Using Benchmarking to Advance Research: A Challenge to Software Engineering

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

Steve Easterbrook
University of Toronto
sme@cs.utoronto.ca

Richard C. Holt
University of Waterloo
holt@uwaterloo.ca

Abstract

Benchmarks have been used in computer science to compare the performance of computer systems, information retrieval algorithms, databases, and many other technologies. The creation and widespread use of a benchmark within a research area is frequently accompanied by rapid technical progress and community building. These observations have led us to formulate a theory of benchmarking within scientific disciplines. Based on this theory, we challenge software engineering research to become more scientific and cohesive by working as a community to define benchmarks. In support of this challenge, we present a case study of the reverse engineering community, where we have successfully used benchmarks to advance the state of research.

1. Introduction

The purpose of this paper is to issue a challenge to software engineering to start defining benchmarks for the key problems in their area. From our experience developing benchmarks and examining other successful benchmarks, we have observed that benchmarking, when embraced by a community, has a strong positive effect on the scientific maturity of a discipline.

Creating a benchmark requires a community to examine their understanding of the field, come to an agreement on what are the key problems, and encapsulate this knowledge in an evaluation. Using the benchmark results in a more rigorous examination of research contributions, and an overall improvement in the tools and techniques being developed. Throughout the benchmarking process, there is greater communication and collaboration among different researchers leading to a stronger consensus on the community's research goals. The technical progress and increased cohesiveness in the community have been viewed as positive side-effects of benchmarking. Instead, we argue that benchmarking should be used to achieve these positive effects.

We initially observed these benefits while building and using a benchmark for program comprehension tools[18, 19]. This experience led us to examine benchmarking more carefully to understand how it has these community effects. We subsequently applied this theory in the development of a second benchmark for fact extractors for the C++ programming language [17].

The critical insight of our theory of benchmarking is this: Within research communities, benchmarks operationalize scientific paradigms, that is, they are a statement of the discipline's research goals and they emerge through a synergistic process of technical knowledge and social consensus proceeding in tandem. The concept of scientific paradigms was first described by Thomas S. Kuhn in the ground-breaking book, *The Structure of Scientific Revolutions* [12]. A scientific paradigm is the dominant view of a science consisting of explicit technical facts and implicit rules of conduct. A benchmark is a succinct statement of both these aspects of a discipline.

By viewing a benchmark as both a technical and a social artefact, we were better able to understand its role with a scientific community. We have developed some guidelines on how to manage the process, thereby making the benefits more accessible. Benchmarking is widely applicable because the technique can be used with a broad range of technology. Some research is more obviously amenable to benchmarking because the performance measures are straightforward, for example, speed or throughput. In software engineering, the performance measures can be quite complex, because our tools and techniques are intended to help with the creation of large software systems.

Benchmarking can be used by any research community that is sufficiently well-established and has a culture of collaboration. Evidence of the former include an existing collection of diverse research results, and an increasing concern in validation of these results. Evidence of the latter includes multi-site research projects, multi-author publications, and standards for reporting, file formats, and the like. From this base, a consensus-based process, led by a small number of key people, can be used to construct a benchmark that is endorsed by the research community.

In this paper, we will argue that benchmarking has a strong positive effect on research, then we will show how these effects can be realised in software engineering. Our arguments about the power of benchmarking are found in the first two sections. Section 2 gives three examples of successful benchmarks from outside of software

Proceedings of the 25th International Conference on Software Engineering (ICSE'03)
0270-5257/03 $17.00 © 2003 IEEE

engineering. In Section 3, we present our theory of benchmarking. It includes a definition, an account of how it functions within a research community, and an explanation of its effectiveness. The next two sections demonstrate how the theory can be applied. Section 4 presents some practical advice on how to apply the theory and Section 5 illustrates the application of the theory using the fact extractor benchmark. The final section of the paper is a re-statement of the challenge to software engineering, using requirements engineering, model checking, and software evolution as specific examples.

2. Examples of Successful Benchmarks

In this section, we present three benchmarks that were successful in that they advanced a discipline by improving the science and increasing the cohesiveness of the community. Although many publications report on the technical aspects behind the development of various benchmarks, we found relatively few that discuss the social and cultural aspects. The three benchmarks in this section are exceptional in that these non-technical aspects have been discussed in the literature. Nevertheless, we believe that these examples are representative of benchmarking in general, because they are consistent with each other and our own experience.

The three examples are TPC-A™ for database management systems (DBMS), SPEC CPU 2002 for computer systems, and the TREC Ad Hoc Retrieval Task for information retrieval systems.

