1 Project Description

1.1 Introduction and Motivation

Dynamic analyses, such as testing and profiling, play a key role in state-of-art approaches to software quality assurance (QA). With rare exception, these analyses are performed in-house, on developer platforms, using developer-provided input workloads. A benefit of in-house analyses is that programs can be instrumented and analyzed at a fine-grained level of detail since QA teams have unrestricted access to the software. The shortcomings of focusing on in-house QA efforts alone are severe, however, including (1) increased cost and schedule for extensive QA activities and (2) misleading results when the input test-cases and workload differs from actual workloads or when the in-house system or execution environment differ from that found in the field.

While existing in-house analysis techniques may be adequate for some systems, they are increasingly ineffective for performance-intensive infrastructure software, such as that found in

- *High-performance computing systems*, such as those that support distributed database servers, financial transaction processing, and scientific visualization.
- *Distributed real-time embedded systems* that monitor and control real-world artifacts, such as avionics mission- and flight-control software and automotive braking systems.
- *The operating systems, middleware, and language processing tools* that support high-performance computing systems and distributed real-time embedded systems.

These types of performance-intensive software systems are increasingly to subject to the following trends:

1. Demand for user-specific customization. Since performance-intensive software pushes the limits of technology, it must be optimized for particular run-time contexts and application requirements. Consequently, general-purpose, one-size-fits-all software solutions are often unacceptable.

2. Severe cost and time-to-market pressures. Global competition and market deregulation are shrinking budgets for in-house infrastructure software development and QA, since performance-intensive users are unable or less willing to pay for specialized proprietary software. The net effect on infrastructure software developers is that they have fewer resources to spend on software development and validation activities.

3. Distributed development. Today's development processes often involve developers distributed across geographical locations, time zones, and even business organizations. The goal of distributed development is to reduce cycle time by having developers work simultaneously, with minimum direct inter-developer coordination. Distributed development can increase churn rates in the software base, however, which in turn increases the need to identify faulty changes quickly.

As these trends increase and continue to converge, they present substantial challenges to developers of performance-intensive software systems. Some of the most vexing challenges include:

Software configuration space explosion. To support the customization demanded by users, performance-intensive software must run on many hardware and OS platforms and typically have many options to configure and subset the software at compile- and/or run-time. For example, today's object request broker (ORB) and web services middleware have dozen or hundreds of options. While this flexibility promotes customization, it creates an enormous number of potential system configurations, each of which should ideally be subject to extensive QA.

Widening gaps in in-house QA. The increasing configuration space, coupled with shrinking development resources, makes it infeasible to handle all QA in-house. For instance, developers often do not have access to all the platforms on which the system will run. They must therefore release their systems with many configurations that have not been subject to extensive QA.

Inability to reason about development decisions. The combination of an enormous configuration space and severe development constraints means that team members must make development decisions without precise knowledge of their consequences in the field.

These trends and their resulting challenges create an environment in which software systems tested by in-house developers and QA teams differ greatly from the systems run by users. We contend that techniques and tools needed to bridge this gap effectively will save substantial time, effort, and resources throughout the software industry. We therefore propose a multi-year, collaborative, and interdisciplinary project to achieve the following objectives:

1. Devise technologies and tools that can improve software cost, quality, and performance substantially.
2. Greatly augment in-house resources with *fielded* resources to gather data needed by technologies and tools, thereby shortening the gap between in-house and in-the-field performance.

3. Minimize human effort significantly by automating analysis steps where possible and by using state-of-the-art statistical techniques to minimize analysis efforts without compromising analysis insights.
4. Minimize unnecessary end-user overhead, without compromising their privacy and security.

To realize these goals, we propose to conduct dynamic analyses around-the-world and around-the-clock, leveraging fielded resources during local, off-peak hours. To do this we will devise and/or redesign analyses so they are (1) *highly distributed and lightweight*, e.g., from the perspective of individual participants and (2) *incremental and adaptive*, i.e., they will change their behavior over time based on earlier results. At a high-level, our approach involves multiple iterations through the following steps:

1. Generate a set of judiciously instrumented instances of a program,
2. Place these instances in the field and subject them to realistic inputs and workload,
3. Gather partial analysis information from each program instance, and
4. Combine these partial results into an overall analysis of the program.
5. Feed available results back to the overall analysis to generate the next set of analysis questions.

