Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 94 (2004) 93–102
www.elsevier.com/locate/entcs

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

Reverse engineering commonly uses fact extraction to transform source programs to factbases. These factbases are in turn used to determine particular views or aspects of the program, such as its architecture or its anomalous structures. Fact extraction is usually defined in an ad hoc manner, which is 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.

Keywords: schema, formalization, fact extraction, software architecture

1 Introduction

Fact extraction is a process of parsing source code and generating a factbase that stores information about it. Producing an accurate and reliable factbase

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

doi:10.1016/j.entcs.2004.01.001

is important because reverse engineering and program comprehension tools usually depend on factbase to analyze the software. It is also important for software tools that exchange information via factbase. However, writing a fact extractor, a program that extracts facts from source code, is often a challenging engineering problem, especially for languages like C++ due to the complexity of the language. Related research shows that fact extractors commonly don't agree and emit different facts on the same source program [1][12][8]. This tends to undermine users' understanding of the program and decrease their confidence in the extractor. This inconsistency among fact extractors also leads to trouble in exchanging information among reverse engineering tools.

According to [8], a *schema* is a description of the form of the data, in terms of a set of entities with attributes and relationships that prescribe the form of the instance data. There are particular schema for factbases, for example, Datrix and Columbus[2,9]. These schemas describe the form of factbase and relate entities in factbase to the source language. They also serve as a guideline of how to build the factbase.

In this paper, we will concentrate on fact extractors that input source code and parse it according to a context-free grammar. Not all fact extractors take this approach. For example, some use lexical, non-grammar approaches, and some analyze object code. However the approach we are concentrating on is commonly used.

Specifications of schema 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 on his own the detailed form of factbase, which leads to errors and inconsistency in factbases. This paper describes our approach to formalize fact extraction and create a complete, consistent and comprehensible basis for fact extraction. Our formalization of fact extraction consists of three parts:

1) syntax of source language
2) syntax of factbase and
3) a sequence of steps that map from source language to factbase.

A specification written using this approach is reasonably short and understandable. In the case of the C++ language, it is about five times the size of the C++ context-free grammar. Programmers should have little difficulty reading it. Our approach has the further advantage that the specification is executable and can be tested on real programs.

After careful inspection and testing, the specification can give a consistent and complete standard for developing and verifying the extractor and provides users of the factbase with a clear understanding of the meaning and form of the extracted facts.

2 Formalization of fact extraction

A fact extractor can be considered to be a mapping or transformation from source programs to factbases. Therefore, the specification of fact extraction can be divided into three parts:

1) specification of the domain (syntax of source language),
2) specification of range (syntax of factbases), and
3) the mapping from domain to range.

In our approach, the first part is given as the context-free grammar for the source programs. The second part gives the form or schema of the factbase, again as a context-free grammar. The third part, the mapping, is responsible for maintaining the semantics (or particular aspects of the semantics) for use in tools that input the resulting fact base.

A common mechanism for dealing with parsed source programs is to represent them internally as an Abstract Syntax Tree (AST), and our approach will use this mechanism. The AST can be *decorated* to become an Abstract Semantic Graph (ASG) by adding various semantic information, usually at least links connecting variable references to corresponding declarations. In the case of Datrix, the factbase represents a variant of the ASG, so this approach is particularly convenient.

As mentioned earlier, the range of fact extraction, i.e., the form of the factbase, can also be specified as another context free language. The factbase can be considered to be an (possibly simplified version of) ASG, which can easily be represented in a form that has a context-free grammar. To illustrate this, consider this C++ expression:

```
a + 3;
```

In our case study, the textual form of the ASG for this expression is represented as:

```
cOperator (op = bplus)
{
   a:  nameref (refers id = 100)
   3:  literal
}
```

The first line indicates that this is an additive expression, the brackets denote the variable a and constant 3 are contained by, or parts of, the expression, where a refers to a variable defined elsewhere and given an identifier 100. This fragment of the ASG gives the flavour of how our approach transforms a source program into an AST, later into an ASG, and eventually into a factbase.