2.1 TPC-A™

The full name for TPC-A is the Transaction Processing Council (TPC) Benchmark™ A for Online Transaction Processing including a LAN or WAN network[8]. It evolved from the *DebitCredit* test originally published in 1984. This effort was spearheaded by Jim Gray but had so many contributors from industry and academia that the author on the paper was given as "Anon et al." This paper struck a chord in the database community. Researchers began to publish refinements and variants of the test and vendors used the version and interpretation that made their product appear most efficient, further fuelling "benchmarketing" wars. Eventually, a number of representatives from industry and academia formed the TPC for the purpose of standardising and supervising benchmarks for DBMSs. Developing TPC-A™ took over two years and required nearly 1200 person-days of effort contributed by researchers and 35 database vendors who were members of the consortium. The process involved many meetings as well as laboratory work by the members. The TPC-A™ specification is over 40 pages long with 11 different clauses covering issues such as transaction and terminal profiles, scaling rules, response time, and rules for full disclosure.

2.2 SPEC CPU2000

SPEC (Standard Performance Evaluation Corporation) is also a consortium with committees that create a variety of benchmarks. CPU2002 is the one for evaluating computer systems. It supersedes CPU95 and is scheduled to be replaced in 2004. Members of the committees include hardware and software vendors, universities and customers [9]. 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" [9, p. 30].

CPU2000 consists of 26 programs (12 with only integer arithmetic and 14 with floating point) to be compiled and run on a computer system.

2.3 TREC Ad Hoc Retrieval Task

TREC (Text Retrieval Conference) is an annual conference on information retrieval sponsored by NIST, and DARPA. Every TREC has included challenges or problems that are open to all participants. One of the two challenges at the first TREC in 1992 was the Ad Hoc Retrieval Task [22]. This task consists of searching a text corpus for articles that match a specific topic. An example of a topic is "What legal actions have resulted from the destruction of Pan Am Flight 103 over Lockerbie, Scotland, on December 21, 1988?" The corpus used in this task originally contained about 740 000 articles and by 1998 had grown to more than twice this size. It consists primarily of news items, such as those from the *Wall Street Journal* and the *Associated Press*. The formatting of the topics has also evolved since the inception of the task.

Members of the community participate in two ways. When the new set of topics is selected and they help in judging the relevance of the articles. They also run their information retrieval systems using the topics and submit the results to the conference organisers. At the conference, organisers and participants unveil and discuss their results.

The Ad Hoc Task was discontinued after 1998 because performance over the previous few years had levelled off. The benchmark was no longer pushing research forward, so the organisers felt that their energy would be better spent on other benchmarks. (By then there were 9 tasks at TREC.) While the Ad Hoc Retrieval Task was by no means solved, researchers had eight years worth of test collections to use in future research.

3. Theory of Benchmarking

Using examples such as those in the previous section and our own experience developing benchmarks, we have formulated a descriptive theory of how benchmarking functions within a scientific research community. A descriptive theory is an explanatory framework to help us better understand the past. We can then use this understanding to guide our use of benchmarking in the future.

This section has four parts, each explaining an aspect of the theory. We begin by giving the scope of theory by describing where the theory does and does not apply. Based on this, we give a definition of what a benchmark is. From there, we examine how a benchmark emerges and is used within a community and why a benchmark is effective for advancing a scientific community.

3.1 Scope

Our theory is concerned primarily with benchmarks that are *created and used by a technical research community*. Benchmarks that are created by a single individual or laboratory and are not used widely tend not to have the same impact on the community and research results. The community of interest may include participants from academia, industry, and government, but they are all primarily interested in scientific research. Benchmarks designed for business or marketing purposes are optimised for different goals, so this theory does not extend to cover those.

3.2 Definition

We use a general definition of benchmark to ensure wide applicability. It applies equally to benchmarking of tools and techniques, as well as other technologies, so we will use these terms interchangeably. We define a benchmark as a test or set of tests used to compare the performance of alternative tools or techniques. A benchmark has three components. A benchmark can be designed by selecting the components hierarchically. However, when a benchmark emerges as part of a community effort, the components can be developed in any order.

Motivating Comparison. This component encompasses two concepts, comparison and motivation. The purpose of a benchmark is to compare, so the comparison that is at the heart of a benchmark must be clearly defined. The motivation aspect refers to the need for the research area, and in turn the benchmark itself and the work on it. Thus, the Motivating Comparison captures both the technical comparison to be made as well as the research agenda that will be furthered by making this comparison.

Task Sample. The tests in the benchmark should be a representative sample of the tasks that the tool or technique is expected to solve in actual practice. Since it is not possible to include the entire population of the problem domain, a selection of tasks acts as surrogates.

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 technology, but is the relationship between the technology and how it is used. As such, performance is a measure of fitness for purpose.

A proto-benchmark is a set of tests that is missing one of these components. The most common proto-benchmarks lack a performance measure and are sometimes called case studies or exemplars. These are typically used to demonstrate the features and capabilities of a new tool or technique, and occasionally used to compare different technologies in an exploratory manner.