To accomplish these steps effectively, we will solve the following research challenges:
Lightweight instrumentation. For our approach to work properly, instrumentation overhead must be largely transparent to individual users. This includes overhead both in terms of program execution and human effort.
Distributed compositional analysis techniques. Since each user provides only a small amount of data, traditional analysis that is localized to an individual user machine will not work. We will therefore develop new dynamic analysis techniques that decompose the analysis into smaller steps and use state-of-the-art statistical techniques to distribute the steps among multiple users and then to fuse each user's results into an accurate solution to the original problem.
Managing fielded instances and acquiring data. We will develop the tool infrastructure necessary to conduct distributed analyses on production user programs. This infrastructure will include mechanisms for deploying instrumentation in fielded instances, acquiring data from them, and storing and analyzing the data. It will also involve placing hooks in individual systems and modifying development tools, such as configuration management and bug-tracking systems.
Scenario implementation and empirical evaluation. Empirical evaluation and iterative improvement is a fundamental aspect of our research approach, which we will demonstrate as follows:

1. We will assess our performance on six specific challenge scenarios. Three scenarios will involve *white-box* data acquisition, where we instrument program source code to (1) calculate test coverage, (2) calculate procedure profiles, and (3) compute dynamic call graphs. Section 1.3.2 describes these scenarios and our empirical evaluation approaches in detail. The other three scenarios will involve *black-box* scenarios, which include (1) regression testing, (2) calculating operational profiles, and (3) anomaly detection, i.e., identifying a system configuration whose performance is unusual when compared to its own previous behavior and the behavior of other configurations. Section 1.3.2 describes these scenarios and our empirical evaluation approaches in detail.

2. We will evaluate our performance on real-world software. Our focus will be on two large-scale, production-quality, performance-intensive infrastructure software projects: *ACE* [Sch01] and *TAO* [Centy], which are widely-used, open-source middleware projects developed at the University of California, Irvine (UCI) and Washington University, St. Louis (WUSTL) and now supported commercially by Riverace, Object Computing Inc. (OCI), and PrismTechnologies using open-source business models [O'R98]. ACE [SH02] is object-oriented *host infrastructure middleware* [SS01] containing a rich set of frameworks and components that implement key patterns [SSRB00] for high-performance and real-time systems. TAO [SLM98] is *distribution middleware* [SSM$^+$01] that implements the Common Object Request Broker Architecture (CORBA) [Obj01] using the frameworks and components in ACE to meet the demanding quality of service (QoS) requirements in distributed, real-time, and embedded systems. ACE+TAO have over one million lines of source code, scores of contributing developers, and more than 20,000 active users who work for thousands of companies in dozens of countries around the world [Gro].

We will use ACE+TAO to demonstrate empirically that our technologies and processes can enable real-world developers and users to tailor their QA tools, techniques and processes to improve such areas as fault detection, performance evaluation, memory footprint minimization, and power reduction. We have chosen to focus on the ACE+TAO projects because (1) we control their development process and source code, (2) they are production-quality software that embody many characteristics of performance-intensive infrastructure software, and (3) they exemplify the three trends outlined above:

• **Highly customizable.** ACE+TAO run on dozens of OS and compiler platforms that change over time, e.g., to support new features in the C++ standard, different versions of POSIX/UNIX, as well as different versions of non-UNIX OS platforms, including all variants of Microsoft Win32 and many real-time and embedded operating systems. ACE+TAO are also highly configurable, e.g., numerous interdependent options supporting a wide variety of program families [Par79] and standards. Common examples of different options include multi-threaded vs. single-threaded configurations, debugging vs. release versions, inlined vs. non-inlined optimized versions, and complete builds vs. many subsets. Examples of different program families and standards include the baseline CORBA 2.5 specification, Minimum CORBA, Real-time CORBA, CORBA Messaging, and many different variations of CORBA services.

• **Cost-conscious QA processes.** Since ACE+TAO are freely distributed, open-source projects, serving a broad user community, their developers wrestle constantly with the need to maintain quality across an enormous configuration space while reducing cycle time and decreasing design, implementation, and QA costs [SP01]. To support these demands, the ACE+TAO development process involves many partially and fully automated support tools that we will leverage. For example, ACE+TAO source code resides in a CVS repository, which provides revision control and change tracking [Sou99]. Software defects are tracked using the Bugzilla bug tracking system (BTS) [The98], which is a Web-based tool that helps ACE+TAO developers resolve problem reports and other issues in a timely and robust manner. The ACE+TAO software distributions also provide a substantial regression testing infrastructure containing hundreds of functional and performance tests.

