Experience with ANSI C Markup Language for a cross-referencer

Hayato Kawashima and Katsuhiko Gondow
Department of Information Science,
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, JAPAN
{hayato-k, gondow}@jaist.ac.jp

Abstract

The purpose of this paper is twofold: (1) to examine the properties of our ANSI C Markup Language (ACML) as a domain-specific language (DSL); and (2) to show that ACML is useful as a DSL by implementing an ANSI C cross-referencer using ACML.

We have introduced ACML as a DSL for developing CASE tools. ACML is defined as a set of XML tags and attributes, and describes ANSI C program's syntax trees, types, symbol tables, and so on. That is, ACML is the DSL which plays the role of intermediate representation among CASE tools. ACML-tagged documents are automatically generated from ANSI C programs, and then used as input of CASE tools.

ACML is self-descriptive and has CASE-tool specific information, which results in high productivity of CASE tools. To show this, we experimentally implemented an ANSI C cross-referencer based on ACML. In the implementation, we had a good result; it took only 0.5 man-month.

1. Introduction

CASE (Computer-Aided Software Engineering) tools can be quite useful to reduce the cost of software development, but the cost of developing CASE tools itself is very high. One reason is because internal data in CASE tools is usually not available to other tools, although such data can be shared among CASE tools. As a result, most CASE tools have their own individual parsers and analyzers, resulting in low maintainability. It is a key issue to find or develop some technique to facilitate data sharing, or exchanging among CASE tools in an elegant and cost-effective manner.

In order to cut the cost of developing CASE tools for ANSI C programming language [17] (e.g., program slicers, cross-referencers and test-case generators), we have introduced ACML (ANSI C Markup Language) [6]. ACML is a kind of domain-specific language (DSL) to describe a common data format specific to ANSI C in the domain of CASE tools. That is, ACML is defined as a set of XML [1] tags and attributes, which describe ANSI C program's syntax trees, types, symbol tables, and relationships among language constructs.

A DSL is a programming language dedicated to a particular domain or problem. For example, Lex and Yacc are used for lexers and parsers; VHDL for electronic hardware description. Furthermore, even markup languages can also be viewed as DSLs, because they can be good languages to describe some data's structures, relationships and semantics specific to some particular domain, even though they are likely to lack the language features like variables, control statements, and/or procedures. In fact, HTML and LaTeX are widely recognized as DSLs [18][19][23].

The first purpose of this paper is to examine the properties of ACML as a DSL. ACML is a markup language, and is considered as a DSL for the same reason described above. ACML-tagged documents are automatically generated from ANSI C programs, and then used as input of CASE tools. ACML provides to programmers a way to easily obtain information about ANSI C programs in a self-descriptive manner, which is useful to develop CASE tools. To show this, we already implemented Weiser's slicer as a preliminary experiment in [6]. But the slicer only processes limited ANSI C programs, and utilizes only the syntax structure in ACML. Thus further experiment is required.

The second purpose of this paper is to show that ACML is useful as a DSL by implementing an ANSI C cross-referencer using ACML. A cross-referencer is a CASE tool that lists all identifiers including variables, functions, labels and type names in a given program, and allows programmers to relate the use of an identifier to its definition, and vice versa. An elaborate cross-referencer considerably helps programmers to understand and maintain their

source programs. However, the existing implementations like Cxref [14] and GNU GLOBAL [12] are still not good enough, since their analysis is likely to be imprecise as a result of their focusing on execution speed. Furthermore, all information in ACML is required to implement our cross-referencer. The cross-referencer processes any ANSI C programs. Thus, in this experiment, the effectiveness of ACML is examined more comprehensively. In the experiment, we had a good result; it took only 0.5 man-month.

The rest of this paper is organized as follows. Section 2 gives a short overview of data integration for CASE tools and the existing standards: CDIF and PCTE. Section 3 discusses the properties of our ACML as a DSL. Section 4 describes our XCI (Experimental ANSI C Interpreter), which also works as a converter from ANSI C source code to ACML-tagged documents. In Section 5, our preliminary experiment of implementing an ANSI C cross-referencer is given. In Section 6, we have discussion through the implementation of our cross-referencer. Section 7 describes related works. Finally, Section 8 gives conclusion and future works.

2. Data integration in CASE tools

2.1. Problem

Generally the cost of developing CASE tools is very high. One reason is because internal data in CASE tools is usually not available to other tools, although such data can be shared among CASE tools. In other words, such data is closed, or even if it is open, it is not very useful. For example, we cannot easily use internal data of GCC [8], such as symbol data, syntax data, and type data, for other CASE tools (Figure 1).

