EXACT: The EXperimental Algorithmics Computational Toolkit

William E. Hart Jonathan W. Berry Robert Heaphy Cynthia A. Phillips
Sandia National Laboratories
Mail Stop 1318, PO Box 5800
Albuquerque, NM
{jberry,wehart,rheaphy,caphill}@sandia.gov

ABSTRACT

In this paper, we introduce EXACT, the EXperimental Algorithmics Computational Toolkit. EXACT is a software framework for describing, controlling, and analyzing computer experiments. It provides the experimentalist with convenient software tools to ease and organize the entire experimental process, including the description of factors and levels, the design of experiments, the control of experimental runs, the archiving of results, and analysis of results.

As a case study for EXACT, we describe its interaction with FAST, the Sandia Framework for Agile Software Testing. EXACT and FAST now manage the nightly testing of several large software projects at Sandia. We also discuss EXACT's advanced features, which include a driver module that controls complex experiments such as comparisons of parallel algorithms.

Categories and Subject Descriptors

G.4 [**Mathematical Software**]: Algorithm design and analysis; D.2.5 [**Software Engineering**]: Testing and debugging—*testing tools*

General Terms

experimentation, performance, verification

Keywords

experimental analysis, software testing, experimental design

1. INTRODUCTION

Experimental algorithmics and algorithm engineering have been active research areas for at least 15 years. The former is based on the premise that algorithms should be implemented and evaluated empirically in order to augment theoretical analyses, since these hide constant factors and therefore may be deceptive. The latter involves leveraging computer science expertise in hardware and software in order to engineer better implementations of algorithms.

ACM FCRC Workshop on ECS 2007, La Jolla, CA

Computer experimentation can be a burdensome process, and many experimental papers have been completed without the care traditionally taken in physical experimentation. This phenomenon has led David Johnson to publish a famous list of "pet peeves," mistakes (and worse) that he has observed in experimental computer science papers [12].

We developed EXACT to provide the experimental computer science community with a tool to make the whole process of computer experimentation easier and more systematic. We hope to free researchers from some of the burdens of computer experimentation, giving them more time to concentrate on interpreting results and refining experiments to obtain even better results.

EXACT has also been strongly motivated by the need for tools to automate software tests. Before one even begins to experiment with an algorithmic implementation, one must be reasonably sure the software is correct. The recent high-profile retraction of three papers from the journal *Science* [15, 5], and possibly two others, is a powerful example of the potentially large damage caused by even subtle bugs. Because a code flipped two columns of a data matrix, Professor Chang's laboratory group used an inverted electron-density map to derive incorrect protein structures.

Although there exist many tools for defining and managing tests, what is missing from these tools is the ability to use experimental design techniques to explore a wide range of algorithmic combinations in an automated fashion. Thus, the same features that make our toolkit valuable for developing experimental algorithmics studies make it an effective platform on which to build, control, and archive nightly tests. In Section 5.1, we describe FAST, a Framework for Automated Software Testing as a harness to control automated runs of EXACT.

EXACT is now a reliable component of our ongoing experimental algorithmics research and software testing activities. For example, we currently use EXACT for nightly testing of the TEVA Sensor Placement Optimization Toolkit (SPOT) [10]. TEVA-SPOT integrates solvers and other tools for the design of contamination warning systems for water distribution systems. The US Environmental Protection Agency (EPA) uses TEVA-SPOT to design sensor networks for large US cities. Thus testing and quality control are critical. Furthermore, we used EXACT to manage experiments for a recent publication testing methods for improving scalability in TEVA-SPOT [4].

We have recently released EXACT 1.0 and FAST 1.0 to the public under a gnu lesser public license (see `http://software.sandia.gov/Acro` for downloading instructions).

These tools run on a variety of platforms, including linux boxes and PCs running cygwin, mingw, and MS Windows. Further information on EXACT and FAST is available at `http://software.sandia.gov/Acro/EXACT` and `http://software.sandia.gov/Acro/FAST`.

2. BACKGROUND

The concept for EXACT was developed at Sandia National Laboratories in 2003 and prototyped in Adams' undergraduate thesis [3] in 2004. This first-generation prototype modeled the experimental process using objects, storing them in an object-oriented database, and populating these objects with increasing amounts of data as experimental studies progressed. At the time, Sandia researchers had to write custom scripts from scratch for each experimental study. These scripts were often repetitive, and sometimes several different scripts would parse the same data. Any given experimental study may have a specific experimental procedure most natural for it. Given unbounded time one could carefully architect the software scaffolding around each individual experimental study, for example to increase confidence in the results. But given finite time for any given study, such customization can be too costly. We hoped EXACT would allow reuse of many basic building blocks for these sorts of scripts. The EXACT prototype parsed data exactly once, added information to the database, and thereafter the system manipulated only data encapsulated in objects.

For several reasons we decided to rewrite the prototype. There was a fairly constraining taxonomy of objects in the original, and there was no clear distinction between information associated with *experiments*, *controls*, and *analyses*. The EXACT tookit presented in this article makes these distinctions explicit, uses a different taxonomy of objects, and uses XML files as the default medium for storing descriptions of experiments, analyses, and results. This does not preclude the use of a database for archiving results, and it frees EXACT from dependence on third party tools. This has been important for nightly software testing.

