Formalizing Fact Extraction

Yuan Lin [1]

School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON N2L 3G1, Canada

Richard C. Holt [2]

School of Computer Science
University of Waterloo
200 University Avenue West
Waterloo, ON N2L 3G1, Canada

Abstract

Schemas that define the form of facts extracted from source code are usually informal. The informal specifications of these schemas are too often incomplete or inconsistent. This paper takes the position that formal specification of fact extraction is beneficial to the reverse engineering community. A formal specification can serve as an unambiguous and reliable standard for people who use, write or verify a fact extractor. We explain how a formal specification for extracted facts can be derived from the source language grammar in such a way that the relationship between the code and its corresponding extracted facts is made clear. To support our position, we report our experience with formalizing a version of the Datrix Schema.

Key words: schema, formalization, fact extraction, software architecture

1 Introduction

Specifications of schemas for facts extracted from source code are generally informal and incomplete. Due to ambiguities in these informal specifications, each developer of a fact extractor has to decide details of how to emit various facts[1][12][8]. This leads to inconsistency among fact extractors, and therefore to trouble in exchanging information among reverse engineering tools.

[1] Email: y3lin@uwaterloo.ca
[2] Email: holt@uwaterloo.ca

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

This paper describes our approach to formalization of fact extraction. Our approach uses a series of transformations to define the relationship between

- the grammar for the source language and
- the schema for the extracted facts.

Using such a transformation-based definition of the schema, it is relatively straightforward to evaluate the completeness and usefulness of the schema. Such a definition provides the developers of fact extractor with a clear standard for writing and verifying the extractor, while providing users of the facts with a clear understanding of the meaning and form of the extracted facts.

2 Benefits of formalizing of fact extraction

Here is a list of four main benefits of our approach to formalizing fact extraction:

1) Close relationship between grammar and schema
2) Clear relationship between various schemas
3) Clear standard for extracted facts
4) Basis for verification of fact extractors.

We will now describe these in some detail.

2.1 Close relationship between grammar and schema

Our position is that a schema for extracted facts should be closely related to the source language grammar. The starting point of writing a schema should be the source grammar. By enumerating a sequence of transformations to transform a source program to its representation as extracted facts in the schema, the designer of the schema can see that the schema is complete, i.e., that all grammatical variations have been dealt with. These defining transformations make it relatively straightforward to estimate the difficulty of writing a fact extractor based on the schema.

2.2 Clear relationship between various schemas

In our experiments in formalizing the Datrix[2] schema, we found it convenient to start by transforming source code into a standard intermediate representation. This transformation carries out various housekeeping functions, such as producing regularized ways of representing expressions and statements. This standard intermediate representation encodes all grammatical variations without ambiguity. We then defined further transformations to generate facts in the format defined by a version of the Datrix schema. We expect that we can also find the transformations from this intermediate representation that generates facts in the format defined by the Columbus schema[9]. This intermediate schema can save designer of schema time and effort spent on details of a particular language.

With this approach we divide the specification of a schema into a sequence of sub-specifications. The initial sub-specifications are largely independent of the details of the form of the final extracted facts. The final sub-specifications for different target schemas (such as Datrix and Columbus) can be compared to gain a clear understanding of the relationship between these schemas.

2.3 Clear standard for extracted facts

Our approach forces the designer of schema to think more carefully about the concrete grammar of the source language and as a result to give a clearer specification of the schema. Because a reliable schema is vital to exchange of information, the extra work is worthwhile. With a less exacting approach, there is the danger of a lack of consistency among fact extractors because the specification of schemas is ambiguous and tends to omit low level but essential details.

2.4 Basis for verification of fact extractors

TXL[3][6] and similar tools input the specification of a transformation, integrated with a corresponding context free grammar, and perform the transformations on source code. This means that TXL can be considered to be a mechanical interpreter for our schema specifications, because our specifications are based on a sequence of such transformations.

We have found that many of our transformations used in defining a schema are reversible. When this is the case, we can write reverse transformations. When the target schema is source complete (contains sufficient information to reconstruct the source program), or nearly so, we can use these reverse transformations to recover the source code from the factbase. Using reverse transformations we verified CPPX, which is a fact extractor developed by SWAG team in University of Waterloo.

Treating fact extraction as a series of transformations is also proposed by the development team of CPPX[5]. However, our approach uses transformations on a context free grammar instead of on a graph; this has the advantage that context free grammars are supported by many tools.

3 Case study: formalizing the Datrix schema

The Datrix schema[2] defines, for the C/C++ language, the form of extracted facts as an abstract semantic graph (ASG) schema[2]. CPPX (C++ Fact Extractor), based on GCC, was developed as a fact extractor that reads C/C++ and emits facts according to the Datrix schemas[5]. RCPPX (Reverse CPPX) was developed as a tool used in verifying CPPX[11].

The current specification of Datrix schema is informal. It defines the Datrix schema by giving example fragments of C/C++ and corresponding examples fragments of ASGs. This approach defines ASG as an AST with embedded

semantic information. In Datrix schema, the AST is comprised of a nominal tree of nodes representing types, declarations, expressions and statements. Semantic information is added to AST as attributes (public, private, static or volatile, etc.) and semantic edges (from reference of a name to the definition of the name).

By contrast, our specification is a sequence of transformations from source grammar, in current version, C++ to Datrix schema. Because the semantic information in the ASG can be derived from the AST, we divide the specification into two parts: 1) from C++ to AST and 2) from AST to ASG.

