Binary-Level Lightweight Data Integration to Develop Program Understanding Tools for Embedded Software in C

Katsuhiko Gondow
Tokyo Institute of Technology
2-12-1 Oookayama Meguro
Tokyo 152-8552, JAPAN
gondow@cs.titech.ac.jp

Tomoya Suzuki
Elmic Systems, Inc.
4-59 Bentendori Naka
Yokohama 231-0007, JAPAN
t-suzuki@elmic.co.jp

Hayato Kawashima
Japan Advanced Institute
of Science and Technology
1-1 Asahidai Tatsunokuchi
Ishikawa 923-1292, JAPAN
hayato-k@jaist.ac.jp

Abstract

In embedded software development, the programming language C and inline assembly code are traditionally widely used. However, tools for C program-understanding, e.g., cross-referencers or call graph extractors, are not mature still today.

In this paper, we introduce a novel technique for developing program-understanding tools, based on binary-level lightweight data integration. To verify this idea, we first propose a new markup language for DWARF2 debugging information, and then, using the technique, we experimentally developed two cross-referencers (called `dxref` *and* `rxref`*) and a call graph extractor (called* `bscg`*) for C. Our preliminary evaluation shows that the technique enabled us to efficiently develop practical and flexible tools.*

1. Introduction

In embedded software development, the programming language C and inline assembly code are traditionally widely used, because C code can directly observe and/or control hardware devices through inline assembly code and pointer operations, and also because C code with inline assembly code is far more portable, productive and maintainable than the code entirely written in assembly code, while not sacrificing too much execution speed nor memory consumption.

The spirit of C [26] like "Trust the programmer" and "Do not prevent the programmer from doing what needs to be done" not only provides great freedom and flexibility to C programmers, but also provides a great risk of writing dangerous code like buffer overrun. Many tools, e.g. Electric Fence, libsafe [3] and SoftwarePot [18], have already been studied to protect the system from such dangerous programs, and many successes have been made in individual areas, but none of them are almighty for all possible vulnerabilities in C. Thus, CASE tools for C language still play an important role to maintain bugs in C programs.

One of the most important CASE tools is program understanding tools [28] designed to help programmers understand software during maintenance. Unfortunately, tools for C program-understanding, e.g., cross-referencers or call graph extractors, are not mature still today. For example, an empirical study [8] reported that call graphs extracted by several broadly distributed tools vary significantly enough to surprise many experienced software engineers. The situation is unchanged still today.

Generally speaking, the cost of developing CASE tools including program-understanding tools is very high, since CASE tools tend to have individual parsers and analyzers even if they can be shared among CASE tools, and also CASE tools tend not to export their analyzed internal data. For example, GCC internally has much information about syntax structure and static semantics of the program being compiled, but there is no simple way to extract it from GCC.

To allow CASE tools to flexibly cooperate and exchange their internal data, tool integration became a key issue. One of the five integrations stated in [32] is *data integration*, which refers to data sharing among tools and change management among them. We call this data integration *heavyweight*, since it requires special databases, e.g., PCTE [22]'s object management system, and fully uniform common formats, e.g., CDIF [7], that works well for the whole bunch of CASE tools including both upper and lower ones.

As XML [30] emerges as the standard format for Web documents, data integrations using XML, e.g., JavaML [9], GCC-XML [4], Sapid [14, 13], ACML [17, 12] are studied. We call them *lightweight*, since they defined common formats that work quite well among strongly related, but a small number of tools. The success of a metrics tool [11] using JavaML supports the lightweight approach. There are many advantages to use XML for data integration:

- XML documents are highly readable, portable and interoperable due to plain-text format and self-descriptiveness.
- XML is powerful enough to describe complex structures and their relations. Using DTD or XML schema,

we can define and check the structure of XML documents. Also, XML is flexible enough to cope with semi-structured documents (e.g., syntax errors).

- XML documents are easy to query, display and modify thanks to rich standard XML technologies, e.g., XML parsers, DOM, SAX and XPath.

Thus, data integration using XML greatly cuts the development cost; e.g., [17, 12] show a program slicer and a cross-referencer are developed only in 2 man-weeks each.

We say JavaML, GCC-XML, Sapid and ACML are *source-level* data integration, since they are all common formats for source code and the results of static analysis. Unfortunately, when it comes to C language, source-level data integration is very hard because of compiler-specific extensions and ambiguous behaviors in the C standards (see Section 2.1 for details).

To solve this problem of source-level data integration, we propose a *binary-level* data integration, which provides common formats by extracting information from binary code, not from source code. Binary-level data integration has many advantages. For example, compiler-specific extensions and the C standards' ambiguous behaviors do not matter in binary-level approach (see Section 3). As far as we know, little attention has been given to binary-level data integration for CASE tools, so it is worth studying.

The purpose of this paper is to show how binary-level lightweight data integration affects the development of CASE tools. As a testbed for this approach, we first define DWARF2-XML as a common format using XML for DWARF2 debugging information. Debugging information such as DWARF2 is a kind of binary information describing local variables, user-defined types, line numbers, lexical scopes, stack frames, and so on (see Section 2.3 for details). Of course, debugging information is primarily designed for debuggers, but we observe that it is quite attractive even for CASE tools.

Second, using DWARF2-XML, we experimentally develop two cross-referencers (called `dxref` and `rxref`) and a call graph extractor (called `bscg`) as typical program-understanding tools. In the experiment, we had a good result; it took only 1 through 2 man-weeks to implement each tool. Our tools are almost practical both in execution speed and in functionality; especially they have less false-positives than the existing tools. The tool `rxref` is a hybrid of `dxref` and GNU GLOBAL [29], and the success of `rxref` shows that our approach has good flexibility.

This paper is organized as follows. Section 2 reviews some background on C language, taxonomy of cross-referencers, DWARF2 debugging information. Section 3 discusses the (dis)advantages of binary-level data integration. Section 4 introduces DWARF2-XML. Section 5 describes experimental implementation of tools using DWARF2-XML. Section 6 describes related works. Finally, Section 7 gives conclusion and future works.