The specification of transformation from source program to ASG is the

fundamental to our formalization. It is the designer's job to identify pieces of meaningful syntactical structure in source language, such as declaration, expression and statement, and then specify a sequence of transformation steps to map these structures to their equivalent ASGs. For the specification to be valid, every step of the transformation has to guarantee that 1) the generated ASG is well-formed, i.e. follows the syntax of ASG and 2) the structure before transformation and generated ASG have the same meaning. For specification to be complete, it must deal with all the grammatical variations of source language, i.e., with all parts of the source context-free grammar.

We believe that validity could only be achieved by extensive testing and the success of our approach depends on tools that can test our specification automatically. Fortunately, this tool already exists. TXL is a rule-based language that performs transformations on context-free language. It can input our specification and performs transformations in our specification on source code.

In summary, we specify the syntax of source programs and ASGs in context-free grammars, and we can specify the transformation from source program to ASG as transformations on context-free grammar. Because tools like TXL support our specification, we can use testing to validate the consistency and completeness of our specification.

3 Benefits of formalizing of fact extraction

Our formalization of fact extraction and our use of transformations based on context free grammar can benefit schema designers, fact extractor developers and the users of fact extractor. 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.

3.1 Close relationship between grammar and schema

This paper focuses on fact extractors that parse the source program based on a context-free grammar. In such an extractor, the factbase can be expected to be closely related to the source program. Our approach to specifying fact extraction takes advantage of this closeness, thereby minimizing the complexity of the fact extraction specification. The approach requires the designer to gain a thorough understanding of the source program grammar as well as

the form of factbase. Based on this understanding, the designer creates a sequence of transformation steps to map a source program to its representation as extracted facts, being careful to see that all parts of the input grammar are dealt with. Our experience indicates that creating these transformations is time consuming, and both inspection and extensive testing are needed to insure the specification is consistent and complete.

A by-product of this process is that the complexity of these transformations is a rough indicator of the amount of work needed to write a fact extractor based on the schema.

3.2 Clear relationship between various schemas

In our experience in formalizing a version of extraction for the Datrix [2] schema, we found it convenient to start with a set of mapping steps that transform source code into a *standard intermediate representation*. These steps carry out various common housekeeping functions, such as producing regularized ways of representing expressions and statements. This standard intermediate representation resolves any ambiguities that might be present in the source program. We then defined further transformations to map the facts to the format defined by the Datrix schema. We expect that it is not difficult to create transformation sets from this intermediate representation to the format for other schemas such as the Columbus schema. [9] This standard intermediate schema can save the designer of a schema the time and effort spent on details of a particular language such as C++.

Our approach divides the transformation into a sequence of steps. The initial steps are largely independent of the details of the form of the final extracted facts. The final steps are particular for different target schemas (such as Datrix and Columbus) and can be compared to gain an understanding of the relationship between these schemas.

3.3 Clear standard for extracted facts

Our approach forces the designer of schema to think carefully about the concrete grammar of the source language. As a result the specification is more likely to be complete and consistent. Because a reliable schema is vital to exchange of information, this 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.

3.4 Basis for verification of fact extractors

TXL [3][6] and similar tools input the specification of transformations, 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 fact extraction specifications, because our specifications are based on a series 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.

In related work, Andrew Malton et al [5] proposed the idea of treating fact extraction as a sequence of graph transformations. They used this approach in the development CPPX (C Plus Plus eXtractor) in 2001. By contrast, our approach uses transformations based on context-free grammars instead of graphs. Our approach has the advantage that it directly uses existing grammar-based tools such as TXL.

4 Case study: formalizing the Datrix schema

The research described here is part of the SWAG team's effort to make fact extraction more accurate and reliable. In 2000, Bell Canada created Datrix schema [2] for the Abstract Semantic Graph (ASG) for C/C++ programs. In 2001, the SWAG team developed CPPX (C++ Fact Extractor). This fact extractor performs a series of graph transformations on the output from GCC front end until it obtains a factbase compliant with the Datrix schema [5]. In addition, the SWAG team also developed RCPPX (Reverse CPPX), a tool that tests and verifies factbases generated by CPPX [11].

Bell Canada's specification of Datrix schema is informal. It defines the Datrix schema by giving example fragments of C/C++ and corresponding examples fragments of ASGs; an ASG is an AST with embedded semantic information. In the Datrix schema, the AST consists of a 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 a name to the declaration of the name).

