Proceedings of the Tenth International Workshop on Program Comprehension, pp. 114-123, Paris, France, 27-29 June 2002.

On Using a Benchmark to Evaluate C++ Extractors

Susan Elliott Sim
Dept. of Computer Science
University of Toronto
simsuz@cs.utoronto.ca

Richard C. Holt
Dept. of Computer Science
University of Waterloo
holt@plg.uwaterloo.ca

Steve Easterbrook
Dept. of Computer Science
University of Toronto
sme@cs.utoronto.ca

Abstract

In this paper, we take the concept of benchmarking as used extensively in computing and apply it to evaluating C++ fact extractors. We demonstrated the efficacy of this approach by developing a prototype benchmark, CppETS 1.0 (C++ Extractor Test Suite, pronounced see-pets) and collecting feedback in a workshop setting. The CppETS benchmark characterises C++ extractors along two dimensions: Accuracy and Robustness. It consists of a series of test buckets that contain small C++ programs and related questions that pose different challenges to the extractors. As with other research areas, benchmarks are best developed through technical work and consultation with a community, so we invited researchers to apply CppETS to their extractors and report on their results in a workshop. Four teams participated in this effort, evaluating Ccia, cppx, the Rigi C++ parser, and TkSee/SN. They found that CppETS gave results that were consistent with their experience with these tools and therefore had good external validity. Workshop participants agreed that CppETS was an important contribution to fact extractor development and testing. Further efforts to make CppETS a widely-accepted benchmark will involve technical improvements and collaboration with the broader community.

1. Introduction

Fact extraction from source code is a fundamental activity for reverse engineering and program comprehension tools, because all subsequent activities depend on the data produced. As a result, it is important to produce the facts required, accurately and reliably. Creating such an extractor is a challenging engineering problem, especially for complex source languages such as C++ [5, 6].

Consequently, it would be useful to have a convenient means to evaluate a fact extractor. In this paper, we report on our experiences designing and using a benchmark for this purpose. We have prototyped a benchmark for C++ extractors, called CppETS (C++ Extractor Test Suite, pronounced see-pets). We chose a difficult source language because benefits can be realised quickly and the lessons transferred to other source languages. The benchmark consists of a series of test cases each with a set of related questions.

This benchmark has been well-received by all who have used or reviewed it. It has been used by four teams of program comprehension researchers to evaluate Ccia, cppx, Rigi C++ parser, and TkSee/SN. The results were presented and discussed at a workshop at CASCON 2001 in November of last year [15]. Despite being a prototype, the CppETS worked very effectively. The teams used the benchmark primarily to test their extractors, while we used the workshop to evaluate the benchmark. The participants generally felt that the test cases were representative of reverse engineering problems and the ratings of their extractors were fair. Following the workshop, developers from IBM and Sun have downloaded the benchmark and are using it as part of their internal test processes. All the materials for CppETS are available online [15]. We plan to refine the prototype into a widely-accepted benchmark, by applying it to additional tools and soliciting feedback from the community.

Before discussing the CppETS itself, we review benchmarking and previous work on evaluating source code extractors.

1.1 Benchmarks

A benchmark is a convenient way to encapsulate the materials and procedure for an empirical study and can be used to answer a wide variety of questions. Walter Tichy defines a benchmark, as "…a task domain sample executed by a computer or by a human and computer. During execution, the human or computer records well-defined performance measurements. [19, p. 36]" Extending Tichy's definition, a benchmark has three components: a motivating comparison, a task domain sample, and performance measures.

1. **Motivating Comparison.** The design of the benchmark is motivated by a particular comparison that its users would like to make. This comparison is made for a purpose, such as, making a purchase or engineering a tool (or technique or technology) to meet a goal. A particular tool can be compared against itself over time, e.g. during development. Or, different tools can be compared against each other.
2. **Task Domain Sample.** The tests in the benchmark should be a representative sample of the tasks that the tool is expected to solve in actual practice.

3. **Performance Measures.** These measurements can be made by a computer or by a human, and can be quantitative or qualitative. Performance is not an innate characteristic of the tool, but is the relationship between the tool and how it is used. As such, performance is a measure of fitness for purpose.

The motivating comparison drives the selection of task domain sample, which in turn drives the selection of the performance measures.