3.3 Benchmarks as Paradigms

Benchmarks function within research communities in the same manner as scientific paradigms. Kuhn argued that scientific progress occurs through revolutions, in much the same way that social and political progress occurs. (The conventional view at the time was that science moved in a linear fashion towards a progressively more accurate truth.) The dominant view in a discipline is called a *paradigm* and a revolution is needed to move a discipline from paradigm to another.

A paradigm is the collection of knowledge that is needed to function within a scientific discipline [12]. Within a scientific discipline, the current paradigm captures the community consensus on which problems are worthy of study, and determines what are scientifically acceptable solutions. In this manner, paradigms convey implicit rules for working within the community along with the explicit rules or factual knowledge.

Kuhn's descriptions of paradigms also includes a lifecycle model for a scientific community. A new research field can be characterised as *pre-scientific* if there is not yet consensus on which problems need solving or on what set of research methods are valid approaches to these problems. When a consensus emerges, working within a paradigm constitutes doing *normal science*. A paradigm can become degenerative if little further progress is made. A scientific revolution occurs when the dominant paradigm is displaced by a new one.

A benchmark operationalises a paradigm; it takes an abstract concept and makes it concrete, so it can serve as a guide for action. The motivating comparison and task sample are a statement of the problems that are worth solving. The performance measures show which solutions are held in higher esteem. The benchmark also contains implicit information about how the problem ought to be solved.

Like paradigms, benchmarks emerge though a process of scientific discovery and consensus. Both must progress together for a standard benchmark to emerge, because

neither alone is sufficient. For example, a discipline may have the research results necessary to design a good benchmark, but lack the agreement that such an effort ought to be undertaken. This tight relationship between knowledge and consensus means that the existence of a benchmark is indicative of the maturity of a scientific discipline.

3.4 Effectiveness

Using paradigms as a model, an explanation of the success of benchmarks needs to account for both sociological and technical aspects. We will discuss the sociological reasons, before considering the technical reasons.

Sociological Factors. As established in the previous subsection, benchmarks encapsulate paradigms. As concrete problem solutions, they make statements about the valid scientific problems and solutions in the area and the *implicit rules for conducting research*. The presence of a benchmark states that the community believes that contributions ought to be evaluated against clearly defined standards. The benchmark itself promotes the conduct of research that is collaborative, open and public.

Collaboration in benchmarking occurs in two ways. During development, researchers work together to build consensus on what should be in the benchmark. During deployment, the results from different technologies are compared, which requires researchers to look at each other's contributions. Consequently, researchers become more aware of one another's work and ties between researchers with similar interests are strengthened.

Evaluations carried out using benchmarks are, by their nature, open and public. The materials are available for general use, and often so is the technology being tested. It is difficult to hide the flaws of a tool or technique, or to aggrandise its strengths when there is transparency in the test procedures. Moreover, anyone could use the benchmark with the same tools or techniques, and attempt to replicate the results.

These factors together, collaboration, openness, and publicness, result in frank, detailed, and technical communication among researchers. This kind of public evaluation contrasts sharply with the descriptions of tools and techniques that are currently found in software engineering conference or journal publications. A well-written paper is expected to show that the work is a novel and worthy contribution to the field, rather than share advice about how to tackle similar practical problems. Benchmarks are one of the few ways that the dirty details of research, such as debugging techniques, design decisions, and mistakes, are forced out into the open and shared between laboratories.

Technical Factors. The technical reasons for the success of benchmarking lie in its strengths as an empirical method. It has characteristics from both case studies and experiments, and consequently shares features of each of these two well-understood empirical methods. as summarised in Table 1.

Table 1: Comparison of Benchmarking as an Empirical Method to Experiments and Case Studies

Characteristics from Experiments	Characteristics from Case Studies
Features • Use of control factors • Replication • Direct comparison of results	**Features** • Little control over the evaluation setting, (e.g. choice of technology and user subjects) • No tests of statistical significance • Some open-ended questions possible
Advantages • Direct comparison of results	**Advantages** • Method is flexible and robust
Disadvantages • Not suitable for building explanatory theories	**Disadvantages** • Limited control reduces generalisability of results

Like experiments, control of the task sample is used to reduce variability in the results—all tools and techniques are evaluated using the same tasks and experimental materials. However, there is little control over the selection of tools or techniques to be evaluated and the individuals operating them in the evaluation; in this sense, benchmarking is more like a case study because these factors are determined by the situation. Furthermore, tests of statistical significance are not always performed on benchmark results.

Another advantage of benchmarking is that replication is built into the method. Since the materials are designed to be used by in different laboratories, people can perform the evaluation on various tools and techniques, repeatedly, if desired. Also, some benchmarks can be automated, so the computer does the work of executing the tests, gathering the data, and producing the performance measures.

