David R. Cheriton School of Computer Science, University of Waterloo, Ontario, Canada

A Program Extractor Suite for C and C++: Choosing the Right Tool for the Job

Jingwei Wu [1] and Richard Holt [2]

Software Architecture Group (SWAG)
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, ON, Canada

Abstract

This report describes a suite of program extractors which we developed by adopting and extending existing program parsing and extraction techniques or tools. This suite is called CX because it is mainly targeted at extracting facts from C and C++ programs. This suite is currently composed of four extractors: CPPX, BFX, LDX and CTSX. The main goal of creating CX is to provide a convenient set of program extractors that can complement each other and work in a systematic manner. The benefits of this extractor suite will be discussed in terms of two practical applications: (1) creating program comprehension pipelines to support various understanding tasks, and (2) building an open source software evolution database (EvolDB) to support empirical research on software evolution.

1 Introduction

Program extraction is important to program comprehension and maintenance tasks [FSG04][MN96]. For example, an architect may need to monitor changes made to a software system to identify changes violating the design constraints of the system. In this case, a program extractor is needed to collect structural information to support continual maintenance. Empirical research on software evolution often requires using a robust and efficient extractor to collect structural artifacts from a large number of versions, sometime up to several hundred versions. A researcher may also be interested in deriving a domain reference architecture by studying several systems from the same domain (*e.g.*,the compiler domain and the web browser domain). In these various examples, one needs to choose a program extractor by making an appropriate tradeoff among

[1] Email: `j25wu@uwaterloo.ca`
[2] Email: `holt@uwaterloo.ca`

requirements on accuracy, efficiency and robustness. In an empirical study of software evolution, a researcher is not likely to use an accurate extractor which spends months in extracting structural artifacts from many historical versions of a long-lived system (*e.g.*, Linux [Lin04]). A robust and efficient extractor, though it may be less accurate, is more likely to prevail in such a case.

No single extraction technique can meet the highest standards on accuracy, efficiency and robustness and in the meanwhile support as many tasks as possible [SHE02]. Instead, a very small set of complementary extractors should be developed to support a wide range of tasks and meet extraction requirements to varying degrees. This paper describes our effort in developing a suite of program extractors for the C and C++ programming language. The main design goals of this suite are as follows.

- It should be simple and convenient to use.
- It should scale up to handle large programs.
- It should support diverse program analysis tasks.
- It should cover the entire build process of software.
- It should efficiently handle large numbers of versions.

CPPX [CPP02] is the first extractor we added to the CX suite. CPPX is a C/C++ source code extractor based on the front end of GCC [GCC02]. It was developed by the SWAG group at the University of Waterloo. CPPX outputs detailed information of a program at the abstract syntax graph level. However, CPPX is neither efficient nor robust in handling very large programs over many versions. This motivated us to develop several lightweight extractors to complement it. These new extractors and CPPX form a suite that provides a simple and cost-effective solution for a wide variety of program analysis tasks (see Section 3).

The rest of this paper is organized as follows. Section 2 describes the four extractors from the CX suite and their pros and cons. Section 3 describes two major applications of the CX suite. Lessons learned from each application are also summarized. Section 4 further compares four CX extractors and discusses their role in supporting systematic extraction of C/C++ programs. Section 5 considers related work. Section 6 draws the conclusion.

2 The CX Suite

The CX suite consists of four program extractors which are created by means of adapting free open source tools, including GNU Compiler Collection (GCC) [GCC02], GNU Binutils (binary utilities) [BUM02], Ctags [Cta04] and Cscope [Csc04]. These extractors are briefly summarized below.

- **CPPX** is a C/C++ source code extractor based on the GCC frontend. It relies on the preprocessing, parsing, and semantic analysis of GNU g++

and can produce program facts as detailed as abstract syntax graph (ASG).

- **BFX** is a binary code extractor built on the Binary File Descriptor (BFD) library. It parses binary code to locate definitions of functions and variables and outputs symbol references to these definitions.
- **LDX** is a binary code extractor based on the GNU code linker LD. It reuses BFX to process individual binary files and then resorts to the real code linker in the resolution of cross-references among different binary files under a specific system configuration.
- **CTSX** is an efficient and robust source code extractor built upon Ctags and Cscope. The Ctags is invoked to locate source program entities and Cscope is used to extract references to source program entities.

We now describe the four CX extractors and their pros and cons in detail.

2.1 CPPX

CPPX is a general-purpose parser and fact extractor for C and C++ programs. It relies on the preprocessing, parsing, and semantic analysis of GNU g++, and produces an abstract syntax graph in accordance with the Datrix model [Bel01]. The produced fact base is in TA [Hol02], GXL [GXL02], or VCG [San95] format.

Abstractly speaking, CPPX output is an E/R graph, which is essentially the abstract syntax graph of the source program being extracted. The vertices represent the program's templates, classes, methods, compound statements, and expressions down to the lowest level of constants and variable references. The graph edges represent syntactic relationships as well as semantic facts linking identifiers to their declarations, function calls to their targets, objects to their types, and most things to their enclosing scopes. From the CPPX output graph it is (almost) possible to reconstruct the original program [LHM03].

CPPX is suitable for use in architectural recovery, data flow analysis, pointer analysis, program slicing, query techniques, source code visualization, object recovery, refactoring, restructuring, re-modularization, and the like. It has been used in both industrial software development environments and academic software engineering research. However, CPPX has several problems. For example, CPPX may take an unusually long time (up to months) to extract facts from large software systems (up to several million lines of code); and the abstract syntax graph, which is overly detailed for many downstream analysis, often needs lengthy transformations to filter out large quantities of unwanted data such as compound expressions and statements. These problems make it necessary to develop more efficient and simplified program extractors. As a result, we have developed three lightweight program extractors, each of which is detailed in the following.

