Collaborative Research: CNS Core: Small: RUI: Intelligent Developer Infrastructure

Charles Curtsinger and Daniel W. Barowy

1 Overview

We propose to design a software development framework that reduces the burden of common developer tasks. Under this vision, developers no longer have to bridge the semantic gap between their intentions and the domain-specific logic for common software development tasks. Instead, developers provide examples of what they want, and the framework automatically translates their examples into domain-specific logic. We exploit a common observation for each developer task we aim to simplify: combined with domain knowledge, observing the behavior of a program or system gives us sufficient information to understand developer intentions. We apply this observation to improve three key tasks:

Building: Observing an initial build exposes the commands that perform that build, along with their dependencies and outputs. Our proposed work automatically finds the optimally minimal incremental rebuild with respect to changed dependencies.

Debugging: Observing a program's execution path demonstrates what conditions must have held to reach a particular point in the program's execution. Our proposed work automatically explains the set of program facts that caused the program to behave differently than the programmer expected.

System Orchestration: Observing a system orchestration task, such as updating a deployed service or invoking a test suite across a pool of machines, exposes the dependencies and effects of those actions in a distributed system. Our proposed work automatically generalizes and repeats the necessary actions so that users can reason at the level of a single computer.

2 Intellectual Merit

Our proposed work has the potential to significantly impact the day-to-day work of software developers. Specifically, our work will *reduce costs* by eliminating the need for developers to write low-level, domain-specific logic; *eliminate bugs* by ensuring that the domain-specific logic for software development tasks is correct by construction; and *improve productivity* by automating common developer tasks using domain-specific insights, all while freeing developers to focus on other aspects of their work.

Our proposed work contributes key scientific and engineering knowledge to the state of the art. First, we will formally describe the domains of building, debugging, and deployment tasks, capturing all of the essential components of each domain as a formal language. Second, we show that program tracing is a powerful window into not just the dynamic behavior of programs in a workflow, but when combined with domain knowledge encoded as a formal language, also enables more general automated reasoning about executions not observed. Finally, we plan to build and distribute practical, usable tools for build, debug, and system orchestration tasks that will facilitate software development by industry, researchers, and students.

Our proposed work would serve as a significant departure from currently-used techniques; existing build systems, debuggers, and system orchestration tools require developers to translate high-level intentions into low-level configurations specific to each task. Prior work in build systems has automated away specific tasks, such as specifying dependencies, but even these approaches rely on developer effort to break builds into separate steps and sequence those actions correctly; our proposed work would entirely replace the task of specifying a build system, instead requiring developers to provide a single example build that is automatically mapped to incremental actions and dependencies. Prior work on debugging has enabled greater exploration of program state through time-travel debugging, but developers are still left to reason about why a program took a particular path; our proposed work would directly answer queries about why a program behaved a particular way. And finally, past work on system orchestration has focused on checking whether modules to perform specific tasks are fully specified and idempotent, but users of these systems must still translate their intentions to invocations of specific modules; our proposed work would allow users to simply demonstrate a task they would like to automate.

3 Broader Impacts

Access to Computer Science Education: We are committed to improving access to computer science education for all students, particularly students from underrepresented groups. Both PIs serve in faculty positions at liberal arts institutions with no graduate students. This grant enables both PIs to continue their research agenda, which will necessarily involve undergraduate students. Previous research projects by both

PIs have included significant numbers of women and domestic students of color, and this grant would support the PIs as they continue these efforts.

Infrastructure for Research and Industry: It is our intention to publicly release all software tools, data, experimental results, and any other products of this research. The PIs have consistently released and maintained software originally developed as part of their research work, including DTHREADS, AUTOMAN, STABILIZER, CHECKCELL, COZ, and EXCELINT [**?**, **?**, **?**, **?**, **?**, **?**]. These projects have been used for further research by the PIs and others, incorporated into standard Linux distributions, adopted for use by industrial partners, and remain available under open-source licenses. We also routinely and voluntarily submit our software artifacts to software artifact committees for additional peer-review as a part of our commitment to verifiable and reproducible research. We expect the work supported by this grant to produce software that has substantial positive impacts on research and industry developers.

Impacts on Computer Science Curricula: The challenges addressed by the proposed work are common to any type of software development, including software developed by students. Our proposed work reduces the burden of software development tasks not usually taught in the classroom. These tasks, being necessary for all software development, nevertheless pose obstacles that distract and deter students from pursuing studies in computer science. Both PIs intend to include the tools developed as part of this work in their courses, and will release instructional guides to support other teachers if they choose to use these tools in their own instruction.

4 Results from Prior NSF Support

PIs Curtsinger and Barowy have received no prior NSF support.

5 Example-Driven Build Systems with RIKER

An important but often under-appreciated component of any piece of software is its build system. A build system specifies how code and assets should be transformed into runnable software. These systems capture important compilation procedures left unstated in the source code itself, such as how and in what order modular software components should be correctly built. Critically, build systems make the process of building software more reliable since the programmer need not remember and correctly reproduce the sequence of steps necessary to produce a working executable. To be useful, build systems must satisfy two sometimes-competing goals: builds must be correct, and they must be fast.

To illustrate the challenge in making builds both fast and correct, we will begin with an example build system that uses `make` [**?**]. A `make`-based build system is written in a semi-declarative domain-specific language, stored in a file called `Makefile`. The `make` tool is responsible for reading this specification and performing a build. The simplest build system to specify with `make` is one that performs *monolithic builds*, which will rebuild the entire project when any of its source files are changed. The `Makefile` below shows a monolithic build system for a program with three source files and two headers.

```
program: main.c x.c x.h y.c y.h
    gcc -o program main.c x.c y.c
```

This build system always produces correct builds, as the final output of the build will always match what another developer would build from a clean copy of the source code. Unfortunately, this build system is inefficient; changing any of the program's source or header files will trigger a complete rebuild. The cost of a complete rebuild is low for a small project, but large projects can take hours to build completely. As a result, many large projects use build systems that perform *incremental builds*. An incremental build runs only the commands whose inputs have changed since the last build, resulting in dramatic speedups for builds during development. An incremental build for the same program might use the following `Makefile`:

```
program: main.o x.o y.o
    gcc -o program main.o x.o y.o
main.o: main.c x.h y.h
    gcc -c -o main.o main.c
x.o: x.c
    gcc -c -o x.o x.c
y.o: y.c
    gcc -c -o y.o y.c
```

This updated build system specifies how to build each of the intermediate `.o` files from their source files, and how to combine these files into the final target `program`. Updating `x.c` will require generating a new version of `x.o`, and all object files will need to be relinked, but the other two `.c` files do not need to be rebuilt. However, there are hidden dangers in incremental builds: any missed dependency can produce incorrect outputs. For example, if `x.c` depends on `x.h`, changing `x.h` will not rebuild `x.o` because this dependency is not listed in the `Makefile`. Because of this missing dependency, a developer with a working copy of the source code would build a different executable than another developer with a clean copy of the source. Any time an incrementally-updated build would not match a clean rebuild, we have an incorrect build. Incorrect builds can lead to latent errors that only appear for new users or in released software, or could lead to compilation errors that would not occur if the build system were correctly specified.

