RA: ResearchAssistant for the Computational Sciences

Daniel Ramage
dramage@cs.stanford.edu

Adam J. Oliner
oliner@cs.stanford.edu

Stanford University
Department of Computer Science
Stanford, CA 94305-9025 USA

ABSTRACT

Computational experiments often discard large amounts of valuable data, such as invocation parameters and the lineage of output. Our goal is to identify, manage, capture, and organize this information. These data can be used to make the scientific process simpler and more efficient, and to increase the value of the research by making it more rigorous and reproducible. ResearchAssistant (RA) is an open source Java programming tool that helps to plug this information leak. RA ensures that all console output is valid XML; saves invocation parameters, the random seed, and code version information; automatically checkpoints intermediate results; creates runnable experiment packages; and keeps meticulous notes. This paper presents the design and implementation of RA, and shows how RA easily scales to make complex experiments repeatable.

Categories and Subject Descriptors

K.6.3 [**Management of Computing and Information Systems**]: Software Management—*software development, software process*; D.2.6 [**Software Engineering**]: Programming Environments—*integrated environments, programmer workbench*

General Terms

Experimentation, Documentation, Reproducibility

Keywords

Reproducible research, programming tool, XML, Java

> "A computer lets you make more mistakes faster than any invention in human history—with the possible exceptions of handguns and tequila."
> - Mitch Ratcliffe, Technology Review, April 1992

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ExpCS, 13–14 June 2007, San Diego, CA.
Copyright 2007 ACM 978-1-59593-751-3/07/06 ...$5.00.

1. INTRODUCTION

As the quantity of data available for analysis increases, scientists turn to computers as a promising and natural platform for research. Most software, however, is designed to solve problems, not to conduct scientific experiments; existing software either obscures or discards information that is pertinent to the scientific method. There must be a fundamental shift in the nature of these tools. The key insight behind the ResearchAssistant (RA) toolkit is to provide a suite of abstractions, at the level of the programming language, that gives scientists suitable control over their computational experiments.

Computational scientists seek to understand properties of the physical universe, of data sets, of an algorithm or computation, and sometimes of the computer itself. Code and data serve as the input to a computer, which performs computations on this information to generate output, elucidating the property of interest. While this process can generate useful data, there is also much that it discards; this paper is concerned with identifying, managing, capturing, and organizing this otherwise-lost information.

The flow of information in a computational experiment is illustrated in Figure 1. This information is divided into three classes: input, runtime, and output. The *input* to an experiment includes the invocation parameters, data sets, and random seeds. We also consider the program's code to be an input. The *runtime factors* include circumstances that may affect the output but which are properties of the computer (like the scheduling of threads), or environmental influences (like manual intervention). The last category of information is *output*, which includes messages printed to the console, the display, or files. As with any taxonomy, the distinctions are gray at the intersections; we have found this terminology useful and present it here to facilitate discussion.

A computational experiment typically involves the inadvertent destruction of data. The exact invocation parameters are lost, figures are orphaned from the computations that produced them, output streams are merged irrevocably to the terminal, records of when and how experiments were performed are never written down, random seeds change, and data sets are lost or altered.

There is value in the lost data. Reproducibility is a key characteristic of good scientific research. We distinguish between reproducibility (using the principles from a body of work to solve a problem with a distinct implementation) and repeatability (recreating the exact circumstances and results of an experiment). We assert that repeatable research is

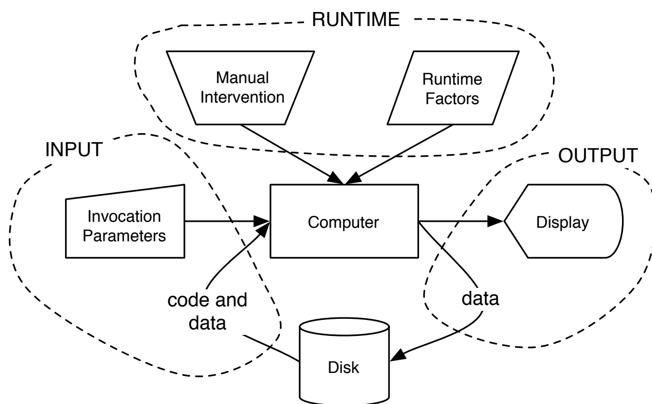

Figure 1: The flow of information in a computational experiment, divided into three classes.

more amenable to being reproduced. To achieve repeatability, we must record the inputs and runtime factors that influence the computation; otherwise the output cannot be recreated. Furthermore, in order to compare experiments, we must isolate what has changed in the input and, consequently, in the output. For example, did a recent adjustment to an algorithm improve the results, or does a subtle change elsewhere deserve credit? Throughout this paper, we focus our discussion on repeatability, with the implication that this is an invaluable resource for reproducibility.

The salient capabilities of RA include automatic bookkeeping, checkpointing, XML output formatting, and portable experiment packaging. It is free and open source [17], and programs written in Java may begin taking advantage of ResearchAssistant by adding a single line of code to their `main` function. In the following ways, RA automatically improves the research process by leveraging existing information that would otherwise be squandered.

- Records experimental runs along with the version controlled source, invocation parameters, and random seeds. Maintain the lineage of output data.
- Automatically checkpoints intermediate states; the experiment may then be run piecemeal to save time and redundant computation. Furthermore, even large data sets may be recorded in a more manageable, though partially processed, state.
- Formats output in a way that reveals when and how it was generated, and that can be readily consumed by existing tools.
- Collects all experimental code, parameters, and data sources into portable, runnable packages. A package is, therefore, a repeatable experiment.

This paper answers the following questions, among others. How can RA identify and capture input and runtime data, and how can it help manage and organize output data? What are the requirements and costs of RA? How can RA make research simpler, more efficient, and more pleasant?

2. BACKGROUND

When data was rare and experiments were arduous to conduct, scientists would meticulously record whatever might be relevant to their work in a lab notebook. Now, experiments can be performed with the push of a button, on large inputs, and with liberal repetition. Scientists turned to computers to solve the problem of data ubiquity that computers themselves had created. Historically, the challenge was framed in terms of *reproducible research*: capture the inputs and runtime factors with sufficient completeness to generate the same output.