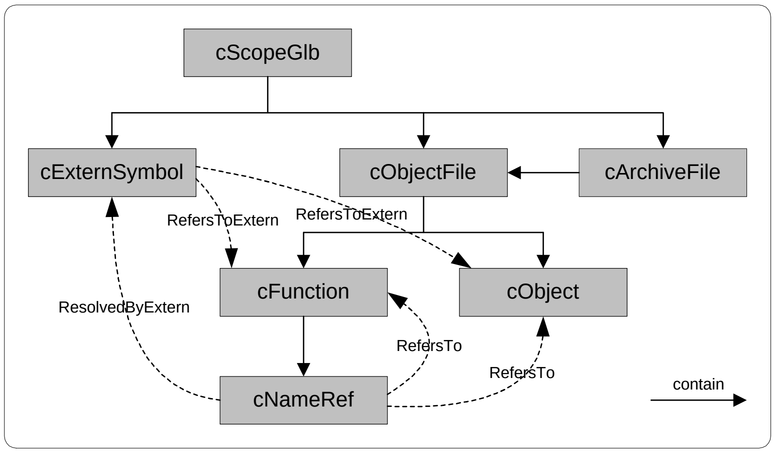

Fig. 1. BFX fact schema

2.2 BFX

BFX (Binary File Extractor) is built on the Binary File Description (BFD) library which is shipped with the GNU Binutils toolkit [BUM02]. Unlike CPPX which deals with source code, BFX extracts facts from binary code (machine code). It can process object modules (`.o`), archives (`.a`), dynamic libraries (`.so`) and executables (`.exe`). The output is in TA or GXL format and conforms to the fact schema shown in Figure 1.

The schema in Figure 1 shows that the BFX output has three levels of granularity: the level of object modules, the level of functions and variables and the level of name references. The `cObjectFile` class represents files ending with `.o`, `.so` and `.exe`. The `cArchiveFile` represents archive files (`.a`). The `cFunction` and `cObject` represent functions and variables respectively. The `cExternSymbol` represents unique string names. There are four relations: a structural relation `contain` and three reference relations `cRefersTo`, `cRefersToExtern` and `cResolvedByExtern`. The `contain` relation for any binary module always forms a tree with a universal root of type `cScopeGlb`. The `cRefersTo` relation refers to resolved references to symbols defined within the same binary module. The `cResolvedByExtern` relation denotes references to externally defined symbols and `cRefersToExtern` means that a definition can be looked up globally by searching for its name[3], *i.e.*, `cExternSymbol`.

In the above schema, `cNameRef` has an attribute called `kind`, which can be assigned a value of `F` or `V`. The `F` value means a function call and the `V` value means a variable use.

2.2.1 Pros

BFX is built on the BFD library. Technically speaking, BFX is equivalent to the binary code dump tool *objdump* [BUM02]. BFX is independent of source code compilation. Therefore, it is able to extract programs written in

[3] For C++ programs, an external definition has a mangled name.

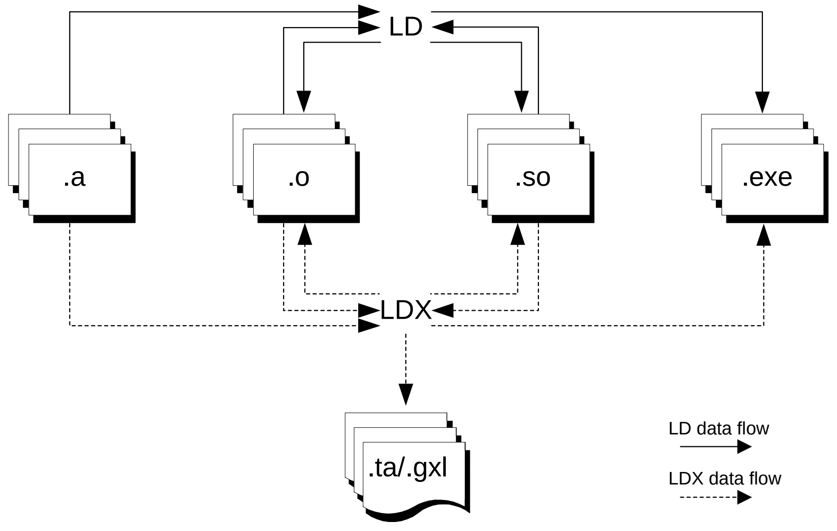

Fig. 2. LDX as a substitute for LD

programming languages other than C and C++, which include, for example, Fortran and Pascal. The BFX output is more than an order of magnitude smaller than the CPPX output since it extracts only function calls and variable uses. In terms of speed, BFX normally operates in a matter of seconds or at most minutes to extract large systems with thousands of object units. BFX's performance will be further discussed in Section 4.

2.2.2 Cons

BFX is used only after source files are compiled into object units. This results in the loss of a significant amount of structural information related to various programming constructs such as abstract data types (*e.g.*, `union` data types) and macros. The extracted function calls and variable uses reflect only what object units contain and what external symbols are referenced. There exists a gap between this kind of structural information and what the programmer sees in the source code. For example, a function-like macro, which is often treated and used like a real function by the programmer, can not be extracted using BFX since it is expanded in the step of preprocessing.

2.3 LDX

LDX (Linker Based Extractor) is built on the GNU code linker LD [BUM02]. LDX performs both code linking (including symbol name resolution across the boundary of object modules) and fact extraction. Its output includes everything produced by BFX as well as build dependencies between object modules, archive files, dynamic libraries and executables. The output is in TA or GXL format. As shown in Figure 2, LDX operates as a full substitute for LD during the extraction of a program.

Compared to BFX, LDX has two distinct features. First, it relies on the actual configuration of the program and the internal linking logic of GNU LD to resolve cross-references among separately extracted object units. Therefore, the extraction and linking of program facts are correctly and simultaneously carried out as the target system is being built. Second, LDX captures build dependencies among object units as perceived by the code linker during link time. For example, a simple hello world program (*e.g.*, `Hello.exe`) compiled on a Linux machine normally depends on an object module (*e.g.*, `Hello.o`) and a particular version of the C dynamic library (*e.g.*, `/lib/libc.so.6`).

