Towards A Portable XML-based Source Code Representation

Ying Zou and Kostas Kontogiannis

Dept. of Electrical & Computer Engineering
University of Waterloo
Waterloo, ON, N2L 3G1, Canada
{yzou, kostas}@swen.uwaterloo.ca

Abstract

Program representation is a critical issue in the area of software analysis and software re-engineering. It heavily relates to the portability and effectiveness of the software analysis tools that can be developed.

This paper describes an approach that focuses on source code representation schemes in the form of Abstract Syntax Trees that are encoded as XML documents. These XML source code representations conform with a DTD we call the domain model. By utilizing a domain model for a given programming language we can build tools on top of an XML DOM tree. The XML DOM tree has a standard API that all tools developed using this approach can use. In such a way, software analysis tools can interoperate or be easily integrated in what we refer to as Integrated Software Maintenance Environments.

1. Introduction

One of the greatest challenges for software analysis and software re-engineering is to design and implement parsers in order to access the intermediate representation of the source code. Even for a simple programming language, the effort to develop a parser could be very high [3, 6].

As an alternative approach, software analysis tools can be built on top of the existing CASE environments that keep representations of the source code of the system being analyzed in a proprietary information base format, and offer an API to access these internal source code representations. Such environments include Refine [1], Datrix, and IBM VisualAge for Java and C++ CodeStore [3]. However, any tool, which is developed based on such an information base, lacks portability and interoperability with other software analysis tools.

As a third alternative approach, we present a source code representation framework that is based on a domain model and a Document Type Definition (DTD) and provides a standard API for interoperable and portable software analysis to be built. In a nutshell, the process of software reverse engineering focuses on the decomposition of a software system into objects and relationships that are stored in an information base, and on the creation and transformation of various program views [4]. These views can be generated from the proposed XML representation of the Abstract Syntax Tree. In such a way, software analysis and re-engineering tools can be developed on top of XML based Abstract Syntax Trees instead of proprietary formats. These tools will be fully interoperable since they share the same API as the W3C's DOM tree API. In this paper, we present this approach and discuss its advantages and limitations.

The rest of the paper is organized as follows. Section 2 provides a brief introduction to the concepts pertaining to Abstract Syntax Trees. Section 3 presents the approach adopted on modeling source code in terms of XML documents and XML DOM trees. In section 4, a case study for developing domain models for the C programming language is presented. Related work is explored in section 5. Finally, section 6 provides pointers for on-going and future work.

2. Abstract syntax tree and XML

Abstract syntax trees

There is a spectrum of levels of granularity at which source code is represented. At the lowest level of detail, Abstract Syntax Trees (ASTs) are used. These contain information about the source program [6] in the form of nodes and edges [2]. Such tree-like structures represent the source program in a top-down matter. For example, C applications are represented at the top level as `applications`, `modules`, and `files`, while at lowest levels as `functions`, `declarations`, `macros`,

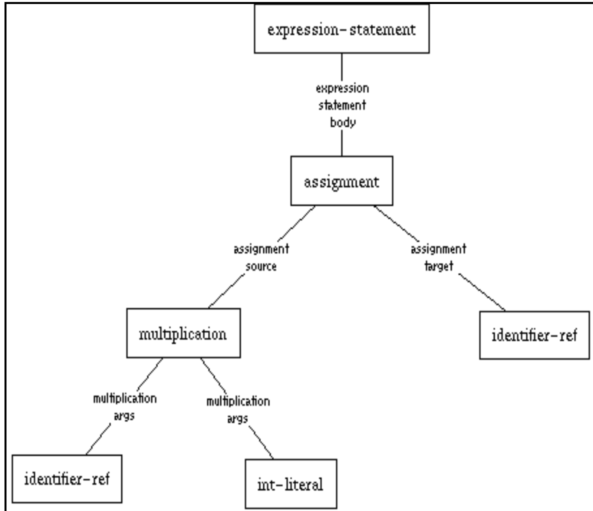


Figure 1: AST Structure for Expression Statement in C

`expressions`, and `identifiers` to name a few [1]. The internal nodes of the AST represent the non-terminal phrases such as `statements`, `operations`, `functions`, and the leaf nodes represent the terminal symbols, such as `identifiers`, and `empty declarators`. An edge denotes tree attributes, which are represented as mappings between AST nodes. AST nodes correspond to programming language constructs such as `If-Statements`, `For-Statements`, and `While-Statements`.