The history of reproducible computational research can be traced back at least as early as Donald Knuth's work on literate programming [9], which dictates that programs should be accompanied by literate descriptions and that code examples should work. Knuth was followed by Jon Claerbout and his colleagues at the Stanford Exploration Project, who led the charge for reproducible research in the computational sciences [19]. The *Claerbout Principle* [3] states,

> An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generated the figures.

Inspired in part by Claerbout, Buckheit and Donoho developed Wavelab [1], a free library of Matlab routines for wavelet analysis, wavelet-packet analysis, cosine-packet analysis, and matching pursuit. Wavelab includes tools that facilitate the reproduction of figures from research publications, and a collection of common data sets. The documents, code, and data used to reproduce a set of results may be assembled into what they term a compendium [6].

In principle, computational experiments should be among the easiest to reproduce. In practice, however, code and data sets are rarely released [10]. Skepticism about results and redundant reimplementation are common consequences. Vlad observes [20] that scientific papers are built on a foundation of ethics and trust. The peer review process should encourage runnable experiments. Where possible, the proof that the implementation and measurements are true to the claims should include, "Here, try it."

Reproducible research has begun to gain traction in some computational sciences such as bioinformatics [5], geophysics, acoustics and signal processing, epidemiology [15], and statistics [18]. Sweave is a tool for statistical research that allows R code to be embedded directly into a document, where the analysis may be rerun in place [12]. WEKA, a toolkit for research in artificial intelligence [4, 7], has tools for saving experiments and parameters, but these remain confined to the WEKA infrastructure. The same is true for YALE [13], a data mining package, and for GATE [2], a natural language processing environment. Indeed, most existing tools for reproducible research are tied to special-purpose frameworks (like WEKA or GATE) or to special-purpose languages (like R or Matlab). ResearchAssistant, on the other hand, is built for Java, a widely used and general-purpose programming language. As described in Section 4, it is only thanks to features of Java like run-time reflection on metadata and serialization that many capabilities of RA are possible.

The challenge of capturing input, runtime, and output data for the scientific process is distinct from replay debugging [11, 14, 21], where a tool is used to isolate misbehavior in a computation. To be valuable, a solution for the research environment must be "always on," rather than imposing a

cost that prohibits daily use.

Despite the plethora of related work, the computational sciences remain dominated by research that cannot be easily verified, tweaked, or extended. This behavior is deeply rooted in the culture of computer and computational scientists. Buckheit and Donoho, the authors of Wavelab, call the current state of affairs "a scandal".

The information that we wish to salvage from computational experiments is not new, and we have discussed numerous existing approaches. They each fall short in some way: the inability to access certain information, implementation at the wrong level of abstraction, onerous burdens on the programmer, or limited applicability. The central technical contribution of ResearchAssistant is to make saving experimental data so simple and pervasive that it qualitatively changes the process of computational science.

3. PROBLEM DESCRIPTION

If we hope to foster rigorous scientific methodology in the computational sciences, we must provide tools that enable research to be conducted efficiently, repeatably, and fastidiously. Therefore, in this section, we characterize the challenges involved with preserving information about computational experiments. Then, with full understanding that there are social challenges in this domain as well, we discuss the design criteria we found valuable in engineering a technical solution.

3.1 Technical Challenges

We now describe specific technical challenges associated with capturing input, runtime factors, and output. Following these descriptions, we present possible solutions and note some caveats and limitations. The implementation of ResearchAssistant is discussed in Section 4.

3.1.1 *Input*

The input to an experiment is roughly those data which provide the following information: what computation to perform, what data to perform it on, and how to perform it. The program code is the computation and the data sets are the targets. Various input data influence how this proceeds, including invocation parameters and random seeds.

Data sets can be problematic, especially when the objective is to share them. Kovačević [10] notes that proprietary data sets may represent a significant obstacle, and suggests arranging for portions of those data sets to be released. Sometimes, releasing even portions of data or code is not an option; it may contain private or sensitive information, or be too large for practical distribution. More commonly, it would be time-consuming to prepare the experiments for release and it is usually not a requirement for publication.

Research code—particularly complex systems with more than one author—is perpetually in flux and has a tendency to degrade over time. Short-term changes driven by a deadline tend to become permanent, often at the expense of other code branches. Regression in performance at an affected task can go months before being noticed. It is crucial that the code used for a particular experiment be recognized as an input and be recorded along with any parameters.

Solutions. Many input data could be saved with more rigorous automatic *bookkeeping*, such as in a log or workbook. This means recording information regarding when an experiment was run, with what parameters, and the random seed. For code, there are already ubiquitous tools for bookkeeping, called version control repositories. By interacting with such repositories, one can record what version number of the code generated a result, and even save the code "diff" (record of incremental code changes). A log message can be annotated with the function that printed it, or with timing information for later aggregate-scale performance analysis.

A *checkpoint* is a complete description of the state of an experiment at some intermediate stage; it is the input to the remainder of the computation. From this perspective, we find a partial solution to dealing with large or proprietary data sets. If the computation distills the initial data into some acceptable format, the checkpoint can stand in place of the data. With a checkpoint, the experiment may be restarted after skipping some initial stages.

Together, bookkeeping and checkpointing capture both the input at the start of the experiment, and the input to certain suffixes of the steps in the experiment.

Caveats. Despite attempts to record and deterministically replay sources of randomness, it is not always possible to do so faithfully. One such case is when private instances of pseudo-random number generators (RNGs) are used, instead of the ones watched by the tool. A user might, for instance, call a procedure that uses a RNG seeded with the system clock. It can be difficult to detect this covert input channel and the experiment may yield nondeterministic results. Instead of a limitation, we see this as an opportunity for scientists to identify the existence of, and to later isolate, changing variables in their experimental setup. If the user believes that a tool is recording all relevant inputs yet still observes nondeterministic output, it implies there are significant external resources that remain unrecognized.