2.3.1 Pros

Like BFX, LDX is multi-lingual and operates at a very fast speed. It adds only negligible overhead to the build process of a target system. Being a link time extractor, LDX utilizes the system configuration information (*i.e.*, build dependencies among object modules) and LD's symbol resolution functionality to derive cross-reference facts correctly. LDX does not introduce any erroneous cross-references.

2.3.2 Cons

LDX has similar cons BFX has. In addition, LDX causes slightly more interference to the build process of a software system than BFX since it needs to be substituted for LD.

2.4 CTSX

The three extractors described above require a successful build of the target system. The extraction can run into serious trouble when large numbers of versions need to be extracted. For a long lived system, it is common that many versions can not be compiled successfully on a specific platform. A robust extractor is required to tolerate erroneous and incomplete source code as well as system configuration problems. In addition, the extractor needs to be efficient in handling large numbers of versions. For example, the Linux kernel has more than 550 versions publicly posted on its official archive Web site [Lin04] until July 2005. It can be a daunting task to use CPPX, BFX or LDX to extract Linux versions since considerable manual effort is needed to deal with various configuration and compilation difficulties over the lifetime of Linux (about 12 years).

We developed a lightweight C and C++ source code extractor to support the extraction of a large long-lived software system over hundreds of versions. This extractor is based on Ctags [Cta04] and Cscope [Csc04]. For this reason it is called CTSX in which T stands for C**t**ags and S for C**s**cope.

Ctags is a tagging tool used by source code editors to parse the source code being edited. Using Ctags, editors can provide rudimentary support for code highlighting and searching. Cscope is a cross-referencing tool for browsing the source code. It allows the user to search for source code entities (definitions and regular expression patterns) as well as references to these entities in an

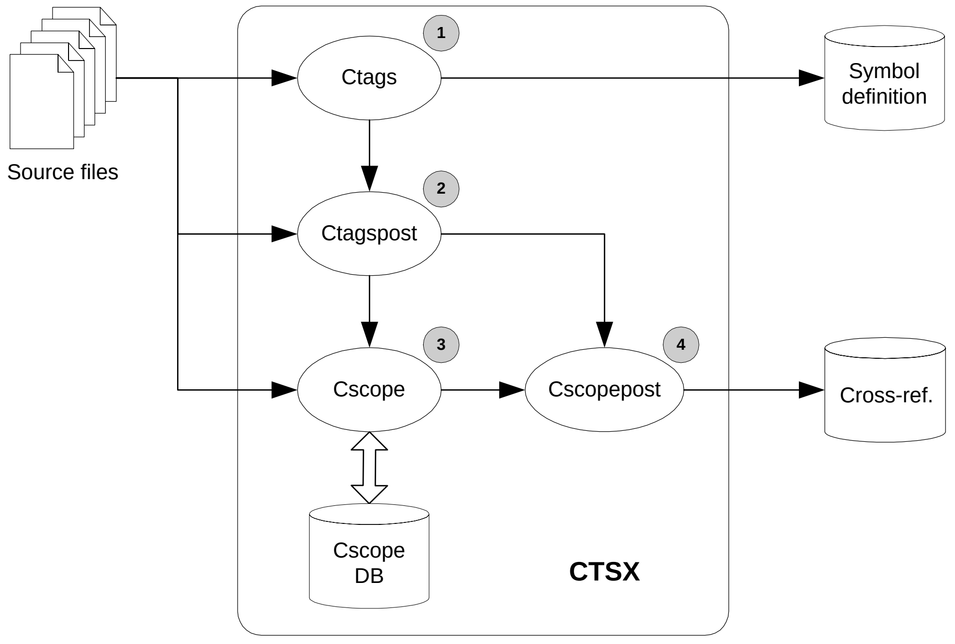

Fig. 3. CTSX built on Ctags and Cscope

interactive mode. Ctags and Cscope are both efficient and robust. They scale up to deal with more than 20 million of lines of code [Csc04].

Figure 3 illustrates the internal implementation of CTSX. CTSX has four components: Ctags, Ctagspost, Cscope and Cscopepost. We instrumented tools Ctags and Cscope to parse new command line options and to read/write facts in a proper format such as TA and CSV (Comma Separated Values). Ctagspost and Cscopepost were written in Perl and they are aimed at reducing errors and adding extra attributes. The execution order of these tools is indicated by numbers. CTSX takes in a list of source files and produces two main text files, which contain symbol definitions and cross-references respectively. The main functionality of each component is detailed as follows.

- **Ctags** extracts various program entities including functions, global variables, local variables, macros, abstract data types (`class`, `struct`, `union`, `enum` and `typedef`), enumerators (values inside an enumeration), function prototypes, namespaces, and external variable declarations. Ctags has command line options for specifying which kinds of program entity to extract. By default, CTSX instructs Ctags to extract all entities listed above.
- **Ctagspost** post-processes Ctags output to add extra information in three ways: (1) extract function parameters ignored by Ctags; (2) determine modifiers (*e.g.*, `inline` and `static`) for functions or variables; and (3) calculate the effective scope of local variables and function parameters. To be simple, the effective scope of a local variable is treated to be equal to the scope of the function enclosing the variable.

  Ctagspost needs to parse the source code briefly to collect extra information because additional texts associated with each program entity located by

Ctags are insufficient for collecting the extra information mentioned above.

- **Cscope** parses the entire source code to build an initial cross-reference database. It then reads program entities from Ctagspost output and retrieves static references to those entities. Depending on the command line options the user specifies for CTSX, Cscope can retrieve different kinds of cross-reference. By default, it outputs references to functions, global variables, macros, and data types (including `typedef`s).

  Every name reference produced by Cscope is assigned a `kind` attribute to indicate its type. Cscope itself can determine which reference is a function call. However, the type of other references is determined by using regular expressions to match relevant code syntax. For example, a type reference can be determined if the type name appears as part of a declaration or a type cast. The references to variables and macros are not distinguished in the current implementation.
- **Cscopepost** filters out references to static functions, static variables, local variables and function parameters through the use of the scoping and accessibility information produced by Ctagspost. For example, a reference to a static function within the same file is filtered out as a non cross-reference. A reference to a global variable is filtered out as an erroneous reference if it falls in the scope of a local variable which has the same name as the global variable. The Cscopepost output contains only references that cross the actual boundary of source files.