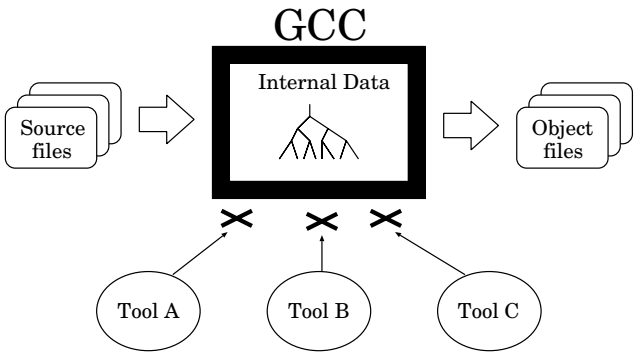

Figure 1. GCC Internal Data

As a result, most CASE tools have their own individual parsers and analyzers, resulting in low maintainability. From this point of view, CASE tools development really costs. Such data should be shared among CASE tools. Thus, it is a key issue to find or develop some technology to facilitate data sharing, or exchanging among CASE tools in an elegant and cost-effective manner. To solve the issue, the idea of using common data formats have already been introduced in CDIF[5] and PCTE[3]. But, unfortunately, these technologies have not come into wide use in CASE tools yet. Thus, the quest for the ideal format is still continuing.

2.2. Data integration

The purpose to integrate CASE tools is to make CASE tools more open, interoperable, and reusable. In [4], CASE tool integration is categorized into data integration, control integration, presentation integration, process integration and framework integration.

Data integration is our major interest, since our goal is to cut the cost of developing CASE tools by introducing common data formats. It is not easy to find a good solution for data integration. We can understand this fact by seeing that even simple data integration like integrating newline characters in text files (CRLF, CR, and LF) could be problematic. For example, JDK1.2 has several bug fixes related to CRLF. Things are even worse for CASE tools, because:

- There are various software products like specification diagrams, design documents, programs, memoranda, manuals, test data, and so on.
- For a product, there are many formats for the product. For example, there are many programming languages, which have their own syntax (i.e., formats).
- For a format of some product, there are many tools for it. For example, there are many CASE tools for ANSI C programs like compilers, debuggers, program slicers, and cross-referencers.
- There are complex relationships among products mentioned above.

Therefore, it is not quite trivial to determine the proper level of abstraction or granularity of the common representation for various kinds of products.

Another problem is pointed out in [22]. Data integration assumes that the connected CASE tools share information. This typically involves the development of a database or repository to store the information shared among the tools. This can be a disadvantage of data integration, since such a repository is likely to be a quite large and complicated database system. Also, such environments tend to be closed rather than open, both because any new tool is needed to integrate with the database system and because the database itself tends to be designed to operate with a particular language.

Our idea is that we can avoid these problems for data integration by supporting only one programming language (i.e., ANSI C) and using XML, which relieves us from developing a complicated repository. When it comes to using XML, it is appropriate to restrict a target language to only one, since it is too complicated and ambitious to represent some programming languages on only a DTD.

2.3. The existing data integration technologies

In this section, we briefly outline CDIF and PCTE from the existing data integration technologies. These technologies contributed to the progress of data integration, but they have not been widely used in CASE tool development yet.

- CDIF : CASE Data Interchange Format.

  CDIF is not a specification of CASE tools or repositories, but a standard format to be applied to data interchange among CASE tools.

  CDIF provides a clear separation between the semantics of data and its presentation. This distinction allows programmers to easily understand the underlying semantics, since all presentation information is removed from the semantics. CDIF is directed to the upstream part of software development. This is why we use XML, not CDIF to define our ACML. The expressive power of CDIF and XML is under research. For example, there is an experiment which converts CDIF into XML [15].

- PCTE : Portable Common Tool Environments.

  PCTE is a standard of open software repositories; PCTE is produced commercially, and standardized by ISO/IEC and ECMA (European Computer Manufacturers Association) in 1995. As a part of the environment supporting software development, PCTE offers a platform-independent interface for the set of data-handling functions. The important concepts are objects, links, attributes and schemata, which construct the entity relationship model. PCTE provides coarse-grained CASE data integration through file-level objects. We took a fine-grained approach to define ACML, so we did not use PCTE.

3. DSL, ACML and, ACML as a DSL

In this section, after a brief overview of DSLs and ACML is given, we discuss the properties of ACML as a DSL. Also, we discuss that XML can be a good tool to implement DSLs.

3.1. DSL: Domain Specific Language

A domain-specific language (DSL) is a programming language dedicated to a particular domain or problem. Usually, DSLs are smaller and easier than general-purpose languages (GPLs) in exchange for their limited domain. For example, Lex and Yacc are used for lexers and parsers; SQL for handling relational databases; VHDL for electronic hardware description [18].

