Electronic Notes in Theoretical Computer Science 65 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume65.html 20 pages

Towards ANSI C Program Slicing using XML

Katsuhiko Gondow [1] Hayato Kawashima [2]

Dept. of Information Science,
Japan Advanced Institute of Science and Technology (JAIST),
1-1 Asahidai Tatsunokuchi Nomi Ishikawa, 923-1292, JAPAN

Abstract

In this paper, we consider ANSI C program slicing using XML (Extensible Markup Language). Our goal is to build a flexible, useful and uniform data interchange format for CASE tools, which is a key issue to make it much easier to develop CASE tools such as program slicers. Although XML has a great potential for such data interchange formats, we first point out that there are still a lot of challenging problems to be solved. Then, as a first step to our goal, we introduce ACML (ANSI C Markup Language), which describes the syntactic structure and static semantics for ANSI C code. In our preliminary experiment, we had a good result; it took only 0.5 man-month to implement Weiser's slicer based on ACML, whereas it took about 2 man-months to implement an ANSI C parser and static semantics analyzer of XCI (Experimental C Interpreter).

1 Introduction

There are a great amount of research on program slicing [22], originally introduced by M. Weiser [24], which contributes to reduce the cost of software development and maintenance, especially of debugging, testing, and program understanding. Unfortunately, however, software slicing tools have not yet come into wide use in the field of software development, although several program slicing tools [23][14] are available. One reason is because the cost of developing software slicers is very high. CASE tools such as software slicers tend to have individual parsers and analyzers that can be shared among CASE tools. LALR(1) parser-generators like Bison [17] reduced the cost of development parsers, but the cost is still high, since, for example, it is required to cleanly solve the `typedef` name problem in the case of ANSI C,

The aim of this paper is to show that our approach of applying XML [3] (Extensible Markup Language) to software slicing is useful in the sense that a

[1] Email: `gondow@jaist.ac.jp`

[2] Email: `hayato-k@jaist.ac.jp`

well-designed DTD (Document Type Definition) for ANSI C and XML technologies including DOM [26] and XML parsers [9] can greatly cut the cost of developing program slicers.

Our goal is to build a flexible, useful and uniform data interchange format for CASE tools, which is a key issue to make it much easier to develop CASE tools such as program slicers. XML has a great potential for such data interchange formats, since XML has both advantages of plain text and a structured format. Moreover, XMI [16] (XML Metadata Interchange Format) allows us to handle UML diagrams as XML documents, so XML could bridge the great gap between the upstream and downstream parts of software development.

To achieve the goal, there are still a lot of challenging problems to be solved. For example, it is not quite trivial to select the best underlying grammar, to determine the degree of abstraction or the granularity of the representation, and to decide how to handle a variety of derived data. Furthermore, it can be quite difficult to define the proper representation of program semantics including system calls and inline assembly code, incomplete (buggy) programs, and the intention of programmers.

As a first step to our goal, we introduce ACML (ANSI C Markup Language), which describes the syntactic structure and static semantics for ANSI C code. In our preliminary experiment, it took only 0.5 man-month to implement Weiser's slicer based on ACML, whereas it took about 2 man-months to implement an ANSI C parser and static semantics analyzer of XCI (Experimental C interpreter). Thus, roughly speaking, we saved 2 man-months in implementation of Weiser's slicer, since we did not have to reimplement the ANSI C parser and static semantics analyzer by using ACML and XCI.

This paper is organized as follows. Section 2 describes problems for developing a DTD for software slicers, although XML has a great potential for data interchange formats of CASE tools. Section 3 introduces ACML (ANSI C Markup Language), which is intended to be convenient for developing static slicers for ANSI C. ACML includes information about the syntactic structure and static semantics like types, symbols, and relationships among language constructs. 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 is given. Section 6 describes related works. Finally, Section 7 gives conclusion and future works.

2 Why XML for Program Slicing?

2.1 Advantages of XML for CASE Interchange Formats

The cost of developing CASE tools such as program slicers is very high. One reason is because the 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 technology to facilitate data sharing, or exchanging among CASE tools in an elegant and cost-effective manner.

The idea of common formats for CASE tools is not new. For example, CDIF [6] (CASE Data Interchange Format) and PCTE [5] (Portable Common Tool Environments) have already been proposed. But, unfortunately, these technologies have not yet come into wide use in CASE tools. Thus, the quest for the ideal format is still continuing.

XML has been emerging as a standard format for a variety of types of data, especially for the Web documents. XML can be an effective tool for our goal, since XML has the following positive characteristics.