The risk of missed dependencies sometimes leads developers to include unnecessary dependencies in a `Makefile`; for example, the `Makefile` above could be modified to list every `.h` file as a dependency for each `.o` target. This change will produce correct builds, but including extra dependencies will lead to unnecessary work when rebuilding the project. A better alternative is to use `gcc`'s dependency generation feature (e.g. the `-MMD -MP` flags) that will produce a list of dependencies in `make` format that can be included from the `Makefile`. However, dependency generation is only available for languages with tool support. Even projects that use languages with integrated build systems like Go and Rust will miss dependencies in multi-language projects, or when projects use generated code or dynamic dependencies. These unsupported cases will still require developers to specify dependencies manually, and to keep those dependencies up to date as code evolves.

Existing build tools force users to choose between a simple, correct build system, or a complex, efficient build system. However, developers need not face this tradeoff. We propose to develop a new, example-driven approach to build systems. Developers typically know the commands they would run to build a system from start to finish; RIKER, our proposed example-driven build system, could automatically identify incremental build steps and dependencies rather than forcing developers to close the semantic gap between what they want built, and the steps required to build it. RIKER will allow developers to easily specify build systems that are both correct and efficient.

5.1 Overview of Proposed Solution and Preliminary Work

We begin our discussion of RIKER with the working example from the introduction, a C language project with three source files and two headers. The first step in building this program with RIKER is to create a `Rikerfile`. A `Rikerfile` is simply an executable shell script (or any other executable file) that performs a complete build of the program. To build our example program, we use the `Rikerfile` below:

```
gcc -o program main.c x.c y.c
```

5.1.1 Building with RIKER

After creating a `Rikerfile`, we can use the `riker` command to build the program. Our preliminary work includes a prototype `riker` tool that builds the example described here. In this initial state, RIKER runs the entire build by executing the `Rikerfile`. In addition to executing the build, RIKER traces the system calls of every process involved in the build using a combination of `ptrace` and Berkeley Packet Filters. Tracing system calls allows RIKER to discover input dependencies and outputs, as well as incremental steps of the build. Any point where a build issues a `exec` system call is a point that RIKER can directly invoke on a later build; each such invocation is treated as a separate *command*. Using this trace information, RIKER constructs a *build graph* of commands and files, which serves as the basis for future builds. Figure **??** shows a simplified build graph for the `Rikerfile` above.

As you can see from the build graph in Figure **??**, the simple, one-line build in the example `Rikerfile` produces a relatively complex build graph. The graph shows both commands (light gray ovals) and files (white rectangles) that make up the build. Dashed lines between commands show child command creations; any such line indicates that a parent command called `exec` in its own process or a child process, starting a new child command. The solid lines indicate file inputs and outputs; a line from a file to a command is an input, while a line from a command to a file is an output. This graph is a simplified version of the graph that RIKER produces automatically for this build; the full graph distinguishes between different modifications to a file (e.g. writing, truncating, etc.) and would include dependencies on system files file creation and deletion edges, as well as dependencies on system files.

Looking at the structure of the graph in figure **??**, we see that `Rikerfile` invokes `gcc`, which then launches three instances of `cc1` to compile source files to `.s` files. The `gcc` process also invokes three instances of `as` to assemble the `.s` files into `.o` files. The `gcc` command also launches the `collect2` command, which in turn runs `ld`, the linker, with its `stdout` and `stderr` redirected to temporary files. This redirection allows `collect2` to check for linking errors, and to invoke the linker a second time in some cases. Dealing with circular dependencies is one of several challenges that must be addressed to correctly run incremental builds that we discuss in Section **??**.

5.1.2 Rebuilding with RIKER

At this point, RIKER has completed an entire build and has a graph showing the commands that produced that build. Along with the structure of the graph, RIKER has also collected fingerprints for each of the files involved in the build. This allows RIKER to detect whether any files have changed since the completion of the build, first checking the modification time, and then verifying that the file contents actually differ using the fingerprint. Every command node in the graph can be invoked on its own, so RIKER can skip execution of commands whose dependencies have not changed, and instead descend to that command's children. A command may read from files that are modified or removed by later commands in the build; to invoke a command with such a dependency, RIKER will need to cache these intermediate files and restore them before running the command.

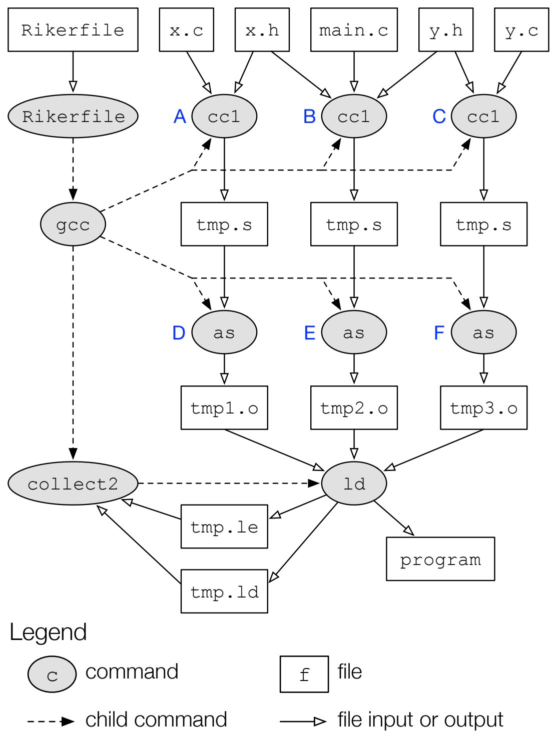

Figure 1: A graph showing RIKER's build of an example program. The same program is built by the `Makefile` examples from the beginning of this section.

As an example, assume a user has added a new function to `main.c`, and added a comment to `x.h`. To run a rebuild, the user invokes `riker` as before. RIKER loads the graph saved from the previous build, and will detect no changes to dependencies for the `Rikerfile` command, nor its child command, `gcc`. Descending past `gcc`, there are now seven child commands that may need to run: three instances of `cc1`, three instances of `as`, and `collect2`. RIKER detects the changed dependencies for the two leftmost `cc1` commands (labeled A and B in blue); command A depends on the changed `x.h`, and command B depends on both changed files, `x.h` and `main.c`. While RIKER observed the commands A and B run in some order for the original build, it is free to choose either command to run at this point because both commands' dependencies are fully satisfied, and the two commands do not interact during their executions. For this example, we will assume RIKER runs command B first, producing a new version of the file `tmp.s` that is read by the `as` command labeled E. Notice that all three `cc1` commands place their output in the file name `tmp.s`; this is `gcc`'s actual behavior, not just a simplification in the build graph. Because these commands share an output file, RIKER must run command E before it can run command A, producing a new version of the file `tmp2.o`. RIKER can now run the `cc1` command labeled A, but recall that the only change to file `x.h` was to add a comment. Because comments are not included in the assembly output, this run of `cc1` will produce a file with the same fingerprint as the original version, so the `as` command labeled D will not need to run. RIKER then restores `tmp3.o` from its cache and runs `collect2`. RIKER chooses to run `collect2` rather than `ld` because of how it must deal with cycles, described in greater detail in section **??**. The `collect2` command invokes `ld`, which re-links the program from the three `.o` files.