The transformation from C++ to AST can be further divided into four steps:

1) eliminate ambiguity over type and variable
2) flatten
3) postfixify
4) generate Datrix AST.

We explain these four steps.

3.1 Eliminate ambiguity of types and variables

The C++ grammar[13] is known to be ambiguous regarding types and variables. More seriously, these ambiguities cannot resolved at the level of a context free grammar[10], which means tools like TXL cannot tell a variable from a type in particular circumstances. Our solution is to introduce an oracle that can identify names of type in a program. In our experiment, we use JLEX to generate this program.

3.2 Flatten

We will illustrate the flattening process by an example. The following declaration

```
int x, y;
```

has the same meaning as the flattened form:

```
int x;
int y;
```

The flattened form regularizes the intermediate representation, by decreasing the number of ways to represent equivalent source code fragments. The flattened form is closer to target ASG specified Datrix schema. In the flattening process, we use transformations on context free language, which can be carried out by TXL, to transform the first declaration to the second one. The result of this step is one declaration contains only one variable and all attributes of the variable are a part this declaration.

3.3 Postfixify

As this transformation begins, all types in the source program have been identified and tagged[4][7], and declarations have flattened. We can now deal with a program as a series of separate declarations, expressions and statements. The current version of Datrix schema also treats a program as a series of separate expressions, statements and declarations. It specifies the emitted facts informally, using fragments of source code and ASG. By contrast, we specify the schema using transformations based on context free grammar. In order to make our specifications useful beyond Datrix schema, we translate the program into an intermediate representation and then translate it into Datrix representation later. We believe that we can also translate the intermediate representation into other schemas.

We choose a postfix form of expressions, statements and declarations as the intermediate representation. Postfix form of expression is well-known, but the postfix form of declaration and statement is not used very often, although it is not difficult. After transformations of this step, the source program becomes a series of statements, expressions and declarations in postfix form.

3.4 Generate Datrix AST

The last step is to generate Datrix AST. Datrix AST can be expressed as a bracket-denoted language, in which children nodes are contained inside bracket. The transformation from postfix representation to Datrix AST is straightforward in most cases.

3.5 Advantages of our approach

In some cases, a formal specification is too long to be reasonably validated by human, and therefore has only limited use. Our specification is not long. It is about five times as long as a context free grammar for C++, and programmers should have little difficulty reading it. Our approach has the further advantage that our specification is executable and can be tested on real programs. In summary, our approach allows us to ensure the quality of our specification by inspection and exhaustive test.

4 Conclusion

Our experiment indicates that the formalization of fact extraction can provide a reliable schema specification and a clear understanding of the relation among different schemas. Because most transformations we specify can be implemented by tools such as TXL, our specification is also a powerful tool to verify fact extractors. When efficiency is not the main goal in a project, our specification and TXL can also serve as a fact extractor prototype.

Our position is that formalizing fact extraction has potential to benefit

reverse engineering, by clarifying the relation between grammar and schema, by clarifying the relationship among various schemas, by providing a clear and unambiguous standard for extracted facts, and as a basis for verifying fact extractors.

5 Bibliographical references

References

[1] M.N. Armstrong et al., *Evaluating Architectural extractors*, Fifth Working Conference on Reverse Engineering (WCRE 1998)**9** (1998), 30–39.

[2] Bell Canada, "DATRIX(tm) Abstract Semantic Graph: Reference Manual" Version 1.4, Bell Canada Inc., Montreal, 2000.

[3] J.R. Cordy et al., *Source Transformation in Software Engineering using the TXL Transformation System*, Special Issue on Source Code Analysis and Manipulation, Journal of Information and Software Technology 44,13 **9** (October 2002), 827–837.

[4] A. Cox et al., *Representing and Accessing Extracted Information*, IEEE International Conference on Software Maintenance (ICSM 2001) **9** (Nov. 2001), 12–21.

[5] T. R. Dean et al., *Union Schemas as a Basis for a C++ Extractor*, Proceedings of WCRE 2001: Working Conference on Reverse Engineering, Stuttgart, Germany **9** October 2001).

[6] T. R. Dean et al., *Grammar programming in TXL*, Proceedings of Second IEEE International Workshop on Source Code Analysis and Manipulation. Montral.**9** (October 2002).

[7] M. Favre, *CPP denotational semantics*, SCAM 2003, associated with ICSM 2003

[8] R. Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, Tibor Gyimothy, *Towards a Stardard Schema for C/C++*, WCRE 2001: Working Conference on Reverse Engineering, Stuttgart, Germany.**9** (October 2001).

[9] Ferenc R. et al., *Columbus - Reverse Engineering Tool and Schema for C++*, Proceedings of the International Conference on Software Maintenance (ICSM 2002), IEEE Computer Society **9** (2002).

[10] John C. Martin,"Introduction to Languages and the Theory of Computation" 3rd Ed., McGraw-Hill, 2003.

[11] Yuan Lin, Richard C. Holt, Andrew Malton, *completeness of a fact extractor*, Working Conference on Reverse Engineering, Victoria BC Canada.**9** (November 2003).

[12] G. C. Murphy et al., *An Empirical Study of Static Call Graph Extractors*, ACM Transactions on Software Engineering and Methodology, 7(2) **9** (April 1998), 158–191.

[13] Bjarne Stroustrup, "the C++ programming language" Special Edition, Addison-Wesley, Pearson Education, 2001