- XML has both advantages of plain text and structured data. XML documents are human-readable without any XML-specific application. Traditional text processing tools like `sed`, `grep` and `perl` can also be applied to XML documents.
- It is relatively easy to exchange XML-documents among different platforms, since text data is easily exchangeable without any problems of data conversion like byte-ordering.
- Program structure can be represented as natural nesting of XML tags. Relationships among programs can be represented as ID/IDREF links.
- Program structure and relationships are likely to be complex, but XML's self-descriptiveness makes it much easier to understand them.
- Various XML tools like XML parsers [9], XSLT [27] and DOM [26] are available to flexibly process XML documents.
- XML tags are extensible to better describe application-specific data.
- Even incomplete data can be stored as an XML document as long as the data is represented as plain text, although the XML document might not be valid.

2.2 What is challenging?

Although XML has a great potential ability as mentioned in Section 2.1, XML is not a "silver bullet" for developing CASE tools. DTD provides a *way* to define tags, but not concrete tags themselves. It is still challenging to define flexible, useful and uniform tags for describing a wide variety of software objects and complex relationships among them. Thus, it is crucially important to study XML by applying XML to CASE tools.

In this section, we enumerate challenging problems in developing a DTD for ANSI C, which is far from just translating an ANSI C grammar to DTD. Although ACML, introduced in Section 3, is trying to solve only the first 3 problems, we believe ACML is a good step to our goal where all the problems are solved. ACML is a DTD for ANSI C including information of abstract syntax trees, types, symbols, and so on.

- Selecting the best ANSI C grammar.

  The grammar used in GCC [18] is probably not appropriate, since it is complicated due to non-standard extensions like `asm`-syntax for inline assembly code, or symbol names like *notype-declarator*, which does not appear in the standard grammar [12]. Standard symbol names (i.e., nonterminal and terminal names) are important for understandability, but even the standard grammar [12] can not be appropriate, since, for example, there are 18 kinds of expressions like *primary-expression* and *additive-expression* only to introduce precedence and associativity for operators. The grammar can be simplified using EBNF available in DTD while preserving essential information, but what is essential depends on applications. Furthermore, Badros argued in [2] that it is not appropriate for his Java Markup Language (JavaML) to use the underlying grammar, which is used for parsing, as it is, since it is too verbose for his purpose. One of Badros's goals is to develop a markup language that can be commonly used for object-oriented programming languages. On the other hand, fine-grained information in the underlying grammar would be required, for example, for XML-based program transformation. Thus, selecting the best grammar is not quite trivial.

- Representation for the different codes that have the same meaning.

  Some code fragments in ANSI C have different ASTs, but are semantically equivalent. For example, the following 8 lists of type-specifiers are all legal, and denote the same type:

  `unsigned long int`, `unsigned int long`, `long unsigned int`, `long int unsigned`, `int unsigned long`, `int long unsigned`, `unsigned long`, `long unsigned`

  Is it better to introduce a unique (canonical) XML representation for them? The answer depends on the situation. For example, a unique representation is convenient for checking type equivalence, but not to distinguish '`unsigned long`' and '`int long unsigned`'. We almost always use the former, but never use the latter. The difference can be crucially important when coding styles need to be taken into account. This suggests we need to handle both abstract and concrete syntax tree at the same time.

- Size/time tradeoff in handling derived data.

  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 be little worth storing as XML documents. XML documents are likely to be large because of many tags and attributes, so the file size also matters. Now, how can we determine if a given derived data should be stored or not. Obviously, there is a tradeoff between the file size and the computation time when handling derived data.

- Various kinds of information to be stored.

There are many candidates for program information to be stored such as static semantics (e.g., types, symbols), human activities (e.g., what is done for testing and debugging), dynamic semantics (e.g., history of variable values and communication with other processes or OS), lexical information (e.g., indentation, brace placement, comments), and analyzed data for them. How well should we design a DTD for them? Especially, it is difficult to design compact, flexible, useful and uniform DTD for dynamic semantics, which is still an active topic of research (e.g., [20]), since raw traces tend to be too voluminous and too unstructured. Furthermore, the DTD should be available for different programming languages and different dynamic analysis tools.

- Representation for programmer's intentions.

  The following code fragment is a legal external declaration in ANSI C.

  ```
  extern int x = 999; /* legal declaration */
  ```

  An external declaration [3] with an initializer like '=999' is treated as a definition. Although this code is quite legal, other programmers or reviewers could feel that something is wrong with this code, since this code is unusual. Actually, ANSI C programmers sometimes do something unusual by intention, since the ability that they can do so is a big advantage of ANSI C. Now, how can we represent the programmer's intentions? For example, in Lint[11], lint-specific comment `/* NOTREACHED */` suppresses the warning that there is an unreached code. Thus, a simple intention can be represented by introducing some directives like `/* NOTREACHED */`. But how about complex intentions? For example, it is not easy to describe what kind of UNIX commands are expected to be set to the environment variable `PAGER` for the command `man`, since the intention of making the manuals look better is vague.

- Representation for constraints in programs.

  There are two kinds of constraints in programs:

  · Constraints on dynamic semantics.