5.2 Advances over Prior Work

The example build in the previous section illustrates several key advantages of RIKER over `make` and other comparable build systems:

Dependency Tracking. RIKER captures all input dependencies with no developer involvement.

Build System Updates. The `Rikerfile` is part of the build graph; when the build commands are changed, RIKER will update the build accordingly.

Incremental Rebuilds. Rebuilds with RIKER are incremental, even if the original build is specified as a single shell command.

Efficient Rebuilds. RIKER can update builds with strictly fewer steps than make, as build decisions are made based on file contents, even for intermediate files.

Together, these features make RIKER easier to use and more efficient than make, while guaranteeing that incremental builds will correctly match the behavior of the complete build written in the Rikerfile. There are several other benefits to an example-driven build system like RIKER, which we briefly discuss below.

Migrating to RIKER. RIKER makes it easy to set up a build system correctly for new projects, but migrating to RIKER from a complex build system is also quite simple, unlike alternative build systems like cmake or grunt, which would require a complete rewrite of an existing build system. Any other build system would work as a starting point, but we will use a migration from make as an example. The following Rikerfile invokes make to build a program from scratch:

```
make --always-build
```

After running riker with this Rikerfile, the build graph captures all of the dependencies for the project, whether they were listed in the Makefile or not. After editing source code, a user can simply rerun riker to perform an incremental build. If the Makefile itself is edited, RIKER will rerun the build from that stage to update the build instructions to match what is currently in the Makefile. This makes it easy to use RIKER as a replacement for make in established projects, or as a development tool for individual users who do not want to manage make dependencies for an existing project. Of course, running a complete build any time Makefile is edited could be costly; section **??** describes a mechanism that would allow RIKER to run only the commands impacted by the change to the Makefile.

Building with wildcards. An important detail that does not appear in the earlier example is dependencies on directories. A project's Rikerfile is executed using /bin/sh by default (RIKER works with any Rikerfile as long as it is executable). As a result, users can write a Rikerfile that uses shell wildcards:

```
gcc -o program *.c
```

This Rikerfile performs the same build as the original example at the beginning of this section, with one additional benefit. Any time the user creates a new file in the current directory, RIKER will notice that the . dependency of the Rikerfile command has changed and re-execute the Rikerfile command to include new source files in the compilation. Of course, future rebuilds will rerun all compilation steps, even for files that have not changed since the last build; as with the example of migrating from make to RIKER, handling this efficiently requires skipping commands that have not changed since the last build, which we discuss in section **??**.

Tracking system dependencies. While the example in section **??** illustrates RIKER's basic operation, we have omitted some detail from the build graph in Figure **??**. These details allow RIKER to handle complex build scenarios that make simply does not capture. For example, the RIKER build graph includes both local and system files, such as include files in /usr/include or the gcc, cc1, and as executable files. Updating an installed library and header files would automatically trigger a rebuild with RIKER. Failing to update a build when system files change could lead to a situation where an incremental build produces a different result than a complete rebuild. Users of make often address this issue by running make clean all to force a full rebuild, but RIKER resolves this issue automatically by rerunning any commands affected by the changes to system files, just as it does for local files.

Correct parallel builds. RIKER tracks both input dependencies and output files, which makes it possible to correctly execute builds in parallel. Many parallel builds with make fail because different commands write to the same files, or because dependencies between commands are not specified globally [**?**]. RIKER's precise dependency information makes it possible to decide precisely which commands can run in parallel, enabling faster compilation without the risk of failed builds inherent in most parallel Makefile builds.

Reproducible builds. A build system that uses RIKER has a record of every file involved in the build, including header files, shared libraries, compiler binaries, and all sources. This information could be used to create a container that will run exactly the same build on any compatible system. A containerized build could be used to reproduce build issues with specific tool versions, user configurations, etc; currently, diagnosing such issues requires developers to inquire about software versions and make guesses about which libraries, tools, or include files could cause an issue. A container with all of the build dependencies for a project is also useful for archiving software so it can be built in the future, even on much newer platforms. This application of RIKER is one motivating use case for LOCUTUS, an extension of RIKER to enable automation of general tasks across distributed systems, which we discuss in depth in Section **??**.

5.3 Research Challenges

We have implemented a tracing layer for RIKER, which uses `ptrace` to monitor a build process and construct a build graph. This preliminary work has convinced us that completing RIKER will be possible, but several important research challenges remain.

5.3.1 Handling Changes to Dependencies and Outputs

The entire point of having a build system is to build software repeatedly as it evolves over time. As a result, build systems must be able to cope with changes to the dependency graph underpinning the build system. Many build systems rely on developer effort to maintain dependency information, but RIKER must capture changes in dependency information without developer involvement.

Consider a simple edit to a C program, where a developer adds a new `#include` line to a source file. In this case, RIKER will see that the source file has changed, rebuild it, and discover the new dependency. This is a natural result of the trace-based dependency discovery that RIKER employs on every build. However, consider the case where this newly-included file is itself generated by some tool. If RIKER relies on the old dependency graph, it may run the compilation for the modified source file before updating the newly-included header file. This case, and similar cases where changes to a file can introduce new dependencies or outputs must be handled. Our intuition is that RIKER can safely update the dependency graph and restart the build when it discovers a new dependency or output, but addressing with full generality will require careful consideration. An alternative approach we may explore is to pause a command at the point where RIKER discovers it has a new dependency; at this point, RIKER could start other commands to satisfy this new dependency without the need to restart the entire build.

5.3.2 Build Graph Transformations

The tracing layer for RIKER will produce a complete build graph with all dependencies and command edges, but this graph can contain a number of features that complicate decisions on rebuild. Two specific graph features we need to identify and handle are cycles and communicating commands.

Cycles would cause a problem when RIKER is deciding whether or not to run a command during a rebuild. The example in Figure **??** has a cycle involving the `collect2` and `ld` commands. When RIKER is deciding whether or not to run `collect2`, it must consider not only whether the inputs to that command have changed since it was last run, but also whether those inputs *will* change as a result of a child command invoked by `collect2`. There is never a situation where RIKER could safely run just `ld`, since this command could always modify inputs to `ld`'s parent command. To resolve this issue, RIKER will identify strongly-connected components analysis and collapse any commands involved in these components to their least common ancestor before serializing the build graph. Without loss of generality, this will allow RIKER to make build decisions locally rather than globally; a build graph without cycles makes it possible for RIKER to determine whether a command must rerun by looking only at its direct inputs. As a result, we expect RIKER's build algorithm to run in linear time with respect to the size of the build graph.