2. Background

2.1. Why source-level data integration is hard in C

As mentioned Section 1, source-level data integration is difficult because of compiler-specific extensions and ambiguous behaviors in the C standards.

First, source-level tools need to cope with compiler-specific extensions, but it is difficult. Most C programs, especially for embedded software, use compiler-specific extensions like GCC's `asm` for inline assembly or `__attribute__` for adjusting alignment, interpositioning, etc. Even system header files like `<stdio.h>` use such extensions, a simple 'Hello, world' program may have compiler-specific extentions. This implies tools only for ISO- and ANSI-C standard [23, 24, 25] conforming programs can be useless for embedded software.

Second, source-level tools need to precisely imitate the ambiguous behaviors defined by the C compiler being used, but it is quite difficult. Some parts of the C standards are intentionally left ambiguous, categorized into *implementation-defined*, *unspecified* and *undefined*. For example, whether a host byte order is big-endian or little-endian (or else) depends on the platform, which is an implementation-defined behavior. If possible, it is preferable not to use such ambiguous behaviors, since they make programs less portable, but sometimes we must use them. For example, when we implement `htonl`, which converts a `long` value from host byte order to network byte order, we probably can do nothing but write code depending on the host byte order.

Actually, most C programs depend on such ambiguous behaviors. Tools need to precisely imitate the ambiguous behaviors to obtain the analysis results consistent to the compiler behavior. But it is quite difficult since unspecified and undefined behaviors are not required to be documented, so we have to read the compiler source code to know such behaviors. Low-level information like byte ordering is crucial in developing embedded software, so tools for embedded software have to consider this problem.

2.2. Taxonomy of cross-referencers

Generally, cross-referencers provide a means for relating the use of a name to its definition, and vice versa. This section gives taxonomy of cross-referencers, including our `dxref` and `rxref`, to show that the implementation methods of tools greatly affect their characteristics like the quantities of false-positives and false-negatives, execution speed, scalability, applicability, and the development cost.

We classifies implementation methods of cross-referencers into the following five methods.

1. source-level partial parsing
2. source-level full parsing
3. binary-level symbol tables
4. binary-level debugging information (`dxref`)

5. hybrid of 1 and 4 (`rxref`)

The first three are the existing methods, and the last two are our new methods. Most existing cross-referencers, e.g., GNU GLOBAL [29], cxref [1], LXR [19], cscope [6], use the *source-level partial parsing method*, which extracts symbols and their relations from source code by string pattern matching or partial parsing. While this method can process huge source code very fast, it cannot understand scopes or name spaces. For example, GNU GLOBAL cannot identify two symbols with the same name in the different name spaces (e.g., variable `foo` and label `foo`). Also, the following code fragment is legal, but Cxref emits a parse error confusing `typedef` name `foo` and label `foo`. Thus this method is very fast, but less informative or incomplete.

```
typedef int foo; /* typedef name */
foo: goto foo;   /* label */
```

At first glance, the *source-level full parsing method* (e.g., Sapid and ACML), by which tools fully parse source programs at the same level as compilers, seems to solve the above problem. The answer is yes but very expensive to develop because of the problems described in Section 2.1, although it has few false-positives and few false-negatives. Actually, to avoid high development costs, Sapid and ACML ignore these problems, which results in less applicability. Section 5.2 shows two examples of source code that Sapid and ACML cannot process.

The *binary-level symbol table method* takes another approach; it extracts symbols and their relations from symbol tables and line number tables in executables. We think Visual Studio .Net uses this method, which is, however, not confirmed yet. This method is lightweight, low development cost, few false-positives, but less informative (i.e., many false-negatives and few true-positives). Symbols are likely to be restricted to global variables and functions. Actually, Visual Studio .Net cannot deals with local variables, tag names, `typedef` names, label names, and so forth.

This observation led us to propose *binary-level debugging information method*, by which binary symbol information is complemented by debugging information. As will be shown in Section 5.2, `dxref` cannot handle references in expressions and statements while it has several good characteristics. Thus, *hybrid method*, which uses both methods of binary-level debugging information and source-level partial parsing, is introduced in Section 5.2.

2.3. DWARF debugging information

DWARF Debugging Information Format Specification, Version 2.0 [31] (DWARF2 for short) is a binary format for debugging information, and widely supported by major compilers (e.g., GCC) and debuggers (e.g., GDB). DWARF2's primary target languages are C, C++, FORTRAN, Modula2 and Pascal. Debugging information in DWARF2 format includes, for example, source language types (including user-defined ones), nested blocks,

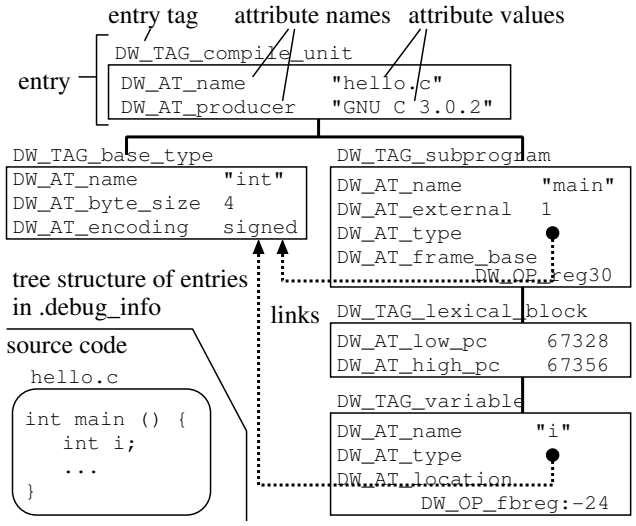

Figure 1. Example of tree structures in `.debug_info`

line numbers, function and object names (including local ones), their addresses on the machine, their accessibility (like `public` and `protected`), how to unwind a stack frame, and so on. All these information is encoded in highly compressed form, and stored in several sections whose names are of the form `.debug_*`.