  For example, a constraint on variable's value such as `x>0` is typical. They can be usually represented using the `assert` macro (e.g., `assert(x>0);`).

  · Other constraints including coding styles (e.g., naming conventions), coding rules (e.g., prohibition of implicit casting), and design policy (e.g., use of some design patterns), which are related to the above "programmer's intentions", and can be difficult to represent in a machine processable format.

- Representation for programs with errors.

  Even if a given program has errors, the program should be stored as an XML document, since it can provide valuable information for debugging or testing. Therefore, DTD should be designed to accept programs with

[3] A declaration outside a function is called an external declaration.

errors. Now the problem is how to represent, as XML documents, syntax errors, static errors (e.g., used but undeclared variables) and dynamic errors (e.g., memory leak). Syntax errors might be represented as XML documents by isolating and simply enclosing with XML-tag like `<syntax error>` by programmer's annotation or compiler's information of error recovery.

- Cooperation between the upstream and downstream parts of software development.

  XMI (XML Metadata Interchange Format) [16] includes DTD for UML. This means that the large gap between the upstream and downstream parts could be bridged with XML technologies. To make it a reality, we need to find a way to relate ACML to XMI neatly.

- C extensions and dialects.

  Typical C extensions and dialects (e.g., `asm`-syntax, dollar signs in identifiers) might be worth being stored as XML documents, since a lot of programs use them.

- Library functions and system calls.

  It is quite difficult to represent the semantics for C library functions and system calls. For example, a C standard library function `signal` determines how subsequent signals will be handled. Once `signal` is called with two arguments: a signal number and a function pointer, the function is called when the specified signal occurs. Obviously, `signal` can affect the control flow of programs. Therefore, `signal` should be taken into account when slicing program including a `signal` call. But how? A signal can occur at any place in the program. How can we construct the control flow graph? How can we represent such semantics of `signal` as XML documents? This discussion is almost valid for other system calls like `fork` or `execve`.

3 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 in XCI Homepage [7]. An ACML-tagged program has information of the abstract syntax tree and the static semantics like types, symbols, control flows and relationships between declarations and references. 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 experimental implementation of Weiser's slicer, which will be shown in Section 5.

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 is because most advanced slicers require the syntactic details. As a result of the fine-grained approach, ACML-tagged programs tend to be very large. As shown in Table 1, even for a 'HelloWorld' program with `#include <stdio.h>`, the size of the converted ACML-tagged code is

	Source code	ACML-tagged code
HelloWorld (without `#include <stdio.h>`)	85 bytes	19,571 bytes
HelloWorld (with `#include <stdio.h>`)	71 bytes	547,617 bytes
`xml.c` in XCI	54,375 bytes	3,581,613 bytes
`global.c` in GNU Global [28]	27,908 bytes	3,202,088 bytes

Table 1
Sizes of source codes and ACML-tagged codes

over 500K bytes, because the header file `stdio.h` introduces numerous declarations, types (typically, `struct` definitions and `typedef` names) and symbols.

In this section, first we show an overview of ACML features, then we explain how ACML tags describe ANSI C programs.

3.1 Overview of ACML features

- ACML can tag any ANSI C programs.

  We selected ANSI C from full-fledged programming languages to gain practical experience, so there is no sense if we only support a subset of ANSI C.

- ACML tags are named after nonterminals in the standard ANSI C grammar [12].

  The ANSI C grammar is very complex; the grammar [12] includes 183 productions and 65 nonterminals, which most programmers are not familiar with (e.g., *direct-abstract-declarator*). If ACML defined different tag names from the standard grammar, the situation would be worse, since it would be more difficult to understand what nonterminals stand for. This is a reason why the GCC grammar is difficult.

- ACML-tagged codes have information about the AST of the original code and its static semantics in a fine-grained manner.

  Although we mentioned in Section 2.2 that there are various kinds of information to be stored, we decided to limit the content of the current ACML to the AST and static semantics, since it is too ambitious to incorporate all information into ACML. Fine-grainedness is required since most program slicing techniques use some syntactic details.

- ACML-tagged code can be unparsed into the original ANSI C code.

  This is one of the criteria to show if ACML-tagged code have enough information or not. We have already implemented an unparser for ACML, although preprocessing directives (e.g., `#include`) and lexical information (e.g., comments) are not recovered.

3.2 Syntactic structures

Although nesting of ACML's elements for syntactic structures is almost the same as the structure of the standard grammar, ACML is simplified by:

- Using the EBNF notations like '`?`' (optional), '`*`' and '`+`" (repetition) in DTD,
- Uniting 18 different nonterminals for expressions (e.g., *primary-expression*) to the '`expression`' tag,
- Introducing an attribute '`rhs`' to distinguish different productions that have the same children,

while preserving information required for static CASE tools, especially, for static program slicing. ACML has 45 productions and 17 nonterminals, whereas the standard grammar has 183 productions and 65 nonterminals.