Another issue that appears in the build graph is communicating commands. If two commands C and D both write a file, and there is at least one write by D between writes by C we have evidence that both C and D must be running at the same time to produce the same output file. Likewise, if C writes a file, D reads that file, and C writes it again, we have evidence that D depends on an intermediate state of the file that is only present during C's execution. It has not escaped our attention that these definitions closely match the definitions used to detect atomicity violations in concurrent programs [**?**]; however, in this case there is not an error, just an interaction that RIKER could not reproduce by running either command independently. We envision two possible approaches to handle communicating commands: RIKER could collapse any communicating commands to their least common ancestor, or RIKER could start all communicating commands as a single unit rather than one at a time. We plan to explore both approaches, with a slight preference for the second as collapsing commands reduces the number of incremental steps available in the build.

5.3.3 Skipping Recognized Commands

Changes to certain files may lead RIKER to re-run a command early in a build process; that command may invoke many child commands that do not need to run, as their dependencies are unchanged. We have outlined two such cases in our earlier discussion: a RIKER build migrated from `make`, or a build that uses shell wildcards to compile all `.c` files in a directory. In these cases, if RIKER could recognize a command as equivalent to one in its build graph and establish that this command's inputs are unchanged, RIKER could skip the command and maintain build correctness. The difficulty in matching new commands to commands in the build graph is with temporary file names; a command may include temporary files as arguments, so a future invocation of a command may not be an exact match for the command in the build graph.

To match and skip these commands, RIKER must employ a fuzzy matching policy that does not over-approximate matches; skipping a command that must run would lead to an incorrect build, so any only false-negatives are acceptable when comparing commands to the build graph. The approach we intend to employ will first look at the command in the build graph. Any input file whose path appears in the command arguments will be marked, along with a fingerprint of that input file at the time the command started. When a new command starts, RIKER will check if that command matches each command in the build graph, with the exception of the regions marked as paths in the previous step. If a command matches, RIKER will then compare the fingerprint of the file referenced by the matched path to the fingerprint from the build graph. RIKER can safely skip any command where *all* paths in the command line arguments match fingerprints for commands from the build graph. Any missed matches because of path differences (e.g. relative rather than absolute path) or unusual path encoding would only result in missed matches. This error should be acceptable, as it will lead to unnecessary commands being run, not a broken build.

5.4 Evaluation Plan

The two primary goals for RIKER are to build software correctly and efficiently. To evaluate RIKER with respect to these goals, we plan to use RIKER to build a number of large software projects such as Firefox, LLVM, Clang, and other widely-used applications with complex build systems. By building each of these projects, we hope to show that RIKER can correctly build the program as the code evolves, that builds with RIKER run a minimal set of commands to update the build, and that runtime overhead with RIKER is sufficiently low. Preliminary results show that overhead from tracing is 10–15% for building C and C++ programs; we expect to overcome this overhead with RIKER's parallel building, which will be enabled by default.

To accurately model developers' interaction with a build system, we will evaluate RIKER as we advance through commits to each project. At each commit, we will first test whether the software builds with the existing build system. If so, we will build the software with RIKER, starting from the state left after the previous non-breaking commit. At each point, we can check the following properties:

Efficiency. How long did RIKER take to update the build from the previous commit? How does this compare to the project's default build system?

Correctness. Did RIKER skip any commands that the default build system ran? If these commands did not depend (transitively) on changed files, the default build system is not minimal. Otherwise, RIKER is incorrect.

Minimality. Did RIKER run any commands that the default build system did not? If these commands depended (transitively) on changed files, the default build system is incorrect. Otherwise, RIKER is not minimal.

Automating this process should allow us to evaluate RIKER across a wide range of projects with different source languages, existing build systems, and domains. Demonstrating improvements in build performance while maintaining correctness would make a compelling case for RIKER.

6 Conterfactual Debugging with SCOTTY

Debugging is an essential component of building software. Research shows that developers routinely spend 35% to 50% of their time debugging their programs [?, ?]. Although substantial research has been invested in reducing this overhead, from tools like stepping [?, ?] or reversible debuggers [?], program animation [?, ?] to automated debugging techniques such as root cause analysis [?] and delta debugging [?], the amount of time developers spend debugging appears to have remained constant [?].

We argue that an important reason for this discrepancy in productivity, despite vastly better tools, is because a mismatch remains between the abstraction level of program misbehavior and the abstraction level of the debugging task. When programs go wrong, developers must reason abstractly about what conditions *may have caused* a program misbehavior, and to fix the bug, they must again reason abstractly about what change to conditions *would fix* the problem. To do this, developers must either simulate an accurate mental model of the program, or else they must use low-level tools to step through an execution and inspect program state. Instead, we propose raising the abstraction level so that developers can directly edit a program state, effectively asking the program to explain why that state did not hold.

To illustrate the problem, consider the C function in the program shown in Figure **??**. This program has a subtle bug that causes segmentation faults on certain inputs. Assume that `get_length` returns the length of the file `f` and `-1` on error. The problem is that `fread`'s `count` parameter (the third parameter) takes an `unsigned int`. Because `get_length` returns an `int`, `len` is silently coerced into an `unsigned int`. When `len` is `-1`, the parameter to `fread` is effectively interpreted as the largest possible `unsigned int`, `UINT_MAX`. In the failing case, the segmentation fault occurs inside `fread`. We show a slightly simplified `fread` implementation in Figure **??** [**?**]. When `fread` is called, the `count` is `UINT_MAX`, so `fgetc` overflows `buf`, eventually causing a segmentation fault when the program writes to an invalid page.

```c
int example(FILE *f) {
    char buf[1024];
    memset(buf, 0, 1024);

    int len = get_length(f);

    if(len > 1024){
        fprintf(stderr, "file is too big\n");
        return -1;
    }

    if(fread(buf, 1, len, f) > 0){
        /* do something */
    }

    return -1;
}
```

Figure 2: A program that segfaults when `len` is less than `0`.

```c
size_t fread(void* buf,
             size_t size,
             size_t count,
             FILE* f) {
    size_t bytes_requested = size * count;
    size_t bytes_read = 0;
    while (bytes_read < bytes_requested) {
      ((char *)buf)[bytes_read] = fgetc(f);
      bytes_read++;
    }
    return bytes_read == 0 ?
      0 : (bytes_read - 1) / size;
}
```

Figure 3: A basic `fread` implementation.