DWARF2 defines the following 9 debug sections:

`.debug_info`, `.debug_abbrev`, `.debug_frame`, `.debug_line`, `.debug_str` `.debug_pubnames`, `.debug_aranges`, `.debug_loc`, `.debug_macinfo`,

The section `.debug_info` is the most important. It logically describes tree-structured entries. Each entry consists of an *entry tag* and a series of *attributes*. DWARF2 defines 47 entry tag names and 59 attribute names.

Figure 1 shows an example of entry tree structures in `.debug_info`, where an entry for compilation unit `hello.c` has two subentries for a type `int` and a function `main`, and the entry for `main` has a subentry for a block, which has a subentry for a local variable `i`. Each entry has 0 or more attributes. For example, the entry with the tag `DW_TAG_compile_unit` in Figure 1 has two attributes `DW_AT_name` and `DW_AT_producer`, whose values are `"hello.c"` and `"GNU C 3.0.2"`, respectively. Some attributes like `DW_AT_type` are links to other entries, which contributes to size reduction.

To embed DWARF2 debugging information in the executable file (`a.out`), you can compile source code using `gcc` with the option `-g3 -gdwarf-2`.

```
% gcc -g3 -gdwarf-2 hello.c
```

Note that the compiler produces DWARF2 debugging information and embed it into `a.out`, so the problems mentioned in Section 2.1 hardly arise here. For example, the value of DWARF2 attribute

`DW_AT_data_member_location` represents the offset of a structure member. The offset is an implementation-defined behavior, but, with very high probability, the offset in DWARF2 is the same value as that in machine code, since both values are generated by the same compiler.

One characteristic of DWARF2 is that DWARF2 entries are loosely structured in the sense that:

- Vendor-specific tags and attributes are allowed in DWARF2. The values in range between `DW_TAG_lo_user` and `DW_TAG_hi_user` inclusive are reserved for vendor specific tags. This implies we cannot simply map DWARF2's tags and attributes to XML's tags and attributes, respectively, when defining DTD for DWARF2.
- No constraint on which entries have which subentries, or which entries have which attributes.

  DWARF2 entries form a tree structure, which reflects the syntax structure of source languages. For example, an entry `DW_TAG_lexical_scope` has a subentry `DW_TAG_variable` in Figure 1. Unlike high-level programming languages, DWARF2 has no rule on which entries have which subentries. In other words, a DWARF2 entry can have 0 or more subentries with any tags. This also holds for attributes.

There already exist some utilities to display the contents of the DWARF2 sections in `a.out`. For example, `readelf` in GNU binutils is very useful for us to understand concrete DWARF2 debugging information, but, unfortunately, not for computers to process it, since:

- `readelf`'s output is not *tree-structured*. For example, entries in `.debug_info` logically form a tree-structure, but they are represented as a linear sequence of entries in `readelf`'s output, although `readelf`'s output has enough information to reconstruct the original tree structure.
- `readelf`'s output is *half-cooked*. For example, mapping between source line numbers and addresses are stored in `.debug_line` as a stream of bytes, which is a program in a byte-coded language to produce the mapping. This kind of encoding is required to reduce the size of DWARF2 debugging information.

  `readelf` emits an execution sequence of the byte-coded program as the mapping information. To obtain the mapping like (line 69, address 0x11930) from `readelf` outputs, we need to interpret opcodes like `Copy` (`DW_LNS_COPY`), which means "produce a pair of (*line*, *address*) as line information using the current values of the state-machine registers".

This observation led us to design fully-structured and fully-cooked DWARF2-XML even if it makes the file sizes of DWARF2-XML documents larger. See also Section 4.

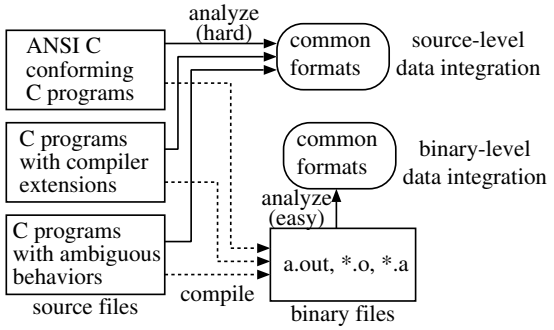

Figure 2. source- v.s. binary-level data integration

3. Binary-level data integration

In Section 2.1, we described the cost to implement practical source-level data integration for C is very high because of compiler-specific extensions and ambiguous behaviors in the C standards. Our idea to solve this problem is binary-level data integration. Compilers like GCC emit binary code that conforms to ABI[1] standards. The emitted binary code includes not only machine code, but also various information (e.g., symbol information, line information, relocation information, debugging information).

In this paper, *binary-level data integration* refers to providing common formats to such various kind of information in binary files and to derived data from it. Figure 2 shows that binary-level data integration avoids the difficulty of compiler extensions and ambiguous behaviors by analyzing binary files, not source files.

The advantages of binary-level data integration are:

- High applicability.

  Binary-level data integration can process programs in binary with compiler extensions (e.g., inline-assembly), with ambiguous behaviors, without source code, and compiled from other languages, as long as they conform to ABIs.
- More true-positives for low-level information.

  Binary-level data integration provides platform-dependent information such as function/variable addresses, sizes and offsets of structure members, stack layouts. This kind of information is often important in embedded software development, but not available in source-level data integration.
- Low development cost.

  The cost of developing source-level data integration is very high, since we need to develop (possibly full) parser and analyzer from the scratch, or modify the existing (possibly huge) compilers like GCC. On the other hand, the cost of developing binary-level data integration is low, since we only have to decode binary

1 application binary interface

code to obtain symbol tables, relocation information, debugging information and so on.