For example, the following function declaration,

```
int plus (int m, int n)
{
    return m + n;
}
```

is converted into the following ACML-tagged document[4] by XCI. (XCI will be described in Section 4.)

```
<function_definition>
  <int/>
  <declarator rhs="pointer_null">
    <declarator rhs="func_new">
      <declarator rhs="id">
        <identifier rhs="identifier">plus</></>
      <parameter_declaration rhs="dec">
        <int/>
        <declarator rhs="pointer_null">
          <declarator rhs="id">
            <identifier rhs="identifier">m</></>
        </></>
      <parameter_declaration rhs="dec">
        <int/>
        <declarator rhs="pointer_null">
          <declarator rhs="id">
            <identifier rhs="identifier">n</></>
        </></></></>
  <statement rhs="compound">
    <statement rhs="return">
      <expression rhs="add">
```

[4] For presentation, all attributes but '`rhs`' are pruned, and all end tags are abbreviated as `</>`. The full result is found in [7].

```
        <expression rhs="identifier">
          <identifier rhs="identifier">m</></>
        <expression rhs="identifier">
          <identifier rhs="identifier">n</></>
      </></></></>
```

3.3 Types

In Section 2.2, we discussed how to represent the same meaning, but different codes, where the question is if it is better to introduce a unique representation or not. We solved this problem by providing two ways to represent types in ACML. For example, the following two declarations,

```
unsigned long x;
int long unsigned y;
```

are converted in the AST part of ACML as follows,

```
<unsigned/><long/>       <!-- for x -->
<int/><long/><unsigned/> <!-- for y -->
```

and also converted into the same type representation.

```
<type>
  <t_prim>
    <t_int is_long="true" is_short="false"
           is_signed="false" is_unsigned="true"/>
  </></>
```

Thus, we have a unique representation for two type specifiers: '`unsigned long`' and '`int long unsigned`' while reserving information enough to distinguish them in the AST part of ACML. The reason why both '`is_long`' and '`is_short`' exist is because we would like to represent even illegal declarations such as '`long short z;`' in ACML.

In general, type information requires graph structures, which is represented using ID/IDREF attributes. For example, the following list structure, which is a typical recursive data type,

```
struct list {
    int          data;
    struct list *next;
};
```

is converted as follows (see also Fig. 1),

```
<type id="X2618d70">                <!-- ID -->
  <t_struct tag="list" num="2">
    <t_field name="data">
      <type><t_prim><t_int/></></></>
    <t_field name="next">
      <type><t_pointer>
        <type type_ref="X2618d70">  <!-- IDREF -->
```

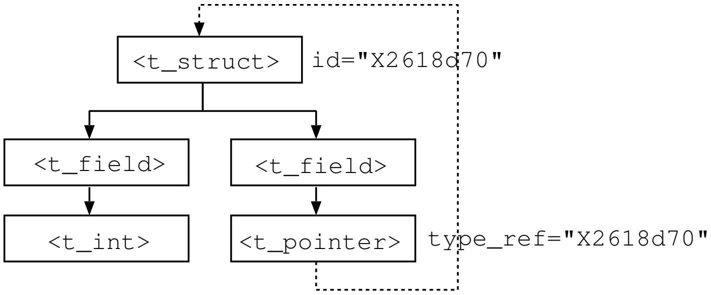

Fig. 1. Graph structure for `struct list`

```
           <t_empty/></></></></></></>
```

3.4 Symbols and Links

Information for symbols is simply placed as a sequence of `<symbol>` elements just behind `<translation_unit>` for external symbols, and just behind `<statement rhs="compound">` for local ones. For example, for an external declaration '`int x`', a symbol '`x`' is represented in ACML as follows[5]:

```
<symbol name="x"
    type_ref="X2618c70" ast_ref="X25f8190"
    namespace="normal"  namelevel="file" />
```

The attribute `type_ref` refers to the corresponding `<type>` element, and `ast_ref` to `<declarator>` in AST part. The ANSI C has 4 name spaces that do not interfere with one another; `namespace` indicates the name space that '`x`' belongs to. The attribute `namelevel` indicates if '`x`' is declared outside a function or not.

As mentioned above, ID/IDREF attributes establish several links for relationships among ACML elements. Besides these links, two kinds of links are represented using ID/IDREF.

- Links for control flows.

  For example, a label statement '`foo:  goto foo;`' is converted as follows.

  ```
  <statement rhs="label" id="X25f84b8">
    <identifier rhs="identifier"> foo </>
    <statement rhs="goto" goto_ref="X25f84b8">
      <identifier rhs="identifier"> foo
      </></></>
  ```

  where the destination of the `goto` statement (i.e., `goto_ref`) is the label statement with ID value of `X25f84b8`. Attributes `case_refs`, `default_ref`, `continue_ref`, and `break_ref` play similar roles.

- Links from reference to definition.

