within the interchange document itself by clearly separating and identifying the metamodel and model data. These are called the *Scheme* and the *Fact* sections, respectively."

The current version of Datrix-TA does not provide the scheme section, only the fact section. The implications of that choice will be discussed in section 5.

4 OBSERVED ADVANTAGES OF USING DATRIX-ASG/TA

1- ASCII-based format:

The ASCII-based and thus human-readable format of Datrix-TA enabled us to easily debug input/output functionalities and tools. In addition to being ASCII-based, no effort was made to shorten the keywords used, such as *cInheritance*, which would have yielded a smaller output. Having kept full-length terms greatly enhances the ASG's readability and comprehensibility, as opposed to having used more cryptic ASCII-based notations, which are difficult to understand and debug. This translates in a very open format that allows anyone interested in the output to study it and comprehend with relative ease before further processing.

As an example of readability, the inheritance relationship between two classes is simply termed *cInheritance*, instead of being represented by a hieroglyph such as @ or &. With such a format, almost any input/output problems can be detected with the aid of any simple text editor (emacs in our case). Even though the format is not and was not intended to be compact, its ASCII content makes it a perfect candidate for compression utilities (zip, gzip, etc). Furthermore, the fact that the output contains a lot of repeated strings makes it a good candidate to compression algorithms such as *Lempel-Ziv*.

2- Linearized format:

Having to take as input a linearized representation of the model makes the reader tool (parser for DATRIX-ASG/TA) as easy to build as possible. It is in fact composed of a simple 10 rule YACC-based grammar.

3- Model Completeness

So far, all analysis tools we have built can be run using only the DATRIX-ASG/TA as input. The source code is not needed. That means that a given system can be parsed only once (usually a lengthy phase for system made of multimillion lines of code), even though many further analyses can then be performed.

5 SIMPLE IMPROVEMENTS NEEDED TO DATRIX-ASG/TA

1- Lack of Scheme:

The lack of a Scheme description part in the output header makes it harder to communicate changes in our model/format, having to rely on added informal documentation to tell changes. Furthermore, this added documentation, in the form of an attached *pdf* document, probably is rarely thoroughly read, is detached from the tools themselves and takes a significant effort to keep current.

2- Lack of identification of the metamodel:

Because we currently do not provide administrative information such as the model name and version, it is demanding to keep track of model changes and maintain tool compatibility through time.

3- Subgraph extraction:

Due to the linearized representation of the code in the DATRIX-ASG/TA format, when one wants to extract a subgraph of the ASG (for example the function call graph) from an ASG generated by one of our parsers, it cannot be done using only grep-like tools. The whole ASG structure needs to be read, traversed, and selected info extracted. This can become a burden to potential users of our tools. However, we could make readily available generic converters (e.g. ASG2FunctionCallGraph) for people who do not want to get into parsing/traversing/extracting work if all they need is a subgraph of the ASG.

4- Terminology:

We currently use a traditional terminology, as opposed to a more broadly accepted terminology such as the one used in the UML model. Although the same concepts are represented both in our model and in this popular model, having used the same terminology as the UML would have enabled easier discussions between users of both our and UML models.

Note that the fact that the output takes a lot of disk space (compared to binary, i.e. object file), is not considered a disadvantage in itself; as mentioned above, Datrix-TA files can be easily compressed with great efficiency using readily available tools.

6 CONSIDERATIONS FOR FUTURE WORK

6.1 Obtaining an ASG at system level

As of the writing of this paper, the Datrix team has built a C++ and a Java parser, which both output a partial ASG. In the case of the C++ parser, the ASG reflects the semantically enhanced AST computed from a preprocessed source file (which corresponds to a compile unit (cu)); we therefore call this ASG an ASG_{cu} . Because the notion of preprocessing is not a part of the Java language, a compile unit corresponds exactly to the scope of the source file; in this context, our Java parser outputs an ASG_f , which corresponds to the semantically enhanced AST of the source file (f) parsed.

In order to make useful findings in a project under study, it is not sufficient to have only access to semantic links between elements at file or compile unit level; the full ASG at system level is needed (ASG_{sys}). However, computing this ASG_{sys} implies several non-trivial computations in order to:

- establish the linking relationship between the set of available ASG_f/ASG_{cu} , in order to group them into linkage units (libraries and executables) and system (collection of linkage units);
- establish semantic relationships between elements of different ASG_f/ASG_{cu} 's, relationships which were not possible to establish before the linking phase was done.

The possible paths to computing an ASG_{sys} from ASG_f/ASG_{cu} 's can be represented in figure 1.

As of the writing of this paper, we have not yet thoroughly analyzed the implications of the different paths leading to an ASG_{sys} . However, our preliminary considerations include:

- preferably have a unique linker tool and a unique semantic enhancer tool taking as input any ASG_{cu} 's coming from the C++ parser or any ASG_f 's coming from the Java parser and producing an ASG_{sys} ;
- some information required for establishing what are the constituents of an LU are not available in the source code. It can only be found in the build information (such as makefiles for C and C++ systems).

Preliminary thoughts has lead us to assume that the linking phase should be done first, and afterwards the semantic enhancements could be done more easily. However, it is possible that another viable approach be

based on interleaving linking and semantic enhancements phases, until the full ASG_{sys} is obtained.