Furthermore, even markup languages can also be viewed as DSLs, because they can be good languages to describe data structures, relationships and semantics specific to some particular domain, even though they are likely to lack the language constructs like variables, control statements, and/or procedures. In fact, HTML and LaTeX are widely recognized as DSLs [18][19][23]. Generally, software development with DSLs primarily aims at achieving faster and more productive development. Even if a programmer is not skilled, he or she can write more concise and higher level programs in less time, since the language constructs of the DSL are simple due to the focus on a specific domain [19].

Unfortunately, it is difficult to design and implement a good DSL. There are many issues to be considered on designing DSLs (e.g., listed in [20]). In this paper, we list two issues, which are discussed later in Section 6.1 and 6.2.

3.2. ACML: ANSI C Markup Language

ACML is an ANSI C Markup Language that we have developed. The entire DTD for ACML is 400 lines and found on the XCI Homepage [7]. An ACML-tagged document has information of the abstract syntax tree and the static semantics like types, symbols, control flows and relationships between declarations and references. ACML provides a concise way to decorate the information as following. See [6] for details.

- Syntax structures are represented as nesting of XML tags.
- Types and relationships are represented using ID/IDREF links.
- Symbols are represented as a sequence of `<symbol>` elements.

ACML is intended to be useful for CASE tools that statically analyze programs like static slicers, cross-referencers and static test-case generators. Actually, we found ACML is useful in an experimental implementation of Weiser's slicer [6].

We took a fine-grained approach for ACML; not only functions and statements, but also all language constructs including literals and variables are tagged with ACML. This approach is necessary to CASE tools for downstream part

of software development (e.g., slicer and cross-referencer), since they require syntactical information in detail, although ACML-tagged code is two orders of magnitude bulkier than the source program [6].

3.3. ACML as a DSL

In this section, we discuss the properties of ACML as a DSL.

The domain of ACML is ANSI C programs. ACML aims at reducing the cost of developing CASE tools primarily for ANSI C programs. ACML is a markup language to decorate ANSI C programs with the information of the syntax and static semantics. Thus, ACML is inherently declarative. Like most markup languages, ACML does not have the language constructs for variables, control statements, and procedures, which are required to describe some computation or behavior, not to describe some property or semantics of data. This is because ACML is a DSL to describe a common data format specific to ANSI C for CASE tools development.

ACML-tagged documents are automatically generated from the existing ANSI C programs by XCI [7], and then used as input of CASE tools. That is, ACML is the DSL which plays the role of intermediate representation among CASE tools. From this point of view, PostScript and RTL (Register Transfer Language) can be categorized into the same type of DSL as ACML. Here we call them 'DSLs to read', each of which usually require a DSL generator, such as XCI. On the other hand, 'DSL to write' is what programmers directly write. Thus, we consider intermediate representation a part of the DSL (i.e., DSL to read).

Figure 2 shows this difference: (1) in DSL to write, programmers write programs in the DSL, and then a DSL processor (typically a compiler) processes the programs to execute, (2) in DSL to read, DSL codes are generated or translated by the DSL processor (e.g., XCI for ACML), then programmers read the codes or other tools read the codes as input to further process.

Generally, the advantages of the DSL approach to software development is higher programmer productivity because programs written in a DSL tend to be more concise, quicker to write and maintain [23]. Also representations in a DSL can translate complicated descriptions of data structures into more concise and simpler ones, in order to read and understand.

Therefore, by attaching importance to reading, ACML helps programmers to understand programs better in less time, or to develop other tools in less costs. While developing a CASE tool, programmers of the CASE tool read, rather than write, ACML-tagged documents. ACML is self-descriptive and has CASE-tool specific information in detail, which results in high productivity of CASE tools.