- Few false-positives.
  Binary information is based on compiler's analysis, so false-positives are much less than in source-level partial-parsing method. (Of course, binary-level data integration suffers from more false-negatives in exchange for few false positives.)
- Information to cope with C preprocessor problem.
  As shown in [10], it is very difficult for software tools to cope with the C macro preprocessor, Cpp. DWARF2, for example, provides the following information that can help to solve the Cpp problem:
  - Information on `#ifdef` and `#define` macro definitions in `.debug_macinfo`. (But no information on where the macros are used.)
  - Precise mappings each of which maps a DWARF2 entry to the line number of unprocessed C source code.

On the other hand, binary-level data integration has the following disadvantages:

- More false-negatives.
  For example, DWARF2 lacks the information about expressions and statements in source code. This implies DWARF2 itself has no ability to output information about expressions or statements.
- Platform, ABI or binary format dependency.
  For example, binary-level data integrated tools for ELF/DWARF2 on SPARC Solaris are not compatible with those for PE on i386 Windows. It is often the case that embedded software tightly depends on some platform, so this problem does not matter in our opinion.
- Difficulty for automatic source code modification.
  Generally, compiling is irreversible, so, for tools only using binary-level data integration, it is difficult to automatically modify source code.

Thus, binary-level data integration seems to be quite attractive, even though it has some drawbacks. As far as we know, little attention has been given to binary-level data integration for CASE tools, so it is worth studying. This paper gives a first step to examine the power of binary-level data integration.

4. DWARF2-XML: a markup language for DWARF2

In Section 2.3, we pointed out that DWARF2 entries are loosely structured, and the outputs of `readelf` are half-cooked and not tree-structured. Considering these issues, we developed DWARF2-XML a markup language for DWARF2, as a testbed of binary-level lightweight data integration. The entire DTD for DWARF2-XML is about 80 lines and found in [16]. DWARF2-XML currently supports the following 7 sections,

`.debug_abbrev`, `.debug_info`, `.debug_line`, `.debug_aranges`, `.debug_pubnames`, `.debug_frame`, `.debug_macinfo`

since C does not use `.debug_loc` and GCC does not emit `.debug_str` (strings are stored in `.debug_info`).

The design policies of DWARF2-XML are as follows.

- DWARF2-XML preserves as many logical structures and links in DWARF2 as possible. Also DWARF2-XML fully decompresses DWARF2 information, interpreting all opcodes like `DW_LNS_COPY`. This solves the deficiency of `readelf`'s outputs.
- DWARF2-XML maps a DWARF2 tag to an XML attribute value, not to an XML tag. For example, an entry of the form `DW_TAG_`*foo* is mapped to an XML tag `<tag name="DW_TAG_`*foo*`">` to allow us to deal with vender-specific tag names without modifying DTD. This also makes it natural to cope with the problem of no constraints on which entries have which subentries. DTD for DWARF2-XML simply defines a tag includes any attributes and/or any tags as follows.

```
<!ELEMENT   tag   (attribute*, tag*)>
```

Figure 3 shows[2] an example of `.debug_info` in DWARF2-XML document for `{ int i; ...}`, which means "This lexical block is compiled into the address range between 67328 and 67356[3]. A local variable `i` of the type `int` is defined in the block, and allocated at the offset `-24` to the frame base register (`%fp` in SPARC). `int` is signed and its size is 4 bytes."

DWARF2 can represent all data types including user-defined types, type modifiers like `const` and `volatile` and storage classes like `extern` and `static`. The data types are shared in DWARF2-XML using XML's ID/IDREF links to reduce the file size. For example, the value of `DW_AT_type` is a link to `DW_TAG_base_type` with the ID value `"id:161"`.

Like other XML-based markup languages, DWARF2-XML documents are bulkier than the original DWARF2 data. Table 1 shows the file or section sizes (in bytes) of DWARF2 (`.debug_*`), DWARF2-XML, etc., where `hello1.c` includes the `stdio.h` header file, but `hello2.c` does not; `x_debug` is a debugger using DWARF2-XML (details are not described in this paper); `readelf+` is introduced in Section 5. From Table 1, we can see DWARF2-XML documents become 15 times larger than the original DWARF2. In our opinion, the file increase is acceptable in exchange for high readability, flexibility, and low development cost.

Interestingly, the size increase is almost canceled by compressing DWARF2-XML with `gzip`, which suggests that general compression utilities like `gzip` may supersede DWARF2-specific compression techniques.

---

2 For lack of space, some attributes are pruned, and all end tags are abbreviated as `</>`.

3 Decimals are used here since XPath does not support hexadecimals.

```
<section name=".debug_info">
    ...
    <tag name="DW_TAG_lexical_block" offset="id:27">          <!--block-->
        <attribute name="DW_AT_low_pc"  value="67328"/>
        <attribute name="DW_AT_high_pc" value="67356"/>
        ...
        <tag name="DW_TAG_variable" offset="id:27">           <!--variable-->
            <attribute name="DW_AT_name" value="i"/>
            <attribute name="DW_AT_type" value_ref="id:161">  <!--IDREF-->
            <attribute name="DW_AT_location">
                <description>DW_OP_fbreg: -24</description></></></></></>
    ...
    <tag name="DW_TAG_base_type" offset="id:161">             <!--type, ID-->
        <attribute name="DW_AT_name" value="int"/>
        <attribute name="DW_AT_byte_size" value="4"/>
        <attribute name="DW_AT_encoding" value="5">
            <description>signed</description></></></>
```

Figure 3. Example of `.debug_info` in DWARF2-XML document for `{ int i; ... }`

	source code	a.out	DWARF2 (.debug_*)	DWARF2-XML	gzipped DWARF2-XML
hello1.c	86	8,788	2,075	11,703	2,445
hello2.c	70	11,156	4,440	34,841	4.954
x_debug.c	28,923	111,324	83,674	1,333,456	86,518
readelf+.c	319,863	549,496	288,687	3,823,263	243,802