  For example, the following code fragment

[5] 10 attributes are omitted here for lack of space.

```
int x;    /* declaration */
return x; /* statement */
```

is converted as follows.

```
<declaration>
  <int/>
  <declarator id="X25f8100">    <!--  ID  -->
    <declarator id="X25f80b8">
      <identifier ref="X25f8100">
        x </></></></>
<statement rhs="return">
  <expression>
    <identifier ref="X25f8100"> <!-- IDREF -->
      x </></></>
```

where the ID value of X25f8100 connects the reference of ‘x’ to its definition.

3.5 Discussions

We have already found several issues to improve ACML.

First, as you can see in [7], the content model of ACML is non-deterministic. The requirement of deterministic content model is non-normative, so it depends on XML parsers if ACML is rejected or not. Actually, ACML works fine with XML4J 3.1.1[9], but not with JAXP1.1.3 [21]. We cannot do validity checking when using JAXP1.1.3. We would like to define DTD without being annoyed with such limitation. So it might be worth considering to use other schemata like RELAX [10] for the next version of ACML.

Second, we did not use XML Namespace [25] in ACML. We should use XML namespace for ACML to be more interoperable to other markup languages like XMI.

Finally, the current XCI generates ID values using pointer values in XCI’s AST. Although this guarantees that each element has a unique ID value, different versions for the same file do not share the same ID values, which makes XML-based versioning more complicated.

4 XCI: Experimental C Interpreter

We have developed XCI [7] (Experimental C Interpreter), which converts any ANSI C programs into ACML-tagged documents, when invoked with the ‘--xml’ option. Without the ‘--xml’ option, XCI interprets a given C code. Fig. 2 shows this function of XCI. XCI consists of about 14,000 lines of ANSI C code, and currently runs under Cygwin 1.3.6 on Windows2000 and Solaris 8. It took about three man-months to implement the first version of XCI. Here is a sample XCI session of executing C code `arg.c`, which simply prints

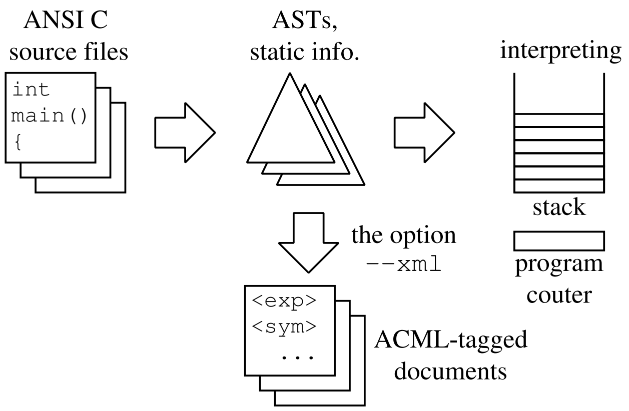

Fig. 2. Overview of XCI execution

the command line arguments.

```
% xci.exe arg.c --args 10 20
argv [0] = arg.c
argv [1] = 10
argv [2] = 20
%
```

At an early stage of development, XCI was designed to have the API to allow programmers to directly access the internal AST data structure in XCI. This is the similar way as used in Sapid [15]. But, in our view, it is likely that such API costs much for programmers to learn, and that the API depends on XCI's internal implementation. To solve these problems, ACML was introduced, and the XCI was changed to output ACML-tagged programs when invoked with the '`--xml`' option. We believe that the decision is right since XML played an important role for 'separation of concerns' in experimental implementation of Weiser's slicer using XCI and ACML (which will be mentioned in Section 5).

4.1 AST traversal

One of the distinctive features of XCI is interpretation by AST traversal. XCI does not use any intermediate code like Java Bytecode or RTL (Register Transfer Language) used in GCC. Instead of them, XCI traverses AST for interpretation. This feature is preferable for developing CASE tools, since the correspondence between the running status and AST is clear whereas intermediate codes tend to be too low-level to relate them.

The variable `vm.pc` is the program counter in XCI, which is of the type `struct PC`:

```
struct PC {
  struct AST *ast; /* node in AST under execution */
  int         nth; /* from where you came */
} pc;
```

The field ast points to the AST node under execution. The field nth keeps information about the last node before visiting the current node: 0 for the parent node and $i (i > 0)$ for the i-th child (child[i-1]).