Finally, benchmarking is an accepted and familiar evaluation technique in computer science. They do not usually require additional training or special knowledge to use or understand. Using a benchmark is similar to doing a homework assignment, something most researchers have experienced.

4. Applying the Theory

In the previous section, we presented our theory of how benchmarking functions with scientific research communities. In this section, we examine how the theory can be used to guide practice. Given that benchmarks are

indicative of the cohesiveness of a discipline on a technical and sociological level, we hypothesise that benchmarking can be applied proactively to advance the maturity of a scientific community, rather than simply enjoying this maturity as a side-effect. This hypothesis suggests that benchmarking can help whenever a research area needs to become more scientific, needs to codify technical knowledge, or needs to become more cohesive. We will offer evidence that this hypothesis is correct in Section 5.

First, we give some guidance on how to determine whether to begin development of benchmarks in a particular research area. For receptive areas, we describe some principles for the benchmark development and high-level requirements for the end product.

4.1 Preconditions and Caveats

Our theory suggests that there are two conditions that must already exist within a discipline before construction of a benchmark can be fruitfully attempted.

The first precondition is that there must be a minimum level of maturity in the discipline. During the early days, when a research area is becoming established, it is necessary and appropriate to go through a stage where diverse approaches and solutions proliferate. At this time, the bounds of the area are being established and different methods are being applied. This proliferation is desirable, so there will be a variety of tools and techniques to be compared by the benchmark.

Evidence that a community has reached the required level of maturity and is ready to move to a more rigorous scientific basis comes in many forms. Typical symptoms include an increasing concern with validation of research results and with comparison between solutions developed at different laboratories; attempted replication of results; use of proto-benchmarks (or at least attempts to apply solutions to a common set of sample problems), and finally an increasing resistance to accept speculative papers for publication.

This precondition is important because there is a significant cost to developing and maintaining the benchmark, and a danger in committing to a benchmark too early. Walter Tichy writes:

> "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" [21] (p. 36).

A community must be ready to incur the cost of developing the benchmark, and subsequently maintaining it. Continued evolution of the benchmark is necessary to prevent researchers from making changes to optimise the performance of their contributions on a particular set of tests. Too much effort spent on such optimisations indicates stagnation, suggesting the benchmark should be changed or replaced.

Locking into an inappropriate benchmark too early, using provisional results, can hold back later progress. The advantage of having a benchmark is that the community works together in one direction. However, this commitment means closing off other directions, albeit temporarily. Selection of one paradigm, by definition, excludes others.

The second precondition is that there must be an ethos of collaboration within the community. In other words, there must be a willingness to work together to solve common problems. A past history of collaboration demonstrates the presence of a good working relationship and sets up the expectation that community members take part in such work. This ethos, familiarity, and experience creates a community that is more receptive to the results and therefore more likely to use the benchmark.

If these preconditions are not met, it does not mean benchmarking cannot be employed. Rather, it means that steps to establish these preconditions must first be undertaken. The first precondition can be addressed by time and scientific results. In some situations, it is merely sufficient to wait for a critical mass of sentiment and technology before embarking on the road to a benchmark. In other situations, preliminary research on benchmark components may also be necessary.

The second precondition can be addressed through a series of planned activities, but again, patience is necessary because it takes time to change the ethos of a group. Some ways to build up a collaborative ethos include informal meetings and joint projects between groups, time set aside for discussions at conferences, and workshops where new problems and ideas are explored.

4.2 Process of Benchmark Development

The key principle underlying the benchmark development process is this: scientific knowledge and community consensus progress in lock-step during creation of benchmark. The selected benchmark components must be endorsed by the community.

Choosing the task sample will undoubtedly be controversial. Tichy observed:

> "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 " [21] (p. 36).

Creating performance measures will be particularly difficult in software engineering. In most cases, no obvious measures are available prior to the benchmarking effort. Although many people associate benchmarking with test of simple one dimensional attributes (such as speed), in fact the measures can address any aspect of

fitness for purpose, and hence can be quite complex. A good starting point for identifying suitable measures is existing work on empirical studies in software engineering (see for example [2] [11] [15]).

Since knowledge and agreement must be built up together, a successful benchmark development process needs to have the following attributes.

The effort must be led by a small number of champions. This group keeps the work active and the work often becomes identified with one or two key members of the group. They act primarily as organisers, co-ordinating activities such as opportunities to provide feedback and publication of results.

Design decisions for the benchmark need to be supported by laboratory work. The benchmark should use established research results where possible. In addition, prototypes and small tests may be needed to further understanding or to support decision-making.

The benchmark must be developed by consensus. There must be many opportunities for the general community to participate, to provide feedback, and to endorse the benchmark. Pursuant to Tichy's observation, there must be lots of opportunity to debate. These opportunities can come in a variety of formats. They can be newsgroups or mailing lists for informal discussion. A more formal, written Request for Comment (RFC) procedure could also be used. We have found that face-to-face meetings that occur as part of conferences and seminars are essential.