To our knowledge, the ExpLab project [11] is the most similar to EXACT. The goals of ExpLab are to (1) provide a simple way to set up and run computational experiments, (2) to provide a means of automatically documenting the environment in which an experiment is run, and (3) to eliminate some of the tedium involved in collecting and analyzing output by providing basic text output processing tools. ExpLab consists of a set of Python scripts for initializing, running and summarizing experiments, including facilities for running experiments on a cluster of workstations.

EXACT differs from Explab in its integrated treatment of experiments and analyses. In Section 2.2, we introduce EXACT's taxonomy of objects, which includes both of these concepts. The extensible XML interface in EXACT allows users to specify and control experimental designs, runs of experiments, and analyses of their results in one concise description.

Systems like Condor [13] help control experiments on remote machines. Many other software efforts, such as advanced random number generators [14], and tools for the design of experiments [1, 2] aid the experimental process at specific stages. However, EXACT is a framework for describing and controlling the whole experimental process.

2.1 The Experimental Process

A typical process of computer experimentation is a feedback network involving the formulation of problems, the design and customization of algorithms, the implementation of those algorithms, the design of experiments to test and compare them, preprocessing of input data to prepare for experimental runs, the runs themselves, the archiving of results, the postprocessing of these results to prepare for analysis, and finally, the analysis and visualization of data. Anomolies or bugs discovered during this process may suggest different ways to perform any of these steps, and the whole process may need to be repeated many times. It is important to be able to modify experimental studies conveniently, and ultimately, to reproduce any experiments that are published or otherwise retained.

2.2 A Taxonomy of Objects

We define an *experimental study* to be the top-level object expressable in EXACT. A study encapsulates one or more *experiments* and/or one or more *analyses*. *Experiment* objects include *factor* objects and the possible *level* settings for the factors, as well as one or more *control* objects that store information such as the environment in which an experiment is to be run and the program or script that will actually perform the run. *Analysis* objects describe the type of analysis to be run and how it is to be accomplished.

2.3 The Capabilities and Usage Model of EXACT

EXACT itself is a collection of Python classes with a driver program that invokes their methods. A user executes this driver from the command line and provides it with arguments indicating the experimental study to be processed, along with optional filters that describe exactly which experiments and analyses are to be run, whether to randomize the order in which experiments are to be run, etc. EXACT then launches the appropriate processes, logs their progress, and compiles structured result files.

The EXACT user is responsible for providing a single script or program that takes three arguments:

- an *input* file containing ordered pairs of *(factor,level)* assignments, along with auxiliary information, such as random seeds,
- the name of a *log* file that will hold the raw output of the experiments and/or analyses to be run, and
- the name of an *output* file that is to contain *(measurement, value)* pairs.

EXACT comes with a suite of experimental studies that are used to test its own functionality. This includes an example script, which many users will be able to adapt for their own experiments.

3. A SIMPLE EXAMPLE

A user of EXACT performs computational experiments and analyses using the python script `exact`. The `exact` script processes an XML input file that specifies an experimental study, which can specify multiple experiments and multiple analyses.

Figure 1 provides an example of an XML specification for a simple experimental study. This example illustrates

the use of EXACT to perform a hypothetical experiment to compare hashing strategies. The root of this specification is an `experimental-study` element, for which a `name` attribute is required. Three XML elements are supported in `experimental-study`: `tags`, `experiment` and `analysis`. The `tags` element is discussed in the Section 4.

```
<experimental-study name="example1">
  <tags>
    <tag>example</tag>
  </tags>

  <experiment name="ht">
    <factors>
      <factor name="hashfn">
        <level>Jenkins</level>
        <level>FNV</level>
      </factor>
      <factor name="collisions">
        <level>chaining</level>
        <level>linear-probing</level>
        <level>quadratic-probing</level>
      </factor>
      <factor name="data">
        <level>dataset1</level>
        <level>dataset2</level>
      </factor>
    </factors>
    <controls>
       <executable>hash_script</executable>
    </controls>
  </experiment>

  <analysis name="LoadFactorUB" type="validation">
    <data experiment="ht"/>
    <options>_measurement=LoadFactor _value=0.75</options>
  </analysis>

</experimental-study>
```

Figure 1: A simple experimental study of hash table options, expressed in EXACT.

3.1 Defining An Experiment

An `experiment` element specifies the factors that defined an experiment (in a `factors` element), as well as how the experiment is executed (in a `controls` element). A `factors` element is comprised of one or more `factor` elements. Each factor represents an experimental choice that can be made in an experiment. For example, when evaluating hash tables, you can independently vary the hashing function used and the collision resolution scheme employed. Further, in computational experiments it is convenient to treat the data set or problem as an additional factor.

The `controls` element specifies how the experiment will be run, including specifications for the design of experiments, replications and random number seeds, and the executable. The default experimental design is a full factorial design with no replications. The `executable` element specifies the command that will be used to run a single experimental *treatment*, which represents a set of factor-level choices.[1] A treatment can have many *trials* or *replications*, individual runs with the same factor-level choices. Running multiple trials is essential for any code with inherent nondeterminism (randomized algorithms), and is usually necessary to account for nondeterminism such as system load effects.

The executable used in example1 has the following syntax:

```
hash_script <input-file> <output-file> <log-file>
```

It is generally useful for the user to write an executable script that parses this input file and calls the application code(s). The EXACT Python module includes routines for parsing an input file. These are particularly handy when factors contain *experimental options* (see below).