The following macros are defined for AST traversal: GO_PARENT() for returning to the parent, and GO_CHILD(i) for visiting child[i].

```
#define GO_PARENT()  (vm.pc.nth = vm.pc.ast->nth + 1,     \
                      vm.pc.ast = vm.pc.ast->parent)
#define GO_CHILD(i)  (vm.pc.nth = 0,                      \
                      vm.pc.ast = vm.pc.ast->u.child[(i)])
```

The following code fragment is a part of the file eval_block.c in XCI, which interprets while statements.

```
switch (vm.pc.ast->ast_type) {
  ...
case AST_iteration_statement_while:
    switch (vm.pc.nth) {
    case 0: /* from the parent */
    case 2: /* from child[1] */
        eval_exp (vm.pc.ast->u.child [0]);
        if (vm.pop_i ()) GO_CHILD (1);
        else             GO_PARENT ();
        break;
    case 1: /* falling through */
    default: assert (0);   break;
    }
    break;
  ...
}
```

The control is passed to the internal statement (child[1]) in the while statement (see also Fig. 3), as long as the evaluated value of the expression (child[0]) is non-zero. If the internal statement has further internal statements, the control can be passed to them. When the result value of child[0] becomes zero, the control is passed to the parent node. That is just the semantics of the while statement.

5 Preliminary Experiment — Implementing Weiser's Slicer using ACML —

To show how useful ACML is for software slicing, we implemented Weiser's static program slicer without procedure calls [24] (Weiser's slicer for short) as a preliminary experiment. 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.

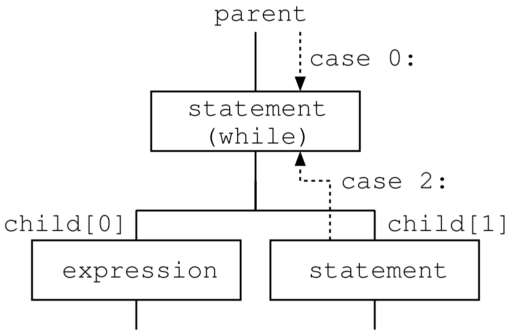

Fig. 3. AST traversal in `while` statement

In this section, we explain the experiment in some detail, and then we discuss some experiences obtained in the experiment.

5.1 Weiser's Program Slicer

A *program slice*, introduced by M. Weiser [24], is the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a *slicing criterion*. Slicing criterion is (n, V), a pair of line-number and variables of interest. Slice computes the same values for the variables in V as the original program does, whenever the statement n is executed.

Program slicing has been proven to be helpful for debugging, testing, software understanding, impact analysis and so forth. So, various slicing techniques have been proposed for programs with procedure calls, arbitrary control flow, composite datatypes, pointers, and interprocess communication (e.g., surveyed in [22]). In our experiment, we implemented a simple slicer based on Weiser's slicing algorithm for simplicity; the implemented slicer has the following limitations:

- No procedure calls (only `main` function)
- No composite datatypes, no pointers (only `int`)
- No control flow statements except `if-else` and `while`
- No system calls, no library functions except `fgetc`[6] and `printf`

Fig. 4 shows an example of slicing, where declarations are omitted to save the space. The original program (Fig. 4 (a)) counts the number of lines (`nl`), words (`nw`) and characters (`nc`) for a text from the standard input. Fig. 4 (b) shows a program slice on criterion (18, {`nl`}). Statements that do not affect the `nl`'s value are all eliminated in the slice. The slice computes the same value for the variable `nl` as the original program.

[6] In current implementation, ACML cannot handle ANSI C programs with preprocessing directives like `#include` or `#define`; all programs are marked up with ACML *after* they are preprocessed with the command `gcc -E`. So we cannot use C macro libraries like `getchar` here, since they are expanded away before marking up with ACML.

```
 1 iw = 0; /* in a word */
 2 nl = 0;
 3 nw = 0;
 4 nc = 0;
 5 c = fgetc(stdin);
 6 while (c != EOF) {
 7   nc = nc + 1;
 8   if (c == '\n')
 9     nl = nl + 1;
10   if (isspace(c)) {
11     iw = 0;
12   } else if (iw == 0) {
13     iw = 1;
14     nw = nw + 1;
15   }
16   c = fgetc(stdin);
17 }
18 printf("%d\n", nl);
19 printf("%d\n", nw);
20 printf("%d\n", nc);
```

(a) The original program

```

 2 nl = 0;


 5 c = fgetc(stdin);
 6 while (c != EOF) {

 8   if (c == '\n')
 9     nl = nl + 1;






16   c = fgetc(stdin);
17 }
18 printf("%d\n", nl);
```

(b) Slice on criterion (18, {nl})

Fig. 4. An example of slicing

In Weiser's slicer, directly or indirectly relevant statements to a given criterion are computed according to data flow and control flow dependences. Informally, data flow and control flow dependences are defined as follows. (Refer to [24] [22] for the formal definitions.)

- Data flow dependence

  Statement j is *data flow dependent* on statement i, denoted by $i \rightarrow_d j$ in this paper, if there is a variable v such that v is assigned at i and referenced at j without intervening assignments to v.

```
1  x = 10;
2  y = 20;
3  y = x;
4  a = b;
5  z = y;
```

  For example, in the above program, $3 \rightarrow_d 5$ (which means that statement 5 is data flow dependent on 3), $1 \rightarrow_d 3$, but $4 \not\rightarrow_d 5$ and $2 \not\rightarrow_d 5$. Statement 5 is *not* data flow dependent on 2, because assignment statement 3 to y intervenes between 2 and 5. In other words, the value 20 assigned to y at 2 does not reach 5.