2.4.1 Pros

Compared to writing an extractor from scratch, the reuse of Ctags and Cscope significantly speeds up the development of CTSX. Only a few days were spent in instrumenting Ctags and Cscope and writing the postprocessing utilities in Perl scripts.

CTSX is a useful program extractor when (1) the target system is extremely large and time consuming to build, (2) the system cannot be built due to configuration or compilation errors, and (3) the correctness of extracted facts is not of critical concern to downstream software analysis. These characteristics make CTSX suitable for extracting program facts of reasonable quality from the evolution history of a long-lived software system in a timely and cost-effective manner. The benefits of CTSX will be demonstrated in section 3.2.

2.4.2 Cons

As a lightweight program extractor, CTSX is more error prone than CPPX, BFX and LDX. This is mainly caused by three factors. First, Ctags is reliant on fault tolerant parsing and it thus may result in the missing of program entities or the recognition of wrong entities. Second, Cscope has no knowledge of the typing of symbol references except function calls. The lexical analysis

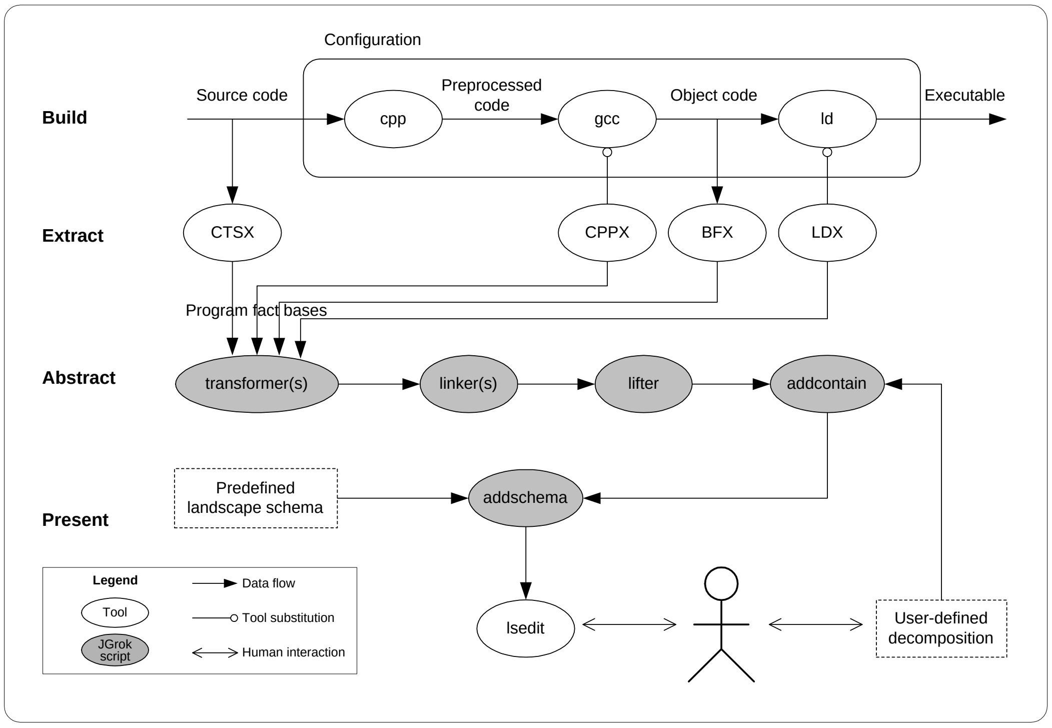

Fig. 4. Program comprehension pipelines built on the CX extractors

based on regular expressions can only alleviate this typing problem. Third, Cscope does not differentiate references to local and global program entities. To reduce the undesirable impacts of these factors, Ctagspost and Cscopepost are added to search for more semantic clues. However, without complete semantic analysis, it should not be expected that CTSX or any lightweight parsing techniques (commonly based on regular expressions [MN96] and island grammars [Moo01]) can produce results as accurate as those produced by the extractors based on full parsing and semantic analysis (*e.g.*, CPPX).

3 Applications

This section describes two main applications of the CX suite to demonstrate its benefits in practice: (1) creating various program comprehension pipelines to aid software developers and researchers; and (2) building an open source software database to support empirical research on software evolution. These applications involve a number of open source software systems. Each involved system is briefly described in Appendix A.

3.1 Creating Comprehension Pipelines

A program comprehension pipeline commonly consists of three main steps: *extract*, *abstract* and *present*. The SwagKit uses such a pipeline to manipulate CPPX facts [Swa02]. As three new extractors (BFX, LDX and CTSX) were developed, we have extended the old SwagKit by implementing new comprehension pipelines to support more diverse software analysis tasks cost-effectively, for example, the recovery of reference architecture for an application domain [GG05].

Figure 4 illustrates the main components shared by various comprehension pipelines currently supported by the extended SwagKit. Each extractor from the CX suite serves as the starting point of a specific comprehension pipeline. The four main steps are as follows:

- **Build** is an extra yet important step in a program comprehension pipeline. The build process of a C/C++ software system normally consists of configuration, compilation and linking. For extractors which are created by adapting build related tools such as the compiler and linker, the build process provides a simple vehicle for carrying out program extraction. However, build is not needed for extractors which ignore system configuration information and perform their own parsing and semantic analysis from scratch.
- **Extract** refers to the extraction of program related facts by using appropriate tools. Broadly speaking, the extracted facts can be as abstract as software architecture or as concrete as cross-references among various program entities (*e.g.*, functions). The latter is what a program extractor normally produces.