Benchmarks, like standards, are created through a process of community consultation and technical refinement. The composition of each component needs to be scrutinised; what tasks should be included and what measures should be used. As Tichy wrote,

> The most subjective and therefore weakest part of a benchmark test is the benchmark's composition. Everything else, if properly documented, can be checked by the skeptic. Hence, benchmark composition is always hotly debated. [19, p. 36]

This debate can be contentious in research areas where innovative tools may not have a well-defined task domain or performance criteria. These are often determined progressively through investigation and peer review. The motivating comparison is controversial, because it is symbolic of the goal of a research area

Controversy and the ensuing discussions are highly beneficial for a research community, particularly when they reach consensus on the three components of a benchmark. A standard benchmark translates into agreement on the goals of the discipline and how to measure progress in the field by setting well-defined objectives and a foundation for subsequent work. Quoting Tichy again, "…a benchmark can quickly eliminate unpromising approaches and exaggerated claims" and "benchmarks can cause an area to blossom suddenly because they make it easy to identify promising approaches and discard poor ones. [19, p. 36]"

Two well-known benchmarks are TPC-A for databases and SPEC CPU2000 for computer systems. Their development paths illustrate the amount of community involvement required to create a widely-accepted and widely-used benchmark. Both of these were developed with extensive collaboration between industry and research, as well as consultation with the broader user community. The Transaction Processing Performance Council's TPC Benchmark™ A, more briefly TPC-A, was first published in 1989 and had evolved over several generations from a benchmark *DebitCredit* first described in a paper in 1984 [7]. (This paper had so many contributors from various organisations that the author was given as "Anon et al.") Developing TPC-A required nearly 1200 person-days of effort contributed by 35 database vendors who were members of the consortium.

SPEC (Standard Performance Evaluation Corporation) is also a consortium with different committees responsible for creating different benchmarks. The committees have representatives from all the major hardware vendors as well as researchers from universities [8]. Requirements, test cases, and votes on benchmark composition are solicited from committee members and the general public through SPEC's web site. The committee uses "benchathons" to refine the benchmark. John Henning explained:

> The point of a benchathon is to gather as many as possible of the project leaders, platforms, and benchmarks in one place and have them work collectively to resolve technical issues involving multiple stakeholders: At a benchathon, it is common to see employees from different companies looking at the same screen, helping each other. [8, page 30]

It is useful to keep these two examples of mature benchmarks in mind when considering our prototype, CppETS.

1.2 Evaluating Extractors

A number of studies have evaluated source code extractors in the context of examining static call graph extractors 14], architectural extractors [3], and program comprehension tools [16]. All of these studies found that extractors varied significantly in terms of accuracy, reliability, richness of facts emitted, usability, and features. Our work draws conceptually on the study by Murphy et al. [12] and methodologically on the studies by Armstrong and Trudeau [3], and by Sim and Storey [16].

Murphy et al.'s study found that a set of nine call graph extractors all produced false positives and false negatives at different rates for each of three subject systems. In other words, no extractor made the errors consistently across the three programs. In their conclusion, they wrote:

> It may be possible to engineer tools that guarantee certain behavioral properties… Or, it may be sufficient to more effectively communicate the design decisions that specific extractors have made… *Another possible approach is to develop new tools and techniques for helping an engineer assess the kind of call graph extracted*. [12, p. 182, italics added]

While CppETS does not help an engineer determine the kind of call graph extracted, it does develop a technique (and results) to help a tool designer or tool user select an extractor that is appropriate for the tasks they wish to undertake.

For their study, Armstrong and Trudeau created a small C program, called `degen`, that contained a number of features that are problematic for source code extractors. They tested five extractors using `degen`, and none of them were able to handle all the features. Our benchmark borrows significantly from `degen`, in that we tested extractors using small source code examples and examined how they handled specific features.

The benchmark approach also builds on work started with the structured demonstration approach used by Sim and Storey [16]. Similar to that study, we put source code and tasks together in a package that researchers and developers could use with their own tools. Results were also shared and discussed in a workshop setting.

1.3 Overview

The remainder of this paper is organised as follows. In Section 2, we describe the collection of tests in CppETS and the approach used to select them. Our experience and results from using the benchmark with four extractors are presented in Section 3. Lessons learned and their implications for refining the benchmark are discussed in Section 4. The paper concludes with Section 5, where we discuss future work and reflect on the evolution of benchmarks.

2. Composition of the Benchmark

In this section, we describe the design of the CppETS. Our work began with the motivating comparison. From there, we selected the task domain sample, or test buckets, and the performance measures.

2.1 Motivating Comparison

We reviewed the extractors evaluated in the studies discussed in Section 1.2, in order to characterise the design space for these tools. It appeared that these extractors traded accuracy for robustness.

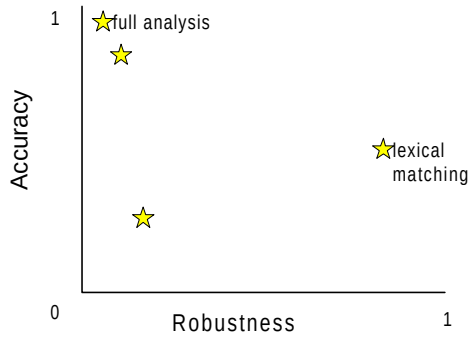

Figure 1: Initial Conceptualisation of Design Space for Extractors

Some extractors used a compiler-based approach and performed a full analysis of the source to produce facts. While these extractors tended to be highly accurate, they could not handle constructs from outside their grammar. Examples of these tools are Acacia [4] and rigiparse [11]. Others used more approximate approaches, such a lexical matching, and these could handle unexpected constructs more easily. SNiFF+ [3] and LSME [12] are examples of this second approach. Their philosophy can be summed up as, "it's not perfect, but something is better than nothing."