• **Highly distributed development.** ACE+TAO are maintained and enhanced by a core, yet geographically distributed, team of $\sim$40 developers. Many of these core developers have worked on ACE+TAO for years. There is also a continual influx of new developers into the core, many of whom start out as graduate students at WUSTL or UCI. Since the programmers are distributed throughout the world, development and QA decisions are therefore made in a highly decentralized fashion. There is also a user community of over 20,000 users who serve as beta-testers. ACE+TAO thus provide an ideal context in which to explore the costs and benefits of our distributed dynamic analyses using lightweight instrumentation.

1.2 Motivation and Related Work

Our proposed work is motivated both by problems we have encountered individually in our earlier research efforts and by the efforts of other researchers, as described below.

1.2.1 Motivation from Our Previous Work

The computer science researchers in our project team have all worked on problems that rely on dynamic analyses. In each case we have been frustrated by the inability to capture how programs and users behave in the field.

PI Notkin's work (with Murphy and Sullivan) on software reflexion models (CCR-9506779) provided a mechanism for summarizing large source models, e.g., call graphs, in terms of a user-defined high-level model and mapping [MNS95, Mur96]. The most extensive use of reflexion models was by a Microsoft software engineer who used the tools to define and extract a large component from the Excel codebase, which then consisted of about 1.2 million lines of C code [MN97]. The reflexion model tools used statically-extracted source models originally. However, Murphy conjectured that combining static and dynamic call graphs would be more effective information that using either alone. Had it been relatively easy to extract accurate dynamic source models from users of Excel, it would have accelerated and improved the accuracy of the Microsoft engineer's analysis by using the reflexion model approach.

PI Notkin's work (with Ernst, Griswold, and others) on the discovery of program invariants from dynamic traces is based on finding accurate trace data [ECGN99, ECGN01]. Our initial results focused more on discovery mechanisms and algorithms, with further refinements focusing on improving the relevance of reported invariants [ECGN00, Ern00]. We relied heavily on the use of random test cases (which works in a very limited domain) and on the use of pre-defined test cases. Acquiring accurate trace data would have provided–and still can provide–a significant boost to this research.

PI Porter's work (with Harrold, Miller, and Rothermel, CCR-9707792) is aimed at creating and evaluating an infrastructure for building and evaluating program-analysis algorithms that can be applied to large-scale software systems. The resulting system is called Aristotle and it has been used it to implement and evaluate several regression test selection (RTS) techniques [HLL$^+$95]. RTS techniques typically use static analysis to remove unnecessary test cases from the regression test suite. Porter's research into RTS techniques has also relied on pre-canned test inputs. This research would have gone much faster and had more external validity had it been cost-effective to get test data that reflected actual user behaviors.

The CCR-9707792 research results led to a new project (CCR-0098158) aimed at developing and evaluating data-driven (i.e., based on each test case's prior performance) techniques for prioritizing test case execution, determining the order in which changes should be integrated, and pruning and optimizing regression test suites. The current approach uses static data and assumes a traditional development model, which limits the accuracy of the techniques in fast-changing environments. Using dynamic data, gathered with the approaches discussed in this proposal, would allow these techniques to better track changes in the underlying source code and usage patterns.

PI Schmidt's work (with Cytron, Varghese, Turner, and Parulkar, NCR–96–28218 and NCR–97–14698) on the open-source ACE+TAO middleware projects has yielded patterns [SSRB00] and framework techniques [SH02, SLM98] that simplify the development of distributed applications by automating and optimizing distributed object discovery, connection and memory management [SMFGG01], parameter (de)marshaling [GS99a], event and request demultiplexing [OSN01], error handling and fault tolerance [NGSY00], object and server activation [POS$^+$00], and concurrency [Sch98]. These capabilities allow applications to interoperate across networks without hard-coding dependencies on their location, programming language, operating system platform, communication protocols and interconnects, and hardware characteristics.