- Control flow dependence

  Statement j is *control flow dependent* on i, denoted by $i \rightarrow_c j$ in this paper, if i can choose to execute j or not.

```
1  if (i > 0) {
2       x = 10;
3  }
```

  For example, in the above program, statement 2 is control flow dependent on 1. ($1 \rightarrow_c 2$.)

Using data flow ($\rightarrow_d$) and control flow ($\rightarrow_c$) dependences, program slice S on criterion (n, V) is defined as follows:

$$S = S_\infty \cup \{n\}$$

$$S_0 = \{i | (V \cap \mathrm{Def}(i) \neq \emptyset \land i \rightarrow_d n) \lor i \rightarrow_c n\}$$

$$S_{k+1} = \{i | \exists j \in S_k, i \rightarrow_d j \lor i \rightarrow_c j\} \cup S_k$$

where Def(i) means the set of variables that are assigned to or modified at statement i. Intuitively, a slice S collects all relevant statements to variables V at statement n, by inversely tracing the two relations $\rightarrow_d$ and $\rightarrow_c$. Computing S is terminated when a fixpoint is obtained, that is, when some k s.t. $S_{k+1} = S_k$ is found. Now assume the following code.

```
1  i = 0;
2  while (i > 0) {
3      x = x + 1;
4      i = i - 1;
5  }
```

There are the following 6 dependences:

$$2 \rightarrow_c 3,\ 2 \rightarrow_c 4,\ 1 \rightarrow_d 2,\ 1 \rightarrow_d 4,\ 4 \rightarrow_d 2,\ 4 \rightarrow_d 4$$

The slice on criterion (3, {x}) is {1, 2, 3, 4}, obtained by computing S_0, S_1 and S_2 as follows.

$$S_0 = \{2\}, S_1 = \{1, 2, 4\}, S_2 = \{1, 2, 4\}, S = S_2 \cup \{3\} = \{1, 2, 3, 4\}$$

5.2 Results and Discussions

We experimentally implemented in Java Weiser's slicer described in Section 5.1. To obtain and process abstract syntax trees neatly, we used JAXP [21] (Java API for XML Parsing) as well as our XCI and ACML. Implementation was mostly straightforwardly done. Several examples including Fig. 4 work fine. It took only 0.5 man-month to implement Weiser's slicer resulting in 2000 lines Java code, whereas it took 2.0 man-months to implement XCI's parser and static analyzer. By using XCI and ACML, we did not have to reimplement ANSI C parser and static analyzer, so, roughly speaking, XCI and

ACML saved 2.0 man-months. Actually we found that both of our system (XCI and ACML) and XML technologies like DOM greatly cut down the cost of developing the slicer.

We had valuable experiences through the experiment.

First, we again recognized that we need to elaborate ACML repeatedly through development of many applications like the implementation of Weiser's slicer. For example, we found ACML lacks an attribute `context` that tells how many sibling nodes there are before the node, which is very convenient for tree walking, especially for tree ascending.

Second, we found that we need to build class libraries for CASE tools that can be commonly used, which includes

- To extract all statements with a given identifier.
- To find the innermost statement including a given identifier.

XSLT and DOM are quite useful, but they cannot handle these operations directly.

6 Related Works

- GraX [4] — GraX is a graph-based interchange format for exchanging software reengineering related data. The concrete notation for GraX is defined using XML as `grax.dtd`. The purpose of GraX is to offer a general format for a wide variety of software objects. Thus, GraX does not offer a specific format for ANSI C like ACML, although GraX can be applied to ANSI C formats.
- SmartTools [1] and The Synthesizer Generator [8] — Both of them are software development environment generators. SmartTools is more related to our work, since SmartTools automatically produces a structure editing environment from a DTD, that is, SmartTools utilizes XML technologies. Unlike these works, we focused on ANSI C program slicing using XML technologies to gain practical experience like the `typedef` name problem.
- Portable XML-based Source Code Representation [29] — This work seems to focus on XML-based source code representation only for ASTs. Unlike this work, ACML has static semantics information, too.
- Wisconsin Program-Slicing Tool [23] and Unravel Program-Slicing Tool [14] — Both of them are tools written in C for slicing ANSI C programs. They do not utilize XML technologies.
- Adding Semantics to XML [19] — This paper proposed to apply attribute grammars to check semantic consistency among XML documents in a declarative manner, which can be useful, for example, to check used but undeclared variables in ACML.
- Sapid [15] — 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. Sapid does not use XML, although Sapid is trying to incorporate XML-based features.