We used accuracy and robustness as the two dimensions for evaluation in CppETS, see Figure 1. (The data points have been included for illustrative purposes and do not represent any existing extractors.) Full analysis approaches would be situated in the top left corner of graph, with high accuracy but low robustness. Lexical matching approaches would be situated in the bottom right corner, with low accuracy but high robustness. The ideal extractor would have both high accuracy and highly robustness. As we will see in Section 4 on lessons learned, this characterisation has some shortcomings, both in the dimensions selected and the relationship between them.

2.2 Task Domain Sample

Having selected a motivating comparison, we needed to create a corresponding task domain sample. For a C++ fact extractor, this would be a collection of source code or test cases that were representative of the problems the extractor would have to deal with in actual practice. We began by enumerating mundane and problematic C++ language features, analysis problems, and reverse engineering issues. This list was then used to create a series of test buckets.

The source code for the test buckets came from a variety of sources. Some were specially written for the benchmark. Others were donated by IBM and by Michael Godfrey. Some were taken from books and web sites. These test cases were small, typically less than 100 lines of code, and none more than 1000 lines. We considered using C++ compiler test suites such as the one distributed with GNU g++ [18] and the commercial C++ validation suites products from Perennial [13] and Plum Hall [14]. However, these suites test the minutiae of the C++ language using thousands or tens of thousands of test cases, typically using an automated testing harness. Unfortunately, there are too many test cases with too much detail to include any suite completely in CppETS.

We created two categories of test buckets, Accuracy and Robustness, corresponding to the two dimensions of our motivating comparison. CppETS 1.0 contains 25 test buckets, 14 in the Accuracy category and 11 in the Robustness category. These test buckets and the rationale for them will be discussed in the remainder of this section.

2.2.1 Accuracy Category

Figure 2 lists the groups and test buckets in the Accuracy category. All of the test buckets in this category used only ANSI standard C++ syntax. However, not all of them followed modern (i.e. post-ANSI standard) C++ idiom.

The preprocessor directives present their own class of difficulties, so they were given their own test group(#1-3). The purpose of the Preprocessor group is to determine whether the extractor analyses the source code before or after preprocessing and the correctness of the facts produced. An extractor that analyses the source code before preprocessing often does not extract the correct information about the resulting source code. For example, preprocessor directives may re-define a keyword or macros can be

combined to create source code. An extractor that analyses the source code after preprocessing does not get information about preprocessor directives such as macros.

Preprocessing
1. Macros
2. Conditional Compilation
3. Pragmas

C++ Syntax

Data Structures
4. array
5. enum
6. union
7. struct

8. Variables
9. Functions
10. Templates
11. Operators
12. Exceptions
13. Inheritance
14. Namespaces

Figure 2: Test Buckets in Accuracy Category

The second group (#4-14) is concerned with C++ language features. The purpose of this group is to test identification of language features and resolution of references, mainly calls to functions and uses of variables. These test buckets include many of the potential extractor problems identified by Armstrong and Trudeau [3], such as an implicit call to a function using a pointer, array traversal using indices and pointer arithmetic, multiple variables with the same name, and usage of data structure elements.

Incomplete Information
15. Missing source
16. Missing header
17. Missing library

Dialects
18. GNU g++
19. MS Visual C+
20. IBM VisualAge C++

Heterogeneous Source
21. C and Fortran
22. Embeded SQL

Generated Code
23. lex/yacc
24. GUI Builder
25. Stateflow

Figure 3: Test Buckets in Robustness Category

2.2.2 Robustness Category

Figure 3 lists the test buckets in the Robustness category. These test buckets are intended to represent the kinds of problems encountered in reverse engineering.

The Incomplete Information test buckets (#15-17) are standard C++ source code, but with a file missing. On a reverse engineering project, the client may have neglected to provide a file, or worse, may not be able to provide a file. The test buckets in the Dialects group (#18-20) contain compiler extensions. These tests can be considered to be C++ with extra keywords. These test buckets are representative of those situations where the legacy source code was developed using a compiler that has a slightly different grammar than the extractor.

The Heterogenous Source tests (#21-22) are C++ (or C) together with statements from another source language. Programming languages are often combined to perform special purpose tasks, for example embedded SQL for interfacing with databases and FORTRAN for scientific computing. The non-C++ code is normally handled by another tool, such as a preprocessor for embedded SQL and another compiler for FORTRAN. Unfortunately, appropriate tools for fact extraction are rarely available.

The Generated Code (#23-25) tests contain files that were not C++ at all, but contain descriptions used to generate C++. These descriptions may be grammars, state charts, or resources, and they are the maintenance artifacts, not the generate source code. Consequently, they view the inputs to the code generator as the maintenance artifacts. Often, the appropriate tool is not available to generate the source code or analyse the initial descriptions.

