Designing an XML-based Exchange Format for Harmonia [1]

Marat Boshernitsan and Susan L. Graham
Computer Science Division
University of California at Berkeley
Berkeley, CA 94720-1776 USA
{maratb,graham}@cs.berkeley.edu

Abstract

In this paper we present our design for a program data exchange format for Harmonia, a framework for constructing language-sensitive interactive CASE tools. We discuss the various design issues we faced while developing an encoding for the syntax tree information in the XML format, including chosing an appropriate encoding, the generation of data schemas based on programming language syntax, and representing program text along with its structure.

1. Introduction

One of the greatest challenges facing designers of new CASE tools is ensuring that their tools work "en suite" with other development tools such as editors, viewers, browsers, compilers, debuggers, etc. A similar problem is faced by CASE tool researchers who wish to benefit from the existing artifacts of tool building such as program parsers and analyzers. One solution to this problem is development of a common format for exchanging the data about programs. In this paper we present an XML-based interchange format being designed for the Harmonia tool-building framework developed at University of California at Berkeley.

The Harmonia framework is a collection of libraries and APIs for constructing interactive language-based programming tools. A distinguishing feature of the Harmonia framework is its reliance on incremental program analysis technology which maintains a fully analyzed representation of a program as it is being manipulated [6]. The Harmonia framework is multilingual, simultaneously supporting many languages [2]. The Harmonia framework also provides bindings for many popular programming languages including C++, C, Java, Emacs Lisp, and Tcl, allowing for rapid prototyping of programming tools of varying complexity.

The exchange format for Harmonia needs to cater both to the needs of small tools that operate on individual source files and to the needs of larger systems which manipulate entire software projects. Additionally, the exchange format should allow communication both among the tools constructed within the Harmonia framework, and with other existing tools that do not employ Harmonia technology.

This short paper considers the issues of data integration among these various tools and presents an exchange format we are designing to meet the requirements above.

2. Program Representation in Harmonia

Since one of the requirements for the Harmonia exchange format is to facilitate communication among tools built within the framework, the proposed format needs to model the program representation as it is implemented within Harmonia. The internal program representation in Harmonia is a syntax tree constructed (and incrementally maintained) from program source code by a syntactic analyzer (parser) provided by the framework.

As Harmonia is a multilingual framework, the parser is driven by a declarative specification of the language grammar and the syntax tree produced by the parser is, in fact, a concrete syntax tree whose shape is dictated by that same grammar specification. (This feature will come into light again when we discuss the exchange format schemas later in this paper). However, the Harmonia framework employs two special techniques for abstracting the grammar: a variant of the EBNF [9] notation for expressing sequences and optional productions, and the GLR parsing technology [5] which transparently incorporates ambiguity, including unbounded syntactic lookahead. Our approach lets us express language grammars at a fairly abstract level without

[1] This research was supported in part by NSF grant CCR-9988531, and by an NSF Fellowship to Marat Boshernitsan.

[2] It should be noted that the languages handled by Harmonia are primarily those that have a syntactic and semantic structure akin to programming languages. Since there is no generally-accepted word for artifacts written in those languages, we use the term "programs". The context should indicate when the discussion pertains only to programming languages.

the typical contortions associated with producing a grammar amenable to traditional parsing techniques. In fact, we use the terms syntax tree and abstract syntax tree (AST) interchangeably in this paper, reflecting their synonymity in the Harmonia framework.

Since Harmonia syntax trees serve as the sole representation for manipulation and presentation of programs, the parser retains all the keywords and punctuation within a tree. It is generally recognized that even when pretty-printing is used for presenting source code, the ability to maintain user-provided formatting is essential for good user interface [1], and so special provisions exist for maintaining user-provided whitespace and comments within Harmonia syntax trees [8].

Further program analysis such as static semantics, flow analysis, etc. may be provided by the Harmonia framework if implemented for the particular programming language. This analysis will typically annotate the AST, providing additional cross-references among syntax tree nodes (e.g. links from the use of an identifier to its declaration) and other information. Such annotated syntax trees are typically called Abstract Syntax Graphs (ASG). Figure 1 gives an example of a Java program fragment undergoing the syntax analysis to yield an AST, followed by the static semantic analysis producing an ASG.

3. Exchange Format Choices

From the beginning we were determined to use XML [10], an emerging standard for data interchange, as an encoding for our exchange format. This resolution led to a number of important decisions to be made about representing syntax trees in XML.