Understandably, the user would like to know why this program segfaulted. Printing `len` does not obviously explain the program's behavior since the value is `-1`. Since `fread` is a C standard library function, the programmer is unable to insert `printf` statements inside `fread` to learn where the program goes wrong. In this case, they will likely need to resort to a stepping debugger like `gdb`. Because the programmer may not have access to the source code for `gdb`, they will need to resort to stepping through machine instructions in assembly language form. The user will also need to closely inspect the man page for `fread` to understand why `-1` is not an acceptable value. It is clear that the semantic gap between the programmer's query, "Why did my program segfault?," and the steps they must perform in order to answer that query is large. Such *program reachability* questions are among the most common questions asked by developers while debugging [**?**].

Instead, we propose an approach that answers the programmer's query with an *explanation* using dependence information from the value most likely to have caused the fault in the first place. In this case, the explanation is "*The error occurred because* `len` *was* `-1` *which was coerced to* `4294967295`*. It should have been greater than 0 and less than or equal to 1024.*" The key insight in answering the user's query in this form is to transform it into different query that has more explanatory power. For instance, we could naively report all of the program conditions that held when the segfault occurred. However, there could be thousands or even millions of values in the program's state at the time of the error. Instead, we transform the query to ask "Why wasn't one of the branches taken in the failing `if` statement in the `example` function?" Our approach learns this value-based explanation by leveraging dependence information obtained by tracing the program, using a form of concolic execution.

6.1 Overview of Proposed Solution and Preliminary Work

We begin our discussion of SCOTTY with the working example from the introduction. We plan to implement SCOTTY as a plugin for the `gdb` debugger. Currently this project is in the theoretical development phase. The first step in diagnosing the problem using SCOTTY is to compile the program as usual and then to start it using `gdb`.

6.1.1 Running SCOTTY

A user starts running their program with SCOTTY by first starting `gdb` and then by invoking the `scotty` plugin. Suppose that our program is called `program` and that we want to give it a path to a text file called `input.txt`.

```
$ gdb program
(gdb) scotty < input.txt
```

Queries can be provided in one of three ways: by providing no query, by setting a breakpoint and changing a value when the debugger breaks, or by setting a breakpoint in unreachable code. Since, in this case, the user has not provided SCOTTY with a query we assume a default query of "Why did my program crash?" SCOTTY executes the program, tracing it in order to observe the program's control flow. If the program does not crash, SCOTTY does not need to do any further work. However, if the program crashes, as it does in this case, SCOTTY computes the solution to the above query. Answering this query requires transforming it into a satisfiability problem to be given to an SMT solver. In essence, we ask a solver an equivalent query, which is "How should values be changed such that either the `true` or `false` branch after the conditional line 12 are reachable?" As we describe in the next section, our SMT formulation includes all of the so-called *path conditions* that lead the program to its point of failure.

Tracing to gather path conditions. As SCOTTY runs, it gathers path conditions for every instruction executed. These path conditions concisely capture necessary and sufficient conditions for a line of code to be executed. For example, at the point of failure in our example, the path conditions shown in Figure **??**.

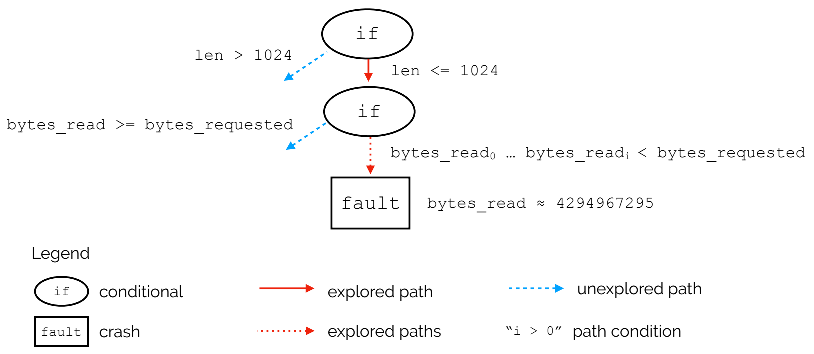

Figure 4: A control flow graph showing explored paths at the time that the segfault is raised. Red lines denote explored paths; dashed blue lines denote unexplored paths. The dotted red line is a shorthand for the sequence of paths. Each path is also annotated with the condition that must have held for that branch to be taken.

Expanding unexplored paths. Since the trace recorded by SCOTTY does not include either branch of the conditional on line 12 of Figure **??**, SCOTTY determines that it needs to explore other paths. Recall that these branches are a part of the query issued by the user. The process of expanding unexplored paths is related to concolic execution techniques used in coverage-directed fuzz-testing tools [**?**].

The last conditional before the segfaulting branch compared the ith value of `bytes_read` to `bytes_requested`. To explore other paths, SCOTTY collects and fixes all path conditions up to but not including the last condition, negates the last condition, and formulates an SMT query to find new variable values such that the formula is satisfied. In principle, there are two ways to satisfy the negated final condition, `bytes_read >= bytes_requested`: find a new value for `bytes_read` or a new value for `bytes_requested`. SCOTTY uses a set of heuristics to prune and prioritize which values to synthesize. In this case, `bytes_read` is computed entirely within the loop body, which makes it a bad candidate for synthesis. `bytes_requested` is generated from two function parameters, `size` and `count`. These are better candidates.

size is ultimately derived from the user-encoded literal program value 1 found on line 12 of Figure **??**, while count is ultimately derived from the user input f, a file. SCOTTY prioritizes synthesizing values derived from user inputs before literal values, since it is reasonable to assume that program bugs are likely to come from corner cases not considered by the programmer.

An SMT solver is given a formula of the form $\texttt{len <= 1024} \wedge \texttt{bytes\_read}_0 \ldots \texttt{bytes\_read}_{i-1} < \texttt{(size * count)} \wedge \texttt{bytes\_read}_i \texttt{ >= (size * count)} \wedge F$, where all program variables are fixed values except count and where F is the set of dependence relations induced by dynamic program slicing on the count variable [**?**]. SCOTTY does not consider altering the FILE *f because the meaning of file pointers are opaque. If this formula is satisfiable, SCOTTY uses the new concrete value to alter count and reruns the program, again tracing to collect dependence and control flow information. If the formula is not satisfiable, or the SMT query times out, SCOTTY attempts to synthesize a different value.

Once an alternative value has been obtained, SCOTTY reruns the program with those new values. If the target branch or branches have been reached, SCOTTY stops exploring, otherwise it repeats the process until it either finds the target branch or an reachability heuristic indicates that the target branches are likely not reachable. For this example, SCOTTY expands path conditions until it has the set shown in Figure **??**.