Table 1. File (or section) sizes in bytes of DWARF2, DWARF2-XML and others

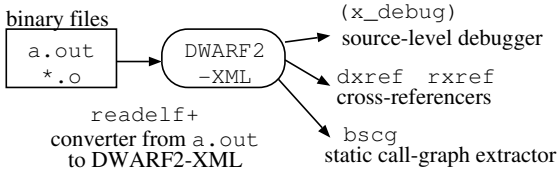

Figure 4. Overview of experimental implementation using DWARF2-XML

5. Experimental implementation of tools using DWARF2-XML

5.1. Overview

As mentioned in Section 1, we developed DWARF2-XML as a testbed for the binary-level lightweight data integration technique. To show how useful DWARF2-XML is, we experimentally implemented two cross-referencers (called `dxref` and `rxref`), a static call graph extractor (called `bscg`), and a source-level debugger (called `x_debug`) using DWARF2-XML. All of them are developed on Solaris 8, and the source code of them and sample outputs are available in [16].

As shown in Figure 4, we developed five tools.

- `readelf+` — translates from `a.out` or `*.o` to DWARF2-XML. `readelf+` is developed by modifying GNU `readelf` (about 12,000 lines in C). The patch is about 2,500 lines. It took about one man-week to develop. Implementing `readelf+` was not technically difficult, but tedious since we had to thoroughly know DWARF2 and `readelf.c`.
- `x_debug` — is a source-level debugger using DWARF2-XML as debugging information. We do not explain `x_debug` any more for the lack of space.
- `dxref` — generates HTML files including cross-reference links, extracted from DWARF2-XML. 2,200 lines in C. One man-week to develop.
- `rxref` — is a hybrid cross-referencer of `dxref` and GNU GLOBAL, developed to compensate for `dxref`'s weakness. 1,200 lines in Ruby. Two man-weeks to develop.
- `bscg` — is a static call graph extractor using DWARF2-XML and a result of disassembling. 1,400 lines in C. One man-week to develop.

In the experiment, we had a good result; it took only 1 through 2 man-weeks to implement each tool. Our tools are almost practical both in execution speed and in functionality; especially they have less false-positives than the existing tools. The tool `rxref` is a hybrid of `dxref` and GNU GLOBAL [29], and the success of `rxref` shows that our approach has good flexibility. So far as the experiment is concerned, binary-level lightweight integration us-

```
% gcc -g3 -gdwarf-2 test.c
% readelf+ -X a.out > test.xml
% dxref test.xml
% w3m HTML/dxref_top.html
```

Figure 5. Example of `dxref` execution steps

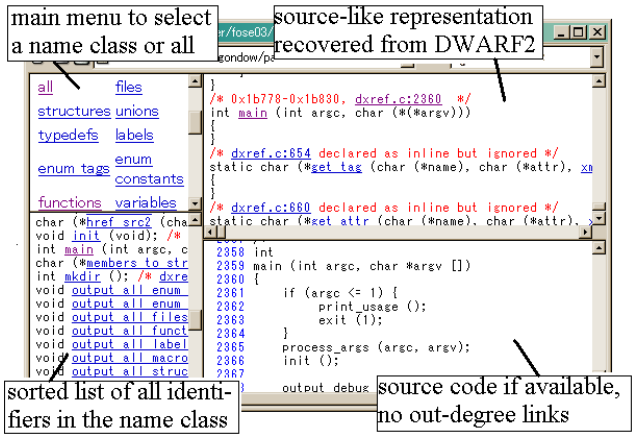

Figure 6. Screen snapshot of `dxref` output

ing DWARF2 is an effective technique to cut the cost of developing program understanding tools.

5.2. Cross-referencers: `dxref`, `rxref`

`dxref` uses only DWARF2-XML on purpose to know the ability or limitations of DWARF2-XML, while `rxref` uses both DWARF2-XML and the outputs of GNU GLOBAL.

Figure 5 shows `dxref`'s execution steps. `readelf+` extracts DWARF2 into DWARF2-XML. Using DWARF2-XML, `dxref` constructs cross reference information, for example, from the data of "a local variable `i` is defined in

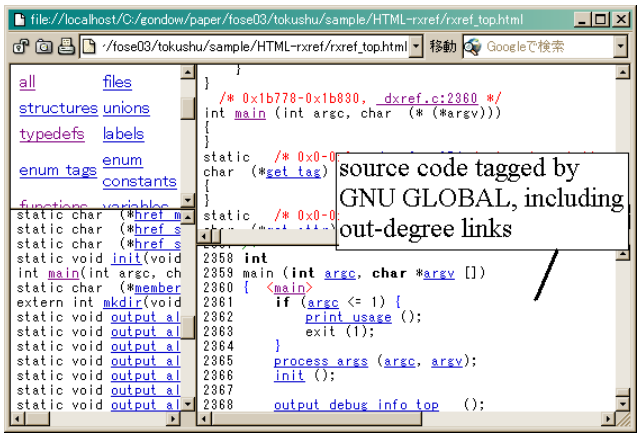

Figure 7. Screen snapshot of `rxref` output

methods (tool names)	applicability	false positive	false negative	scalability	development cost	interoperability
source partial-parsing	G	B	NB	VG	G	B
source full-parsing	B	G	VG	NB	B	B
binary symbol table	VG	VG	B	G	G	B
debug info. (`dxref`)	VG	VG	NB	B	G	G
hybrid (`rxref`)	VG	G	G	B	G	G

Legend: VG=very good, G=good, NB=not bad, B=bad

Table 2. Characteristics of `dxref`, `rxref` and the existing cross-referencers' methods

the block of the address range between 67328 and 67356" in Figure 3. Figure 6 is a screen snapshot that a Web browser Opera displays HTML documents generated by `dxref`.