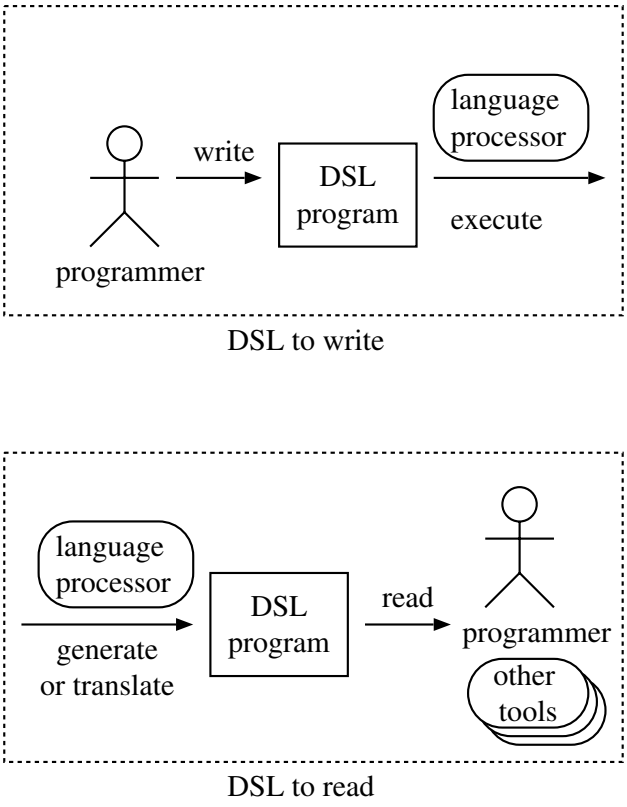

Figure 2. DSLs to write v.s. DSLs to read

In addition to CASE tool development, DSLs to read are attractive, for example, in the following situations.

- More easily understanding and utilizing binary formats like ELF format.

  Not a few people think that ELF format is so difficult that only hackers can understand ELF format, but that is not the case. Data in ELF format is essentially a simple sequence or collection of the information of machine code, symbols, relocation, line numbers, strings and so on. Therefore, if we develop a well-designed DSL to read ELF format (e.g., using XML), the DSL would greatly help us to understand and utilize ELF format.

  GNU BFD (Binary Format Descriptor) provides a common view for different executable formats like ELF, COFF, and PE, and allows us to deal with them more easily to some extent. However, format-specific sections like `.debug` in ELF are not supported in GNU BFD. Thus, GNU BFD is not good enough.

- Better program understanding.

  Comments, i.e., informal explanations in natural languages embedded in programs, are widely used in

practical programming to make the programs easier to understandable. But comments are often ambiguous, not correct, and not machine-processable. Thus, formal annotations are required for better program understanding. For example, if all identifiers are annotated with their type information in a formal manner, it would help programmers to maintain the programs. It is natural to define such annotations as a DSL to read, since annotations become formal, machine-processable, and possibly automatically generatable.

ACML can tag any ANSI C programs, though ANSI C preprocessing comments and directives like `#include` or `#define` are not marked up, that is, ACML ignores comments, since all programs are marked up with ACML *after* they are preprocessed. This issue is one of our future works.

4. XCI : Experimental C Interpreter

We have developed XCI (Experimental C Interpreter) [7], which converts any ANSI C programs into ACML-tagged documents. ACML code generation by XCI is essential for ACML, as mentioned in Section 3.3.

XCI also works as an ANSI C interpreter. Figure 3 shows that ANSI C programs are translated into ACML-tagged documents or interpreted. Currently, we do not use the function of interpreter in XCI, but it will be utilized when ACML incorporates dynamic semantics (e.g., for dynamic slicers). This is one of our future works.

4.1. The previous experiment: implementing Weiser's slicer

To show how useful ACML is for program slicing, we implemented Weiser's static program slicer without procedure calls [9] as a preliminary experiment [6]. Weiser's slicer is a simple but good example, since it uses an important technique common to all slicers, that is, tracing data and control flow dependences in a given program.

We experimentally implemented in Java Weiser's slicer. To obtain and process abstract syntax trees neatly, we used DOM [10] (Figure 4). Implementation was most straightforwardly done. It took only 0.5 man-month to implement Weiser's slicer resulting in 2000 lines Java code. By using ACML and XCI, we did not have to reimplement ANSI C parser and static analyzer.

Since the slicer only processes limited ANSI C programs, and utilizes only the syntax structure in ACML, further experimentation is required.

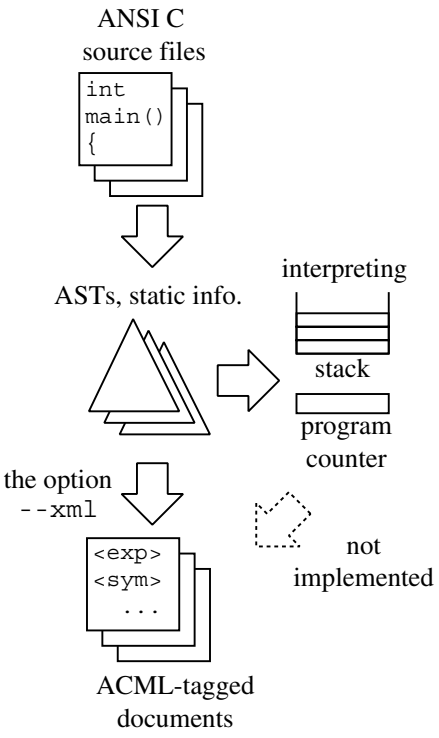

Figure 3. XCI overview

5. Cross-referencer based on ACML

We have introduced ACML as a DSL. In this section, to show that ACML is useful as a DSL, we experimentally implement an ANSI C cross-referencer using ACML.