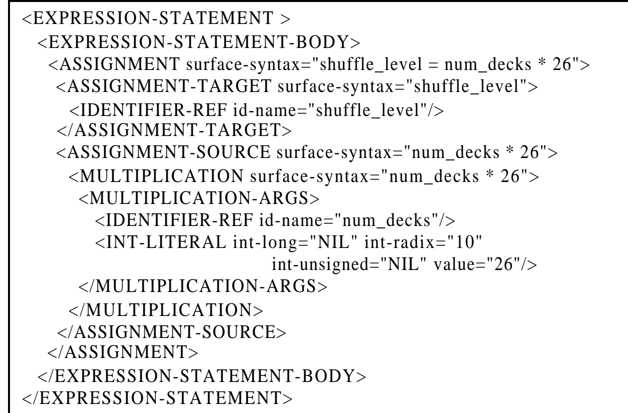
There are several issues for the successful representation of ASTs with the purpose of developing flexible software analysis tools. Namely, these issues include:

1. Decoupling the AST representation from specific parsing environments by utilizing domain models for programming languages,
2. Allowing for AST representations to be extended in order to allow tools to be built in an incremental way,
3. Using a flexible, open and standard API so that such analysis tools can be integrated into collaborative software maintenance environments.

These minimal requirements will ensure that software re-engineering tools can be quickly built, easily maintained, and provide multiple views of the system being analyzed.

Advantages of representing AST in XML

XML provides a unified framework for tagging structured data [5]. It is extensible, and can define meaningful tags that link syntax with the semantics of the entities for a given domain. Similar to AST, an XML document can be thought of as a tree structure with nodes and edges connected by a hierarchy relationship. Such a



```
<EXPRESSION-STATEMENT >
  <EXPRESSION-STATEMENT-BODY>
    <ASSIGNMENT surface-syntax="shuffle_level = num_decks * 26">
      <ASSIGNMENT-TARGET surface-syntax="shuffle_level">
        <IDENTIFIER-REF id-name="shuffle_level"/>
      </ASSIGNMENT-TARGET>
      <ASSIGNMENT-SOURCE surface-syntax="num_decks * 26">
        <MULTIPLICATION surface-syntax="num_decks * 26">
          <MULTIPLICATION-ARGS>
            <IDENTIFIER-REF id-name="num_decks"/>
            <INT-LITERAL int-long="NIL" int-radix="10"
                              int-unsigned="NIL" value="26"/>
          </MULTIPLICATION-ARGS>
        </MULTIPLICATION>
      </ASSIGNMENT-SOURCE>
    </ASSIGNMENT>
  </EXPRESSION-STATEMENT-BODY>
</EXPRESSION-STATEMENT>
```

Figure 2: XML Element Structure for Expression Statement in C

tree is called a Document Object Model (DOM) tree.

Compared to the custom-made software extractors, the DOM, provides a standard API for programmatic access to XML documents, and manipulates the structural data. In such a way, tool developers can consistently interact with, and compute XML-based documents. In addition, the DOM APIs are widely supported and can be bounded with different programming languages, such as Visual Basic, C++, Java, and JavaScript.

Finally, XML documents can be transported by the HTTP protocol over the Internet. Therefore XML allows for an AST information base to be independent of the parser, and be published in a Web-based repository, such as a BizTalk server [7]. The software analysis tools can either download the required information over the network or store the ASTs locally for further processing. Results from the analysis of large systems can also be represented as annotations of the AST in an XML form, and directly be stored in a Web repository.

3. Mapping AST into XML

There are two approaches to extract the Abstract Syntax Tree and encode it in XML. The bottom-up approach, utilizes the concept of a domain model definition that denotes the syntactic structures of a programming language such as Pascal, Fortran, C, C++ and Java. Tools that utilize this approach include Refine for C, Datrix for C++/C/Java, JavaCC for Java [8], and IBM VisualAge for C++ and Java [8].

The other approach, referred to as the top-down approach, examines the grammar of the specific programming language, and defines a standard logical structure for an annotated Abstract Syntax Tree. By following the rules, different parsers can extract the necessary information from the source code and encode it in a uniform and application-independent format.