  CTSX can be used independent of a system's configuration. By contrast, the other three extractors must be applied after the system is configured. CPPX and LDX need to be embedded into the build process through tool substitution. Program facts are extracted as the system is being built. BFX can be embedded into the build process or applied directly on all object modules after a successful compilation.
- **Abstract** is the step of manipulating program facts through transformation, linking and lifting. Facts produced by different extractors are transformed into an appropriate form with unwanted program entities and relationships removed and new program entities and relationships created. A sequence of *transformers* is applied until a desired form is achieved. The transformed program facts for individual files or units are linked to form a graph model representing a part of the system or the whole system. The *linker* resolves references across the boundary of files. The *lifter* abstracts lower level cross-references into higher level dependencies. The *addcontain* script imposes a subsystem decomposition hierarchy on the flat system model. The resulting model can be further lifted if it is too large to handle in a subsequent step.
- **Present** is the last step. The *addschema* adds a standard schema to the abstracted system model to produce a software landscape view which a user can explore by using the visualization tool *lsedit* developed by the SWAG group[Swa02]. The user can examine relationships among various program entities at different levels of granularity.

After program facts are transformed, all the pipelines share the remaining tools: *linker*s, *lifter*, *addcontain*, *addschema* and *lsedit*. These tools are implemented in JGrok, a scripting language designed for manipulating sets and relations [JGr04]. Several different linkers can be used to combine individual small program models into a large system model. Depending on the linker, the resulting system model can contain erroneous dependencies to varying degrees [WH06].

Pipeline	Example for Architecture Recovery	Example for Domain Reference Architecture Recovery	Example for Software Evolution Analysis
CPPX Pipeline	Emacs (2003) InnoDB (2002) MySQL (2002) PostgreSQL (2002)		PostgreSQL (2003) [ZG03]
BFX Pipeline	DB2 UDB (2003) KSpread (2004) Mozilla (2004)	**Web Browser** (2004) [GG05]: Dillo Epiphany Flower Konqueror Lynx Mozilla Safari **Instant Messenger** (2004): CenterICQ EG-lite Gaim Kopete Miranda	KSpread (2004) OpenSSH (2004) PostgreSQL (2004) Linux kernel (2004)
LDX Pipeline	DB2 UDB (2003) Gnumeric (2004)	**First Person Shooter** (2003): Cube Quake Quake II	Gnumeric (2004) OpenSSH (2004) PostgreSQL (2004) [WH04] Linux kernel (2004) [WH04]
CTSX Pipeline	OpenOffice (2005)		

Table 1
Uses of comprehension pipelines on software systems

3.1.1 Common Uses

The comprehension pipelines starting with each CX extractor have been successfully used in a variety of circumstances which include industrial and academic environments as well as graduate course teaching. They are mainly used to support three kinds of software analysis: software architecture recovery, software evolution study, and domain reference architecture recovery. Table 1 lists systems on which four pipelines have been applied in each kind of analysis. A brief introduction to the performed analysis is given below.

- Software architecture recovery aims at reconstructing views on the system architecture as-built. A recovered architecture is commonly represented using a hierarchical organization with components and their relationships at varying levels of granularity.

- The empirical study of software evolution is commonly conducted as a longitudinal analysis of one or more software system properties such as the system size and system structural complexity.
- Domain reference architecture recovery aims at generalizing more than one recovered system architectures from the same application domain into a standardized architecture which can be referenced for developing and understanding similar applications.

3.1.2 Lessons Learned

A number of lessons have been learned from observing how these comprehension pipelines were used in practice. The users mainly include graduate students from the SWAG group and students who enrolled in graduate level software engineering courses at the University of Waterloo as well as myself. During the period from 2002 to 2005, I worked as a teaching assistant to aid students in applying these pipelines on a number of open source software systems. Here are some of the lessons learned.

- Perform system extraction after a successful build. This requirement characterizes a common way of using program extractors which need to be embedded into the build process. This has two important implications when a pipeline requires a successful build. First, build success increases the user's confidence of performing an extraction. Second, by building a system, the user can determine proper components (or features) that need to be included as the system is being configured by the user.
- The pipelines based on BFX/LDX are more convenient to use than the one based on CPPX. Two factors contribute to this observation: (1) CPPX has noticeable defects in extracting source code and often causes compilation errors; and (2) BFX/LDX is faster and produces less detailed information than CPPX. This perhaps explains why students tend to use BFX/LDX to deal with multiple versions which either belong to the same system or are from the same domain.
- CTSX is not satisfactory in conducting an accurate extraction of large C++ systems. The sheer volume of overloaded methods, local variables and type casts causes CTSX to produce a fairly large amount of inaccurate facts. In 2005, a student group enrolled in the graduate course on software architecture recovery felt uncomfortable with the accuracy of facts extracted from OpenOffice using CTSX. The group switched to the BFX-based pipeline.

In brief, the four program comprehension pipelines built on the CX suite provide useful support for analyzing large software systems in a variety of ways. Depending on the nature of software analysis, the user can choose an appropriate extractor or use different extractors in combination.

3.2 Building Software Evolution Database (EvolDB)

One of the main factors that impede empirical research on software evolution is the lack of appropriate tool support for collecting historical information from a long lived software system [KS96]. Software structural artifacts such as call graph are important for software understanding and maintenance tasks [MNS95]. A sequence of structural artifacts collected from the historical versions of a system can be valuable for understanding how the system has evolved [CKN^{+}03][ZG03].

This section describes the application of the CX suite on a number of large open source systems to build a large evolution database of software structural artifacts. This database is referred to as EvolDB. We have extracted ten open source systems over large numbers of versions (official releases or daily snapshots). These systems include GCC, KSDK, KOffice, Linux, Mozilla, OpenSSH, OpenSSL, PHP, PostgreSQL and Ruby. The benefits of building EvolDB include the following:

- The extraction of diverse software systems over a large number of versions can subject the CX suite to substantial software development in real life. The CX extractors can thus be contrasted in terms of quality attributes such as robustness and efficiency.
- The resulting evolution database is free for public use. The researchers interested in software evolution can take advantage of this database to advance empirical research on open source software evolution. Possible directions may include validating software evolution laws or explaining open source software evolution from new perspectives.