3.1.2 *Runtime Factors*

Runtime factors are data associated with properties of the computer, including thread scheduling algorithms, hardware errors, and the duration of the computation. We also include some environmental influences, like manual intervention. When the computer itself is not the artifact under study, most runtime factors should not influence the experiment and therefore need not be recorded.

Solutions. While much of this information is inherently difficult or impossible to capture automatically, some of it takes almost no time nor space to record and is frequently useful. For example, RA records invocation and exit times.

3.1.3 *Output*

Output is the data of interest produced by the experiment. This is generally written to the terminal, to a graphical display, or to one or more files. The central challenge in this domain is to generate the output while maintaining its lineage; a result should not be orphaned from the computations that produced it. Consider the following examples of information being lost as it is stored.

By default, messages printed to different output streams (e.g. `stdout` and `stderr`) are intermingled without identifying delineations. They may be directed to separate files, but then information on relative line ordering is lost. Once figures and files have been created, there is often nothing tying them back to the experiment, like the code version or invocation parameters. Multiple sites in the code may write messages to the output that have a common format; without additional annotation, the origination site remains un-

known. Furthermore, output data frequently presents users with a choice: print everything and sift through it, or print less and rerun if the data of interest changes.

Solutions. A general-purpose way to annotate structured output data is to use *XML formatting*. XML serves as a generic intermediate format. If the consumer of the data changes, the code requires no alterations; instead, a different parsing rule, easily defined for XML, could be constructed. For example, a popular tool like `xmlstarlet`[1] can transform XML into a LaTeX table for inclusion directly into a document. This format also affords flexibility in what data is presented; a verbose module could be suppressed according to its XML tag, or a single type of data (like thrown exceptions) could be viewed in isolation.

An *experiment package* consists of all the inputs and outputs of a particular run. This runnable package will deterministically yield the same results, unless some relevant inputs are unaccounted for. This is distinct from a compendium [6] in that we require the package to be directly runnable; it not only includes all the pieces of an experiment, but understands how to put them together.

Experiment packages hold promise in at least three situations: (1) ensuring repeatability, (2) improving peer review, and (3) assisting remote collaboration. Rather than discussing a plot or a number, the experiment package becomes the artifact of interest: this input produces that output. When doing peer review, it is possible to actually try out the experimental claims in the paper, rather than relying solely on trust in the authors and sufficient detail in the document. When collaborating remotely, experiment packages may be passed back and forth, in place of the tedious enumeration of parameters and other inputs.

Caveats. These solutions add overhead to statements that print information to an output stream. Observe that only codes that print with conspicuous verbosity will suffer a meaningful performance penalty. In general, programmers already understand that printing is expensive and do it only when necessary; an incremental cost is negligible.

It is occasionally impractical or impossible to package an experiment for deterministic replay. Data sets may be proprietary or intractably large, the experiment may be intentionally sensitive to computer hardware or other environmental factors, and so on. Under these circumstances, the research may be unreproducible. At least, a tool could still provide bookkeeping, checkpointing, and structured output formatting.

3.2 Design Criteria

For a solution to be widely adopted, it must also meet a number of social challenges. We enumerate some of the design criteria that we consider to be crucial for such a tool.

- **Accessibility.** A valuable tool is easy to find, simple to learn, and worthwhile to master.
- **Part of the Ecosystem.** Researchers use many tools to get their work done. A valuable tool is a good citizen in that community. Formatting output in XML is one such behavior, as is compatibility with pervasive tools like version control repositories.
- **Repeatability.** A valuable tool encourages good scientific practices, such as making experiments repeatable. Experiment packages and checkpoints are steps in this direction.
- **Improved Research Process.** A valuable tool leads not only to higher quality research, but to a scientific process that is more efficient and enjoyable.
- **Regression Testing.** Code and data evolve; regression testing is the standard practice for managing this change. A valuable tool complements this process.

[1] http://xmlstar.sourceforge.net/

4. RESEARCHASSISTANT

ResearchAssistant (RA) is a toolkit that provides programming language-level abstractions for performing computational research. It is freely available, open source [17] software consisting of several thousand lines of Java code. As suggested in Section 3, RA implements bookkeeping, checkpointing, XML formatting, and experiment packaging. Although the principles behind these abstractions are widely applicable, RA leverages several Java-specific features like reflection on metadata and annotations; an analogous tool could be developed for other sufficiently rich dynamic languages such as C# or Python.

This section examines the design criteria and features of RA incrementally from the perspective of a simple example: pre-processing web pages, based on the words they contain, for a clustering system. We assume some familiarity with the Java language and runtime.

Text clustering is a well-studied task in natural language processing and information retrieval, where many successful systems start by considering only the counts of word tokens on each page. We use *token* to mean the linguistic base or *stem* of a character sequence containing no symbols or numbers; e.g. the input “followed” becomes the token “follow” (by stemming as per Porter’s algorithm [16]), whereas “20cc” is ignored because it contains numbers.

A useful component in a clustering application is code that can process an HTML page into tokens and can count them. The Java program `WebCounts`, shown in Figure 2, serves that purpose. It makes an `http` request for the contents of a web page, extracts tokens from the HTML, and prints them to `stdout` along with the number of times that each token occurred on the page.

In the following subsections, we present successive versions of `WebCounts`, progressively adopting more of the RA framework. Each subsection describes another kind of information that we wish to save using RA, explains the steps necessary to do so, and characterizes the impact for the user. Where appropriate, we include implementation details. We found this to be the most natural way to explain the salient contributions of RA. The sections are ordered to present the framework incrementally, but it is not necessary to perform every step listed here to take advantage of RA. Indeed, a single line of runnable code is sufficient to gain several benefits. We conclude the section with a demonstration of `WebCounts` being incorporated into a larger research project.