The input file is automatically generated by EXACT. It contains a set of option-value pairs. These options specify the factor names, the level used for each factor in this treatment, and information about the number of replications. For example, an input file might look like:

```
_exact_debug    0
_experiment_name        example1.ht
_test_name      3
_num_trials     1
seed    $PSEUDORANDOM_SEED
_factor_1_name  hashfn
_factor_1_level level_1
_factor_1_value Jenkins
_factor_2_name  collisions
_factor_2_level level_2
_factor_2_value linear-probing
_factor_3_name  data
_factor_3_level level_1
_factor_3_value dataset1
```

Each line consists of an option name, followed by whitespace, followed by the option value (which may contain whitespace). The `seed` option in this example has a value that is defined by an environmental variable, `PSEUDORANDOM_SEED`. The use of environmental variables for random number seeds has proven particularly convenient, since it enables the execution of different trials with the same input file; the environmental variable `PSEUDORANDOM_SEED` simply needs to be changed for each trial.

EXACT treats the experimental options beginning with '_' as *internal* options, and other options are *external* options. Both internal and external options are included in the input file for a given treatment computation. External options are used by the executable command, and in our use they typically map directly to command-line options in our test code. By contrast, internal options can change the behavior or EXACT. The use of these options is discussed further in Section 4.

The output file has a similar format; each line represents a measurement-value pair. For example, an output file might look like:

```
"Num Items Hashed" numeric/integer 101
LoadFactor numeric/double 0.8
"Termination Status" text/string "OK"
exit_status numeric/integer 0
```

The output file can contain an arbitrary set of measurements. However, the `exit_status` measurement is required for some of the analyses supported by EXACT. This measurement is assumed to be zero if the experimental computation was executed successfully.

Finally, the log file usually contains the precise call to application code, anything printed during the execution, and possibly other miscellaneous information that is useful for analyzing an experimental computation. This is useful in diagnosing execution errors.

[1]This terminology comes from the medical and psychological experimental design community. However, it has become widely accepted within other communities that employ experimental design tools (e.g. see the MicQuality Six Sigma glossary: `http://www.micquality.com/six_sigma_glossary/`).

3.2 Defining An Analysis

The `analysis` element specifies the type of analysis that will be done on one or more sets of experimental data. EXACT currently supports several types of analyses, including validation that measurement values are correct, baseline comparison of one experiment with another, and computing a simple summary of the relative performance of two experiments.

The example in Figure 1 illustrates a validation test. The `data` element specifies the experiment that is tested, and the `options` element specifies the options for this analysis. The options are formatted as a set of option-value pairs. The `_measurement` option specifies the measurement value that is being tested, and the `_value` option specifies the target value for comparison. The default is to test that the experimental measurement is less than this target value, which in this case tests that the load factor for the hash table is less than 0.75.

3.3 Running an Experimental Study

Suppose that the file `example1.study.xml` contains the text in Figure 1. Then the call to EXACT to run this experimental study is:

```
exact example1.study.xml
```

In this example, the `exact` script puts the experimental and analysis results in the `example1` subdirectory. It puts the experimental results for the `ht` experiment in the `example1/ht` subdirectory. It puts the analysis results in the `example1` directory.

The exact script has some command line options to subselect portions of a larger study file:

```
--experiment=<expr>
   Executes one or more experiments that match a
   regular expression.

--analysis=<expr>
   Executes one or more analyses that match a
   regular expression.
```

By default, `exact` detects whether the `example1` subdirectory exists and avoids rerunning experiments and analyses. These options force these executions, which overwrite the previous results. In the case of experiments, the previous experimental directory is deleted and then repopulated with new results. The `--force` option can be used to explicitly rerun all experiments and analyses.

4. ADVANCED FEATURES

The simple experimental study discussed in the previous section illustrates EXACT's core capabilities for defining and performing experiments and analyses. EXACT contains a variety of advanced features that significantly enhance the flexibility and extensibility of this capability. The following sections highlight features that (1) provide a flexible experimental formulation, (2) support the integration of experimental design tools and replication of experimental treatments, (3) ensure robust execution of experiments, (4) enable the integration of driver scripts, (5) enable analyses with external data, and (6) control the execution of a set of experimental studies.

4.1 Experimental Options

The factor levels used in our simple example are comprised of simple values. In this example, these values naturally correspond to options in our hypothetical hash table test code. However, for complex computational experiments it is often necessary have a factor level correspond to a set of options in a test code. For example, there are many ways to configure a branch-and-bound search engine depending on choice of branching strategy, bounding strategy, incumbent search strategy, etc. Each of these elements of the overall search strategy can have a variety of options. An appropriate experimental design would not always treat these option choices as separate factors. In fact, to set up a simple experiment will often require the specification of multiple search options simultaneously.

The following example illustrates how separate experimental options can be simultaneously specified in EXACT:

```
<factors>
  <factor name="search">
    <level> </level>
    <level>initialDive=true</level>
    <level>initialDive=true integralityDive=true</level>
  </factor>
  <factor name="problem">
    <level>_data=bm23 _optimum=34 _opttol=1e-8</level>
    <level>_data=p0033 _optimum=3089 _opttol=1e-6</level>
  </factor>
</factors>
```

This example illustrates a possible experiment for the PICO mixed-integer linear programming solver [7, 6]. The first factor controls the branch-and-bound search. Specifically, it controls which node to expand next in the branch-and-bound tree. In PICO, any parameter left at its default value need not be specified on the command line. In this example, the first level of the search factor is empty. This will invoke default values for the search: best-first search throughout the computation. Thus the next node expanded is the one with the lowest lower bound (for a minimization problem). The value of the second level specifies an initial depth-first search until PICO finds a feasible solution. At that point, it switches to best-first search. By default, this initial dive expands a node at the lowest depth (longest path to the root of the search tree). The value of the third level tells PICO to do an initial dive always expanding a node with a linear-programming relation solution closest to integral among all open nodes.