4.3 Desiderata for Successful Benchmarks

In this section, we present seven requirements or properties of successful benchmarks. These can be used as design goals when creating a benchmark or as dimensions for evaluating an existing one. We took work by Gray [8] and Feather et al. [6], and expanded on them, based on our own observations of successful benchmarks. The first two properties, accessibility and affordability, are primarily concerned with the packaging and delivery of the benchmark. The last five, clarity, relevance, solvability, portability, and scalability, deal with the benchmark problem itself. In the remainder of this subsection, we will describe these properties in detail.

Accessibility. The benchmark needs to be easy to obtain and easy to use. The test materials and results need to be publicly available, so anyone can apply the benchmark to a tool or techniques and compare their results with others. The test and results should easy to use by software engineers with different levels of expertise with empirical studies. Also, if the benchmark is easy to understand, it will be less like to be interpreted incorrectly and will have higher credibility.

Affordability. The cost of using the benchmark must be commensurate with the benefits. The costs commonly involve human resources, as well as software and hardware. Some of these costs can be reduced by automating the benchmark. If the costs are too high or the benefits too low, few people will be likely to use the benchmark. The breakeven point for this relationship will vary according to the maturity of the technology and the status of benchmark results among peers and potential users.

Clarity. The benchmark specification should be clear, self-contained, and as short as possible. This clarity should help ensure that there are no loopholes to be exploited.

Relevance. The task set out in the benchmark must be representative of ones that the system is reasonably expected to handle in a natural (meaning not artificial) setting and the performance measure used must be pertinent to the comparisons being made. This property is as difficult to satisfy as it is important. For some task domains, it is sufficient to use a miniaturisation of a naturally-occurring problem. For others, some ingenuity is required to include enough context to make the problem realistic.

Solvability. It should be possible to complete the task domain sample and to produce a good solution. This feature seems obvious, but is worth stating. A task that is too difficult for all or most tools yields little data to support comparisons. A task that is achievable, but not trivial, provides an opportunity for systems to show their capabilities and their shortcomings.

Portability. The benchmark should be specified at a high enough level of abstraction to ensure that it is portable to different tools or techniques and that it does not bias one technology in favour of others. One implication of this feature is the benchmark may need to implemented more than once for different platforms or for different architectures.

Scalability. The benchmark tasks should scale to work with tools or techniques at different levels of maturity. It should be applicable to research prototypes and commercial products. This property influences the size of task: it should be sufficiently large to showcase the more mature techniques, but not too large to test techniques currently being researched.

5. Case Study: Reverse Engineering

Over the last few years, we have been working to define benchmarks for software reverse engineering tools. During this time we applied and refined the principles described in the previous section.

The first benchmark we developed was for comparing program comprehension tools (i.e. tools that helped programmers understand source code for the purpose of making modifications) [18, 19]. Users of the benchmark had to complete a number of maintenance and documentation tasks on *xfig 3.2.1*, an Open Source drawing program for UNIX.

The second benchmark was CppETS (C++ Extractor Test Suite) [17]. We used a collection of small test programs written primarily in C++ and questions about these programs to evaluate the capabilities of different fact extractors.

Both of these benchmarks were developed in collaboration with other researchers, used by additional researchers and tool developers, and discussed at a workshop or conference. Both benchmarks produced technically interesting results, but a more significant contribution was the deeper understanding of the tools and the research problem that they brought to the community. Since these two benchmarks are similar in terms of their development and impact on the research area, we will describe only CppETS in detail.

5.1 Preconditions in the Reverse Engineering Community

Recall that two preconditions are needed prior to working on a community-based benchmark. These are a minimum level of scientific maturity and a culture of collaborative work. Both of these were present in the reverse engineering community before we began our work.

There was a healthy variety of tools implemented by a number of research groups. Many of these tools had similar features, but used different approaches. It was felt that it would be easier to understand the relative merits of the different tools by applying them to a common software system [1] [3].

The community had a track record of collaboration and discussion. The main conference in the area, Working Conference on Reverse Engineering (WCRE), is organised to maximise discussion time. For example, paper presentations are 20 minutes long instead of the usual 30 minutes and there is a half-hour discussion following three paper presentations. Consequently, there is a culture of public debate and exchange of ideas at conferences. Further evidence of this community's desire to work together can be found in GXL (Graph eXchange Language) [10]. This format for exchanging data between software tools was ratified for use by the community in January, 2001 and has been adopted by over 30 researchers in 8 countries.

5.2 CppETS for C++ Fact Extractors

CppETS is a benchmark for comparing the capabilities of C++ fact extractors. Fact extraction is a fundamental problem in reverse engineering, since all subsequent analysis is affected by the quantity and quality of facts produced. It is a surprisingly difficult problem, particularly for C++, which has led to implementations that use different approaches with varying degrees of success [14]. Consequently, it can be very difficult for a user to select a fact extractor that meets their requirements. We created CppETS to address this need. The design and application of CppETS 1.0 is described in detail elsewhere [17], so we will only describe the salient components here.