Performing semantic enhancements will introduce new relationships, which will require modeling to be added to our ASG_{sys} . This will give us a good opportunity to adopt the UML terminology and thus move our model closer to UML's design and terminology.

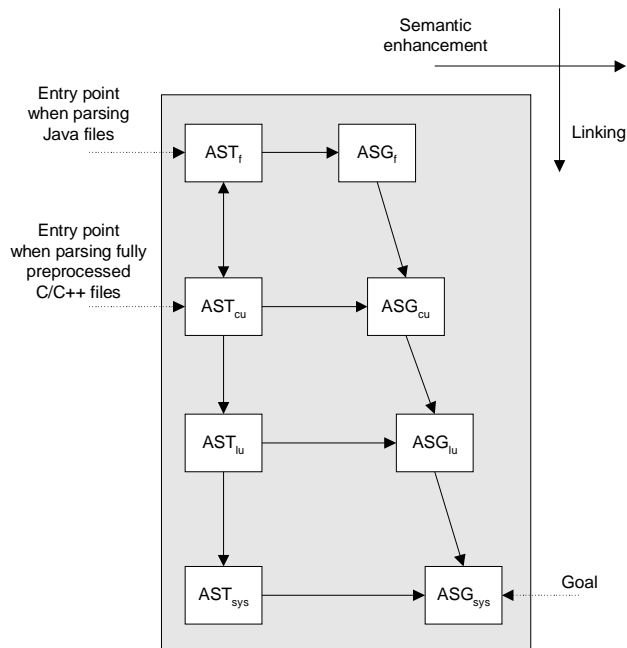


figure 1: possible paths leading to an ASG_{sys}

6.2 ASCII Repository design for Interchange

An extremely important issue that will have to be addressed concerns the physical design of the repository for interchange. While doing analyses in our lab, the ASG_{sys} will probably have to be stored into a database, for it will be huge and would rapidly become too big to handle if it was contained in files. But we want to keep the ability to exchange ASG's, even very large ASG_{sys} 's, with other researchers, using a plain ASCII format. A number of possibilities can be stated (good and bad, subject for discussions), namely:

- 1 file for the complete ASG_{sys}
- 1 file for each ASG_f , partitioned into sections (imports, exports/exportable, includes, internal)
- 1 file for each ASG_f (not partitioned)
- n files for each ASG_f (imports, exports/exportable, includes and internal sections split in distinct files).

Reference naming in an ASG_{sys} is also an issue to be addressed properly. Among the possibilities are:

- the usage of mangled name strings,
- unique numbering of each ASG node

6.3 Communication with our users

Communication with our users is not currently a straightforward process, partly because we mostly rely on e-mails to send informations, updates and new releases of our tools and licenses. In order to facilitate the access to all the information we expect to provide to our community of users a new web site, which will act as a central information center, offering:

- latest and downloadable binaries for each of our parsers, for all platforms (Sun, HP-UX, Linux/pc, WindowsNT),
- latest and downloadable binaries for utilities such as filters (e.g. ASG2FunctionCallGraph, ASG2InheritanceGraph, ...) and release notes,
- latest documentation

7 CONCLUSION

In this article, we presented the main modeling issues that were faced when creating our internal model and interchange format. Experiences using these model and format have proved to be encouraging although an effort could be made to reduce the terminology gap between our model and UML.

REFERENCES

1. Bell Canada Inc., DATRIX – Abstract semantic graph reference manual, version 1.2, Montréal, Canada, July 1999.
2. CSER: Consortium for Software Engineering Research <<http://www.cser.ca/index.html/>>
3. R. C. Holt, “An introduction to TA: The Tuple-Attribute Language”, March 1997.” <<http://www-turing.cs.toronto.edu/pbs/papers.html>>.
4. G. St-Denis, R. Schauer, R. K. Keller, “Selecting a Model Interchange Format, The SPOOL Case Study”, in Proceedings of 33rd Hawaii International Conference on System Sciences, Jan. 2000, Maui, Hawaii
5. Datrix R&D site, <<http://www.iro.umontreal.ca/labs/gelo/datrix/>>
6. Knapen G., Laguë B., Dagenais M., Merlo E., “Parsing C++ Despite Missing Declarations”, in Proceedings of the International Workshop on Program Comprehension, May 99, Pittsburgh, PA, USA.
7. Mayrand, J., Leblanc, C., Merlo E., “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics”, Proceedings of the International Conference on Software Maintenance, Monterey, California, USA, November 4-8, 1996
8. Laguë B., Proulx D., Mayrand J., Merlo E., Hudepohl J., “Assessing the Benefits of Incorporating Function Clone Detection in a Development Process”, in Proceedings of the International Conference on Software Maintenance 97, Nov. 97, Bari, Italy.
9. Laguë B., Leduc C., Le Bon A., Merlo E., Dagenais M., “An Analysis Framework for Understanding Layered Software Architectures”, in Proceedings of the International Workshop on Program Comprehension, June 98, Ischia, Italy.
10. Laguë B., Leduc C., “An Approach to Analyze the Decomposition of Object Oriented Systems into Source files”, OOPSLA 97 Workshop on OO quality, Oct. 97, Atlanta, USA. <<http://www.iro.umontreal.ca/~keller/Workshops/OOPSLA97/Papers/lague.bruno.ps.Z>>
11. P. Devanbu, <<http://seclab.cs.ucdavis.edu/~devanbu/genp/>>