5.1. What is a cross-referencer?

Generally, cross-referencers provide a means for relating the use of a name to its definition [22]. A cross-referencer we particularly aim at, is a CASE tool that lists all identifiers including variables, functions, labels and type names in a

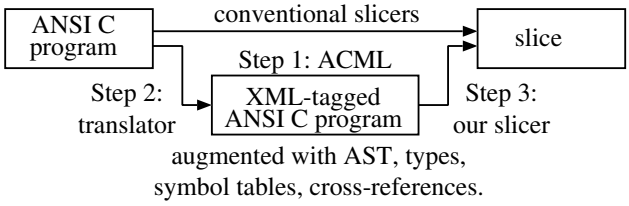

Figure 4. Weiser's program slicer based on ACML

given program, and allows programmers to relate the use of an identifier to its definition, and vice versa.

An elaborate cross-referencer considerably helps programmers to understand and maintain their source programs. However, the existing implementations like Cxref [14] and GNU GLOBAL [12] (Section 7) are still not good enough, since their analyses are likely to be imprecise or less informative as a result of their focusing on execution speed.

5.2. Why we implement a cross-referencer using ACML?

The purpose of this experiment is to show that ACML is useful as a DSL by implementing an ANSI C cross-referencer using ACML. A cross-referencer is an appropriate example to measure the effectiveness of ACML and XCI, because:

- There are several ANSI C cross-referencers widely used, e.g., Cxref and GNU GLOBAL. We can compare our cross-referencer with them for evaluation.
- It requires all information in ACML to implement a cross-referencer, unlike Weiser's slicer. This means the effectiveness of ACML is examined more comprehensively.
- Less restriction on ANSI C programs is required to implement a cross-referencer than to implement a program slicer. Actually, our cross-referencer accepts any ANSI C program, except that it cannot deal with references to other files (i.e., no linkage among files).

5.3. Function of our cross-referencer

Our cross-referencer translates a given ANSI C program via a ACML-tagged document to 13 HTML files, where identifiers are color-highlighted and cross-references are described as HTML hyperlinks. Figure 5 is a screen snapshot of the result that the cross-referencer processed `global.c` in GNU GLOBAL [13].

In the cross-referencer, all identifiers (functions, variables, tags, fields, and labels), typedef names, and enum constants are correctly related to their definitions, by extracting information about their types, name spaces and scopes. On the other hand, Cxref cannot deal with name spaces in some cases. For example, assume the following program, which is legal in ANSI C.

```
int main ()
{
  typedef int foo;
  foo: goto foo;
}
```

Our cross-referencer correctly distinguishes a typedef name `foo` and a label `foo`, but Cxref version 1.5d does not distinguish them and reports a parse error.

5.4. Results

We successfully implemented the cross-referencer as XSLT stylesheets in 2000 lines, which took only 0.5 man-month. We can implement the cross-referencer using DOM instead of XSLT, but we did not, because one of our purposes is to examine the expressive power of XSLT for ACML.

We found that XSLT is very useful to concisely describe the cross-referencer, but its execution speed is very slow. We measured the execution time of our cross-referencer, Cxref and GNU GLOBAL, which is listed in Table 1. Possible reasons are as follows, although the cause of slow execution is still under investigation.

- ACML-tagged documents are repeatedly scanned by 13 XSLT stylesheets, since one XSLT stylesheet can produce only one file. Moreover, there is no way to store intermediate results in XSLT, which can be shared among similar processing, but not shared.
- Naively described XSLT stylesheets are concise, but suffer from slow execution, since the same XML tags are repeatedly scanned by different XSLT templates. Moreover, XSLT has no set or list operations, which are convenient to store collected intermediate results.
- Java I/O is slow. And/or XT is slow since it is a reference implementation.

6. Discussion

6.1. XML as a good tool for DSLs

ACML is defined as a set of XML tags and attributes. In our experience, XML reduces the cost of developing DSL processors. In this section, we discuss the advantages of using XML for developing DSL processors.