4.1 One-Line Migration

By itself, `WebCounts` fails to capture many of its inputs including command line arguments, time of invocation, source code, and other information about its environment. Much of this information could be collected manually, but doing so without the proper tools is burdensome. For instance, users

```
1  public class WebCounts {
2   public static void main(String argv[]) {
3    String text = textFromURL(argv[0]);
4    List<String> tokens = tokensInText(text);
5    Map<String,Integer> counts
6                        = tokenCounts(tokens);
7
8    for (String token : counts.keySet()) {
9      System.out.println(counts.get(token)
10                            + "␣" + token);
11   }
12  }
13 }
```

Figure 2: The `WebCounts` example program downloads the contents of a URL (given in the first command line argument), outputting counts of word tokens contained in its text. This code will be successively modified through Section 4. Methods not shown have obvious behavior based on their names.

```
Grabbing text from http://www.nytimes.com/
39 a
7 about
1 acquit
...
1 young
4 your
1 zone
```

Figure 3: Raw console output from the program in Figure 2. The first output line is sent to `stderr` by `textFromURL`. The remaining lines are counts of stemmed word tokens on the page. Several hundred similar lines were elided at the ellipsis.

have a choice between saving the output from a pair of output streams (`stdout` and `stderr`) to one file or to two. If stored together, the single output file potentially loses consistency of formatting. If stored separately, the two files lose relative ordering information. For example, Figure 3 shows an excerpt of the output from `WebCounts`, where we log both `stdout` and `stderr`. Unfortunately, the program calls a "noisy" subroutine `textFromURL` that prints a status message to `stderr`, breaking the simple tab-delimited format used consistently on `stdout`.

Many of the record-keeping pitfalls enumerated above are avoided by a method call to `RA.begin(argv)` as the first line of a program, e.g. immediately before line 3 in Figure 2. Here `RA` is the name of the Java class we provide, `begin` is the method that causes RA to install itself into the running Java environment, and `argv` is the array of strings of provided command line arguments.

Valid XML output. RA prints output in XML, so simple parsers can extract, reformat, and pass data to other tools. The new output of the program, shown in Figure 4, is guaranteed to be well-formed XML. RA provides a simple, stateful streaming XML generation API. RA intercepts the Java built-in `System.out` stream (corresponding to `stdout`) and replaces it with a version that uses the XML API to output each line in a `<stdout>` tag. An analogous stream replaces `System.err`, writing its lines to `stdout` wrapped in a `<stderr>` tag. Similarly, all lines read on `stdin` can optionally be echoed to `stdout` wrapped in a `<stdin>` tag. RA ensures that the generated XML is valid by escaping control characters and by installing a virtual machine shutdown hook that closes any open XML tags upon termination of the program.

Basic environment logging. RA records basic information about the Java runtime environment at the start of the XML in an `<environment>` tag. This information includes the class whose main method was invoked, the invocation time, command line arguments, and the default Java environmental variables (called `properties`). RA registers interrupt handlers and exit handlers with the virtual machine so that, on exit, RA inserts an `<exit>` tag for recording the total running time and the reason for termination. The reason might be an exit code with the source code line number that called `System.exit()` (when available), the stack trace of an uncaught exception, or a POSIX-style exit signal such as `SIGINT`.

Runtime tracing. In addition to converting print statements to XML tags, RA uses Java's ability to inspect the running stack trace to record the source code line number of the print statement. This type of tracing can be extremely useful when analyzing logs of experimental systems for which interactive debugging is infeasible. Each print statement pays a slight performance penalty for its XML-redirection and call-stack tracing. Few research applications, however, are performance-bound by writing their output. Hence, most research applications that print through RA will see overall runtime performance slowed negligibly.

4.2 RA from the Command Line

The improved recording of the outputs and runtime factors introduced in the previous section is important, but the software still lacks a coherent record of all its inputs for different invocations over time. By providing command line flags to the program, RA can be instructed to save this data, as well.

No additional code is required beyond the call to `RA.begin` as described previously. At initialization, RA parses the command line arguments and takes appropriate action before returning control to the program's `main` method. RA provides helpful usage error messages if the command line is ill-formed, and suggests sensible defaults for optional parameters.

Infrastructure. ResearchAssistant takes meticulous notes in a *workbook*, as well as notes in the output itself. The workbook is a directory on disk for recording and archiving three types of inputs: resources (external files the program may read or manipulate), invocation logs (consistently named records of experimental runs), and checkpoints (saved partial computations). RA accepts the workbook path as a command line argument, defaulting to the current directory, and provides an API for accessing its resources. RA wraps all input and output streams created through the `Workbook` API, with a version that transparently computes the MD5 hash of all bytes read/written. The hash and file name are logged to the XML when the stream is closed or at program termination.

Bookkeeping and repeatability. In addition to logging resource accesses made through the workbook, RA can

```
<invoke class="example.WebCountsOneLineMigration"
        starttime="Wed␣Feb␣14␣16:26:20␣PST␣2007"
        seed="8745931257572013">
  <environment>
    <workbook root="/data/workbook" />
    <arguments>
      <arg>http://www.nytimes.com/</arg>
    </arguments>
    <properties>
      ...
      <entry key="java.runtime.name">Java(TM) SE Runtime Environment</entry>
      <entry key="java.runtime.version">1.6.0-b105</entry>
      ...
      <entry key="os.arch">amd64</entry>
      <entry key="os.name">Linux</entry>
      <entry key="os.version">2.6.17-10-generic</entry>
      ...
    </properties>
  </environment>
  <trace stream="stderr">example.WebUtils.textFromURL(WebUtils.java:111)</trace>
  <stderr>Grabbing text from http://www.nytimes.com/</stderr>
  <ReadResource name="http://www.nytimes.com/" type="EXTERNAL_URL"
                hash="453d1e99248cc5fec2b9cdf2a8a5daed" />
  <trace stream="stdout">example.WebCountsOneLineMigration.main(WebCountsOneLineMigration
.java:23)</trace>
  <stdout>39 a</stdout>
  <stdout>7 about</stdout>
  ...
  <stdout>1 zone</stdout>
  <exit code="0" endtime="Wed␣Feb␣14␣16:26:23␣PST␣2007" runtime="P0Y0M0DT0H0M3.421S" />
</invoke>
```

Figure 4: Output from the same program that generated Figure 3 but prefixed with a call to `RA.begin(argv)`. RA automatically records the invocation and exit times, duration, environmental parameters, invocation arguments, workbook location, random seed, and exit conditions. Output is well-formed XML and the original command line invocation can be reconstructed from the `<arguments>` sub-tree.