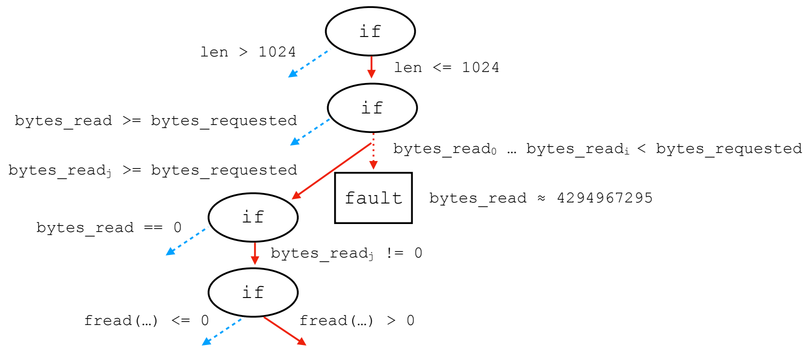

Figure 5: A control flow graph expanded by SCOTTY's re-execution, showing explored paths leading to one of the target branches of the user's query.

Explaining the query result. Because SCOTTY found a set of alternative concrete values, it is able to generate an explanation for the user. The most relevant facts are those that differ from the original execution of the program, namely the value of len. In the trace that reaches the target branch, len is a value greater than zero and less than 1024. This differs from the original execution where len is -1.

Even with causal information localized to the len variable, there are many facts that may be relevant to the user. In addition to informing the user that len was -1, it is also important to convey the fact that the len and count variables do not have the same semantics, even though they refer to the same storage location. This is why we report that len is coerced into the unsigned value 4294967295.

6.2 Advances over Prior Work

The example in the previous section illustrates the key advantages of SCOTTY over other debugging systems.

Intuitiveness. Because SCOTTY surfaces its explanation by allowing a developer to specify the program behavior they wanted, the abstraction level for debugging tasks is significantly raised.

Efficient Inference and Re-execution. The core inference procedure, which is based on concolic testing, is known to scale to billions of program constraints. Furthermore, SCOTTY utilizes the same insights present in RIKER to prevent from having to reexecute an entire program from scratch every time a value changes; hardware instructions are modeled as commands in the RIKER framework and only those instructions necessary to update state are executed.

Minimality. Because SCOTTY reports only the program state changes that are both necessary and sufficient to satisfy a user debug query, users can quickly fix their programs without having to reason abstractly about program behavior.

6.3 Research Challenges

A number of research challenges remain before SCOTTY is practical.

6.3.1 Unreachable Code

An important class of debugging queries posed by a user is why code under a breakpoint did not run. Our motivating example was one such case. Unfortunately, it is possible that the code did not run because there is no possible path to the given breakpoint. Cases like this arise when breakpoints are set in unused functions or in branches that are never exercised. Unless these cases are considered, SCOTTY's re-execution engine will never terminate for unreachable constructs.

In the simplest case, dead code elimination techniques can be used to identify statically unreachable portions of a program [**?**]. Such analyses can be leveraged to explain to the programmer that their breakpoint will never be triggered because they never called the given code.

Other reachability problems are more subtle. Some code may never be reached because a conditional never evaluates in a way that exercises the branch. Although constant propagation techniques can be used to effectively eliminate some branches, this is an expensive static fixed-point procedure. Research is needed to determine whether we can solve the same problem dynamically.

It is likely that unreachable code will remain even after analysis, and so a fallback technique will need to be developed in order to stop SCOTTY's re-execution. A simple approach is a use a global analysis timeout, however this approach is unsatisfying from the standpoint of providing actionable information to the user.

6.3.2 Factual Queries

A *factual* query is asked by the user when they ask a query about a fact that holds during normal execution. For example, a developer might set a breakpoint in a portion of the program that is on a path traced during execution. Alternatively, a developer might indicate that they want to know why a variable holds a given value. These are in contrast to the *counterfactual* queries we have described up to this point.

Although SCOTTY could answer these questions in a simplistic way, either reporting all of the facts that hold when the program breaks, or for value queries, the dynamic slice of the program for that value, we suspect that both of these will be the wrong answer in many cases. A developer who sets a breakpoint in live code is likely trying to understand the execution of their program. In that case, what SCOTTY should return is the optimally minimal explanation that explains why the fact holds. In both these cases, it is not clear what the best minimal explanation is. More research is needed to identify good explanations.

6.3.3 Optimizations

Static control flow graphs. When SCOTTY restarts a computation to explore previously unexplored paths, for certain queries, such as "why not" queries for breakpoints, it is searching blindly. This is problematic from a complexity standpoint: there are a potentially exponential number of alternative paths to explore. This situation could be significantly mitigated if SCOTTY had access to the static control flow graph computed by the compiler. If code under a breakpoint is reachable, SCOTTY need only direct the solver toward negating path conditions clearly on a path to the breakpoint.

Although this approach seems promising, there are caveats. Control flow graphs can only be generated for program text in source code form. It is common practice to distribute libraries, such as the C standard library, in binary form, even to developers. Research is needed to determine whether the control flow graph generated from user code can be combined with a purely dynamic approach described earlier for libraries.

Re-execution. When SCOTTY explores previously unexplored paths, it must evaluate the program with respect to a changed value. The simplest way to do this is to re-execute the entire program. However, this is likely to be a very expensive set of operations. Instead, it would be better to restore the program to the latest state consistent with an already-explored path. In the static case, finding such a state is non-trivial.

We propose leveraging RIKER's dynamic re-execution engine for this purpose. The insight is that we can model hardware instructions as commands with known inputs (registers and memory locations) and outputs (usually registers). Since RIKER already computes the set of commands that need to be re-executed given a change in an input, we can use the same approach to re-execute a program after changing a value. Since this is a relatively low-level technique, research is needed both to map program facts into the user's abstraction level (e.g., *variable* values, not *register* values), and to test its feasibility from a performance standpoint.

6.4 Evaluation Plan

The primary goals of SCOTTY are to be able to answer queries *quickly* and *minimally*. To evaluate SCOTTY with respect to these goals, we plan to use the system to diagnose real bugs. We plan to mine GitHub repositories both for real-world code as well as real-world issues reported by users. Since, as described before, many real bug reports are essentially the kind of query answered by SCOTTY, we suspect that there will be ample data for evaluating the system.

To evaluate speed, we will collect the time it took to gather and process traces as the system runs. Evaluating minimality will require care. By construction, SCOTTY will return the set of facts that differ from the initial execution. However, because SCOTTY is not a static technique, it is possible that there remain other unexplored paths that would answer the same query more minimally. While we could carefully analyze and annotate ground truth for programs by hand, all but the most trivial programs will have very large sets of program values requiring annotation. One promising approach is to generate benchmarks programmatically from known minimal explanations. Another approach is to perform a user study with generated code; although explanations may not be strictly minimal, they may nevertheless be useful for real users.

7 System Orchestration with LOCUTUS

With any sufficiently complex software system, deploying, updating, restarting, and maintaining systems quickly becomes a challenge. Developers often rely on system orchestration tools like `ansible` and `puppet` to automate these tasks. While there are many different system orchestration tools available, we will use `ansible` for our running example; other tools use different domain-specific languages to specify orchestration tasks and make slightly different guarantees, but most tools follow a similar usage model.