3.1. Trees vs. Graphs

One option for encoding ASGs in XML is to employ a general-purpose graph encoding similar to GraX [2] and GxL [3]. In such an encoding, the *MethodCall* fragment of the ASG in Figure 1 may be represented in XML as follows:

```
<node id=1 name=MethodCall>
  <edge target=2 type=child/>
  <edge target=4 type=child/>
  <edge target=5 type=child/>
  <edge target=6 type=child/>
</node>
<node id=2 name=Name>
  <edge target=3 type=child/>
  <edge target=42 type=declRef/>
</node>
<node id=3 name=IDENT text="f"/>
<node id=4 name=LPAREN text="("/>
```

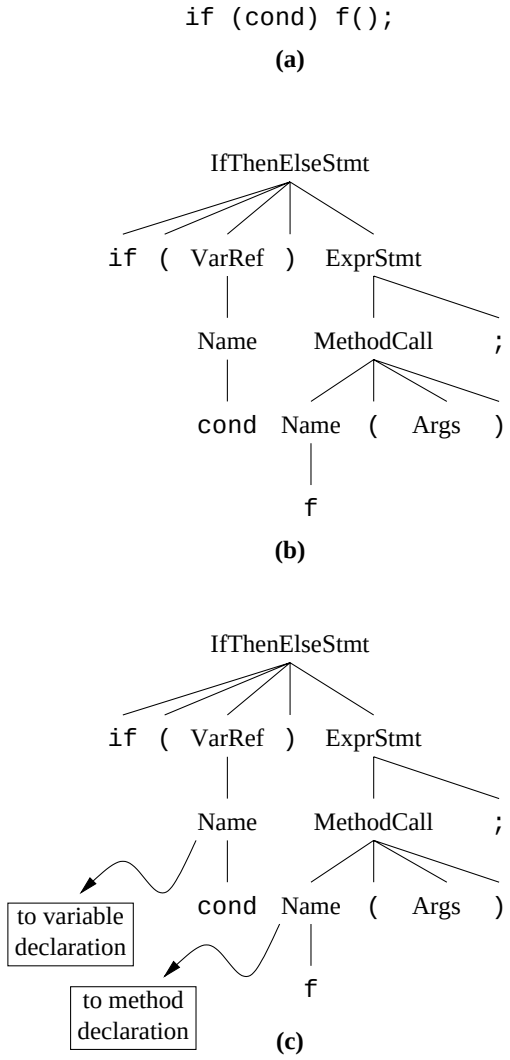

Figure 1. A Java program fragment represented as (a) text, (b) AST, and (c) ASG.

```
<node id=5 name=Args></node>
<node id=6 name=RPAREN text=")"/>
```

(Here, *declRef* is a reference to the method declaration node in the syntax tree whose *id* is 42.)

Alternatively, we may choose to encode the syntax tree directly, taking advantage of the fact that XML syntax is particularly well suited for representing hierarchical data. This approach is also attractive since in Harmonia the tree structure (produced by the parser) is fundamental, as it is the basis for all the manipulations. The attributes resulting from further program analysis (and which indeed induce the graph structure by providing additional links between syntax tree nodes) are secondary and may, in fact, not be present at all if semantic analysis services are not implemented for the par-

ticular programming language. The following illustrates the tree-based encoding:

```
<node id=1 name=MethodCall>
  <node id=2 name=Name declRef=42>
    <node id=3 name=IDENT text="f"/>
  </node>
  <node id=4 name=LPAREN text="("/>
  <node id=5 name=Args></node>
  <node id=6 name=RPAREN text=")"/>
</node>
```

3.2. Designing the Exchange Schema

The XML standard requires that a schema called a *document type definition* (DTD) be provided along with the XML data, if the data is to be validated in any way by the processing tool. For example, the encoding in the preceding example may be described by a very simple XML DTD:

```
<!ELEMENT node (node)*>
<!ATTLIST node ...all attributes...>
```

The disadvantage of this DTD is that it imposes virtually no restrictions on the shape of the syntax tree, making tools using this exchange format amenable to malformed input. Rather than requiring each tool to ensure the validity of the incoming data (which may not be trivial in a multilingual environment), we can employ a readily available validating XML parser by encoding the syntax tree within a more rigorous DTD. We observe that the nodes in the syntax tree are typed according to the grammar describing the shape of the syntax tree, which is precisely the same grammar we use to drive the syntactic analyzer (see Section 2). The following is the running example in such *typed* encoding:

```
<MethodCall id=1>
  <Name id=2 declRef=42>
    <IDENT id=3 text="f"/>
  </Name>
  <LPAREN id=4 text="("/>
  <Args id=5></Args>
  <RPAREN id=6 text=")"/>
</MethodCall>
```