`dxref` has several good characteristics. For example, `dxref` successfully processes C programs written using GCC extensions like `asm` (high applicability); `dxref` provides the information of function/variable addresses, stack layouts and so on (more true-positives for low-level information); `dxref` was developed in one man-week (low development cost); the problem of C's ambiguous behaviors does not arise because of the use of DWARF2 emitted by the compiler being used (few false-negatives); `dxref` provides the information[4] of C preprocessor's directives like `#include`, `#define` and `#undef` (coping with Cpp problem). Furthermore, `dxref` provides accurate identifier lists categorized by name class, i.e., files, structures, unions, typedefs, labels, enum tags, enum constants, functions, variables and macros. Thus, the advantages of binary-level data integration given in Section 3 apply to `dxref`. A qualitative comparative table of cross-referencers' methods from our experience is summarized in Table 2.

Unfortunately, the disadvantages of binary-level also apply to `dxref`. Especially, DWARF2 lacks the information of expressions and statements in C, which results in more false-negatives. That is, `dxref` provides no links to navigate, e.g., from a function use to its definition. To solve this problem, we developed yet another cross-referencer `rxref`. `rxref` is a hybrid of `dxref` and GNU GLOBAL, combining the HTML outputs of both tools. Figure 7 shows a screen snapshot of `rxref`. The lower right frame in Figure 7 is source code where cross-reference links are embedded by GNU GLOBAL. Thus, the drawback of `dxref` is considerably solved in `rxref`.

Table 3 shows the execution speeds [5] of `dxref`, `rxref`

4 `dxref` gives what macros are defined where, but not where they are used, due to the limitation of DWARF2 `.debug_macinfo`.

5 Commands and options used: `readelf+ -X` and `dxref --src2html` for `dxref`, `readelf+ -X` and `ruby rxref.rb -g` for `rxref`, `xci --xml` and `java cref` for ACML, `sdb4` and `spie -l -t html` for Sapid SPIE, `cxref -xref-all -html` for Cxref, `gtags` and `htags` for GNU GLOBAL, `genxref` for LXR, `cscope -b -q` for cscope.

	dxref	rxref	ACML cross. ref.	Sapid SPIE	cxref	GNU GLOBAL	LXR	cscope
hello1.c	0.07+1.7 sec	0.07+4.7 sec	0.12+6.5 sec	4.4+1.3 sec	0.37 sec	0.42+2.0 sec	0.49 sec	1.8 sec
hello2.c	0.08+2.3 sec	0.08+6.0 sec	0.20+11 sec	5.9+1.4 sec	0.41 sec	0.74+2.3 sec	0.48 sec	1.9 sec
x_debug.c	0.4+12 sec	0.4+39 sec	N/A	N/A	1.8 sec	0.91+3.8 sec	1.3 sec	2.7 sec
readelf+.c	0.7+34 sec	0.7+120 sec	N/A	N/A	2.6 sec	1.6+9.1 sec	6.9 sec	2.3 sec

Table 3. Execution speeds of `dxref`, `rxref` and the existing cross referencers (elapsed time)

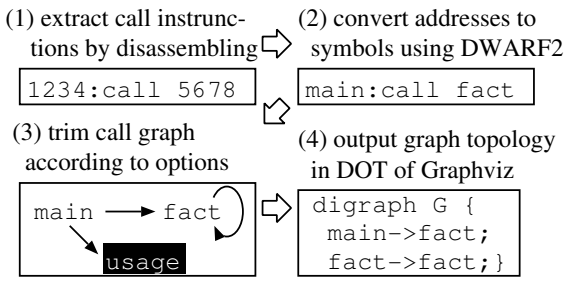

Figure 8. How `bscg` extracts call graphs

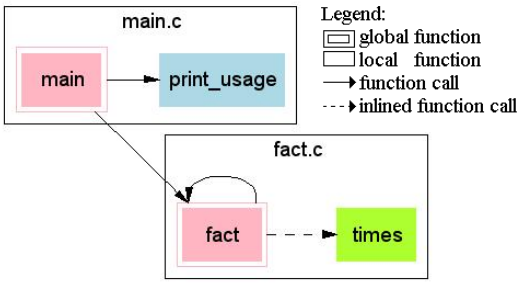

Figure 9. Example output of `bscg`(1)

and some existing cross referencers on Solaris 8 (333MHz UltraSPARC-IIi with 128MB RAM). While `dxref` and `rxref` are slower than cxref, GNU GLOBAL, LXR and cscope, our tools `dxref` and `rxref` run fast, in our opinion, enough for practical use, which is achieved by optimizing XPath usages in `dxref`, and by using SAX, not DOM, in `rxref`. Table 3 also shows the drawback of source-level data integration; Sapid SPIE [13] and ACML [12] failed to process `x_debug.c` and `readelf+.c`[6]. This supports high applicability of binary-level data integration.

5.3. Call graph extractor: `bscg`

Call graph extractors are tools to provide graph-based visual representation for function calls. `bscg` we developed is a static call-graph extractor using DWARF2-XML. Figure 8 shows how `bscg` extracts call graphs from binary files.

Figure 9 shows an example output of `bscg`, which indicates the function `main` calls `print_usage` and `fact`,

6 Sapid has some ability to skip GCC extentions by editing `Sapid.conf`, but we just used the default settings.

options	descriptions
`--nolocal`	exclude local (i.e. file scope) functions
`--only` *files*	include functions in the specified files
`--ignore` *funcs*	exclude the specified functions
`--callers` *funcs*	include callers of *funcs* recursively
`--callees` *funcs*	include callees of *funcs* recursively
`--depth` *num*	max. recursion depth (`all` for infinite)

Table 4. `bscg`'s graph trimming options

and `fact` calls `fact` itself and inlined function `times`; `main` and `fact` are global while `print_usage` and `times` are local (i.e., declared with `static`).

Call graphs are likely to be very huge and "spaghettied" in practical use, making the call graphs unreadable and useless. To solve this issue, `bscg` has the graph trimming options in Table 4. These options allow us to focus on some specific parts of the graph. For example, Figure 10 shows a `bscg`'s output for bash-2.03 with the option "`--depth all --callers reset_mail_timer`", which means "generate a call graph including reset_mail_timer and its (transitive) callers only". The option reduced the number of functions displayed in the output from 900 to 13.