be directed to create runnable snapshots of the invoked code. The `--ra-jar` command line option directs RA to write a new Java Archive (JAR) into the workbook's invocation folder. The JAR is a nearly complete snapshot of the state of the experiment's invocation. RA writes into the JAR a copy of all classes currently visible in the Java classpath (thereby recording the executable state), a verbose version of the XML log written to `stdout`, and all command line arguments. If provided, the `--ra-source` switch directs RA to additionally snapshot a copy of the source folder into the JAR; if the source directory is managed by Subversion version control, RA will save the status of each file with respect to the repository, as well as any diffs and new files needed to fully reconstruct the source tree of the generating program. The JAR can be distributed and re-run on any system with a Java Runtime Environment, thereby enabling repeatability. When run, the experiment JAR can print the saved XML output; it can re-run the code again with different parameters; or it can re-generate the source tree. Future versions of RA will include more advanced functionality in JAR construction and re-running, such as optionally saving copies of input resources at construction time or presenting graphical diffs of XML output between stored runs.

When an experiment is first run, RA's initialization time is proportional to the amount of data that must be logged into the JAR. For instance, it will take longer to save the source than to save just the log. In our experience, these local operations complete in at most a few seconds even on large source trees containing thousands of files.

4.3 Repeatable Randomness

There are many technical complications to making an invocation identically repeatable. In the previous levels, we showed how RA can track explicit program arguments, source code versions, and properties of the runtime environment. These inputs do not guarantee repeatability, however. One major remaining factor is intentional randomness through pseudo-random number generators (RNGs). RNGs are common in many research applications as a source of apparently random numbers, but are actually generated deterministically from a starting *seed value*, which is often a number computed from the current system time. RA can record and modify the seed value, so a later run can access the same pseudo-random numbers in the same order.

Note that RA cannot control other potential sources of non-determinism, such as external resources and inherent stochasticity in the Java API. For example, non-determinism resulting from ordering objects as a function of their address in memory (as the built-in `HashSet` does) is beyond RA's control.

The call to `RA.begin` instantiates a global pseudo-random number generator whose seed can be provided on the command line (`--ra-seed`) or can default to a function of the current system time. Conventionally, Java programmers access random numbers through one of two methods: calling the standard system class's `Math.random` method (returning a double-precision floating point number) or by creating a new instance of the built-in `java.util.Random` RNG directly. To access RA's repeatable RNG, the programmer need only change code references from `Math.random` to `RA.random` and from `new java.util.Random()` to `RA.newRandom()`.

All explicit intentional uses of randomness now derive from a safe, repeatable seed. If the program successfully avoids other sources of non-determinism inherent to Java, then running the same code with the same input, and specifying the same seed on the command line, should result in identical output.

4.4 Custom XML

In the previous sections, we considered how RA records the inputs to an experiment, but have not yet improved the process of generating outputs. A particular challenge stems from recording multiple types of output. The programmer is faced with the choice of writing to multiple files or using a suitable mechanism of tagging lines on `stdout` with their data type. For instance, our `WebCounts` example prints counts of tokens, but could just as well also output the list of all original input character sequences that mapped to a particular token via stemming.

XML is a flexible and robust language for data management and RA uses it to record experimental parameters and output streams. RA allows programmers to explicitly create custom XML to be interspersed into the XML output stream. In practice, this is as easy as printing to any other output stream.

Printing a line of text to `stdout` is accomplished in Java with a call to `System.out.println(lineOfText)`. After calling `RA.begin`, these lines are mapped transparently to a new call, `RA.stream.line("stdout",lineOfText)`, which outputs `lineOfText` wrapped in a `<stdout>` XML tag to the XML stream. Invalid characters and XML control characters are automatically escaped (RA provides a mechanism to directly output pre-formed XML). If `lineOfText` is a multi-line string, it is re-indented to flow naturally in the generated output. Tags with multiple children (such as `<arguments>` in Figure 4) can be opened with `RA.stream.begin(tag)` and closed with `RA.stream.end(tag)`. XML attributes to be stored with the tag may optionally be provided as arguments to `RA.stream.line` and `RA.stream.begin`.

Figure 5 shows further modifications to the `WebCounts` codebase; calls to print methods on `System.out` have been replaced with calls to XML methods on `RA.stream`. Figure 6 gives the output of this code.

Part of the ecosystem. The change from `<stdout>` to `<token>` may seem trivial, but the implications are not. A particular run of an experiment may actually generate many different types of outputs: matrices of correlations, histograms of counts, status messages, tables of values, etc. These can co-exist in the same XML output, and each post-processing tool can be given data only for tags it understands. As an example, we wrote a short shell script that uses an XPath[2] query to select `<token>` lines, which are reformatted and sent to the GNU plotting tool `gnuplot`[3] for display. The same script will work on the output from any other experiment using the same tag conventions, regardless of whether other data is intermixed.

Fine-grained runtime-logging. RA can log output times and generate source code line numbers for any printed tag, not just lines sent to `stdout` and `stderr`.

4.5 Stage-Wise Experiments

In the previous section, we explored how ResearchAssistant can improve bookkeeping and repeatability with min-

[2]http://www.w3.org/TR/xpath
[3]http://www.gnuplot.info/

```java
public class WebCountsCustomXML {
  public static void main(String argv[]) {
    RA.begin(argv);

    String text = textFromURL(argv[0]);
    List<String> tokens = tokensInText(text);
    Map<String,Integer> counts = tokenCounts(tokens);

    RA.stream.begin("tokens", "url", argv[0]);
    for (String token : counts.keySet()) {
      RA.stream.line("token", token, "count", counts.get(token));
    }
    RA.stream.end("tokens");
  }
}
```

Figure 5: The `WebCounts` example, expanded to use custom XML tags by defining new streams. The output of the `for` loop will be enclosed in `<tokens>` tags, and each token will be tagged as `<token>`. These tag names may also be dynamically calculated at runtime.

```xml
<invoke class="example.WebCountsCustomXML"
        starttime="Wed␣Feb␣14␣16:26:23␣PST␣2007"
        seed="8745934849079013">
  <environment>...</environment>
  <trace stream="stderr">example.WebUtils.textFromURL(WebUtils.java:111)</trace>
  <stderr>Grabbing text from http://www.nytimes.com/</stderr>
  <ReadResource name="http://www.nytimes.com/" type="EXTERNAL_URL"
               hash="c1c33b514e5ccb854d92f65cfd1bf68a" />
  <tokens url="http://www.nytimes.com/">
    <token count="40">a</token>
    ...
    <token count="1">zone</token>
  </tokens>
  <exit code="0" endtime="Wed␣Feb␣14␣16:26:27␣PST␣2007" runtime="P0Y0M0DT0H0M3.802S" />
</invoke>
```

Figure 6: The output of `WebCounts` with user-defined streams and tags: specifically, the version in Figure 5.

imal modifications. Now, we present how a novel aspect of RA's architecture can have a positive impact on research code quality and its ease of development.