Motivating Comparison. To find the most accurate and robust (i.e. resistant to non-standard input) fact extractor for C++.

Task Sample. A collection of test buckets each consisting of small C++ test programs and an associated Question File that asked about different facts that could potentially be extracted from the source code. Teams had to perform an extraction on these test buckets and show that the extractor produced answers to the questions. The solution to a test bucket consisted of the output from the extractor and an Answer File that served as documentation and/or concordance to the output.

Performance Measures. Points were earned for correct answers and completeness of documentation. These points were allocated to Accuracy and Robustness categories.

In terms of the desiderata for benchmarks described above, CppETS fared as follows.

Accessibility. CppETS could be downloaded and used by any interested person. The format for the benchmark was straightforward.

Affordability. It typically took about one person working for two to three days to complete the benchmark. Teams often needed more time because they were developing or repairing their extractors at the same time.

Clarity. The problems set out in the benchmark were clear enough to point out inconsistencies or mis-matches in our collective vocabulary. In some cases the vocabulary could be found in other research areas, such as compilers, they were not in wide use. In other cases, the community was learning to apply the same terms to the same phenomenon. This issue will be discussed further in Section 5.4.

Relevance. The benchmark contained test buckets that tested a variety of C++ language features, as well as buckets that tested common reverse engineering problems, such as missing source code.

Solvability. The test buckets came with varying degrees of difficulty and no tool was expected to be able to earn all available points. It was expected that the extractors would be able to parse, if not correctly analyse, all the test programs in the Accuracy category.

Portability. The test programs were portable to a variety of platforms and compilers. Some of the programs in the Robustness categories did contain compiler extensions, i.e. keywords specific to a compiler, and pre-processor directives with implementation-dependent interpretations. These were included to represent extraction problems found reverse engineering.

Scalability. All of the test buckets in the benchmark were relatively small, which made them easy to understand and score by hand. Without some automated

scoring mechanism or an oracle to provide a set of canonical answers, it was difficult to include any test buckets of any significant size. In version 1.1, we included a subset (29 000 lines of code) of the "basesuif" package, which is the core package of Stanford University's SUIF2 compiler system.

5.3 CppETS Development Process

Recall from Section 4.2, that there are three ingredients needed for successful development process. These were champions to lead the process, design decisions that were supported by research, and opportunities for community participation and feedback. All three were present in the development of CppETS.

The benchmark was championed primarily by Susan Elliott Sim. She was joined by Holger Kienle (University of Victoria) for version 1.1. This benchmark built on discussions about C++ fact extraction that were already taking place in the reverse engineering community. Before creating this benchmark, the selection of test buckets was discussed with some researchers who had implemented fact extractors. While developing the benchmark, we used extensive testing in our own laboratories and followed closely two recent papers in the area [4] [7].

To date, we provided two opportunities for the community to participate the development of the benchmark. CppETS 1.0 was discussed at CASCON2001 in November of that year. The following fact extractors participated in this evaluation: Acacia (AT&T, represented by University of Waterloo), cppx (University of Waterloo and Queens), Rigi C++ Extractor (University of Victoria), TkSee/SN (University of Ottawa). CppETS 1.1 was discussed at IWPC (International Workshop on Program Comprehension) in June, 2002. The participants in this workshop were Acacia, cppx, TkSee/SN, and Columbus (University of Szeged and FrontEndART). For both workshops, the benchmark was published in advance and teams submitted their solutions for scoring prior to the workshop. At the workshop, the teams gave presentations on their extractors and how they did on the benchmark. The organisers presented the rankings of the teams and chaired a discussion of the benchmark. At both workshops, the participants and the organisers gained new insights into both their tools and the problem of fact extraction.

The benchmark tasks uncovered errors in all the extractors. Similarly, the teams found mistakes in the benchmark. In both workshops, the fact extractor with the highest score was a surprise to the participants. For the first workshop, the success of TkSee/SN was unexpected because it didn't have a lot of visibility. For the second one, cppx's score was a surprise because it was a significant improvement over the previous workshop.

5.4 Impact of CppETS

According to our theory, a successful benchmark is one that has an impact on the discipline. The full extent of the impact of CppETS will become clear over time, but in the last few months some effects have already appeared.

At the workshops, participants were eager to share their experiences and results. By the end of the day, there was a high level of energy in the room, researchers felt a renewed sense of purpose for their work, and everyone learned a lot. They now have a shared experience of having worked together successfully, which further cements their relationships. These strengthened ties and personal history will make it easier for them to work together in the future. They are looking forward to the next version of CppETS and to co-authoring a paper together.