Even if a well-designed DSL is made, implementing the DSL is often difficult and costly [19]. For example, it is difficult for "domain developers" to create a DSL compiler which translate from the DSL to a target machine or GPL without higher skills in compiler technologies. XML technologies make it easier and of less cost to implement the DSL. For example, if we utilize XML parsers or XSLT processors, we do not have to develop a parser or translator for the DSL from the scratch. Of course, XML is not a silver bullet. It is still difficult and costly to develop the DSL's

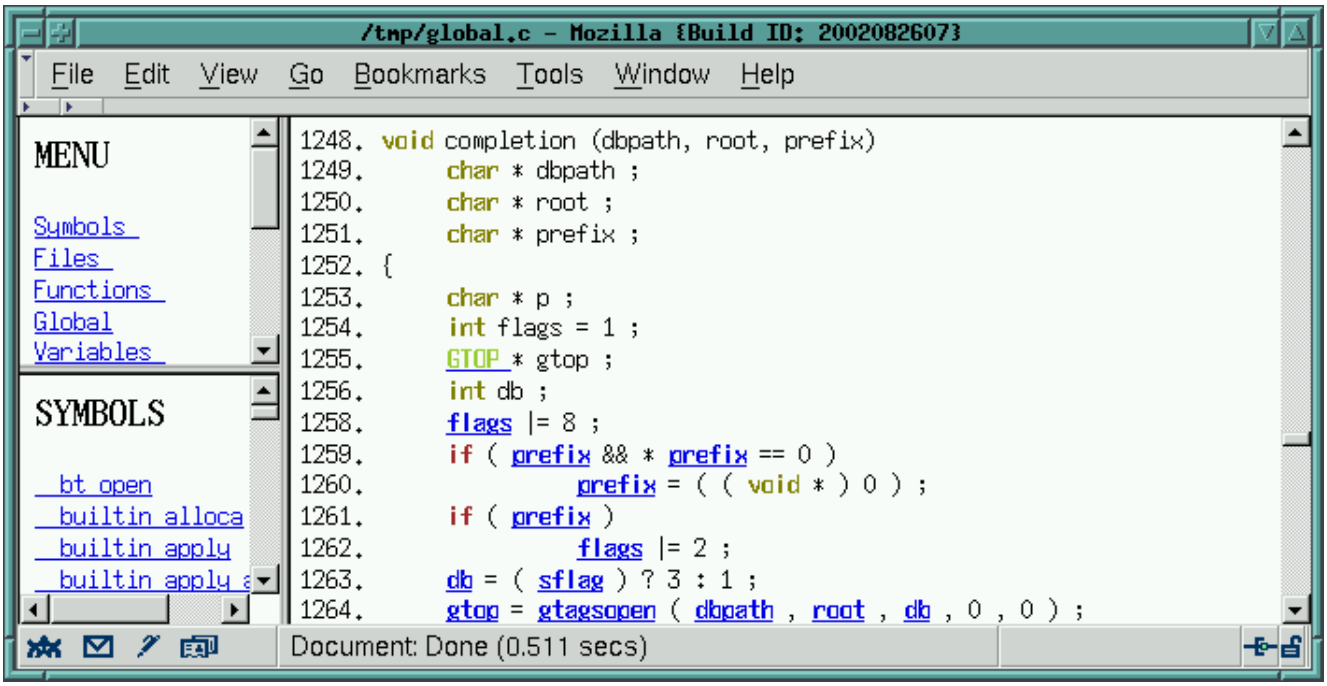

Figure 5. Screen snapshot of our cross-referencer

essential parts like semantic analysis and efficient optimization.

We have developed two tools based on XML: Weiser's slicer (Section 4.1) and cross-referencer (Section 5). Through the development of them, we strongly feel that XML technologies cut the cost of developing DSL processors. On the other hand, we implemented XCI in ANSI C without utilizing any XML technologies, which was time-consuming and error-prone work. To summarize, XML can be an effective tool for developing DSL processors, since XML has the following positive characteristics.

- XML's DTD provides notations that are rich and concise enough to define the DSL syntax. Furthermore, it promotes data integration for various DSL processors.
- Various XML tools like XML parsers, XSLT and DOM are available to flexibly process XML-based DSL programs. Their specifications are open and standardized.
- XML parsers can check if a given XML-based DSL program is, through XML validation check, syntactically correct or not.
- The syntax and semantics of XML-based DSLs are easy to understand, since XML is self-descriptive.
- XML has both advantages of plain text and structured data. This means that it is possible to use traditional text processing tools like `sed`, `grep` and `perl` for simple processing of XML-based DSL programs.

6.2. Designing DTD for ACML

As mentioned in Section 3.1, it is difficult to design DSLs well. That is the case for XML. XML is not a silver bullet for designing a good DSL. This section summarizes what is difficult in designing a DTD for ACML [6].

- Determining the proper level of abstraction or granularity of information in ANSI C programs.

  This highly depends on how to use ACML. We would like to use ACML primarily for program slicers and cross-referencers, most of which require the syntactic details. So, we took a fine-grained approach for ACML; not only functions and statements, but also all language constructs including literals and variables are tagged with ACML.

  On the other hand, JavaML (Java Markup Language) [2] took a coarse-grained approach to model Java independently of the Java specific syntax, which could integrate various object-oriented programming languages into a uniform format.