The DTD for this XML encoding can be generated automatically directly from the same specification of the language grammar used by the Harmonia parser and from the list of attributes for each type of syntax tree node (also part of the declarative language specification in Harmonia). For example, given the following specification fragment for the Java method call

```
MethodCall: Name LPAREN Expr* RPAREN

Name      : IDENT

node Name { NodeRef attribute declRef }
```

the following DTD can be produced by a simple automatic transformation:

```
<!ELEMENT MethodCall (Name, LPAREN,
                      Expr*, RPAREN>
<!ATTLIST MethodCall id ID #REQUIRED>
<!ELEMENT Name (IDENT)>
<!ATTLIST Name
          id ID #REQUIRED
          declRef IDREF #REQUIRED>
```

This approach essentially ties our exchange format to the language grammar, requiring the exchange of schemas between the tools *as well as* the data. An alternative approach is to incorporate all supported languages into a single schema. The DATRIX schema [4] is an example of this design, supporting C, C++, and Java. That approach works well for these languages because they share common concepts. However, it is not practical for Harmonia, which supports many languages of vastly different styles.

3.3. Encoding Program Text

One of the applications of the Harmonia framework is construction of language-sensitive *front-end* tools such as editors, browsers, etc. that may need to manipulate program source code, rather than its structural representation embodied in the AST. For such tools, it is important to provide the means to retrieve not only program structure, but also various non-structural source elements such as comments, whitespace, identifier spelling, punctuation, etc. While this information can be made available as part of the encoded tree structure, an attractive alternative is to simply "markup" the source code with the structural information represented in the syntax tree:

```
<MethodCall id=1>
<Name id=2 declRef=42><IDENT
id=3>f<IDENT></Name><LPAREN
id=4>(</LPAREN><Args
id=5></Args><RPAREN id=6>)</RPAREN>
</MethodCall>
```

An advantage of such an encoding (as opposed to, for example, encoding a node's textual content as one of its attributes) is that a tool interested in program text only needs to strip off XML tags, whereas a tool that only cares about program structure may safely ignore the text.

4. Conclusions and Future Work

In this short paper we presented the exchange format currently being developed for the Harmonia framework. Our design embodies a number of decisions, the most significant of which is the generation of exchange format schemas directly from language grammars, utilizing the very same specification that was used to construct the language parser.

This design raises a number of important questions which we plan to investigate further as the development progresses:

- **Schema and grammar evolution.** Since the exchange format is based on the language grammar, we need to provide a way for tools that utilize different grammars for the same language (or, even more likely, different versions of the same grammar) to make sense of each other's data. We believe that this issue can be addressed by employing readily available XML transformation tools based on the XSLT transformation language [11].
- **Data granularity.** Our exchange format provides a considerable level of detail that may not be required by some tools. Many high-level modeling tools may only be interested in the "declaration level" or even "architecture level" facts about the program, and while such data may be computed and incorporated into the syntax tree by the Harmonia framework, transporting the entire AST (and indeed computing the entire AST) may prove to be unnecessarily expensive.
- **Incorporating revision information.** Among other services, the Harmonia framework provides a very fine-grained versioning mechanism capable of keeping track of structural and textual modifications to the source program [7]. Some version-aware applications may require access to this data, while other tools may wish to ignore it. To facilitate this task, we intent to augment the presented exchange format to incorporate revision information.

References

[1] M. de Jonge. A pretty-printer for every occasion. In *Proceedings of the Second International Symposium on Constructing Software Engineering Tools*, pages 68–77, Limerick, Ireland, June 2000.

[2] J. Ebert, B. Kullbach, and A. Winter. GraX—an interchange format for reengineering tools. In *Proceedings: Sixth Working Conference on Reverse Engineering*, pages 89–99. IEEE Computer Society Press, 1999.

[3] R. C. Holt, A. Winter, and A. Schürr. GXL: Towards a Standard Exchange Format. Fachberichte Informatik 1–2000, Universität Koblenz-Landau, Universität Koblenz-Landau, Institut für Informatik, Rheinau 1, D-56075 Koblenz, 2000.

[4] S. Lapierre, B. Laguë, and C. Leduc. Datrix source code model and its interchange format: Lessons learned and considerations for future work. In *Workshop on Standard Exchange Format*, Limerick, Ireland, June 2000.

[5] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[6] T. A. Wagner. *Practical Algorithms for Incremental Software Development Environment s*. PhD thesis, University of California, Berkeley, 1997. Available as technical report UCB/CSD–97–946.

[7] T. A. Wagner and S. L. Graham. Efficient self-versioning documents. In *CompCon '97*, pages 62–67, San Jose, CA, Feb. 1997. IEEE Computer Society Press.

[8] T. A. Wagner and S. L. Graham. Modeling explicit whitespace in an incremental SDE, 1997. Submitted to Software—Practice & Experience.

[9] N. Wirth. What can be do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):882, Nov. 1977.

[10] Extensible markup language (XML™). `http://www.w3.org/XML/`.

[11] XSL tranformations. `http://www.w3.org/TR/xslt`.