The benchmark has influenced the development of the fact extractors. CppETS pointed out shortcomings in their tools and raised questions the researchers had not previously considered. The lessons learned from the evaluation will be applied to work on creating a standard schema for C/C++ to be used with GXL.

CppETS has improved both the technical results and the cohesiveness of a community by acting as a vehicle to improve our shared understanding of the problem of fact extraction. The benchmark consists of a series of small extraction problems which made it easy to identify points of disagreement in terminology. Once we were able to establish a common vocabulary for the C++ language features and their analysis, we were able to discuss our conceptualisation of fact extraction more clearly.

An example of this improved understanding relates to identifying the location of a code fragment. Consider the question, "On what line does the definition for function 'average' start?" One possible answer is that it starts on the line containing the function signature, e.g. `int average (int *list)`. Another possible answer is that it starts on the line containing the curly brace that denotes the start of the block containing the function body. An argument can be made for both interpretations, and neither can be said to be more correct until we have some idea how this fact will be used by a subsequent analysis. This problem is further complicated by pre-processor directives and subsequent transformations that occur during the compilation process. These operations may cause a feature in the original source code to move, or disappear entirely.

In order to properly frame an evaluation, we will need to specify a context or downstream analysis for the required facts. In other words, we need to have a task sample that consists of more than just source code. By the same token, fact extractors need to be more clear about what analyses their data models support.

6. The Challenge

We have shown that benchmarking can have a strong positive effect on the scientific maturity of a research community. The benefits of benchmarking include a stronger consensus on the community's research goals, greater collaboration between laboratories, more rigorous examination of research results, and faster technical progress. We believe these benefits can be realised in many areas of software engineering. We conclude the paper by identifying several specific areas of software engineering that we believe are ripe for benchmarking.

6.1 Requirements Engineering

Requirements engineering (RE) has matured as a research community considerably over the past decade, with a series of annual conferences now in their tenth year, and a number of smaller regular workshops. Collaborative links such as the recent European-funded RENOIR network of excellence provide strong evidence of an ethos of collaboration. Hence, the preconditions described in section 4.1 are both met.

In some areas of RE, most notably software specification, proto-benchmarks have been used for years. A recent paper by Feather et al. characterizes these as *exemplars* [6], and identifies the uses, strengths and weaknesses of exemplars for evaluating specification techniques. When used for promoting research comparison and understanding, exemplars have both a motivating comparison and a task sample. However, as Feather et al point out, exemplars lack appropriate evaluation criteria:

> "How can we determine whether the specification language used on one exemplar has appropriate expressiveness, scaleability, evolveability, deductive power, development process efficiency, and so on?"[6, p 423].

This list provides an obvious starting point for defining performance measures. We challenge the RE community to turn some of its exemplars into benchmarks.

6.2 Model Checking

Model checking is an obvious candidate for traditional benchmarking, as speed is an essential property of a model checking algorithm. Several proto-benchmarks have emerged (e.g. the elevator problem used by Plath and Ryan [16]), and nearly all model checking papers include a performance evaluation showing how speed varies, typically with the size of the state space. However, these evaluations have not been standardized in any way.

Moreover, the speed of a particular model checker does not depend in any simple way on the size of the state space. Sreemani and Atlee [20] report that model checkers tend to be sensitive to particular modelling choices, including order of introduction of variables, their types, and the modularity of the system. Reports of the raw performance of model checking engines tends to ignore these factors. Here, a benchmarking effort is clearly needed to develop consensus on appropriate task samples, if we wish to know anything about the relative performance of model checkers in particular applications.

However, there is a less obvious and more interesting application of benchmarking for model checkers that we believe will advance the field more dramatically. One can distinguish between a model checking *engine,* which computes the satisfaction relation for a particular temporal logic, and a model checking *framework*, which includes modelling languages for expressing state machine models, abstraction techniques for reducing large models to a size suitable for fast checking, and tools for translating the models into the input language of the checker engine and for visualizing the results. Recent work on applying model checking to UML models and Statecharts falls into the latter category [13]. Benchmarking here would concentrate less on pure performance, and more on finding out which analysis questions a particular framework can handle. We suspect the impact would be similar to that seen for CppETS.

6.3 Software Evolution

In contrast to the previous two, software evolution is an emerging research community, but they are already pursuing a benchmark. Over the last couple of years, key people in the field have published papers[5] and proposed workshops to draw the community's attention to this problem. This work is motivated by a desire to better understand research results by comparing them using common task samples.

Our advice to this community is to continue pursuing this work, but we feel that it is somewhat early to attain a full-fledged community-based benchmark. We feel that they have not quite attained the first precondition, that is, a sufficient proliferation research results. However, the needed multiplicity of approaches will come with time and so will the consensus and results needed to build a the benchmark. We encourage them to continue with this effort, because it will prepare the community for a benchmark.