Using a domain model definition extracted from the specification of a given programming language (i.e. ANSI

```
<?xml version="1.0"?>
<!element EXPRESSION-STATEMENT
                    (EXPRESSION-STATEMNT-BODY)>
<!element EXPRESSION-STATEMENT-BODY(EXPRESSION)>
<!element EXPRESSION(ASSIGNMENT |MULTIPLICATION |...)>
<!element ASSIGNMENT (ASSIGNMENT-TARGET,
                              ASSIGNMENT-SOURCE)>
<!element ASSIGNMENT-TRAGET
                    (EXPRESSION|IDENTIFIER-REF|... )>
<!element ASSIGNMENT-SOURCE
                    (EXPRESSION|IDENTIFIER-REF|...)>
<!element MULTIPLICATION(MULTIPLICATION-ARGS)*>
<!element MULTIPLICATION-ARGS
                    (IDENTIFIER-REF*|INT-LITERAL*|...)>
<!attlist ASSIGNMENT surface-syntax  CDATA #REQURIED>
<!attlist ASSIGNMENT-TARGET surface-syntax CDATA #REQURIED>
<!attlist ASSIGNMENT-SOURCE  surface-syntax CDATA #REQURIED>
<!attlist MULTIPLICATION surface-syntax CDATA #REQURIED>
<!attlist IDENTIFIER-REF id-name CDATA #REQURIED>
<!attlist INT-LITERAL int-long CDATA #REQURIED
                       int-radix CDATA #REQURIED
                       int-unsigned CDATA #REQURIED
                       value NUMBER  #REQURIED>
```

Figure 3: DTD for Expression Statement in C

C), we utilize a hybrid approach to define the logical structure of the entities of an Abstract Syntax Tree in terms of a Document Type Definition (DTD) document by following the steps below.

In the first step, we define the structure of the Abstract Syntax Tree for a specific language by developing a domain model for this language.

By recursively traversing the hierarchy of the domain model entities, we are able to map the given domain model to a Document Type Definition (DTD). Specifically, the tree hierarchical structure of the domain model is mapped to XML elements and attributes. Figure 1 illustrates an Abstract Syntax Tree that conforms to a given domain model and represents the C assignment expression, "`shuffle_level = num_decks * 26`". The non-terminal nodes are `expression-statement`, `assignment`, and `multiplication`.

The leaf nodes represent the terminal token, such as `identifier-ref` (identifier reference), `int-literal` (integer). Similarly the edges represent the attributes as mappings between AST nodes. For example, the edge, named `assignment-source`, is considered as an attribute of the `assignment` node that contains as a value a node of type `multiplication`. Each node and edge in the AST is mapped to an XML element tag as illustrated in Figure 2. The attribute values of an AST node are mapped to the corresponding attribute values of the XML elements.

In the second step, we focus on the enhancement and generalization of the domain model and common schema for the programming language being modeled. The domain model for a given language and its corresponding DTD can be enhanced with information such as unique identifier numbers, linkage, and analysis information. Similarly, domain model generalizations include the introduction of elements that relate to system constructs such as `system`, `module`, and `component`.

	Size of Source Code	Size of AST XML
TwentyOne	28,504 bytes	961,882 bytes
AVL Library	164,401bytes	1,660,167 bytes
Clips	971,294 bytes	22,298,518bytes

Table 1: Comparison between the Size of Source Code and AST XML Document

In this context, the grammar of the programming language being modeled defines the Document Type Definition (DTD) and consequently the organization of the XML document that models the AST of a given source code fragment. Conformance to the grammar of the programming language being modeled, ensures that the AST DTD specifies the logical structure of the XML element hierarchy and relates to the syntactic entities of the source code. Figure 3 illustrates the AST DTD to the AST in Figure1.

In the final step, we address issues on sharing AST XML documents between software re-engineering tools and software maintenance environments. The generated XML AST is usually larger in size than the source code size, because of all the tags and attributes that are added to the source code. The XML document is considered as a dynamic database that can be easily manipulated by the DOM API. Similarly, the XML AST can be stored in commercial relational databases by storing XML documents as one filed in a table, or multiple fields in one or more tables.

4. A case study

To investigate the usefulness of this approach, the domain model for the C programming language was examined and C source code for various systems has been represented in the form of an XML document and an XML DOM tree.

We have used the Refine/C Parser by Reasoning to obtain an XML version of the C source code. In this context, we could have used any parser for this task. We have chosen the Refine parser because of the flexibility of the API it offers and the rich domain model that defines in the form of language entities taxonomy. For example, the C domain model specifies that `multiplication` is a subclass of `arithmetic-expression`, which is in turn a subclass of the `expression` class type. Moreover, the domain model defines the attributes of these object classes in terms of mappings from one object class to another. The domain model also specifies which of these attributes define the structure of a C Abstract Syntax Tree and which attributes define annotations on the tree [1].