A computational experiment often consists of a series of stages of data processing tasks, where the outputs from one stage are used as inputs to another. Generally, the program's `main` method parses command line arguments, loads auxiliary inputs, and orchestrates the whole experiment in one monolithic batch. Correctly tracking the inputs to and outputs from each stage is difficult, so there is a temptation to take shortcuts that produce shoddy code.

RA provides a declarative mechanism to succinctly and robustly identify the inputs, outputs, arguments, and prerequisites for each stage of an experiment by use of a Java language feature called *annotations* introduced in Java 5. An annotation is a syntactic marker for associating arbitrary meta-data on almost any aspect of a Java program, including classes and fields. RA provides annotations for marking source code with its role in a stage-wise experiment as well as utility classes that read those annotations. Using RA for managing a complex experiment is easier than taking the shortcuts.

The first step in creating a stage-wise experiment is to code the set of stages and explicitly annotate their interdependencies. Stages of computation are defined for RA as classes that implement the `Stage` interface, which defines only one method, `run` (accepting no arguments). RA defines an annotation, `@Stage.Requires`, that can be used to record which other `Stage` classes are pre-requisites for a given stage. Other annotations are provided for marking class fields as outputs whose values RA will record, inputs whose values RA will provide before running, and arguments whose values RA will supply from command line arguments.

RA includes a helper class named `StageWise` for managing a set of `Stage` instances throughout their life-cycle. At its core, `StageWise` is a dependency-graph scheduling algorithm that is similar to compilation tools like Ant[4] and `make`[5]. When instructed to run a particular stage class, `StageWise` instantiates all of its pre-requisites recursively, running them in topological (depth-first) order. Before each stage is run, RA ensures that all of its prerequisites are complete and appropriate values are set for all fields that are marked as

[4] http://ant.apache.org/
[5] http://www.gnu.org/software/make/

Algorithm 4.1: RunStage(S)

```
global ARGS, OBJS
for each P ∈ GetPrerequisites(S)
  do RunStage(P)
I ← Instantiate(S)
for each F ∈ GetArgumentFields(S)
  do I[F] ← ARGS[GetName(F)]
for each F ∈ GetImportFields(S)
  do I[F] ← OBJS[GetName(F)]
Run(I)
for each F ∈ GetExportFields(S)
  do OBJS[GetName(F)] ← I[F]
```

Figure 7: Pseudocode for running a stage and its prerequisites, for the case with no loops in the dependency graph. S is a class implementing Stage. ARGS and OBJS are dictionaries mapping global field names to values. In RA, `GetPrequisites` is implemented by reading the `@Stage.Requires` annotation and the `Get*Fields` methods are implemented by reading the appropriate field annotations.

inputs or arguments. `StageWise` then calls `run()` to allow the stage to execute before finally caching the values of all the stage's output fields for possible use as inputs to later stages. Simplified psuedocode for this algorithm is provided in Figure 7. Loops in the stage dependency graph trigger an exception when encountered by RA.

Returning to our `WebCounts` example, consider the code in Figure 8, which is equivalent in function to the example from Section 4.4 but re-written as a series of Stages. The previous code body of main is exactly reproduced by the bodies of the two `run` methods in the refactored example. At runtime, the call to `StageWise` in line 4 instructs RA to build and run a stage dependency graph for the target stage, `Count`. In doing so, RA will discover that `Load` must be run first because it is marked by `Count`'s `@Stage.Requires` annotation on line 24. RA then executes both stages.

Data is communicated between stages through *experiment fields*, which are named fields that RA preserves between stages. Notice that the `String text` field is marked with the `@Stage.ExportField` annotation in line 16. This annotation tells RA that after the `Load` stage completes, the value of `text` should be saved into the experiment's context under the name `"WebCounts:text"`. Similarly, the `Counts` stage marks one of its fields (line 27) with the `@Stage.ImportField` annotation, which informs RA that it should populate that field from the experiment context before calling `run()`. Before the experiment begins, RA walks the dependency graph to ensure that all experiment fields are created by one stage before being read by another.

Notice also that in line 11 we have marked the custom `@Argument` annotation on the `url` field, which is to contain the URL that the `Load` stage will read. The annotation informs RA that `url` is a *stage argument* and that it must be specified, either via a command line switch (line 12), via a file containing many such properties, or with its default (line 13). Any number of arguments can be specified in any relevant part of the code—RA ensures that they are all populated appropriately via reflection on the declared field type.

Improved research process. Explicit decoupling of stages involves worthwhile reasoning about inputs, outputs, and relationships between code sections. RA extends the building-blocks available in Java so that they better align with how Java is actually used in many research applications.

Argument parsing is time consuming. Many research code bases support a large number of command line switches to control execution flow and to set parameters. RA automatically parses many Java types from command line strings or from auxiliary properties files, assigning values directly into appropriately marked fields. This alone is a major improvement over the current, fractured nature of argument parsing in Java, which requires dispersed help messages, conditional statements, and code branches to stay synchronized. Using Java's support for annotations, each argument is self-documenting and self-contained, and RA generates helpful usage messages and exits if any arguments are missing or malformed. To our knowledge, RA provides the most flexible, convenient, and powerful command line argument processing tool available for Java.