One reason for the broad adoption of ACE+TAO in R&D projects and commercial products is its high level of software quality resulting from its open development and automated regression-testing processes described in Section 1.1 on page 2. The core ACE+TAO developers cannot test all possible platform and OS combinations, however, because there are simply not enough people, OS/compiler, platforms, CPU cycles, or disk space to run the hundreds of ACE+TAO regression tests in a timely manner. Moreover, since ACE+TAO are designed for ease of subsetting, several hundred orthogonal features/options can be enabled/disabled for application-specific use-cases. Thus, there are a combinatorial number of possible configurations that could be tested at any given point, which is far beyond the resources of the core ACE+TAO development team to handle in isolation.

The inability to regression test a broad range of potential configurations is problematic since it increases the probability that certain configurations of features/options may break in new releases, thereby reducing end-user confidence and increasing subsequent development and QA costs. Using the dynamic data gathering and analysis techniques presented in this proposal would allow ACE+TAO developers and users to track changes in the underlying source code and usage patterns more effectively and to apply the appropriate optimizations more efficiently.

1.2.2 Motivation from Related Work

Our proposed project is also motivated by related work on dynamic analysis techniques, run-time performance measurement, and regression testing. Below, we outline this related work and contrast it with our approach.

Distributed dynamic analysis techniques. Pavlopoulou and Young [PY99] describes residual testing, which tries to improve structural test coverage by instrumenting uncovered entities during deployment. The thesis of their project is that field data can help validate and refine in-house models and also that the ubiquitous presence of the Internet enables approaches that were infeasible previously. Their specific work on residual testing is similar to our approach, with several specific limitations:

- As described in Section 1.3.2, we are tackling a much broader range of analyses, not just structural test coverage and
- As described in Section 1.3.1, we are developing a general infrastructure and approach to support these analyses by lightly instrumenting many fielded instances, each of which provides partial analysis data, which we collect and compose to conduct an overall analysis.

Voas also presents a conjecture similar to ours in [Voa]. He argues that fielded instances of mass-marketed software generate rich, but currently ignored, data that could be used to calculate "operational profiles." An operational profile is a quantitative characterization of the way in which a particular program is used, e.g., a profile of the frequency of invocation of top-level user commands is a common operational profile. We could find neither an empirical evaluation nor an implementation of this approach.

Voas further notes that such ideas have been tried without success in products such as the Netscape Quality Feedback Agent (NQFA) and in the Pure Coverage (PC) toolset. Nevertheless, he argues that these failures are largely attributable to social issues, such as the desire for data privacy and security, and that it might be possible to remedy these problems with appropriate incentives.

While we agree that these social issues must be managed, we do not agree that they are the only—nor even the most immediate—challenges to deploying these ideas. Instead, we claim that the most critical challenges are technical, such as:

• **Minimizing performance degradation.** In theory, each fielded instance can provide as much information as an in-house instance can. In practice however instrumentation that is acceptable in-house will likely degrade field performance to unacceptable levels. In fact, we claim that for our approach to successful, instrumentation on the fielded instances must be virtually transparent. Gains from increasing the number of instrumented instances will therefore be offset by losses from decreasing the amount of instrumentation placed on any single instance. One of our research goals is to show that this tradeoff does not prohibit us from doing a acceptable full analysis by composing partial results from many instances.

• **Accounting for differences in usage patterns.** Another technical challenge is that some analyses will be robust to differences in the way fielded instances are used, whereas others may not. For instance, a company that develops video games may want to know what percentage of their code has ever been executed. It may be possible to answer this question without knowing differences in usage between novice and expert users. If, however, we must decide whether to use our limited resources to fix bugs in module X vs. module Y, it may be important to know how often, for how long, and by whom these modules are used.

• **Dynamically-modifiable instrumentation.** We cannot predict all of the analyses we may ever want to conduct. Consequently, we must be able to change our instrumentation after it has been fielded. In particular, this may require us to modify running binaries remotely.

In short, simple, static, brute force instrumentation approaches will not work. We must therefore devise techniques that minimize the burden that our instrumentation places on fielded instances. Research is needed to determine acceptable instrumentation limits. To meet these constraints, we propose to design and empirically validate new dynamic analysis techniques that work by processing many small packets of data gathered from fielded instances. These new techniques will require:

1. Dynamically modifiable instrumentation
2. Tools that automatically create and place instrumentation in fielded instances
3. Mechanisms for returning data from fielded instances and
4. Techniques that recombine the data to accurately solve the original analysis problem.