Table 2 summarizes how many versions were extracted and how much time was spent for each target software system. The extraction was conducted on either official releases or snapshot versions checked out from the CVS source control repository. Two computers were used. CPPX, BFX and LDX were used on a Linux machine with one Intel(R) Pentium(R) IV 1.6GHz CPU and 1GB memory. CTSX was used on a Linux server with two Intel(R) Xeon(R) 2.2GHz CPUs and 4GB memory. Although the server has more computing power, it is approximately 40% faster than the first computer due to the running of many backend services and user tasks. A detailed performance comparison of these CX extractors will be described in section 4.1.

3.2.1 Using CPPX

We substituted CPPX for *gcc* to extract abstract syntax graphs from 28 official releases of PostgreSQL, which range from 6.0 to 7.3.4. The extraction ran into two major difficulties: (1) every version before 6.5 had configuration problems such as missing header files and could not be built successfully, and

System	Period		Size (KLOC)		Versions	Extraction time (hours)		
	From	To	From	To		CPPX	BFX+LDX	CTSX
OpenSSH†	1.2pre6 (1999)	3.8p1 (2004)	20	70	60	15.1	2.2	0.11
PostgreSQL†	6.0 (1997)	7.3.4 (2004)	182	519	28	16.5	2.6	0.32
Linux kernel†	2.0 (1996)	2.5.75 (2003)	674	5140	368	-	87.2	-
	1.0 (1994)	2.6.12.3 (2005)	165	5954	581	-	-	36.32
Mozilla†	1.0 (1999)	1.7.3 (2004)	3700	4500	19	-	14.3	4.13
PostgreSQL◇	1997-01-01	2005-01-01	182	519	97	-	13.7	-
PostgreSQL*	1997-01-01	2005-09-09	182	556	3175	-	-	29.70
GCC*	1997-08-12	2005-09-09	582	1559	2951	-	-	122.89
KSDK*	1999-01-01	2004-12-31	3	263	2192	-	-	7.09
KOffice*	1999-01-01	2004-12-31	272	962	2192	-	-	53.81
OpenSSL*	1999-01-01	2005-09-09	164	291	2444	-	-	18.22
PHP*	1999-04-08	2005-09-09	16	645	2347	-	-	23.58
Ruby*	1999-01-01	2005-09-09	74	198	2444	-	-	9.01

†: official release ∗: daily snapshot version ◇: monthly snapshot version (the first day of every month)

Table 2
Open source systems extracted using the CX extractors

(2) CPPX has some defects in handling very large arrays of strings and it ran out of memory for almost each of the 28 versions of PostgreSQL. We modified configuration scripts and source files to achieve a successful compilation on the Linux platform. We also used CPPX to extract a total of 60 official releases of OpenSSH ranging from 1.2pre6 to 3.8p1. Similar problems were encountered and fixed manually.

The measured extraction time given in Table 2 does not include the time we spent on solving problems. After a system could be built and extracted successfully, we cleaned all the generated object code and re-compiled the system from scratch in order to measure the time spent on extraction.

3.2.2 Using BFX and LDX

LDX and BFX were successfully used to extract four large systems (OpenSSH, PostgreSQL, Linux and Mozilla) over a large number of versions. These two extractors are able to handle very large systems at a reasonable speed. The time spent by both BFX and LDX accounts for approximately 5∼10% of the total build time of a system (see section 4.1).

When using BFX and LDX on Mozilla, we did not encounter any configuration problems and compilation errors. For the other three systems, the older the version, the more likely we ran into problems. All encountered problems were fixed manually.

3.2.3 Using CTSX

CTSX was applied to extract all the ten open source software systems. For systems which have a CVS repository, we conducted daily snapshot extraction. For OpenSSH, Linux and Mozilla, only public official releases were extracted. The extraction of 2951 daily snapshots of GCC took the longest time (122.89 hours) to finish. In general, the total time spent by CTSX on each target system is satisfactory. CTSX did not break for any version we have extracted.

3.2.4 Lessons Learned

We now discuss several lessons learned from building large software evolution databases.

- For a large software system, many of its official releases may not be readily configured and compiled because the underlying platform (both hardware and software) changes drastically. This kind of build break often needs to be solved by manually modifying the configuration scripts or even the source code in order to get the system compiled. Installing outdated libraries is also required sometimes. Our experience with Linux and PostgreSQL shows that it is a daunting task to obtain a successful build for every available release. A program extractor depending on build success is not suitable for extracting a large number of versions over a long period of time.
- It is not guaranteed that CPPX can perform a successful extraction even if a software system can be compiled successfully. This is mainly caused by the internal defects of CPPX. For example, a very large array of constants can cause CPPX to break. By contrast, BFX and LDX are relatively more robust than CPPX. They can be applied to extract a version as long as the version can be compiled. CPPX transforms abstract syntax graphs which are more complicated than the format of binary code handled by BFX and LDX. This is the main reason why CPPX is less robust.
- CTSX is more satisfactory than CPPX, BFX and LDX in extracting a large software system in a robust but less accurate manner. In particular, it is capable of performing daily snapshot extraction on a long lived system within just a few hours or at most several days. CTSX is well positioned for supporting software evolution analysis on a large scale.

4 Discussion

4.1 Performance

Performance measures the speed of an extractor in extracting program facts. Performance is an important consideration when extraction is conducted on very large programs, which may have several million lines of code. In particular, if hundreds of versions of such a large system (*e.g.,* Linux) need to be extracted for the purpose of examining various evolution phenomena (*e.g.,* growing complexity), speed becomes a critical concern in the design of a program extractor.