- Size/time trade-offs in handling derived data.

Table 1. Execution time[1](elapsed time in seconds)

	size	our cross-ref.[2]	Cxref[3]	GNU GLOBAL[4]
`hello1.c`[5]	77B	0.09+1.84 sec.	0.09 sec.	0.39+2.50 sec.
`hello2.c`[6]	68B	0.20+5.92 sec.	0.11 sec.	0.41+2.62 sec.
`global.c`[7]	28KB	0.67+28.91 sec.	0.20 sec.	0.52+2.73 sec.
`type.c`[8]	124KB	2.06+64.03 sec.	0.29 sec.	1.09+4.43 sec.

[1]measured on a 800MHz Mobile Pentium 3 with 256MB SDRAM, and running Windows 2000, [2]exec. time of XCI and our cross-referencer, [3]exec. time of '`cxref -xref-all -html`', [4]exec. time of '`gtags`' and '`htags`', [5]without `#include <stdio.h>`, [6]with `#include <stdio.h>`, [7]in GLOBAL, [8]in XCI,

A program slice is a typical derived data from the original program. It often costs much time to compute program slices, so the result of slicing should be stored as XML documents. However, derived data that are easily recomputed (e.g., the length of identifiers) would not be worth storing. XML documents are likely to be large because of many tags and attributes, so the file size also matters. Obviously, there is a trade-off between the file size and the computation time in handling derived data.

- Determining what kinds of information ACML should provide.

  The current ACML has information about the syntax structure and static semantics like types, symbols, and relationships among language constructs. The following are other candidates that current ACML does not provide.

  - Dynamic semantics. (e.g., history of variable values)
  - Human activities. (e.g., logs for debugging and testing)
  - Lexical information. (e.g., indentation, brace placement, comments)
  - Correspondence of source code to design diagrams (e.g., UML) or specification.

- Designing canonical notations for different codes that have the same meaning.

  Some code fragments in ANSI C have different ASTs, but are semantically equivalent. For example, `unsigned long` and `int long unsigned` are equivalent types. Is it better to introduce a unique (canonical) XML notation for them and how? In this case, we decided to introduce two notations:

  - Non-canonical notation to distinguish the difference of coding styles.
  - Canonical notation for checking type equivalence.

6.3. Possible improvements for our cross-referencer

In this implementation, to create hyperlinks, we do not recompute, but simply trace ID/IDREF links among elements in ACML. Thus, our cross-referencer still leaves room for improvements. Possible improvements are as follows.

- To link each definition to its use (i.e., reverse links) as GNU GLOBAL does.
- To provide a table of each definition and use as Sapid [16]'s SPIE does.
- To cope with C preprocessing directive like `#include` and `#define`. GNU GLOBAL, Cxref, and Sapid [16]'s SPIE cope with this problem in some extent.
- To link each definition and use of external objects across files.
- To produce outputs in various formats.
- To allow users to customize what kind of cross-references are produced and how.

7. Related works

- **JavaML** [2] and **GCC-XML** [21]
  JavaML and GCC-XML are both markup languages for programming languages. Unlike ACML, they are so coarse-grained that it is not suitable for developing cross-referencers.
- **Sapid** [16]
  For a given ANSI C source code, Sapid stores information of the syntactic structure and static semantics into

the file in the format called I-model. Using I-model, Sapid offers a great cross-referencer called SPIE. But SPIE is not based on XML. Sapid have been developed to a high degree of perfection, although it is too large scale.

- **GNU GLOBAL** [12]
  GNU GLOBAL is a cross-referencer for C, C++, Yacc and Java. Supporting multi-languages is achieved by not parsing, and by focusing on only file and function names. Thus, GNU GLOBAL cannot deal with name spaces correctly. GNU GLOBAL is light-weight, so GNU GLOBAL can process a large project containing many subdirectories. GNU GLOBAL is customizable with `gtags.conf`.

- **Cxref** [14]
  Cxref is a cross-referencer for C, and produces various kind of documents (in LaTeX, HTML, RTF or SGML) including cross-references. Cxref can handle not only ANSI C, but also K&R and most popular GNU extensions. Unlike GNU GLOBAL or Sapid's SPIE, hyperlinks are not embedded in source code.

8. Conclusion and future works

In this paper, we have considered our ACML as a DSL, and we have shown that ACML is useful in implementing an ANSI C cross-referencer using ACML.