2.3 Performance Measures

Having chosen the test buckets, our next step was to find a method for measuring the performance of the extractors. Taking an arbitrary extractor and examining its output for completeness and correctness is not a simple problem. The facts produced could be stored in memory, in a binary-encoded database, or in a human-readable intermediate format, such as GXL [9]. The output schema of the extractors could also vary significantly, ranging from the abstract syntax tree level to the architectural level [9]. Writing a tool to check the accuracy of facts as specified by a schema can be as difficult as writing an extractor itself.

We handled this challenge by making two simplifying assumptions.

1. The output of the extractors must be stored in a text file that was human-readable. Alternatively, the extractor could be accompanied by a tool that allowed users to query the factbase. This assumption excluded tools that store the facts in memory, such as integrated development environments, from using the benchmark.
2. Operators/users of the extractors would be involved in assessing the output to simplify the problem of comparing output with different schemas and formats.

Using these two assumptions, we devised the following performance measures for the tests in the benchmark. Along with the source code in each test bucket, there was a text file containing questions about the program. The answers to the questions are also provided and it is the responsibility of the

```
#include <stdio.h>
enum days { SUN = 1, MON, TUES, WED, THURS, FRI, SAT };

main()
{
   // enum days day;
   int day;
   char *dayName[SAT + 1] = {"", "Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
"Friday", "Saturday" };

   for (day = SUN; day <= SAT; day++)
     printf("%d%11s\n", day, dayName[day]);

   return 0;
}
```

Figure 4: Source Code from `enum` Test Bucket

person operating the extractor to demonstrate that these answers can be found in the parser output.

The questions covered a variety of topics, including simple recognition and resolution of language constructs and their attributes. For the recognition questions, we asked the extractor operator to show the output for a specified feature, such as a template or exceptions. Sometimes we asked for a comparison of related features, such as a class and a struct. In terms of resolution, we asked questions to determine whether the extractor correctly linked a reference with its declaration or definition. In terms of attributes, we asked for location information in varying combination, file name, start, end, line, character on a line, and byte offset from start of file. The questions covered a wide range of functionality and data models, so we could test a variety of extractors with the same material. Consequently, no single extractor was expected to be able to correctly answer all of the questions.

We used two marking schemes: a quick one that awarded non-numeric grades and a detailed one that gave numerical scores. Both of these marking schemes are explained in detail in the next section, but we give brief descriptions here. There are two reasons for having two marking schemes. One, the quick analysis was used as a "sanity check" to determine whether it would be reasonable to continue to the detailed analysis. These ratings were validated externally against the evaluators' and developers' *a priori* knowledge of the extractors. Two, they give different insights into the results. The quick analysis was concerned with the overall performance of the extractors, while the detailed analysis sought to provide explanations for the performance.

Portions of the enum test bucket will be given here as an illustrative example. Figure 4 is the source code and Figure 5 is an excerpt from the question file. The source code defines a global enumeration type called days, and iterates through it in the main function, printing out the strings from a corresponding dayName array.

Benchmark users were expected to use their extractor on the source code and answer the questions using output from the extractors. This can be done by submitting the extractor output and providing a concordance, e.g. a list of the relevant source lines, nodes, or edges. Alternatively, the user could describe the tools and procedure used to obtain the answer from the factbase. Since the questions in Figure 5 (and all the other test buckets) could be answered by simply inspecting the source code, responding "yes" or repeating the answers given, would not earn full marks.

```
2.  What is the fourth enumeration constant
in enum days?

Answer: WED

3.  What is the (integer) value of the
enumeration constant MON?

Answer: 2
```

Figure 5: Excerpt of questions for `enum` test bucket

3. Application and Results

CppETS 1.0 was distributed to four teams and they were asked to submit their solutions one week in advance of the workshop held at CASCON2001 in November of that year [15]. At the workshop, the teams presented the results and the workshop organisers presented our analysis of their results and the benchmark. In this section, we report on this application of the benchmark and the results of the teams.

The four extractors evaluated were:

- Ccia, AT&T [4]
- cppx, University of Waterloo and Queen's University [2, 5]
- Rigi C++ parser, University of Victoria [11]
- TkSee/SN, University of Ottawa [2]

Table 1 lists the extractors and team members. With the exception of Ccia, the teams consisted of individuals involved in the development of the extractors. Two teams, cppx and TkSEE/SN, used the benchmark as a source of additional tests for their development work. The teams for

Tool	Description	Team Members
Ccia, AT&T	•Part of Acacia tool suite •Built using front end from Edison Design Group •Runs on IRIX, Solaris, SunOS, and LINUX •Emits database according to internal format; intended to be used with rest of Acacia	Mike Godfrey—UofWaterloo faculty Andrew Trevors—UofWaterloo graduate student
cppx, U. of Waterloo and Queen's U.	•Built using GNU g++ as a front end •Runs on same platforms as GCC, binaries available for Solaris and LINUX •Emits TA, GXL, VCG	Ian Bull—UofW graduate student Ian Davis—UofW research associate Andrew Malton—UofW faculty
Rigi C++ parser, U. of Victoria	•Built using Visual Age C++ as front end •Runs on AIX, NT, OS/2; need to purchase VAC++ separately •Emits RSF	Holger Kienle—Uvic graduate student Johannes Martin—UVic graduate student
TkSee/SN, U. of Ottawa	•Built using Cynus Source Navigator as front with additional extraction scripts •Runs on UNIX; web service available at http://kbre9.site.uottawa.ca/parser_online/ •Emits TA++ and GXL	Tim Lethbridge—UofO faculty Sergei Marchenko—UofO research associate

Table 1: Characteristics of Tool Teams

Ccia and the Rigi C++ parser were interested in further exploring the capabilities of those extractors.