Figure **??** shows a simple `ansible` playbook, adapted from an example in the `ansible` documentation, that encodes a series of actions to set up a web server [**?**]. This playbook updates the apache web server package, uses a template to generate a configuration file for apache, ensures the server is started, and if the configuration file changed, restarts the server. None of these steps are terribly difficult to understand, and with some effort a developer can map them to and from the playbook; however, the way these tasks are conveyed to `ansible` is completely different from how the developer would perform them manually. It is likely that the developer used to run this process manually, and later migrated to a system like `ansible` to automate these steps. This process required significant effort: the developer had to learn the `ansible` configuration language, including its peculiar "handler" control flow, and then search through the list of over 3000 modules to find the ones that understand how to invoke `apt` and `service`, and how to generate a configuration file from a template. The developer also had to translate control flow they understood—*if I change the apache configuration, it must be restarted*—to match `ansible`'s representation of conditionals using handlers. This large semantic gap makes the process of automating cloud orchestration tasks both tedious and error-prone.

```
- hosts: server
  remote_user: root
  tasks:
  - name: update apache
    apt:
      name: httpd
      state: latest
  - name: write the apache config file
    file:
      src: httpd.conf
      dest: /etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running
    service:
      name: httpd
      state: started
  handlers:
    - name: restart apache
      service:
        name: httpd
        state: restarted
```

Figure 6: An example `ansible` playbook

7.1 Overview of Proposed Solution and Preliminary Work

LOCUTUS is an example-driven orchestration tool that can replace existing tools like `ansible`. Rather than specifying actions and conditions in a domain-specific language, LOCUTUS observes a user as they perform an orchestration task, tracking the actions, dependencies, and outputs of each step. Using this information, LOCUTUS will be able to update the state of a system automatically any time a dependency changes. Instead of translating an orchestration task to `ansible`'s domain-specific language, a developer could run the same commands they were using to manually execute this task, this time within LOCUTUS:

```
user@localhost $ locutus
locutus:user@localhost $ scp httpd.conf server:~/httpd.conf
locutus:user@localhost $ ssh server
locutus:user@server $ sudo mv httpd.conf /etc/
locutus:user@server $ sudo apt update
locutus:user@server $ sudo apt install httpd
```

```
locutus:user@server $ sudo apt upgrade httpd
locutus:user@server $ sudo service httpd restart
locutus:user@server $ exit
locutus:user@localhost $ exit
```

By tracing the user's actions across this system, LOCUTUS can construct a dependency graph much like the build graphs used by RIKER. In fact, our preliminary work on RIKER has allowed us to explore the design space for LOCUTUS. Unlike RIKER, LOCUTUS will follow actions across remote sessions. This will allow LOCUTUS to trace the dependency on the local copy of `httpd.conf` over `scp` and on to `/etc/httpd.conf` on the remote machine. Restarting the `httpd` service will read this file, so LOCUTUS is aware of this dependency. Critically, the user is not responsible for encoding the exact conditions when the `service httpd restart` command must be run; rather, LOCUTUS can automatically infer this based on the generated dependency graph.

7.2 Advances over Prior Work

LOCUTUS closes the semantic gap for system orchestration tasks in much the same way that RIKER does for build systems. Automating an orchestration task by example ensures that all actions are fully specified; there is no risk of a forgotten parameter or malformed action, both of which are possible when tasks are written in a domain-specific language like `ansible`'s playbooks. In addition to a full specification of actions to be run, LOCUTUS has detailed dependency information that would be laborious to specify manually. This dependency information makes it possible to execution only the necessary steps in the orchestration task on later runs, all without manual intervention. LOCUTUS also eases the migration from manual task execution to automation. In the PIs' own experience, most automated tasks, and are automated once they become too inconvenient. LOCUTUS makes this migration as painless as running the task manually, one last time.

7.3 Research Challenges

A significant amount of LOCUTUS' functionality will derive from RIKER, but adapting RIKER's tracing and update algorithms to work across machines introduces a number of significant research challenges.

Tracing Across Distributed Systems: LOCUTUS needs to trace events locally, and on any other machine accessed as part of the example task. This requires some special handling for the `ssh` and `scp` commands, as LOCUTUS will need to start a tracing process on the remote machine before any work is done on behalf of the user. LOCUTUS needs to combine traces from multiple machines. Traces will be connected through commands that initiate network connections; these commands can depend on and modify files on both machines.

Handling Failure: Any sufficiently complex networked system is likely to suffer from regular failures, and LOCUTUS will need to deal with these failures. Network failure in the middle of a LOCUTUS update may leave the system in an intermediate state. Existing orchestration tools simply report failure and stop the update process, but LOCUTUS may be able to do more than this. LOCUTUS' dependence graph captures information about the state of the system prior to the failure, so it may be possible to undo some or all of the actions that were partially committed. Of course some actions are difficult to revoke: once a process has been stopped or a network connection has been closed, LOCUTUS can no longer undo this change. It may be possible to defer these actions until a final commit phase, or to schedule actions in such a way that LOCUTUS leaves each individual machine in a fully-updated or unmodified state.

Generalizing Orchestration Tasks: Most system orchestration tools have a notion of a machine inventory, with categories for each machine. It is likely that tasks created with LOCUTUS will be run against a number of remote hosts. Users of LOCUTUS could write loops over lists of hosts as part of the demonstration of their orchestration task, but this is an area where LOCUTUS may be able to provide a more convenient abstraction. LOCUTUS could identify hostnames as network connections are initiated, and could parameterize each orchestration task by the names of the hosts accessed by that task.

7.4 Evaluation Plan

We will evaluate LOCUTUS against three criteria: correctness, generality, and performance. Our plan is to find publicly available system orchestration tasks written using other systems to test LOCUTUS. We will need to identify the common types of orchestration tasks that users run; this will require a survey of orchestration tasks hosted on GitHub or other source repositories. Next, we will need to verify that LOCUTUS can accurately capture the actions and dependencies these orchestration tasks require. LOCUTUS should be able to trace an existing orchestration task and accurately reproduce its effects, but we will also need to identify a subset of tasks to manually convert back to command sequences. The first test will allow us to verify that LOCUTUS can correctly run existing orchestration tasks, while the second will show that LOCUTUS could have been used directly, instead of using an existing system orchestration tool like `ansible` or `puppet`. For each orchestration task we evaluate, we will measure the runtime overhead introduced by LOCUTUS' tracing, as well as any performance gains due to parallelization made possible by LOCUTUS' dependence graph.

8 Related Work

8.1 Build Systems

First publicly released in 1976, `make` was one of the earliest build automation tools [?]. `make` takes a file called a Makefile as input, which encodes the relationships between *targets* and their *dependencies*. `make` projects can be defined recursively, where high-level targets are defined in terms of lower-level targets. One of the most important features of `make` is its ability to produce *incremental builds*. `make` is widely-deployed, despite well-known issues such as long build times for complex projects [?]. While many tools have been developed to address `make`'s shortcomings, we focus on notably different approaches below.