The three examples in this section and CppETS illustrate ways that benchmarking can be used in different areas of software engineering. We challenge other areas to start using benchmarking to advance their research. Many branches of software engineering are still in the pre-science stage where there is not yet agreement on a body of knowledge, accepted techniques, or even fundamental problems. Other branches have reached the normal science stage, but are struggling to codify the lasting lessons from research and to have an impact on industrial practice. For areas in both stages, benchmarking will increase the scientific maturity of the area.

A benchmark that has been developed by consensus and is widely accepted provides a beacon for the community. It lights the path for others to follow, shows that there is agreement on what is important, and concentrates attention on a key problem. It is these effects, which emerge out of the benchmarking process, that have a lasting positive impact on a scientific discipline.

Acknowledgements

We would like to thank the following people for their participation in our work with the CppETS benchmark. Mike Godfrey, Jeff Elliott, and Catherine Morton helped with the test cases in version 1.0. Version 1.1 was developed jointly with Holger Kienle. Ian Bull, Rudolf Ferenc, Mike Godfrey, Timothy C. Lethbridge, Andrew Malton, Sergei Marchenko, Volker Riediger, and Andrew Trevors used CppETS and participated in the workshops.

References

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

[2] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach, "The Goal Question Metric Approach," in *Encyclopedia of Software Engineering, Two Volume Set*, Gianluigi Caldiera and Dieter H. Rombach, Eds. New York City: John Wiley and Sons, Inc., pp. 528-532, 1994.

[3] Berndt Bellay and Harald Gall, "An Evaluation of Reverse Engineering Tool Capabilities," *Software Maintenance: Research and Practice*, vol. 10, pp. 305-331, 1998.

[4] Ivan T. Bowman, Michael W. Godfrey, and Ric Holt, "Connecting Architecture Reconstruction Frameworks," *Journal of Information and Software Technology*, vol. 42, no. 2, pp. 93-104, 1999.

[5] Serge Demeyer, Tom Mens, and Michel Wermelinger, "Towards a Software Evolution Benchmark," presented at Proceedings of the International Workshop on Principles of Software Evolution, Vienna, Austria, pp. 172-175, 10-11 September 2002.

[6] Martin S. Feather, Stephen Fickas, Anthony Finkelstein, and Axel van Lamsweerde, "Requirements and Specification Exemplars," *Automated Software Engineering*, vol. 4, pp. 419-438, 1997.

[7] 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.

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

[9] John L. Henning, "SPEC CPU2000: Measuring CPU Performance in the New Millennium," *IEEE Computer*, pp. 28-35, July, 2000.

[10] 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.

[11] Barbara Ann Kitchenham, "Evaluating software engineering methods and tools. Parts 1 to 12.," *ACM SIGSOFT Software Engineering Notes*, vol. 21-23, 1996-1998.

[12] Thomas S. Kuhn, *The Structure of Scientific Revolutions, Third Edition*. Chicago: The University of Chicago Press, 1996.

[13] William E. McUmber and Betty H. C. Cheng, "A Generic Framework for Formalizing UML," presented at Proceeedings of the Twenty-third International Conference on Software Engineering, Toronto, Canada, pp. 433-442, 12-19 May 2001.

[14] 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.

[15] Shari Lawrence Pfleeger, "Experimental Design and Analysis in Software Engineering, Parts 1-6," *ACM SIGSOFT Software Engineering Notes*, vol. 19-20, 1994-1995.

[16] M. Plath and M. Ryan, "SFI: A Feature Integration Tool," in *Advances in Computer Science*, R. Berghammer and Y. Lakhnech, Eds. Heidelberg, Germany: Springer Verlag, pp. 201-216, 1999.

[17] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook, "On Using a Benchmark to Evaluate C++ Extractors," presented at Tenth International Workshop on Program Comprehension, Paris, France, pp. 114-123, .

[18] 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.

[19] Susan Elliott Sim, Margaret-Anne D. Storey, and Andreas Winter, "A Structured Demonstration of Five Program Comprehension Tools: Lessons Learnt," presented at Seventh Working Conference on Reverse Engineering, Brisbane, Queensland, Australia, pp. 210-212, 23-25 November 2000.

[20] Tirumale Sreemani and Joanne M. Atlee, "Feasibility of Model Checking Software Requirements: A Case Study," presented at Proceedings of COMPAS'96, Gaithersburg, Maryland, pp. 77-88, 17-21 June 1996.

[21] Walter F. Tichy, "Should Computer Scientists Experiment More?" *IEEE Computer*, pp. 32-40, May, 1998.

[22] Ellen M. Voorhees and Donna Harman, "Overview of the Eighth Text REtrieval Conference (TREC-8)," presented at Text REtrieval Conference (TREC-8), Gaithersberg, Maryland, pp. 1-24, November 17-19, 2000.