This first factor is actually an example of simple nested factors. The integrality option does not make sense unless PICO is doing an initial dive, so this is a nested choice. Exact does not currently support nested factors, as discussed briefly in Section 6. For small examples such as this, we can currently finesse the issue with enumeration of relevant tuples of values.

The second factor specifies options for the test problem. The levels in this factor also specify the value of the optimum for the corresponding test problem and a tolerance that the analysis module can use later to determine whether PICO found the optimal solution.

EXACT parses experimental options and puts them in the input file as separate option-value pairs. Thus, the experimental command can use the factor level value, or use these-option value pairs directly. As noted earlier, EXACT treats the experimental options beginning with an underscore ('_') as *internal* options; other options are *external* options. We can specify a generic validation test as follows:

```
<analysis name="ValgrindErrors" type="validation">
   <options>_measurement="Valgrind Errors" _value=0
            _cmp_operator='eq'</options>
</analysis>
```

The executable creates an output file where the measurement named "Valgrind Errors" is paired with the number of errors the memory-checking program valgrind reported for the run. This validation test checks to see whether valgrind reported no errors.

The following example illustrates the use of analysis options specified with the experiment:

```
<analysis name="FinalValue" type="validation">
  <options>_measurement='SolutionValue'
           _tolerance=_opttol</options>
</analysis>
```

The executable creates an output file that pairs the measurement named "SolutionValue" with the solution from the computation. This validation test checks whether the computed value is optimal within tolerance. The analysis option `_value` defaults to `_optimum` if it is defined. Both `_optimum` and the internal option, `_opttol` are defined with the data in the first example of this section. In this case, `_opttol` defines a convergence tolerance, which may be different for different problems.

4.2 Experimental Design

The EXACT `control` element can specify an experimental design. The default experimental design is a full factorial design, which contains treatments for all combinations of factor levels. A full factorial experimental design can quickly become prohibitively expensive as the number of factors increases. So, EXACT provides a fractional factorial design generator, which selects a subset of the treatments in a full factorial design. Specifically, EXACT's `xu_doe` DOE tool, specified in the following example, uses Xu's method [16] to find a maximally orthogonal fractional factorial design.

```
<controls>
  <doe>xu_doe</doe>
  <executable>pico_test</executable>
</controls>
```

4.3 Experimental Replication

The EXACT `control` element can specify replication of experimental treatments using pseudo-random number seeds. The following example illustrates seeded replication:

```
<controls>
  <replication>4
    <seeds>83794 1349870 108392 965210</seeds>
  </replication>
  <executable>pico_test</executable>
</controls>
```

Users can also specify seeds from a file. EXACT supports (pseudorandom) seeded replication for reproducibility and debugging. When seeds are omitted, EXACT generates seeds using the system time and a pseudorandom number generator.

To perform simple, non-seeded replication, an executable can ignore the seed provided by EXACT. Simple replication can be used to study the impact of the computing environment on an algorithm (e.g. the impact of network latencies). EXACT does not currently recognize that the seed is being ignored, so this use of EXACT may impact analyses.

4.4 Robust Executable Computations

EXACT leverages new process management mechanisms in Python 2.4 to ensure that the experimental computations are managed in a robust manner on both MS Windows and Unix platforms. It launches subprocesses with a mechanism that avoids zombie processes. Further, the `exact` script incorporates signal handlers to trap user interrupts and kill orphaned subprocesses.

If the experimental command is itself a script, then it is critically important that that script also carefully trap signals. EXACT can be used to create robust command scripts. Users can import the EXACT Python module into their command script, and use the `run_command` function to encapsulate the necessary subprocess management.

EXACT also extends the Python subprocess mechanism to enable the specification of time limits for subprocesses. This feature can ensure that experimental computations do not run indefinitely due to coding errors. Further, it can limit the overall computation time for algorithms that have weak stopping conditions (e.g. heuristic optimization methods). Users can specify time limits as follows:

```
<controls>
  <executable timelimit="60">pico_test</executable>
</controls>
```

This example terminates the `pico_test` command after 60 seconds. The command may still create output files for this execution if it captures the termination signal gracefully.

4.5 Driver Scripts

As we noted earlier, the execution command is generally a script that parses the input file, constructs a command-line for the test code, runs the test code, and parses the output to construct the file of measurement outputs. Although this script has a common flow, a different script will generally be needed for each EXACT application. However, the exact `commands` element can customize the test code execution generically.

Consider the following example, which tells EXACT to use its driver command `exact_timing`:

```
<controls>
  <executable driver="exact_timing">pico_test</executable>
</controls>
```

The `exact_timing` command uses the unix `time` command to print execution time information. For example, the command

```
exact_timing /usr/bin/ls
```

executes `/usr/bin/ls` and then prints a summary of timing information in an easily-parsable format.

When a driver command is specified, EXACT sets the environmental variable `EXACT_DRIVER` to the name of this command. If the execution command is a script, then this environmental variable can be extracted and prepended to the command-line for the test code.