4.6 Checkpointing

Writing an experiment as a directed acyclic graph of stages makes the dependencies explicit, and enables efficiency gains through checkpointing. After a schedule of Stages has been selected, RA can read the `checkpoints` folder in the workbook to see if any sub-sequence with compatible arguments has already been run with its output saved. If so, RA can reload the saved computation and skip those stages.

A Stage class in RA can be marked with the `@Stage.Serialize` annotation. All fields in the stage that are marked with the `@Stage.Export` annotation are automatically written by `StageWise` into the workbook and are thus available for loading later. Line 8 in Figure 8 shows an example, where the `Load` stage is marked with `@Stage.Serialize`. In our example, after `Load` runs, RA will serialize its `WebCounts:text` export. On subsequent runs, if RA sees a `Load` checkpoint with the same URL argument, RA will not run `Load` but will instead deserialize the checkpoint to retrieve the previously exported page text. In this way, RA can dramatically improve running times on more complicated tasks by skipping redundant computation. Furthermore, checkpointing enables repeatability of our `WebCounts` experiment by allowing later stages to be run from a snapshot of the URL page text, even after the contents of the live URL have changed.

Regression testing. The completeness and consistency of output logs between runs simplifies the process of creating research regression tests. Salient data in the output log can be tracked over time as the same experiment is re-run on newer code versions, enabling simple scripts to generate warnings when performance on a particular task changes unexpectedly. Because RA gracefully captures outputs into a single XML stream, comparing even unstructured output between runs is tractable.

Time-saving. Cumulatively, using RA saves the users time by automatically organizing the output, rerunning only necessary stages of experiments, and integrating with the rest of the research workflow.

```java
1  public class WebCountsStageWise {
2    public static void main(String argv[]) {
3      RA.begin(argv);
4      new StageWise().run(Count.class);
5    }
6  }
7
8  @Stage.Serialize
9  class Load implements Stage {
10
11   @Argument("URL␣to␣be␣downloaded")
12   @ArgSwitch("--url")
13   @ArgDefault("http://www.nytimes.com/")
14   String url;
15
16   @Stage.ExportField("WebCounts:text")
17   String text;
18
19   public void run() {
20     text = textFromURL(url);
21   }
22 }
23
24 @Stage.Requires(Load.class)
25 class Count implements Stage {
26
27   @Stage.ImportField("WebCounts:text")
28   String text;
29
30   public void run() {
31     List<String> tokens = tokensInText(text);
32     Map<String,Integer> counts = tokenCounts(tokens);
33     for (String token : counts.keySet()) {
34       RA.stream.line("token", token, "count", counts.get(token));
35     }
36   }
37 }
```

Figure 8: The same experiment as Figure 2, refactored into stages to take better advantage of RA's stage-wise experiment infrastructure. The two inner classes define a `Count` stage and a `Load` stage with an explicit dependency of `Count` on `Load`.

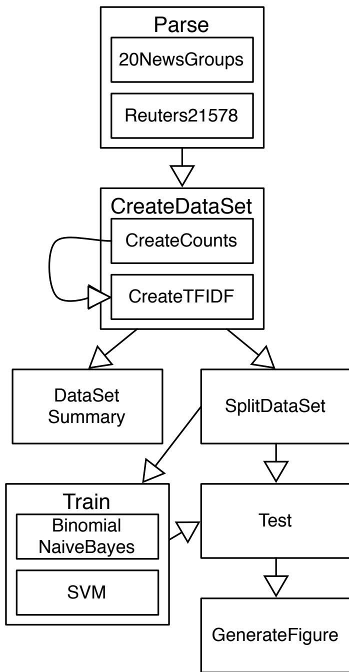

Figure 9: Stage dependency graph of a text classification application. Arrows represent dependency relationships. Sub-boxes represent alternatives for a particular task—e.g. 20NewsGroups and Reuters21578 are two standard corpora in the text classification community.

4.7 Scalability

Computer scientists manage complexity with hierarchies. Coding an experiment as a graph of stages with explicit dependencies is one way to scale to almost any experimental design. In this way, the authors have used RA to develop a text classification experiment platform named TextTickle. Figure 9 illustrates its dependency structure. The program performs a set of relatively standard tasks in this problem domain:

- Parses a corpus of text (`Parse`) to extract relevant information. An extension of `WebCounts` serves as a data source for classifying live pages from the Web.
- Creates a dataset abstraction of that corpus (`CreateDataSet`),
- Outputs summary statistics of that data (`DataSetSummary`),
- Splits the dataset into a portion for model training and another for model testing (`SplitDataSet`),
- Trains a model on the training data (`Train`),
- Tests the model on the testing data (`Test`), and
- Outputs a figure based on the results (`GenerateFigure`).

Like much research software, this code started as a single main method that used helper classes to perform the steps above. Substantial effort was spent managing the command line arguments needed for each sub-stage and by the code required to fork between code branches depending on the requested task. The result was generally poor record keeping and an inability to switch tasks without re-reading code and commenting or uncommenting various code blocks.

The refactored example relies on RA to handle argument parsing and dependency resolution. When invoked, the user specifies which stage they would like to run, and RA creates a dependency graph of the stages marked as requirements. Note in particular that some stages have more than one requirement. `Test` requires both `Train` and `SplitDataSet`; `Train` also requires `SplitDataSet`. RA gracefully handles this case, and `SplitDataSet` is only run once.

TextTickle supports multiple sub-types of `Parse`, of `CreateData`, and of `Train` to specify the text corpus, data processing model, and machine learning algorithm, respectively. RA supports this functionality transparently by querying which sub-type of these generic stages to instantiate via the arguments infrastructure. Adding data sources and algorithms is no larger a task than writing a new class that does exactly what it should. The research programmer is freed from worrying about code fragmentation and dead branches, and can easily construct complex experiments that compare runs across parameters, data sets, algorithms, or any combination thereof. The stage-wise experiment infrastructure enhances the programmer's ability to rapidly prototype, benchmark, and document complex research systems.