Several build systems address `make`'s performance problems. TUP introduces a concise DSL for specifying dependencies and also provides improvements to change detection to produce faster build times [?]. `shake` is a allows users to express build rules using arbitrary Haskell functions. Notably, `shake` constructs the build's dependence graph dynamically, allowing automatically-generated files to be encoded as dependencies [?]. This feature is useful in cases where dependencies are hard to specify ahead of time, such as for platform-dependent header files. `shake` is also much faster than `make` for large recursively-defined software builds [?]. Another alternative is PLUTO, which, while also capturing dynamic dependencies, also surfaces more granular dependencies through a dynamic dependency analysis which enable better incremental builds [?]. However, PLUTO requires language-specific extensions (called *builders*) to support additional languages.

Another issue with `make` is that users sometimes find it difficult to encode dependencies. MEMOIZE and FABRICATE free users from the task of specifying dependencies, instead letting users provide a sequence of build commands [?, ?]. Both MEMOIZE and FABRICATE then utilize system call tracing facilities—such as `strace`—to infer dependencies automatically. Commands are specified as shell scripts or as Python scripts. FABRICATE extends MEMOIZE to Windows, and also provides some facilities for specifying parallel builds.

Finally, BUCK and BAZEL, offer performance improvements like facilitating use on build clusters, notably feature reproducible builds [?, ?]. A *reproducible build system* compiles code deterministically, ensuring that a given input on the same architecture will always produce the same binary. Reproducible build systems allow systems integrators to mitigate software vulnerabilities that target build systems themselves, like Ken Thompson's infamous compiler hack [?]. Like `make`, users of BUCK and BAZEL must specify dependencies manually, although BUCK uses a set of annotations that gives the tool's incremental update algorithm access to more granular dependence information than is explicitly specified.

Like MEMOIZE and FABRICATE, RIKER allows users to specify build commands instead of dependencies. Dependencies are automatically inferred, but unlike MEMOIZE and FABRICATE, RIKER builds are more granular, capturing dependencies from the user's environment, system headers, `mmap`ed files, temporary files, sockets and pipes. These additional dependencies enable more incremental builds since any `exec`ed command can be incrementally executed, but capturing all dependencies is necessary to produce correct builds. A good example is a typical LaTeX paper build using `bibtex` that calls the exact same `pdflatex` command three times, each time producing a different result. Such builds can be thought of as *fixed-point* builds because each run of `pdflatex` augments the build with additional information, finally culminating in a final output. Neither MEMOIZE nor FABRICATE capture build system state changes between `pdflatex` invocations, ultimately producing incorrect builds. RIKER can build LaTeX projects correctly.

8.2 Debugging

Existing debuggers require users to manually set breakpoints and to step through program states to deduce the facts germane to their bug. Although sophisticated debugging tools like reversible debuggers are now commercially available, these tools still delegate the essential *deductive* task of determining why a program fact holds to the end user. Several approaches have integrated examples (specifically, counterexamples) in order to raise the abstraction level for programmers. For example, LALR parsers have been modified to provide counterexamples on parsing conflicts [?]. Counterexamples have also been used in debugging concurrency bugs [?]. We are not aware of any prior work that frames the problem in term of *querying about which changes would lead to alternative program states*.

A great deal of prior work on testing programs looks at systematically altering program inputs in order to find bugs. Some of the earliest work involved fuzz testing, which randomly generates inputs and uses a simple fitness function—does the program crash?—in order to evaluate whether a bug was found successfully [?]. Subsequent research noted that the original fuzz work often repeated work unnecessarily, since even unique inputs can belong to the same equivalence class with respect to control flow, and also that it was a blackbox technique that could not efficiently generate inputs to explore new paths efficiently. Goal-directed fuzzing addresses both of these problems, by inferring equivalence classes, sometimes using hand-written grammars, or by reasoning about them directly from program text or dynamic behaviors [?, ?, ?, ?, ?]. The state of the art in input generation are path-directed techniques based on concolic testing. The most sophisticated of these is the SAGE testing tool, currently in production use at Microsoft, which relies on instruction-level path tracing and SMT [?, ?, ?]. SMT has also been used in a variety of domains, notable in finding concurrency bugs [?]. Although SCOTTY is inspired by much of this work, particularly SAGE, all of this prior work is geared toward finding test inputs that maximize test coverage, not about answering debugging queries.

8.3 System Orchestration

There are numerous tools available for system orchestration, many of which appear in a survey by Delaet, Joosen, and Brabant [?]. The two most widely-used orchestration tools are Ansible and Puppet [?, ?]. While these tools differ in their requirements for deployment—Puppet requires a central server, while Ansible does not—they use similar approaches; users must encode tasks using existing modules that understand how to interact with a particular service or software tool, or perform a common task like filling in a file template or copying a file. The largest difference between orchestration tools is the language they use to encode tasks; Ansible uses YAML, while Puppet uses a custom declarative language to describe tasks. Other orchestration tools include Augeas, which uses a model based on lenses, and NixOS, which uses a lazy functional language to encode actions [?, ?, ?]. LOCUTUS builds on these systems in two important ways; tasks are created through demonstration, not by writing them in a domain-specific language, and LOCUTUS tracks dependencies across machines. Engage and Facebook's orchestration tools also track inter-machine dependencies, but still require users to manually specify tasks [?, ?].

Other work has looked specifically at correcting problems with system orchestration tasks. Rehearsal checks Puppet configurations for idempotence [?]. Tortoise is a tool that can automatically fix Puppet orchestration tasks to reflect a repair action run in a shell [?]. These projects identify and fix existing orchestration tasks encoded in Puppet, but LOCUTUS could potentially avoid these problems entirely.

There is relevant work in tracing across distributed systems, including XTrace and Pivot Tracing [?, ?]. These systems are targeted at performance monitoring and error diagnosis rather than system automation, but the ideas from both may be complimentary to LOCUTUS.

9 Collaboration and Timeline

The PIs will manage the collaboration on this project with weekly conference calls among the participants, including any undergraduate researchers working on the project. In addition to meeting remotely, PIs will arrange to meet at conferences or one of the PI's home institutions at least three times per year. These meetings will allow more intensive collaboration, and will serve as an opportunity for students to establish connections with both PIs. Our planned timeline for the proposed work is as follows:

Topic	Activity	Year 1	Year 2	Year 3
Example-Driven Build Systems	Tracing and Dependency Analysis	•		
	Incremental Build Scheduling	•		
	Publication and Outreach	•	•	•
Counterfactual Debugging	Trace Collection and Analysis	•	•	
	Program State Query Support		•	•
	Publication and Outreach		•	•
System Orchestration	Distributed Tracing	•	•	
	Distributed Update Scheduling		•	•
	Publication and Outreach			•

We plan to release all three tools described in this proposal for public use. It is our intention to continue to support these tools beyond the funding period.