This functionality provides a simple mechanism for augmenting the functionality of an existing command script. EXACT provides the following driver commands:

- `exact_timing` - This computes timing statistics in a standard format.
- `exact_valgrind` - This uses `valgrind` to check for memory reference errors and memory leaks.

- `memmon` - This monitors the maximum virtual memory usage, and prints a summary after the command terminates.

- `exact_pexec` - This launches the test code in parallel (e.g. using `mpirun`).

The driver script also provides a simple mechanism for customizing the execution of experiments on different platforms. For example, the `exact_pexec` command can be customized to support different parallel communication mechanisms. This enables the application of parallel tests for a wide range of experiments, while only needing to customize the platform-specific characteristics in a single script.

4.6 Analyses with External Data

EXACT currently supports three types of analysis: validation of experimental measurements, baseline comparison between experiments, and a comparison of relative performance. The latter two experimental analyses may involve the comparison of two or more sets of experimental results. Consequently, the EXACT `analysis` element may contain an arbitrary number of `data` elements that specify experimental results for the analysis.

The following example illustrates three ways of specifying experimental results in a `data` element:

```
<analysis name="Comparison" type="baseline">
  <data experiment="exp1"/>
  <data experiment="exp2" import="example2.study.xml"/>
  <data experiment="exp3" import="example3.study.xml"
                results="example3.exp3.results.xml"/>
  <options>_measurement=Value _tolerance=1e-5</options>
</analysis>
```

The first `data` element specifies experiment "exp1" in the current experimental study; this is the baseline experimental data for this comparison. The second `data` element specifies experiment "exp2" in the experimental study specified by "example2.study.xml". EXACT imports this entire study, assuming that the files lie in the standard `example2` subdirectory. The third data element specifies experiment "exp3" in the experimental study specified by "example3.study.xml". However, this import is restricted to the data in the results file "example3.exp3.results.xml". This file does not need to lie in the `example3` directory; this can be a file in an arbitrary directory.

4.7 Managing Multiple Experimental Studies

Section 5.1 describes how we use EXACT to support software testing in the Acro (A Common Repository for Optimization)[9] software framework. We use different tests of the same software depending upon our goals. For example, it is convenient to distinguish between "smoke" tests, which are quick tests of a code's overall functionality, and "nightly" tests, which perform a more comprehensive tests that usually take longer to run.

EXACT supports the execution of multiple experimental studies. In particular, the `tags` element associated with an experimental study can group studies into categories:

```
<tags>
  <tag>smoke</tag>
  <tag>nightly</tag>
  <tag>pico</tag>
</tags>
```

EXACT would run the study in this example if it were running `smoke` tests, `nightly` tests, `pico` tests, or any combination of the tests matching its tags. But this study would not run if EXACT were only executing `monthly` tests.

5. CASE STUDIES

The following sections provide examples of how we have used EXACT for software development and experimental algorithmic testing at Sandia National Laboratories.

5.1 A Case Study: Software Testing for Acro

The experimental methods supported by EXACT have many applications including software comparison and evaluation, improving code robustness and performance, and software testing. We developed EXACT in conjunction with several large software projects at Sandia National Laboratories, such as the Acro optimization library [9]. EXACT supports software testing in a generic, automated fashion. The FAST (Framework for Automated Software Testing) test harness uses EXACT's testing summaries to coordinate software testing on a distributed set of platforms.

For large software projects, software testing is a complex endeavor. Though testing may never "prove" that a particular code works correctly, it can provide high confidence in the code's quality. In particular, we then have reasonable confidence in the correctness of the code in applications and experiments.

Acro illustrates many of the challenges that we have seen for large software development projects within the Department of Energy. Acro is supports end-user applications, as well as the development of new research capabilities. Consequently, code stability is a critical issue. Further, Acro requires the integration of a diverse set of third-party libraries. Tracking changes in these libraries, and assessing their impact is a significant challenge. Finally, Acro needs to run on a diverse set of high-performance computing architectures. Thus, code portability is an essential requirement for the deployment of Acro.

EXACT supports canonical software testing techniques that help meet these goals: unit testing, regression testing, memory testing, integration testing and functional testing. Perhaps the simplest application of EXACT is for unit testing. Unit testing tests the basic software components that comprise a large software project, for which a test result simply indicates whether the unit test passed. EXACT does not support the specification of unit tests, as is done in frameworks like cppUnit and jUnit. However, EXACT can help coordinate the execution and summary of unit tests. Furthermore, EXACT supports simple difference-comparison analyses for simple applications where a unit test framework is unnecessary. These difference-comparison analyses also support regression testing. Regression testing compares previously passing tests to the same tests on the modified software to ensure that the modifications have not unintentionally caused a degradation of previous functionality. EXACT also includes baseline comparison analyses that can perform regression for numerical values within specified tolerance.

Memory tests check for memory leaks and errors involving using corrupted or uninitialized data. EXACT's driver scripts support these tests seamlessly. For example, consider the `exact_valgrind` driver. This driver simply augments the measurements reported by the user application

code with information about memory violations.

EXACT's experimental design capabilities have effectively supported integration and functional testing. Integration testing exposes defects in the interfaces and interaction between integrated components (modules), and functional testing tests a code to confirm that it supports adequate functionality (e.g. adequate response time to solve a problem). EXACT's experimental design capabilities can be leveraged to quickly design a large number of tests that exercise a code in a wide range of conditions. For example, EXACT facilitates applying a code to many data sets using many control parameters. Thus, EXACT has been an effective testing mechanism for software tools like PICO, which has a lot of command-line parameters that control its behavior. These parameters impact running time, but should not impact the correctness of the computation. Also, PICO integrates complex linear programming software libraries. Testing the interface between PICO and these libraries is an ongoing challenge as these third-party libraries evolve.