Like `dxref` and `rxref`, the advantages of binary-level data integration (Section 3) also apply to `bscg`. Rather than discussing these issues again, we discuss more detailed characteristics of `bscg`. A qualitative comparative table of cross-referencers' methods from our experience is summarized in Table 5. For example, `bscg` has the following positive characteristics:

- `bscg` can identify inlined functions since DWARF2 has the inlining information, while other binary-level tools cannot due to the deletion of call instructions.
- `bscg` can extract a function call from inline assembly code like `asm ("call fact");`, while other source-level tools cannot.
- `bscg` can exclude library functions (e.g., `printf`), system calls (e.g., `open`) and functions in runtime system (e.g., `_start`, `_fini`), while other tools cannot.

Unfortunately, `bscg` has several drawbacks: no support for macro calls, signals, function pointers and optimization.

Table 6 shows the execution speeds[8] of `bscg` and some

8 Commands and options used: `readelf+ -X` and `bscg` for `bscg`; `genfull -g cobjdump` for CodeViz(cobjdump); the diff. of patched `gcc` and normal `gcc`, and `genfull` for CodeViz(cdepn); `cflow` for Cflow; the diff. of `gcc -pg` and `gcc`, and `gprof-callgraph.pl` for `gprof-callgraph.pl`.

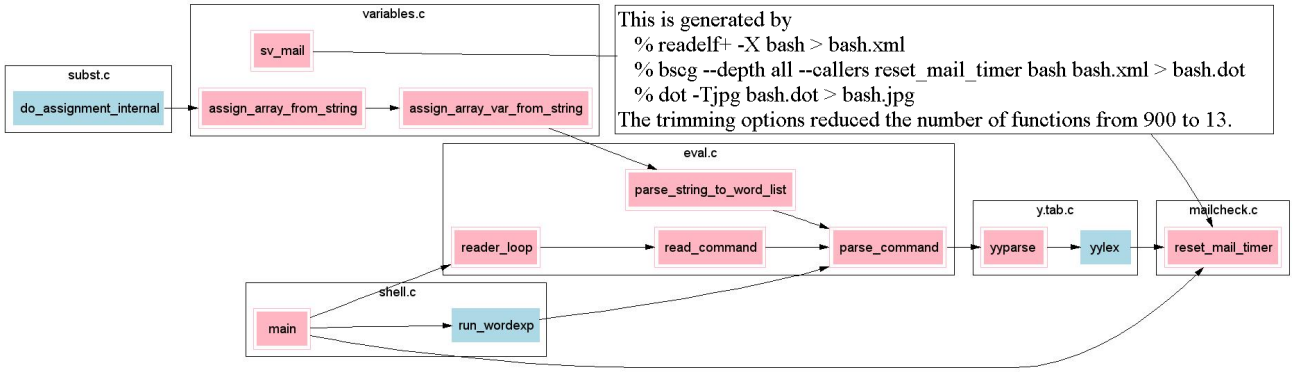

Figure 10. Example output of `bscg(2)`

tool names	methods	macro functions	signals	`#ifdef`	inline functions	library functions	selecting subgraphs	`asm`	function pointers	false positive	optimization	compilation	execution
Cflow	source-level/partial parsing	B	B	G	NB	NB	B	B	B	B	G	NR	NR
CodeViz(cdepn)	extending compilers	B	B	G	B	NB	B	B	NB	G	G	R	NR
CodeViz(cobjdump)	binary symbol table	B	B	G	B	B	B	G	NB	G	B	R	NR
`gprof-callgraph.pl`	profiler	B	B	G	B	B	B	G	G	G	B	R	R
`bscg`	DWARF2 debug info.	B	B	G	G	G	G	G	NB	G	B	R	NR

Legend: G=good, NB=not bad, B=bad, R=required, NR=not required

Table 5. Characteristics of `bscg` and the existing call graph extractors

call graph extractors on Solaris 8 (333MHz UltraSPARC-IIi with 128MB RAM). `bscg` runs fast enough for practical use except the case of bash-2.03. `bscg` has a problem in scalability, since `bscg` constructs a DOM tree to process DWARF2-XML, which consumes quite much memory. Actually, on machine with enough memory (Sun Blade 150, UltraSPARC-IIe 550 MHz, 640MB RAM), `bscg` processed bash-2.03 only in 25 seconds. Clearly, in use of DOM, there is a tradeoff between much memory consumption and low development cost.

6. Related works

- As far as we know, all the existing cross-referencers and call graph extractors do not utilize debugging information, while this paper does. [8] told a call graph extractor xrefdb in the Field [27] accepts an executable (compiled/linked with the debug switch), but xrefdb uses only the executable symbol table to determine the source files to be scanned. (I.e., xrefdb does not use debugging information).
- There are several studies on XML-based source program representation e.g., JavaML [9], GCC-XML [4], Sapid [14, 13], ACML [17, 12]. They are all *source-level* in the sense that they extract data from source code, while our approach extracts it from binary code.
- Clearly, Purify [15]'s OCI(object code insertion) is a *binary-level* technique, but it focuses on the modification of text code, rather than utilizing debugging information to lower CASE tools.

7. Summary

In this paper, we introduced a novel technique for developing program-understanding tools for embedded software, based on binary-level lightweight data integration. To verify this idea, we first proposed a new markup language for DWARF2 debugging information, and then, using the technique, we experimentally developed two cross-referencers (called `dxref` and `rxref`) and a call graph extractor (called `bscg`) for C. So far as the experiment is concerned, we had a good result.