The two and a half hour workshop began with the organisers giving a short introduction to the problem and the benchmark [15].. Each team had 25 minutes to give an introduction to their extractor and a qualitative overview of their solutions to give the audience an impression of how well their extractor performed. The organisers then reported on the quick analysis of the solutions and lessons learned from the workshop.

Both the workshop and the evaluation were a success. There was broad acceptance by both the teams and workshop attendees of the benchmarking approach and CppETS. The teams thought the approach was sound and were enthusiastic about using the test buckets. This excitement climaxed with the unveiling of the results from the quick analysis. All in attendance approved of the final standings and agreed that they were consistent with their impressions and expectations, based on the presentations by the teams.

3.1 Quick Analysis

For the quick analysis, the results were scored as follows. The 25 test buckets contained a total of 93 questions. The solutions to each question was given a grade: Full Answer, Partial Answer, and No Answer. A Full Answer was awarded when the question was answered correctly in its entirety. A Partial Answer was awarded when the answer was not totally correct or complete. A No Answer grade was given when the extractor failed to provide any solution. This occurred when the extractor crashed and failed to parse or if it was not designed to output the facts requested.

For the workshop, the solutions were scored generously and when in doubt the higher of two possible grades was awarded. The performance of the extractors were described using stacked bar graphs that showed the number of Full, Partial, and No Answers on each test buckets. By using green, yellow, and red for the different grades on the bars, the graphs provided a Gestalt view of the results. They also afforded easy comparison of the extractors at the level of test buckets. These graphs cannot be reproduced here effectively, so the interested reader is invited to see them on the workshop web site [15].

Results were produced by summing the scores from the various test cases. Aggregating the scores from test buckets that evaluate different language features gives an effective overview of performance. In this analysis, all test buckets were equally weighted. In a special-purpose evaluation, weightings could be assigned to reflect the relative importance for language features for particular downstream task. Finally, we were able to aggregate scores from Accuracy and Robustness test cases because this operation preserved the measurement scale (i.e. absolute) and external validity of the scores.

The final standings (shown in Table 2) were based on the sum of the number of Full and Partial answers. The extractor with the highest score was TkSee/SN with 51. They were followed by the Rigi C++ parser with 45, Ccia with 41, and cppx with 17. It was clear that cppx had a low score because it was still in development and many features were not yet

implemented. The other three extractors had scores that were very close to each other. This finding was consistent with the presentations by the teams during the workshop.

	Full Answer	Partial Answer	No Answer	Full + Partial
Ccia	32	9	52	41
cppx	7	10	76	17
Rigi C++	26	19	48	45
TkSee/SN	33	18	42	51

Table 2: Results of Quick Analysis

3.2 Detailed Analysis

Since the results of the concise analysis were considered to be externally valid, we continued with a detailed analysis. While this analysis is richer, the final standings of the tools were the same.

For detailed analysis, we used a marking scheme that awarded marks for each request for a fact that was satisfied. The questions from the test buckets were re-grouped to correspond more closely to single language features and each question was labelled as direct (i.e. relevant to feature being tested) or indirect (i.e. relevant to other aspects of the parse.) Under this re-division, there were 106 questions and 444 available marks.

For each question, each mark was categorised into one of three classifications: Correct, Incorrect, or Not Available. Marks were awarded in the Not Available category when the extractor did not have that information available in its data model. In other words, the extractor was never intended to answer such a question. The Correct category meant that the mark was earned for a full, correct answer. The remainder were marked as Incorrect.

For each question, a score was calculated as follows: $\frac{\#\,Correct}{\#\,Correct + \#\,Incorrect}$. The Not Available marks were removed, so each extractor was only evaluated on the facts that it was designed to output. The scores were normalised, so each question and then each test bucket was equally weighted. Each test bucket had a score out of one, so the maximum possible Accuracy score was 14 and for Robustness the maximum was 11.

Figure 6 shows a plot of the Correct scores on the Accuracy category vs. the Robustness category. This simple graph shows some very intriguing relationships among the extractors and hints at the trade-offs we had hypothesised. From this graph, we can see that Ccia is the most accurate on the straightforward C++ test buckets and TkSee/SN is best able to handle non-standard source code. The standings from Table 2 are essentially the sum of the Correct scores from both the Accuracy and Robustness categories. TkSee/SN came ahead of Ccia on overall because its Robustness score more than covered the difference in Accuracy.