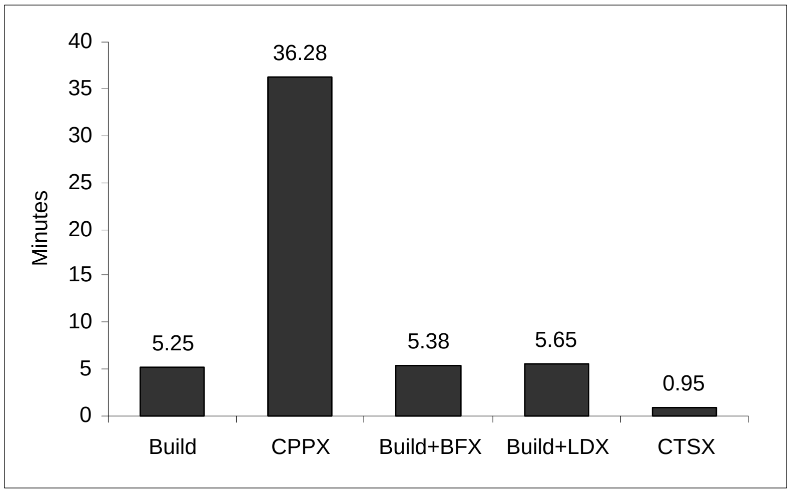

Fig. 5. Performance comparison of the CX extractors

We applied the CX extractors on PostgreSQL 7.4 (an open source database management system) to conduct a performance comparison. Each CX extractor was used repeatedly to extract the system three times. The average extraction time was calculated. The average time required to build the system was also collected in order to provide a standard base for showing speed differences. Figure 5 displays the performance results obtained on a Linux machine with an Intel Pentium IV 1.6GHz CPU and 1GB memory.

It took 5.25 minutes on average to build PostgreSQL by using the standard tool chain which includes *configure*, *make*, *gcc* and *ld*. We reused this build process to conduct program extraction by substituting CPPX for *gcc*. The total expended time was 36.28 minutes which is about 7 times large as the build time. BFX was used after build success and cost only a small fraction of the build time, roughly 2.5%. LDX spent slightly more time than BFX since it resolves cross-references between object modules. CTSX spent the least amount of time, which approximately accounts for 18% of the build time. Overall, CTSX, BFX/LDX and CPPX can have speed differences up to an order of magnitude.

A detailed examination of the four components of CTSX shows that Ctags, Ctagspost, Cscope and Cscopepost roughly account for 10%, 20%, 35% and 35% of the total extraction time respectively. Reusing Ctags and Cscope reduces the development time but also yields a negative impact on the speed of the extractor. However, building a more efficient extractor from scratch is achievable but would require a longer development time. In our work, we have focused on the rapid development of an extractor by means of adopting and enhancing existing tools.

4.2 Accuracy and Robustness

In CppETS (C++ Extractor Test Suite), Sim suggests that accuracy and robustness are two important dimensions for evaluating an extractor [SHE02]. Accuracy measures the correctness of facts extracted by a program extractor, and robustness measures how well the extractor deals with irregularities present in the source code.

A software system, which has been maintained over a long period of time, often contains code written in different programming languages or different dialects of the same language. Languages like SQL and Assembly may also be embedded in the code. It is not uncommon that the system might be targeted at an outdated computing platform or built on obsolete libraries, which are not available nowadays. The missing source files (including header files) and unrelated entities with similar names make matters even worse when extraction needs to be conducted over a long history of releases. All such irregularities complicate building the program extractor. The extractor robustness and the accuracy of the extractor output must be treated properly.

The three extractors CPPX, BFX and LDX are of high accuracy and low robustness while CTSX is the opposite. A detailed explanation is given below:

- Based on the well-engineered compiler frontend of GCC, CPPX theoretically outputs data as accurate as the abstract syntax graph constructed by the compiler. However, the evaluation of CPPX against CppETS yielded a low score of accuracy due to the premature implementation of the extractor itself [SHE02].
- BFX and LDX are very close to a compiler-based extractor in terms of accuracy and robustness. Several factors contribute to this claim. First, BFX and LDX are built on mature software tools (GNU LD) and libraries (BFD) which are of product quality and have been exhaustively tested. Thus the accuracy of their output is equivalent to what a code linker or a binary utility tool (*objdump*) can see from the binary code. Second, BFX and LDX operate only after source code are compiled into object code. If the compiler failed to produce object code, it is not possible for BFX and LDX to finish the extraction. So, both extractors are only as robust as a compiler.
- Given that Ctags and Cscope are mainly based on lexical analysis and robust parsing techniques, CTSX is able to recover gracefully from unexpected syntax and continue parsing without a failure. The direct consequence is that CTSX produces more errors than CPPX, BFX and LDX. In particular, CTSX is not satisfactory in extracting facts from large C++ systems. The use of CTSX in building an open source software evolution database without failure shows its high robustness in reality.

4.3 Systematic Support for C/C++ Extraction

The CX extractor suite supports a systematic extraction of C/C++ programs. By *systematic*, we mean the following:

- The CX extractors cover different steps in the build process of C/C++ a program. CTSX deals with un-preprocessed source code; CPPX works on preprocessed source code in the step of compilation; BFX is used after source compilation; and LDX is used in the step of code linking.

  In fact, the current CX suite lacks a fact extractor targeted at the preprocessing step. As a result, dependencies between macros and other source code entities cannot be extracted. A viable solution will be to instrument the GNU C preprocessor to develop a new fact extractor, thus extending the current suite to cover the entire build process. This remains our future work.
- The CX extractors can be conveniently embedded into the build process to automate program extraction without causing much interference. Especially, CPPX and LDX work as substitutes for GCC and GNU LD.
- The suite is applicable to two most important types of software artifact: source code and binary code. CPPX and CTSX extract program facts from source code but BFX and LDX extract from binary code.

5 Related Work

The C/C++ source code is commonly extracted using parser-based extractors, which can be handwritten from scratch or built on existing parsers or compilers (*e.g.*, GCC). Several extractors such as rigiparse [MOTU93], CPPX [CPP02], CAN [FBMG01] and TkSee/SN [TkS03] belong to this category. These extractors can produce detailed structural information. However, the full parser based approach does not solve the robustness issue (dealing with missing code or syntactical errors).