EXACT is integrated with FAST, a distributed software testing framework developed with Acro. FAST supports a lightweight, distributed build mechanism that facilitates portability tests on a heterogenous set of compute platforms. Further, FAST supports the evaluation of codes with different software configurations on the a homogenous set of workstations. FAST integrates EXACT experimental outputs to provide a dashboard that summarizes testing results (see `http://software.sandia.gov/Acro/testing`). This dashboard organizes test results around `category` labels associated with each EXACT analysis.

FAST runs nightly tests of evolving Acro software projects. Each morning, it sends all developers an email with an summary information about build and test failures. Developers can then follow links to web pages with quick access to log files from failed runs, analysis results, etc. If a developer produces code on one platform and commits that code to a central repository, he then finds out within 24 hours if that code has caused errors on other diverse platforms. Normally it may take users on these other platforms considerably longer to discover a problem. With immediate feedback, the developer knows what changes he just made, and can therefore immediately narrow the search for the source of the new problems.

Because the EXACT nightly tests run automatically and continuously, they can expose rare errors. For example, this nightly testing recently found a bug in the PICO code. PICO is inherently nondeterministic, so the tests grow different search trees for the same problem each night. This bug occurred only when multiple rare conditions happened simultaneously. The logfiles captured the error and gave a seed that reliably reproduced the bug.

5.2 A Case Study: Code Coverage in DAKOTA

The DAKOTA optimization toolkit integrates a wide range of optimization, sensitivity analysis, and uncertainty quantification capabilities [8]. DAKOTA provides a generic, flexible interface to these capabilities, and is particularly well-suited for complex engineering design applications. A general problem formulation supported by DAKOTA optimizers is:

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & l_c \leq c(x) \leq u_c \\ & l_b \leq x \leq u_b \end{array}$$

where $c(x)$ is a set of linear and nonlinear constraints, which may be bounded above and below.

DAKOTA has recently integrated a set of simple problem transformations that can be used to bias the optimization process and make it more numerically robust. These transformations can rescale the search domain in several ways, including logarithmic scaling (`log`), automatic scaling into $[0, 1]$ (`auto`), and scaling by user-specified characteristic values (`value`). Scaling by characteristic values can be combined with logarithmic and automatic scaling, so there are six different scaling configurations: `log`, `auto`, `log/value`, `auto/value`, `value`, and `none`. Similarly, objective functions, nonlinear constraints, and linear constraints can be separately rescaled.

Figure 2 provides an EXACT experimental study that has been developed to test these rescaling mechanisms. The goal of this study is to ensure that DAKOTA tests cover a range of different scaling combinations to ensure that this scaling mechanism works robustly. The factors in this experiment define the optimizers that will be used, the test problems that will be optimized, and the scaling controls that will be exercised.

An interesting aspect of this experiment is that not all factor-level combinations are feasible. The optimization problems are:

- `nlp-linear` - a prototypical nonlinear optimization problem with linear constraints,
- `nlp-nonlinear` - a prototypical nonlinear optimization problem with nonlinear constraints,
- `nlls-linear` - a prototypical nonlinear least squares problem with linear constraints, and
- `nlls-nonlinear` - a prototypical nonlinear least squares problem with linear constraints.

The optimization solvers are tailored for different types of problems; `JEGA` and `SNLL-opt` can only be applied to the `nlp` problems, and `SNLL-lsq` can only be applied to the `nlls` problems. A more intricate constraint on the factor-level combinations is that `log` scaling of the search domain cannot be performed when there exist linear constraints.

To account for these types of constraints, EXACT supports several mechanisms for filtering experimental designs to eliminate treatments that are not feasible. The mechanism illustrated here uses a python function that is applied to test the feasibility of a treatment. The python file `rescaling.py` is imported when EXACT processes this experimental study, and the function `treatment_filter_fn` is used to test treatments.

Figure 3 show the definition of `treatment_filter_fn`. This function accepts a single argument, which is a Python dictionary. The keys of this dictionary are the factor names in the experiment, and the dictionary values are the levels. The `treatment` element in Figure 2 includes a `value` attribute. When this is set to "text", then the level value is in the dictionary. When this is set to "name" (the default), then the level name is in the dictionary. (In this example, levels are not named explicitly, and thus exact gives them a canonical name like "level_2", which is not useful.)

Figure 3 illustrates how `treatment_filter_fn` naturally filters out unwanted or infeasible treatments. The first condition verifies that log scaling is not used with linearly constrained problems. The next two conditions ensure that

```
<experimental-study name="rescaling">

  <experiment>
    <factors>
      <factor name="solver">
        <level>JEGA</level>
        <level>SNLL-opt</level>
        <level>SNLL-lsq</level>
      </factor>

      <factor name="problem">
        <level>nlp-linear</level>
        <level>nlls-linear</level>
        <level>nlp-nonlinear</level>
        <level>nlls-nonlinear</level>
      </factor>

      <factor name="domain-scale">
        <level>auto</level>
        <level>log</level>
        <level>auto/value</level>
        <level>log/value</level>
        <level>value</level>
        <level>none</level>
      </factor>

      <factor name="objective-scale">
        <level>log</level>
        <level>log/value</level>
        <level>value</level>
        <level>none</level>
      </factor>

      <factor name="nonlinear-constraints-scale">
        <level>auto</level>
        <level>log</level>
        <level>auto/value</level>
        <level>log/value</level>
        <level>value</level>
        <level>none</level>
      </factor>

      <factor name="linear-constraints-scale">
        <level>auto</level>
        <level>auto/value</level>
        <level>value</level>
        <level>none</level>
      </factor>
    </factors>
    </factors>
    <controls>
       <executable>rescale_test</executable>
       <filter>
         <python>rescaling</python>
         <treatment value="text">
               rescaling.treatment_filter_fn
         </treatment>
       </filter>
    </controls>
  </experiment>

</experimental-study>
```