By contrast, our version of the Datrix specification is a sequence of transformations based on the C++ grammar to meet the 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. We further divide the transformation from C++ to AST into four steps:

1) Eliminate ambiguity in types and variables,
2) Flatten,
3) Postfixify, and
4) Generate Datrix AST.

We will now explain these four steps.

4.1 Eliminate ambiguity of types and variables

The C++ grammar [13] is known to be ambiguous regarding types and variables. Consider this program fragment:

```
TypeOrVar * p;
```

This example can be a multiplicative expression of two variables *TypeOrVar* and *p* or a declaration of a pointer *p* that points to the type *TypeOrVar*, if *TypeOrVar* is defined as a type. These ambiguities cannot resolved at the level of a context-free grammar [1], which means tools like TXL cannot directly tell a variable from a type in particular circumstances. Our solution is to introduce an *oracle*, which is a program that distinguishes names from types. We use JLEX to generate this program. As for the above example, the oracle inserts the tag <type> when it determines that *TypeOrVar* is a type. As a result of this step, the program fragment is disambiguated by transforming it to:

```
TypeOrVar <type> * p;
```

The elimination of ambiguity is not an inherent requirement for fact extraction, but it is highly desirable, for otherwise the extracted factbase retains the ambiguity, which can lead to wrong interpretations by later tools. Therefore, our approach eliminates ambiguity.

4.2 Flatten

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

```
int x, y; (1)
```

has the same meaning as

```
int x;
int y; (2)
```

Declaration (1) is shorter and more convenient for programmers. However, Datrix does not allow this shorter notation. In the ASG, each time a variable refers to x or y, a *refers* edge is drawn from that variable to x or y. Therefore, x and y have to be separate nodes in ASG.

We call declaration (2) the *flattened form* of declaration (1). The flattened form simplifies 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 declarations to the flattened ones. The result of this step is each declaration contains only one variable and all attributes of the variable are parts of this declaration.

4.3 Postfixify

The initial steps tag any ambiguous variables or types [4][7] and flatten all declarations. In terms of the AST, each declaration corresponds to a particular node and each statement or expression corresponds to a particular sub-tree. We can now deal with a program as a series of separate declarations, expressions and statements. These initial steps are potentially useful for creating factbases other than Datrix, for example, Columbia factbases.

We chose a postfix form of expressions, statements and declarations as the standard intermediate representation. The sub-transformation that creates this format is called *postfixify*. The 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 this step has been carried out, the source program becomes a series of statements, expressions and declarations in postfix form.

4.4 Generate Datrix AST

The fourth step is to generate Datrix AST. The Datrix AST can be expressed as a bracket-denoted language, in which child nodes are contained inside brackets. The transformation from postfix representation to Datrix AST is straightforward in most cases, because the syntax of the Datrix AST is simple. For example, in Datrix, declarations, the most complicated part of C++ language, are all of the following form no matter it is built-in type, pointer or class:

```
p:  cObject
{
   cPointerType
   {
      int:  cBuiltInType
   }
}
```

This example declares p as a pointer to integer.

4.5 From AST to ASG

Up to this point, we have dealt with only the transformation from source code to AST. To create an ASG from an AST, we still need to add *refers* edges from name references to declaration. Because variables in different scopes can have the same name, we generate a unique key (a number) for every variable in the ASG. This mechanism is similar a symbol table, only much easier because the syntax of AST is simple.

4.6 Advantages of our approach

There is the danger that a formal specification can be so long and complicated that it is of limited use. However, with our approach, the specification of fact extraction is not particularly long or complex. It is about five times as long as a context-free grammar for C++, and programmers should not have much difficulty reading it. Our approach has the further advantage that the specification is executable and can be validated by testing it on actual programs.

5 Conclusion

Our experience indicates that the formalization of fact extraction can provide a useful schema specification and a clear understanding of the relation between different schemas. Because most transformations in these specifications can be implemented by program manipulation tools such as TXL, our specification is also a powerful tool to verify both the specification and the actual fact extractors. When efficiency is not the main goal, the specification and TXL can also serve as a fact extractor prototype.

Our conclusion 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.

Acknowledgement

The authors thank the referees for their valuable suggestions for improving the paper.

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