There are a number of parsing techniques based on the idea of partial extraction and regular expression matching [CC03][Moo01][MN96]. Murphy and Notkin present a lightweight lexical technique for extracting information from source code, structured data files, and documentation [MN96]. Their approach allows the user to specify language features using hierarchical patterns and regular expressions. Moonen proposes a formalism called Island Grammars for partially specifying languages that contain irregularities and generating a parser based on the specification [Moo01]. Islands are specified using production rules and regular expressions. Islands are captured by the generated parser. The rest of the source program is treated as water and ignored. An island grammar based parser generator called MANGROVE has been developed. These extraction techniques are generally more flexible, lightweight and fault tolerant. But the extractors built on them normally produce less accurate results than a full parser based extractor.

By contrast, our work is not intended to develop more complicated and advanced extraction techniques but to reuse existing tools to form a small suite of program extractors to support a wide variety of C/C++ extraction needs. We have adopted the CPPX extractor and developed instrumented versions of GNU compiler tools (*e.g.*,LD). These extractors can be conveniently embedded into the build process of a software system to extract structural information. In case that the system cannot be built successfully, CTSX can be used to carry out the extraction. CTSX is satisfactory in terms of performance and robustness though its output is less accurate than what the other three CX extractors produce.

6 Conclusion

CPPX, BFX, LDX and CTSX are complementary C/C++ extractors, which cover different steps in the build process of a C/C++ software system. These extractors provide a range of tradeoffs among accuracy, robustness and efficiency. They provide systematic tool support for a wide variety of program extraction tasks. In particular, CTSX is efficient and robust for extracting a software system over hundreds of versions. The benefits of these extractors are discussed with regard to two major applications: (1) creating program comprehension pipelines and (2) building an open source software evolution database.

References

[Bel01] Bell Canada. *DATRIX Abstract Semantic Graph Reference Manual, Version 1.4*. Website: http://www.casi.polymtl.ca/casibell, 2001.

[BUM02] Binutils User Manual. Website: http://www.gnu.org/software/binutils, 2002.

[CC03] Anthony Cox and Charles Clarke. Syntactic approximation using iterative lexical analysis. In *Proceedings of the 11th International Workshop on Program Comprehension*, pages 154–163, Portland, Oregon, May 2003.

[CKN+03] Christian Collberg, Stephen Kobourov, Jasvir Nagra, Jacob Pitts, and Kevin Wampler. A system for graph-based visualization of the evolution of software. In *Proceedings of ACM Symposium on Software Visualization*, pages 77–86, San Diego, California, June 2003.

[CPP02] CPPX. C++ Source Code Extractor. Website: http://swag.uwaterloo.ca/~cppx, 2002.

[Csc04] Cscope. Website: http://cscope.sourceforge.net, 2004.

[Cta04] Ctags. Website: http://ctags.sourceforge.net, 2004.

[FBMG01] Rudolf Ferenc, Arpad Beszedes, Ferenc Magyar, and Tibor Gyimothy. A short introduction to Columbus/CAN. *Technical Report*, 2001.

[FSG04] Rudolf Ferenc, István Siket, and Tibor Gyimóthy. Extracting facts from open source software. In *Proceedings of the 20th International Conference on Software Maintenance*, pages 60–69, Chicago, Illinois, September 2004.

[GCC02] GCC. GNU Compiler Collection. Website: http://gcc.gnu.org, 2002.

[GG05] Alan Grosskurth and Michael W. Godfrey. A reference architecture for web browsers. In *Proceedings of the 21th International Conference on Software Maintenance*, Budpest, Hungary, September 2005.

[GXL02] GXL. *The Graph eXchange Language*. Website: http://www.gupro.de/GXL, 2002.

[Hol02] Richard C. Holt. *An Introduction to TA: the Tuple-Attribute Language*. Website: http://plg.uwaterloo.ca/~holt/cv/papers.html, 2002.

[JGr04] JGrok. A Query Language for Reverse Engineering. Website: http://swag.uwaterloo.ca/tools.html, 2004.

[KS96] Chris F. Kemerer and Sandra Slaughter. Need for more longitudinal studies of software maintenance. In *Proceedings of the International Workshop on Empirical Studies of Software Maintenance*, Monterey, California, USA, 1996.

[LHM03] Yuan Lin, Richard C. Holt, and Andrew Malton. Completeness of a fact extractor. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 196–205, Victoria, British Columbia, Canada, November 2003.

[Lin04] Linux Kernel. Website: http://www.kernel.org, 2004.

[MN96] Gail C. Murphy and David Notkin. Lightweight lexical source model exraction. *IEEE Transactions on Software Engineering*, 5(3):262–292, July 1996.

[MNS95] Gail C. Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *Proceedings of ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, New York, NY, USA, October 1995.

[Moo01] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 13–22, Suttgart, Germany, October 2001.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to system structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[San95] Georg Sander. VCG – visualization of compiler graphs. *Technical Report A01-95*, February 1995.

[SHE02] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook. On using a benchmark to evaluate c++ extractors. In *Proceedings of the 10th International Workshop on Program Comprehension*, pages 114–123, Paris, France, June 2002.

[Swa02] SwagKit. *The Software Architecture Group (SWAG) Analysis Toolkit.* Website: http://swag.uwaterloo.ca/swagkit, 2002.

[TkS03] TkSee/SN. *A C++ Source Code Extractor Based on Cygnus Source Navigator.* Website: http://www.site.uottawa.ca/~tcl/kbre, 2003.

[WH04] Jingwei Wu and Richard C. Holt. Linker-based program extraction and its uses in studying software evolution. In *Proceedings of the International Workshop on Foundations of Unanticipated Software Evolution*, pages 1–15, Barcelona, Spain, March 2004.

[WH06] Jingwei Wu and Richard C. Holt. On improving linkage resolution in software system model extraction. Technical report, David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada, May 2006.

[ZG03] Lijie Zou and Michael W. Godfrey. Detecting merging and splitting using origin analysis. In *Proceedings of the 10th Working Conference on Reverse Engineering*, pages 146–154, Victoria, BC, Canada, November 2003.