- JavaML [2] — JavaML (Java Markup Language) is trying to model the Java programming language independently of the Java specific syntax. Such abstraction could integrate various object-oriented programming languages into a uniform format. Unlike JavaML, ACML tightly depends on the ANSI C grammar to maximize the power of ACML for slicing ANSI C programs. We have already discussed, in Section 2.2, several difficulties for designing markup languages for a programming language. This diversity between JavaML and ACML suggests another difficulty: the degree of syntax-dependence.
- GCC-XML [13] — GCC-XML generates an XML description of a C or C++ program from GCC's internal representation. Unfortunately, the output of GCC-XML is coarse-grained, so GCC-XML is not suitable for our purpose.

7 Summary

7.1 Conclusion

In this paper, we have considered program slicing for ANSI C based on XML. To investigate how useful XML is for program slicing, first we defined ACML (ANSI C Markup Language), which can describe the abstract syntax tree and static semantics such as types, symbols, and so on. Then we experimentally implemented Weiser's static slicer, and had a good result.

7.2 Future Works

As discussed in Section 2.2, there are a lot of kinds of information to be stored as XML documents. We will extend ACML to support such information. Especially, we have a plan to study the following:

- To extend ACML for dynamic program slicers.
- To extend ACML to support other similar languages like C++ or Java, which is motivated by the fact that GNU Global [28] supports C, C++, Java and Yacc.
- To develop other CASE tools using ACML such as cross-referencers, program browsers (navigators), test case generators, and so on.
- To extend ACML to describe preprocessor directives, lexical information (e.g., comments, coding styles), system calls, and inline assembly code.
- To relate ACML and XMI to bridge the upstream and downstream processes in software developments.
- To reduce the size of ACML-tagged code for efficiency.

Acknowledgements

The authors would like to thank Takuya Katayama, who deeply inspired us to start this work. The authors would also like to thank anonymous referees for valuable comments that have improved this paper.

References

[1] Isabelle Attali, Carine Courbis, Pascal Degenne, Alexandre Fau, Joel Fillon, Didier Parigot, Claude Pasquier, and Claudio Sacerdoti Coen. SmartTools: a development environment generator based on XML technologies. In *Proc. XML Technologies and Software Engineering (XSE2001)* (2001).

[2] Greg J. Badros. JavaML: A markup language for Java source code. *WWW9 / Computer Networks*, 33(1-6) (2000), 159–177. `http://www.cs.washington.edu/homes/gjb/JavaML/`.

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

[4] Jurgen Ebert, Bernt Kullbach, and Andreas Winter. GraX: An interchange format for reengineering tools. In *Proc. 6th Working Conf. on Reverse Engineering (WCRE'99)* (1999).

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

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

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

[8] GrammaTech, Inc., Ithaca, NY. *The Synthesizer Generator, A System for Constructing Language-Based Environments.* `http://www.grammatech.com/products/sg/index.html`.

[9] IBM. *XML Parser for Java (XML4J).* `http://www.alphaworks.ibm.com/tech/xml4j`.

[10] INSTAC XML SWG. *RELAX (Regular Language description for XML).* `http://www.xml.gr.jp/relax/`.

[11] S. Johnson. *Lint, a C Program Checker, Unix Programmer's Manual.* AT&T Bell Laboratories (1978).

[12] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language, 2nd Edition.* Prentice Hall (1988). ISBN 0-13-110362-8.

[13] Brad King. *<GCC_XML description="XML output for GCC">*. `http://public.kitware.com/GCC XML/`.

[14] Jim Lyle. *The Unravel Program Slicing Tool.* National Institute of Standards and Technology, Information Technology Laboratory. `http://www.itl.nist.gov/div897/sqg/unravel/unravel.html`.

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

[16] Object Management Group (OMG). *formal/00-11-02 (XML Metadata Interchange (XMI) version 1.1).* `http://www.omg.org/technology/documents/formal/xmi.htm`.

[17] GNU Project. *Bison.* Free Software Foundation. `http://www.gnu.org/`.

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

[19] Giuseppe Psaila and Stefano Crespi-Reghizzi. Adding semantics to XML. In *2nd Workshop on Attribute Grammars and their Applications (WAGA99)* (1999), 113–132.

[20] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. 23rd Int. Conf. on Software Engineering (ICSE2001)*, (2001), 221–230.

[21] Sun Microsystems. *Java API for XML Processing (JAXP).* `http://java.sun.com/xml/jaxp/index.html`.

[22] Flank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3) (1995), 121–189.

[23] University of Wisconsin. *The Wisconsin Program-Slicing Tools.* `http://www.cs.wisc.edu/wpis/slicing_tool/`.

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

[25] World Wide Web Consortium. *Namespaces in XML.* `http://www.w3.org/TR/REC-xml-names/`.

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

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

[28] 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`.

[29] Ying Zou and Kostas Kontogiannis. Towards a portable XML-based source code representation. In *Proc. XML Technologies and Software Engineering (XSE2001)* (2001), 343–353.