ACML is a DSL for developing CASE tools. ACML is defined as a set of XML tags and attributes, and describes ANSI C program's syntax trees, types, symbol tables, and so on. That is, ACML is the DSL which plays the role of intermediate representation among CASE tools. From this point of view, PostScript and RTL can be categorized into the same type of DSL as ACML. We consider intermediate representation a part of the DSL (i.e., DSL to read). ACML-tagged documents are automatically generated from ANSI C programs, and then used as input of CASE tools. This process is supported by XCI and XML technologies. Thus, while developing a CASE tool, programmers of the CASE tool read, rather than write, ACML-tagged documents.

ACML is self-descriptive and has CASE-tool specific information, which results in high productivity of CASE tools. To show this, we experimentally implemented an ANSI C cross-referencer using ACML. In the implementation, we had a good result; it took only 0.5 man-month. Roughly speaking, we saved 2 months, since it took 2 months to develop an ANSI C parser and static analyzer for XCI and we did not have to reimplement the ANSI C parser and static semantics analyzer in implementing the program slicer and the cross-referencer.

We will extend ACML to support a lot of kinds of information. Especially, we have a plan to study the following:

- To elaborate our cross-referencer as mentioned in Section 6.3.
- To feed back analyzed results (e.g., a program slice) into ACML documents.
- To develop an XML-based C preprocessor to better cope with the problem of C preprocessor.

References

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. `http://www.w3.org/TR//REC-xml`.

[2] Greg J. Badros. *JavaML: A markup language for java source code.* `http://www.cs.washington.edu/homes/gjb/JavaML`.

[3] ECMA (European Computer Manufacturers Association). *Portable Common Tool Environment (PCTE) - Abstract Specification*, 1997. ftp://ftp.ecma.ch/ecma-st/Ecma-149.pdf.

[4] ECMA and NIST. *Reference Model for Frameworks of Software Engineering Environments, Draft Edition 3 of Technical Report ECMA*. TR/55 and NIST Special Publication 500-201, 1993.

[5] Electronic Industries Association CDIF Technical Committee. *CDIF CASE Data Interchange Format - Overview, EIA/IS-106,* 1994. http://www.eigroup.org/cdif/.

[6] K.Gondow, H.Kawashima. *Towards ANSI C Program Slicing using XML*, 2nd Int. Workshop on Language Descriptions, Tools and Applications (LDTA'02).

[7] K.Gondow, H.Kawashima. *XCI (Experimental ANSI C interpreter) Homepage*. Japan Advanced Institute of Science and Technology (JAIST). `http://www.jaist.ac.jp/~gondow/xci`.

[8] GNU Project. *GCC*. Free Software Foundation. `http://gcc.gnu.org`.

[9] M. Weiser. Program slicing. *IEEE Transaction of Software Engineering*, SE-10(4):352-357, 1984.

[10] WWW Consortium (W3C). *Document Object Model (DOM)*. `http://www.w3.org/DOM`.

[11] WWW Consortium (W3C). *XSL Transformations (XSLT) Version 1.0*. `http://www.w3.org/TR/xslt`.

[12] Tama Communications Corporation. *GNU GLOBAL source code tag system.* `http://www.gnu.org/software/global/`.

[13] Shigio Yamaguchi. *GNU GLOBAL-Source Code Tag System for C, C++, Java and Yacc*. `ftp://ftp.gnu.org/gnu/global/global-4.1.tar.gz`.

[14] Andrew M. Bishop. *The Cxref Homepage*. http://www.gedanken.demon.co.uk/cxref.

[15] Organization for the Advancement of Structured Information Standards (OASIS). *The XML Cover Pages*. http://xml.coverpages.org/xml.html.

[16] Fukuyasu Naoki, Yamamoto Shinichirou and Agusa Kiyoshi. *An evolution framework based on fine grained repository*. In Int. Workshop Principles of Software Evolution (IWPSE99) pages 43-47, 1999.

[17] B.W. Kernighan and D.M. Ritchie. *The C programming Language, 2nd Edition*. Prentice Hall 1988.

[18] Diomidis Spinellis. *Notable design patterns for domain specific languages*. Journal of Systems and Software, 56(1):91-99, February 2001.

[19] Scott Thibault, Renaud Marlet and Charles Consel. *A Domain Specific Language for Video Device Drivers: from Design to Implementation*. Conference on Domain-Specific Languages, 1997. USENIX Association.

[20] A. van Deursen and P. Klint. *Little languages: Little maintenance?*. Journal of Software Maintenance, 10:75-92, 1998.

[21] Brad King. *<GCC_XML description="XML output for GCC">*. http://www.gccxml.org/HTML/Index.html.

[22] Steven P. Reiss. *Software Tools and Environments* ACM Computing Surveys, Vol. 28, No. 1, March 1996.

[23] Proc. Conf. on Domain-Specific Languages. *USENIX ;login: - DSL'97 Conference Summaries*. http://www.usenix.org/publications/library/proceedings/dsl97/summaries/.