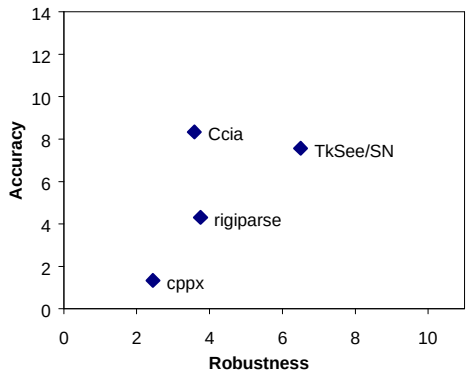

Figure 6: Accuracy vs. Robustness of Four Extractors

4. Lessons Learned

Fundamentally, the CppETS 1.0 was a success. The basic approach of using test cases to evaluate an extractor was good, because it allowed us to query the extractors directly about their functionality. In some cases, the tools were the only reliable authority about their capabilities and the developers were not able to reliably tell us whether their tool could handle a particular feature without a test program. It is noteworthy that we were able to uncover bugs in all of the extractors.

Developing CppETS and validating it in a workshop was very enlightening. The lessons learned from this experience fell into two categories: those that relate to the development a benchmark and those that relate to the problem of fact extraction in general. These lessons will be the focus of this section.

4.1 Benchmarking C++ Extractors

CppETS 1.0 proved to be a good general-purpose benchmark for C++ extractors. There are some minor problems, such as typographical errors and mistakes in the canonical answers, that can be addressed with a 1.1 release, without re-designing the entire benchmarks. While the design decisions we made initially have been borne out through implementation and usage, we still need to consider the alternatives. In this examination, we re-visit the components of a benchmark: a motivating comparison, a domain task sample, and performance measures.

4.1.1 Motivating Comparison

Recall from Section 2.1 that our characterisation situates the extractors along two dimensions, accuracy and robustness(see Figures 1 and 6). The problem with this characterisation is that it would be possible for a poor extractor to be highly accurate and highly robust. For example, an extractor that accepted any input and emitted no

facts would be both highly robust and highly accurate (it output everything correctly according to its empty schema.) A more appropriate characterisation would define robustness as accuracy in the presence of noisy data, as shown in Figure 7.

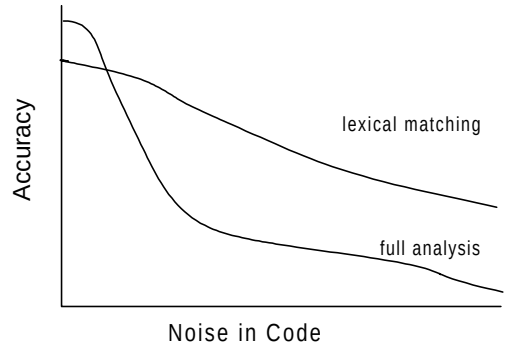

Figure 7: Re-conceptualisation of Robustness

Examples of noise in source code include syntax errors, missing information, compiler extensions, and heterogeneous source languages. The graph postulates that extractors that use full parsing and analysis will be more accurate than extractors that use approximate matching on source code with no noise. But their relative performances will change as the amount of noise increases. Extractors that use full analysis approaches are more sensitive to noise and their performance will degrade faster. In contrast, extractors that use approximate approaches are more resistant to noise, so their performance will degrade more slowly.

Our motivating comparison is tightly focused on the data output by the extractors. There are other requirements and aspects of performance that users consider when selecting an extractor. Performance characteristics such as speed, memory requirements, and size of factbase are factors that become important when working with larger programs. A user may have other requirements beyond the schema of the extractor, such as supported platforms, research and commercial availability, cost, ease of use, interoperability, technical support, and quality of documentation. While these features that are difficult to test with a benchmark, this information can be collected as part of the evaluation process.

4.1.2 Task Domain Sample

The task domain sample for CppETS 1.0 is a series of test buckets with small pieces of source code. In general, the size was appropriate for the approach that we used, but they were both too big and too small for different reasons.

In the prototype, results were sometimes conflated because a single test attempted to measure different things at the same time. This problem would be solved by using smaller test cases that separate the tasks of identifying a language feature, resolving references to it, and recognising its attributes. These three steps have analogues in the compilation domain: compiling, linking, and code generation. These smaller test cases would allow similar tests with different levels of difficulty within a single test bucket.

While the problems posed by the test cases are representative of problems for an extractor, the test cases are far too small to be representative of program comprehension or reverse engineering problems. We had plans for a third category (Accuracy-and-Robustness), but no suitable scoring mechanism could easily be found, so these plans were abandoned. We had hoped to evaluate the accuracy of the extractors on extant programs, such as those used in previous structured demonstrations (xfig 3.2.1 [16] and SORTIE [17]). But with even these moderate sized programs, it is non-trivial to obtain a complete set of correct answers. The answers could be created using painstakingly manual methods or we could use a trusted oracle to tell us the right answers. Unfortunately, no such oracle exists, for if it did, we would not be working to improve the extractors that we have.