Of course, the result of this paper is preliminary, and not enough to verify the effectiveness of our approach. we need more research and experiment. Our future works include:

- To apply the technique to other lower CASE tools like memory profilers, test coverage tools, and so on.

	lines in C	bscg	CodeViz (cobjdump)	CodeViz (cdepn)	Cflow	gprof-callgraph.pl
hello1.c	6	0.07+0.10 sec	0.80 sec	0.02+0.67 sec	0.20 sec	0.25+0.63 sec
hello2.c	6	0.80+0.16 sec	0.80 sec	0.02+0.64 sec	0.24 sec	0.25+0.64 sec
x_debug.c	1,100	0.4+2.7 sec	1.2 sec	0.03+0.94 sec	0.65 sec	0.37+0.97 sec
readelf+.c	12,000	0.7+8.1 sec	4.7 sec	0.16+7.7 sec	1.43 sec	0.33+0.84 sec
bash-2.03	44,000	4.8 sec+29min19sec[7]	13.3 sec	0.53+11.1 sec	15.0 sec	1.08+2.2 sec

7 25 seconds if enough memory is available (Sun Blade 150, UltraSPARC-IIe 550MHz, 640MB RAM)

Table 6. Execution speeds of bscg and the existing call graph generators (elapsed time)

- To apply the technique to other binary data format, or develop a new binary format suitable for lower CASE tools, e.g., by extending DWARF2.

Acknowledgments

Prof. Steven Reiss kindly explained what the Field's call graph extractor extracts from executable files. Prof. Shinichiro Yamamoto gave us a detailed explanation of how Sapid deals with GCC extensions. This research was partially supported by the Ministry of Education, Science, Sports and Culture, Grant-in-Aid 14780202, 2002-2004. We gratefully acknowledge their helps and contributions.

References

[1] A. M. Bishop, The Cxref Homepage, http://www.gedanken.demon.co.uk/cxref

[2] AT&T Labs-Research, Graphviz – open source graph drawing software, http://www.research.att.com/sw/tools/graphviz/

[3] A. Baratloo, T. Tsai and N. Singh, Transparent Run-Time Defense Against Stack Smashing Attacks, Proc. USENIX Annual Tech. Conf., 2000. http://www.research.avayalabs.com/project/libsafe/

[4] B. King, <GCC_XML description="XML output for GCC">. http://public.kitware.com/GCC_XML/

[5] Cflow, ftp://ftp.netsw.org/softeng/lang/c/tools/cflow/

[6] CSCOPE, http://cscope.sourceforge.net/

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

[8] G. C. Murphy, D. Notkin, and E. S.-C. Lan, An Empirical Study of Static Call Graph Extractors, 18th Int. Conf. on Software Engineering (ICSE), pp.90-99, 1996.

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

[10] G. J. Badros and David Notkin. A Framework for Preprocessor-Aware C Source Code Analyses, Software Practice and Experience, 30(8), pp.907-924, 2000.

[11] A Class Design Metrics Collector Using JavaML and Its Application (in Japanese), Hirohisa Aman, Kazunori Sakai, Hiroyuki Yamada and Matu-Tarow Noda, IPSJ JOURNAL 43(12):pp.4005–4008, 2002.

[12] H. Kawashima and K. Gondow, Experience with ANSI C Markup Language for cross-referencers, Proc. Domain-Specific Language Minitrack, 36th Hawaii Int. Conf. on System Sciences (HICSS-36), 2003.

[13] H. Ohashi and S. Yamamoto, SPIE – Source Program Information Explorer, http://www.sapid.org/html2/mkSpec/SPIE-0.html (in Japanese), 2002.

[14] H. Yoshida, S. Yamamoto, K. Agusa, A Generic Fine-grained Software Repository Using XML (in Japanese), IPSJ JOURNAL 44(6):pp.1509-1516, 2003.

[15] IBM Corp., Rational Purify, http://www-306.ibm.com/software/awdtools/purify/.

[16] K. Gondow. Homepage for DWARF2-XML. Japan Advanced Institute of Science and Technology (JAIST). http://www.jaist.ac.jp/~gondow/dwarf2-xml/

[17] K. Gondow and H. Kawashima, Towards ANSI C Program Slicing using XML, 2nd Int. Workshop on Language Descriptions, Tools and Applications (LDTA02), Electronic Notes in Theoretical Computer Science (ENTCS), 65(3), 2002.

[18] K. Kato and Y. Oyama, SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation, Software Security – Theories and Systems, LNCS 2609, pp.112-132, 2003.

[19] LXR Cross Referencer, http://sourceforge.net/projects/lxr

[20] M. Gorman, CodeViz – a call graph generation utility for C/C++, http://www.csn.ul.ie/~mel/projects/codeviz/

[21] P. Reinholdtsen, http://www.student.uit.no/~pere/linux/gprof-callgraph/gprof-callgraph.pl

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

[23] Programming Language C, ANSI X3.159-1989.

[24] Programming languages–C: ISO/IEC 9899:1990.

[25] Programming languages–C: ISO/IEC 9899:1999.

[26] Rationale for American National Standard for Information Systems – Programming Language – C, http://www.lysator.liu.se/c/rat/title.html

[27] S. P. Reiss. The Field Programming Environment: Friendly Integrated Environment for Learning and Development. Kluwer Academic Publishers, 1995.

[28] S. R. Tilley and D. B. Smith, Coming Attractions in Program Understanding, Carnegie Mellon University, CMU/SEI-96-TR-019, 1996.

[29] S. Yamaguchi, GNU GLOBAL – Source Code Tag System for C, C++, Java and Yacc. ftp://ftp.gnu.org/gnu/global/global-4.1.tar.gz

[30] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Ed.). World Wide Web Consortium. http://www.w3.org/TR/REC-xml

[31] Tool Interface Standards, DWARF Debugging Information Format Specification, Version 2.0, 1995.

[32] Wasserman, Anthony I., Tool Integration in Software Engineering Environments, in Software Engineering Environments: Proc. Int'l Workshop on Environments, F. Long, ed., Springer-Verlag, pp. 137-149, 1990.