Based on ANSI C [10] grammar and leveraging the concepts in C domain model [1], we have created an AST XML structure that was specified in terms of a DTD.

Figure 3 shows part of the resulting DTD. Basically, it reflects the hierarchical nature of a recursive definition of a C language entity.

Examples of AST XML document can be found at [11]. Table 1 provides some comparison statistics related to the size of the original source codes and the size of the generated XML documents.

5. Related work

There is a growing momentum of activities related to XML representation of source code. In [8], a system for annotating object-oriented languages, including C++ and Java with XML tags has been presented. The work has utilized the top-down approach as discussed above. That is, the mappings from the Java and the C++ grammar to the corresponding AST DTDs are specified at the beginning. Consequently, semantic actions to a custom made parser have been added in order to annotate the input stream (source code) with XML tags. The C++ AST XML was generated from IBM VisualAge C++ Compiler, which retains the C++ AST in its CodeStore. The Java AST XML was based on a parser generator tool, called Java Compiler Compiler (JavaCC). The common structure between these two object-oriented languages is also specified and represented as OOML.

In [14], an InterMediate Langugage (IML) is proposed. It modules source language and keeps the programming language constructs for sophisticated data-flow and control-flow analyses. It was not designed for the flexible data exchange. In [13], based on IML, the Resource Graph (RG) is built on top of IML. It abstracts the globe information, such as call, type, and use relations. Such information is suitable for analysis, but not for fine-gained representation of source code.

In [6], the Graph Exchange Language (GXL) is proposed. It is a data exchange format among software analysis tools. GXL is designed for the representation of typed graphs.

Other program representation schemes, such as ASG, Program Dependence Graph, Rigid Standard Format (RSF), Tuple-Attribute Language (TA) and AsFix, represent the source code at different abstraction level, and are used in different program analysis tools. The high-level exchange schema between them is discussed in [15].

6. Conclusion

Abstract Syntax Trees provide the most detailed information of the source code used for software analysis. Representing ASTs as XML documents makes them widely accessible to various software analysis tools and heterogeneous software maintenance environments.

In this paper, we briefly discussed an approach that utilizes a domain model and consequently map these entities of the domain model to a DTD. A parser is used to generate a representation of the source code in terms of an XML document that is compliant with the given DTD. The XML document effectively corresponds to an AST that has a standard structure and API. We consider that such an XML based representation of the source code can be thought of as a first step towards CASE tool interoperability.

Future work includes the utilization of the XML based source code representation documents to develop tools that perform code transformations and facilitate source code migration from one operating platform to another [12].

7. References

[1] Reasoning Systems, "Refine/C Programming's Guide", June 1995.

[2] A. W. Appel and Maia Ginsburg, "Modern Compiler Implementation in C", Cambridge University Press, 1998.

[3] J. Martin, "Leveraging IBM VisualAge for C++ for Reverse Engineering Tasks", CASCON'99.

[4] R. S. Arnold, "A Road Map Guide to Software Reengineering Technology".

[5] "Why XML", http://msdn.microsoft.com/xml/general/whyxml.asp.

[6] R. C. Holt, et al., "GXL: Toward a Standard Exchange Format", 7th WCRE 2000.

[7] Microsoft Corporation, "BizTalk Framework 2.0 Draft: Document and Message Specification", http://msdn.microsoft.com/xml/articles/BizTalk/biztalkfwv2draft.asp.

[8] E. Mamas, K. Kontogiannis, "Towards Portable Source Code Representation Using XML", 7th WCRE'2000.

[9] "XSL Transformations (XSLT) Version 1.0: W3C Recommendation 16 November 1999", http://www.w3.org/TR/xslt.

[10] H. Schildt, "The Annotated ANSI C Standard American National Standard for Programming Languages – C ANSI/ISO 9899-1990", McGraw-Hill, 1990.

[11] http://www.swen.uwaterloo.ca/~yzou/AST/.

[12] K. Kontogiannis, P. Patil, "Evidence Driven Object Identification in Procedural Systems''. STEP'99, September 1999, pp. 12-21.

[13] Jörg Czeramski, et al, "Data Exchange in Bauhaus",. 7th WCRE'2000.

[14] R. Koschke, J.-F. Girard, and M. Würthner, "An intermediate representation for integrating reverse engineering analyses" , 7th WCRE'2000.

[15] Michael W. Godfrey, "Defining, Transforming, and Exchanging High-Level Schemas", 7th WCRE'2000.