4.1.3 Performance Measures

The performance measures in CppETS were simple, but were able to give a rich description of the extractors' capabilities. The question that arises regarding performance is whether we are measuring the right thing.

Recall from Section 1.1 that performance is an indication of fitness for purpose. While the stated purpose of these extractors is "reverse engineering" or "program comprehension," these categories are so broad that it is difficult to formulate clear performance measures. The solution is to narrow the purpose of the evaluation, but at the same time keep it sufficiently general that it is still worthwhile to construct a benchmark. One option is to define the evaluation relative to a downstream application, for example, slicing, architecture recovery, or code migration. Another possibility is to define the benchmark relative to a standard schema, such as the Dagstuhl Middle Model [1] or the forthcoming C++ abstract syntax tree schema [6]. The features examined and the questions posed can be selected from the schema.

A second problem is that the characterisation treats accuracy as a binary characteristic, in other words, can it handle a particular language feature? As Murphy et al. found, extractors do not operate according to their specification all the time. They sometimes mis-analyse a language feature idiosyncratically. One of their many examples is the following:

> Field, when run on mapmaker, missed the call (main;banner), apparently because main is not defined with a return type. Editing the definition of main to add a return type allows Field to extract the call. Not all functions, however, need to be declared with a return

type for Field to extract calls. For example, a small test case of defining a function without a return type did not result in a false negative [12, page 172].

The benchmark should also consider accuracy statistically over a large code base. Information retrieval statistics such as precision and recall could be used.

The final problem with our performance measures is more fundamental. One assumption of the evaluation is that extractors could be evaluated separately from other reverse engineering and program comprehension tools. In practice, it was difficult to isolate the extractor both in terms of usage and designing the evaluation criteria. The quality of the solutions were influenced by i) how experienced the team members were with extractor output and ii) the tools available for querying the factbase. It is not clear how this problem can be addressed. One possibility is to accept it as a shortcoming in the design of the evaluation. Another is to include a downstream analysis tool in the evaluation. This modification would permit the participation of IDEs, but would make it more difficult to verify the submitted solutions.

4.2 Fact Extraction

After placing the four extractors side-by-side and comparing them, interesting lessons emerge regarding their design, implementation, and shortcomings. All of the extractors that participated in the workshop use COTS (commercial off-the-shelf) components as front ends. The extractor with best score, TkSee/SN, used Cynus Source Navigator as a front end plus some home-grown scripts to augment the factbase. Source Navigator is a code browser, so it is intended to be used to extract and display information about source code rather than compile it. The remainder relied on the intermediate representations from a compiler: Ccia used a front end from Edison Design Group; cppx used GNU gcc; and Rigiparse used VisualAge C++. Consequently, the capabilities of these extractors are quite similar.

This transfer of features from compiler intermediate representations would be fine if parsing and analysis for compilation were the same as parsing and analysis for reverse engineering, but they are not. There are many other examples of technologies that do not transfer from one domain from the other. Reverse engineers needed to create their own intermediate formats, for example GXL, TA, and RSF, rather than adopting a compiler intermediate format wholesale, such as SUIF [9, 10]. The cppx extractor transforms GNU g++ internal data suitable for compilation into one more suitable for reverse engineering [5]. As mentioned in Section 2.3, we could not use compiler test suites in their entirety for this benchmark because the they were too fine-grained and focused on language syntax.

Fact extractors for reverse engineering and compiler front ends attempt to solve related problems, but they have different problem domains and thus are intended to meet different requirements. The tendency to use compilers as front ends can be attributed to familiarity: A course on compilers is standard in undergraduate computer science curricula and many researchers in this field have a background in compiler research. As reverse engineering matures, these differences emerge so we need to distinguish ourselves as a field. Consequently, we need to articulate our requirements for an extractor more clearly. Work is under way to create a standard schema for C++ at the AST level [6] and for a middle-level schema [1]. These efforts will advance the state of fact extraction and help us to articulate our contributions to improving software engineering.

5. Future Work

CppETS 1.0 has proven to be a simple and effective benchmark for evaluating C++ fact extractors. It has been used to evaluate four extractors and been discussed in a workshop. The participating team members and interested audience members agreed that CppETS was well-designed and made an important contribution to the problem of implementing C++ fact extractors. It was found to give valid characterisation of their Accuracy and Robustness.

We are encouraged by this success and plan to continue this research. This experience could easily be generalised to other source languages such as C (and the corresponding benchmark called CETS), Java (JETS), and even Cobol (CoETS). Future work for CppETS will take a two-pronged approach: improving the design of the benchmark and consultation with the broader community to gain acceptance.

The three components of a benchmark (a motivating comparison, a task domain sample, performance measures) are best selected and refined through community involvement and debate, not through insular technical work. The benchmarks from TPC and SPEC are widely respected and this stature was attained in part through collaborative work by stakeholders from multiple institutions with different perspectives.