Parallelization. Although not currently implemented, the features of RA extend naturally to the field of parallel and distributed processing. Stages that don't share prerequisites may be run on different processors. XML tags may be assigned to outputs based on their local timestamps, source of origin, or intermediate routing points. Although such applications are typically obsessed with performance, recall that printing is reserved for when something is being actively debugged or when it is actively breaking. In either case, the overhead is acceptable and the additional information is welcome. As we noted earlier, however, this introduces fresh sources of non-determinism that we do not currently attempt to capture.

Real world usage. RA has already been used in published natural language processing research [8] that presents a model quantifying the relatedness of word pairs. The research software consists of seventeen stages and dozens of auxiliary classes. The `StageWise` infrastructure eased the management of multiple code branches and the handling of three dozen command line arguments. Checkpointing dramatically shortened the code-run-debug cycle because of two long initialization stages; one stage compacts 349 MB of dictionary files down to a 132 MB internal representation and the second pre-processor further compresses this signature to 86 MB, which is directly used by later experiment stages. Running times for the most heavily run target stage was 68 seconds when resuming from the checkpoints, down from a baseline of 125 seconds when checkpointing was disabled. Custom XML tags were used to manage the different types of data exported by any single run of the experiment. The final figures for the paper were automatically generated by scripts that extracted and plotted the appropriate data from the generated XML. Qualitatively, we found that ResearchAssistant substantially improves the process of developing research code in the ways outlined in this paper.

5. CONTRIBUTIONS

Technology has made a tremendous impact on the computational sciences by generating new data sets, making enormous calculations tractable, and enabling powerful simulations. Unfortunately, software has not realized its potential to augment the scientific process. Toward that end, this paper presents the ResearchAssistant (RA) toolkit, a set of programming abstractions that give computational scientists access to, and control over, information flow in their experiments. This includes checkpointing intermediate results, generating structured output in valid XML, and producing runnable, packaged experiments.

This work makes the following contributions:

- Characterizes the information that is lost by computational tools and motivates its value to scientists. (Sections 1 and 3)
- Discusses the history of literate programming and reproducible research. (Section 2)
- Presents RA, a toolkit of programming abstractions for experimental research. (Section 4)
- Illustrates with the `WebCounts` example how RA gives scientists superior control over computational experiments. (Section 4)
- Proffers RA and all code discussed in this paper to the research community [17].

6. ACKNOWLEDGMENTS

The authors would like to thank the following people for their insightful comments: Christopher D. Manning, Alex Aiken, Paul Heymann, Jenny Finkel, Peter Hawkins, and William Morgan. We are also grateful to our anonymous reviewers for their interest and helpful criticisms. Daniel Ramage was funded in part by an NDSEG fellowship. Adam Oliner was funded in part by the U.S. Department of Energy High Performance Computer Science Fellowship.

7. REFERENCES

[1] J. Buckheit and D. L. Donoho. Wavelab and reproducible research. *Wavelets and Statistics*, 1995.

[2] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: A framework and graphical development environment for robust NLP tools and applications. In *Proceedings of the 40th Anniversary Meeting of the Association for Computational Linguistics*, 2002.

[3] J. de Leeuw. Reproducible research. The bottom line. In *Department of Statistics Papers*, number 2001031101. Department of Statistics, UCLA, March 2001.

[4] E. Frank, M. A. Hall, G. Holmes, R. Kirkby, B. Pfahringer, I. H. Witten, and L. Trigg. WEKA—A machine learning workbench for data mining. In O. Maimon and L. Rokach, editors, *The Data Mining and Knowledge Discovery Handbook*, pages 1305–1314. Springer, 2005.

[5] R. Gentleman. Reproducible research: A bioinformatics case study. *Statistical Applications in Genetics and Molecular Biology*, 4(1), January 2005.

[6] R. Gentleman and D. T. Lang. Statistical analyses and reproducible research. *Bioconductor Project Working Papers*, May 2004.

[7] G. Holmes, A. Donkin, and I. Witten. WEKA: A machine learning workbench. In *Proc Second Australia and New Zealand Conference on Intelligent Information Systems*, Brisbane, Australia, 1994.

[8] T. Hughes and D. Ramage. Lexical semantic relatedness with random graph walks. In *Proceedings of EMNLP-07*, 2007.

[9] D. Knuth. Literate programming. In *CSLI Lecture Notes*, number 27. Center for the Study of Language and Information, 1992.

[10] J. Kovačević. How to encourage and publish reproducible research. In *Proceedings of the IEEE Intl. Conf. on Acoustics, Speech and Signal Processing*, April 2007.

[11] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. In *IEEE Transactions on Computers*, volume 36, pages 471–482, April 1987.

[12] F. Leisch. Sweave: dynamic generation of statistical reports using literate data analysis. In *Adaptive Information Systems and Modelling in Economics and Management Science*, number 69. March 2002.

[13] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: Rapid prototyping for complex data mining tasks. In *Proceedings of the 12th ACM SIGKDD, International Conference on Knowledge Discovery and Data Mining*, pages 935–940, Philadelphia, PA, 2006.

[14] R. Netzer and B. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of ACM Supercomputing*, pages 502–511, Minneapolis, MN, November 1992.

[15] R. D. Peng, F. Dominici, and S. L. Zeger. Reproducible epidemiologic research. *American Journal of Epidemiology*, 163(9):783–789, 2006.

[16] M. F. Porter. *Readings in information retrieval*, pages 313–316. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[17] D. Ramage and A. J. Oliner. ResearchAssistant. http://www.stanford.edu/~dramage/ra/, February 2007.

[18] L. P. Schumm and R. A. Thisted. Reproducible research using Stata. North American Stata Users' Group Meetings 2005 16, Stata Users Group, July 2005. available at http://ideas.repec.org/p/boc/asug05/16.html.

[19] M. Schwab, N. Karrenbach, and J. Claerbout. Making scientific computations reproducible. *Computing in Science and Engineering*, 2(6):61–67, November 2000.

[20] I. Vlad. Reproducibility in computer-intensive sciences. *Ad Astra*, 1(2), 2002.

[21] L. D. Wittie. Debugging distributed C programs by real time replay. In *Proceedings of the ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging*, volume 24, January 1989.