Figure 2: An experimental study for testing rescaling in DAKOTA.

```
def treatment_filter_fn(self,combination):
    if combination["domain"][:3] == "log" and \
       combination["problem"][-6:] == "linear":
       return False
    if combination["linear-constraints-scale"] != "none" and \
       combination["problem"][-6:] != "linear":
       return False
    if combination["nonlinear-constraints-scale"] != "none" and \
       combination["problem"][-8:] != "nonlinear":
       return False
    if combination["problem"][:4] == "nlls" and \
       combination["solver"] != "SNLL-lsq":
       return False
    if combination["problem"][:3] == "nlp" and \
       combination["solver"] == "SNLL-lsq":
       return False
    return True
```

Figure 3: The Python filter function used in the rescaling experiment.

scaling is not considered for constraints that are not represented in the problem. The last two conditions ensure that the appropriate solver is used for each problem.

5.3 Parallel Testing for PICO

This section gives an example of parallel testing for the PICO mixed-integer programming solver. PICO was designed to scale to thousands of processors, but it can run on any number of processors. If we are testing PICO on a machine with many processors, the test problem should be sufficiently large and difficult to stress the system. This same test problem might be infeasible for a smaller machine.

The experiment in Figure 4, has two factors. The first gives two integer programming problems. Problem tiny.mps should only run on machines that have few processors and problem huge.mps should only run on machines that have a lot of processors. The second factor specifies the number of processors for a run. As in the DAKOTA example, we can provide a filter to suppress unreasonable pairings of factor values. Suppose every machine we will use for parallel testing defines an environmental variable specifying a maximum number of processors. For a network of workstations, this could depend on the number of machines to which the workstation can launch a secure remote shell, the number of cores on the workstation, and the number of "virtual" machines it can reasonably emulate through multiple processes. The filter might, then, for example, run the tiny problem only on machines with at most 6 processors and run the huge problem only on machines with at least 32 processors.

This script can run on any platform that supports MPI-like interprocessor communication. PICO orginallly had its own set of parallel testing scripts (called a qa-suite), which we have migrated to EXACT.

6. CONCLUSIONS AND FUTURE WORK

EXACT has proven to be a flexible framework for defining, executing and analyzing computational experiments. Development of EXACT has been motivated by the need for tools to support experimental algorithmics research, as well as robust techniques for software testing. EXACT is now in regular use for software testing and development at Sandia, and software releases for Acro and related projects now rely on it as part of their release process. Furthermore, we now use EXACT to manage computational experiments

```
<experiment>
  <factors>
    <factor name="instances">
      <level>tiny.mps</level>
      <level>huge.mps</level>
    </factor>
    <factor name="numProcessors">
      <level>1</level>
      <level>2</level>
      <level>4</level>
      <level>32</level>
      <level>128</level>
      <level>256</level>
    </factor>
  </factors>
  <controls>
     <executable driver="exact_pexec">
          pico_test --milp
     </executable>
     <filter>
       <python>parallel_pico</python>
       <treatment value="text">
            parallel_pico.treatment_filter_fn
       </treatment>
     </filter>
  </controls>
</experiment>
```

Figure 4: An experimental study for testing PICO in parallel.

for algorithmic research [4].

There are many commercial and open-source software testing tools available. To contrast our use of EXACT, we have noted that EXACT's experimental design capabilities enable the rapid application of a large number of diverse tests. Further, this capability supports empirical comparisons of performance, which is not a goal of traditional software testing tools. However, these tools provide a more integrated ability to define tests and track how they relate to software features. This tracking is more difficult with the experiments defined by EXACT, as is the confirmation that specific code feature requirements have been met (at least, without targeted experiments).

The active use of EXACT continues to drive the development of this tool. The Python module that underlies EXACT has proven quite extensible, as has the XML formats for defining experimental studies and experimental results. For example, the DAKOTA scaling study was suggested to us while writing this article. This study required a more sophisticated filtering mechanism for experimental factors than had previously existed in EXACT. The filtering mechanism described in Section 5.2 was easily added to EXACT, and a working example of this for the scaling example was working within a few hours.

Our use of EXACT has highlighted a number of capabilities that would significantly enhance the functionality of EXACT. These can be broadly categorized as follows:

Experimental Design.

The experimental design capabilities currently supported in EXACT are quite simple. Several projects, such as the PICO example described in Section 4.1 would benefit from explicit support for nested factors. This would allow more concise representation of experiments involving nested factors, but our design-of-experiments code does not currently handle nested factors. We are still exploring the best way to support nested factors. We also plan to support other external experimental design tools. In particular, applications like the DAKOTA scaling example highlight the need for experimental design tools that can integrate constraints in the design process.

Experimental Control.