A benchmark needs to be continually criticised and matured. On this issue Tichy wrote,

> Constructing a benchmark is usually intense work, but several laboratories can share the burden. Once defined, a benchmark can be executed repeatedly at moderate cost. In practice, it is necessary to evolve benchmarks to prevent overfitting. [19, p. 36]

A benchmark needs to be maintained and renewed to reflect that state of the art in a discipline. The SPEC CPU 95 benchmark was superseded by SPEC CPU 2000 to better reflect the workloads of modern computer systems and new knowledge about measuring systems performance [8]. A benchmark should be retired when it no longer pushes a field to improve on previous work or when a field reaches a sufficiently high level of performance that further research would bring only minimal returns.

CppETS is a prototype and has already made contributions to improving the development and evaluation of fact extractors. The refinement of CppETS promises further gains in these areas and improved fact extractors. We look forward to the day when C++ fact extraction is a solved problem and CppETS can be retired.

6. Acknowledgments

We would like to thank the participants in the CASCON workshop: Holger Kienle, Johannes Martin, Tim Lethbridge, Sergei Marchenko, Andrew Malton, Ian Bull, Ian Davis, Mike Godfrey, and Andrew Trevors. Marin Litoiu and Paul Smith were supportive CASCON workshop coordinators. Jeff Elliott, Mike Godfrey, and Catherine Morton helped out with the C++ test cases. Special thanks go to Holger Kienle for the insightful comments and the suggesting method used for the quick analysis. This research was supported by NSERC, CSER and IBM.

7. References

[1] "Model for program entity level information," http://scgwiki.iam.unibe.ch:8080/Exchange/2, last accessed 8 January 2002.

[2] "C/C++ Parser with TA++ and GXL Output," http://www.site.uottawa.ca:4333/dmm/, last accessed 8 January 2002.

[3] Matthew N. Armstrong and Chris Trudeau, "Evaluating Architectural Extractors," presented at Working Conference on Reverse Engineering, Honolulu, HI, pp. 30-39, October 12-14, 1998.

[4] Yih-Farn Chen, Emden R. Gansner, and Eleftherios Koutsofios, "A C++ Data Model to Support Reachability Analysis and Dead Code Detection," *IEEE Transactions on Software Engineering*, vol. 24, no. 9, pp. 682-693, 1998.

[5] Thomas R. Dean, Andrew J. Malton, and Ric Holt, "Union Schemas as a Basis for a C++ Extractor," presented at Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, pp. 59-67, 2-5 October 2001.

[6] Rudolf Ferenc, Susan Elliott Sim, Richard C. Holt, Rainer Koschke, and Tibor Gyimóthy, "Towards a Standard Schema for C/C++," presented at Eighth Working Conference on Reverse Engineering, Stuttgart, Germany, pp. 49-58, 2-5 October 2001.

[7] Jim Gray, "The Benchmark Handbook: For Database and Transaction Processing Systems," . San Mateo, CA: Morgan Kaufman Publishers, Inc., 1991.

[8] John L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, no. July, pp. 28-35, 2000.

[9] Richard C. Holt, Andreas Winter, and Andy Schürr, "GXL: Toward a Standard Exchange Format," presented at Seventh Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 162-171, 23-25 November 2000.

[10] Holger Kienle, Jörg Czeranski, and Thomas Eisenbarth, "Exchange Format Bibliography," presented at Workshop on Standard Exchange Format (WoSEF): An ICSE 2000 Workshop, Limerick, Ireland, pp. 2-9, .

[11] Hausi A. Müller and Karl Klashinsky, "Rigi- A System for programming-in-the-large," presented at Tenth International Conference on Software Engineering, Singapore, pp. 80-86, April 11-15, 1988.

[12] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan, "An Empirical Study of Static Call Graph Extractors," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 2, pp. 158-191, 1998.

[13] Perennial Inc. "C++ Validation Suite," http://www.peren.com/pages/cppvs.htm, last accessed 8 January 2002.

[14] Plum Hall Inc. "C and C++ Validation Test Suites," http://www.plumhall.com/suites.html, last accessed 8 January 2002.

[15] Susan Elliott Sim. "C++ Parser-Analysers for Reverse Engineering: Trade-offs and Benchmarks," http://www.cs.utoronto.ca/~simsuz/cascon2001, last accessed 7 January 2002.

[16] Susan Elliott Sim and Margaret-Anne D. Storey, "A Structured Demonstration of Program Comprehension Tools," presented at Seventh Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 184-193, 23-25 November 2000.

[17] Margaret-Anne D. Storey, Susan Elliott Sim, and Ken Wong, "A Collaborative Demonstration of Reverse Engineering Tools," *Applied Computing Reviews*, forthcoming, 2002.

[18] The GCC Team. "GCC Home Page- GNU Project-Free Software Foundation," http://gcc.gnu.org/, last accessed 8 January 2002.

[19] Walter F. Tichy, "Should Computer Scientists Experiment More?," *IEEE Computer*, no. May, pp. 32-40, 1998.