Currently, EXACT does not support randomization of experiments. Although that is easy to add, a more fundamental feature is blocking factors. For example, in many contexts we wish to compare algorithms on a given set of test problems. Such experiments could be blocked by test problem to ensure that different algorithms are tested on the same problem at approximately the same time, in order to minimize the variance due to changing environmental conditions such as network or computer loads.

Another control issue is the management of replications of seeded experiments. Software like the PICO integer programming solver are inherently nondeterministic. PICO's management of cuts (added constraints) relies on timing information, and its asynchronous parallel computation may be sensitive to network delays. However, some aspects of PICO can be controlled with a pseudo-random number generator. Thus, EXACT should support replications of a seeded experiment.

Management of the execution of computational experiments is also an area for future work. Although process management is robust in most cases, the use of EXACT under Cygwin with native Windows applications remains a challenge in some cases (e.g. when processes fail due to memory errors). Also, parallelization of computational experiments would be particularly nice for interactive analysis of large experiments. The execution process in EXACT is quite modular, so we could easily create a more general mechanism.

Statistical Analysis.

Perhaps the most glaring omission in EXACT is the integration of statistical analysis techniques. Although we have prototyped the use of R for performing simple statistical tests, this capability was not sufficiently mature to include in the initial EXACT release. However, the framework for analyses in the EXACT Python module was specifically designed to enable the easy integration of new analysis classes. New analyses classes can be registered with a simple mechanism, so we expect that it will be straightforward to support many different statistical analyses in EXACT.

Experimental Artifacts.

The default experimental artifacts generated by EXACT are XML files with log information, experimental measurements, and experimental analyses. To facilitate post-experimental analysis, we plan to also support the generation of data tables that can easily be loaded into R and Splus. Although EXACT will eventually support a standard set of statistical analyses, a more flexible environment like R and Splus is often needed for a complete analysis in real-world applications.

EXACT is currently used to generate a database of experimental results for software tests. This artifact is not closely integrated with the current EXACT release, though this was a core design feature of the earlier version of EXACT developed by Adams [3]. Reintegrating this capability into EXACT provides support for persistent experimental

results, which can be accessed from multiple sites if the underlying database is supported on a network server. Further, this capability can support the comparative analysis of experiments at different points in time, which facilitates the robust application of baseline experiments.

Acknowledgements

We thank Stefan Chakerian for collaborating on the interface of FAST and EXACT. We also thank Brian Adams for suggesting the use of EXACT to test DAKOTA's rescaling mechanism. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94-A L85000.

7. REFERENCES

[1] DDACE: Distributed design and analysis of computer experiments. http://csmr.ca.sandia.gov/projects/ddace.

[2] Jmp. http://www.jmp.com.

[3] Adams, K. Exact: The experimental algorithmics computational toolkit. Undergraduate thesis, Computer Science Department, Lafayette College, May 2004.

[4] Berry, J. W., Carr, R. D., Hart, W. E., and Phillips, C. A. Scalable water network sensor placement via aggregation. In *Proc. World Water and Environmental Resources Conference* (2007), American Society of Civil Engineers.

[5] Chang, G., Roth, C. B., Reyes, C. L., Pornillos, O., Chen, Y.-J., and Chen, A. P. Retraction. *Science 314*, 5807 (Dec 2006), 1875.

[6] Eckstein, J., Hart, W. E., and Phillips, C. A. Massively parallel mixed-integer programming: Algorithms and applications. In *Frontiers of Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds. SIAM, 2006.

[7] Eckstein, J., Phillips, C. A., and Hart, W. E. PICO: An object-oriented framework for parallel branch-and-bound. In *Proc Inherently Parallel Algorithms in Feasibility and Optimization and Their Applications* (2001), Elsevier Scientific Series on Studies in Computational Mathematics, pp. 219–265.

[8] Eldred, M. S., Hart, W. E., Bohnhoff, W. J., Romero, V. J., Hutchinson, S. A., and Salinger, A. G. Utilizing object-oriented design to build advanced optimization strategies with generic implementation. In *Proc Sixth AIAA/USAF/NASA/ISSMO Symp on Multidisciplinary Analysis and Optimization* (1996), pp. 1568–1582.

[9] Hart, W. E. The ACRO optimization home page. http://software.sandia.gov/acro.

[10] Hart, W. E., Berry, J. W., Riesen, L. A., Murray, R., Phillips, C. A., and Watson, J.-P. SPOT: A sensor placement optimization toolkit for drinking water contaminant warning system design. In *Proc. World Water and Environmental Resources Conference* (2007), American Society of Civil Engineers.

[11] Hert, S., Kettner, L., Polzin, T., and Schäfer, G. ExpLab: A tool set for computational experiments. http://explab.sourceforge.net.

[12] Johnson, D. S. A theoretician's guide to the experimental analysis of algorithms. In *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, M. H. Goldwasser, D. S. Johnson, and C. C. McGeoch, Eds. American Mathematical Society, Providence, 2002, pp. 215–250.

[13] Litzkow, M. J., Livny, M., and Mutka, M. W. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems* (1988), pp. 123–130.

[14] Mascagni, M., and Srinivasan, A. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Transactions on Mathematical Software 26* (2000).

[15] Miller, G. A scientists's nightmare: software problem leads to five retractions. *Science 314*, 5807 (Dec 2006), 1856–1857.

[16] Xu, H. An algorithm for constructing orthogonal and nearly-orthogonal arrays with mixed levels and small runs. *Technometrics 44*, 4 (Nov 2002), 356–368.