We describe our proposed work on these topics in Section 1.3.

Run-time performance measurement. Our proposed research draws heavily on the methods and techniques of run-time performance measurement, particularly issues related to program instrumentation. At a high level, run-time performance measurement involves the following iterative process:

1. Developers first instrument a program, run it, and gather the data provided by the instruments. This data is called the program's *profile*.
2. Developers next analyze the profile to address specific questions, such as "where does the program spend most of its time?" or "what percentage of the time do we take the true branch of this if-statement?"
3. Developers then use the results of this analysis to better optimize the program's performance.

Our work differs from traditional performance evaluation in at least two ways:

1. Traditional techniques tend to be done in-house and thus can assume unfettered access to resources and control over when, where, and how to run the program. Since our program instances may be in the field we may have less access and control.
2. Traditional performance evaluation tends to focus on low-level issues, such as measuring cache behavior or doing profile-directed compiler optimizations. Instead, we are focusing on higher-level software engineering questions, such as testing, reliability, maintenance, and usability.

A common way to instrument a program is to place probes (instrumentation code) at all important program points (e.g., entrances to basic blocks). As the programs runs, instrumentation code executes each time control reaches them. Executing the probes causes data to be calculated and recorded. This data forms the program's profile, which is later analyzed. Such a profile is called a *complete profile*.

Complete profiles at low levels of granularity, however, can generate enormous amounts of data. Hollingsworth presents an example that generated over 2Mb of data per second per node [HLM95]! Moreover, such instrumentation can substantially perturb the very performance developers are trying to observe. Consequently, researchers have explored techniques to lower instrumentation overhead. A significant result from these efforts is the realization that reduced data collection implies increased post-processing if we insist on high-quality information. General strategies for lowering instrumentation overhead include: (1) collect coarser-grained data, e.g., place probes at the entry to every new scope rather than every basic block, (2) sample over time by periodically enabling/disabling the probes, (3) sample across program locations by dynamically repositioning probes, and (4) change probe code functionality dynamically rather than statically preconfiguring probes with all potentially useful functionality.

Traub et al. [TSS00] present an approach called *ephemeral instrumentation* in which they use few probes and collect data infrequently. With this approach probes are statically inserted into the program. At run time the probes cycle between enabled and disabled states based on various parameters settings. Approaches like this yield incomplete profiles. Therefore, a key issue is whether more complicated analyses can be done with any accuracy. In some cases it does appears that statistical techniques can allow us to extrapolate from the data to the actual program behavior. For example, Traub et al. examined the problem of superblock scheduling. This requires information about the relative execution frequencies of each program node and edge, but their technique provided only branch biases. Still, they were able to transform a control flow graph weighted with branch biases into a control flow graph weighted with relative execution frequencies with reasonable accuracy. They did this by reducing the problem into one of finding the limiting probabilities of an irreducible, finite-state Markov chain.

Miller, Hollingsworth and colleagues [MCC$^+$95] have developed a dynamic run-time instrumentation system called ParaDyn and an associated API called DynInst. This system allows developers to dynamically change the location of probes and their functionality at run-time. This general mechanism can be used to implement strategies like ephemeral instrumentation, but also allows probes to change functionality on the fly. For instance, this system was used to automatically identify performance bottlenecks [HMG$^+$97]. It worked by first coarsely measuring performance and then repositioning probes at increasingly finer levels of granularity until the bottleneck was located.

Anderson et al. [ABD$^+$97] use no software probes and collect data infrequently. While the program is running they randomly interrupt it and capture the value of the program counter (or other available hardware registers). Using this information they statistically estimate the percentage of time each instruction is executed. Assuming they run the program long enough, they claim that they can generate a reasonably accurate model with overheads of less 5%. A chief disadvantage of this approach is that it is very limited in the kind of data it can gather. We believe that software probes will be more appropriate for our research.

There are numerous other approaches to instrumentation; among the most prominent are techniques including ATOM [SW94], EEL [LS95] ETCH [RVW$^+$97] and Mahler [WP87].

Regression testing. Our research is also influenced by previous work on software testing, particularly regression testing. After modifying software, developers typically want to know that unmodified code has not been adversely affected. When it has, we say that a *regression error* has occurred. Developers often do regression testing to search for such regression errors. The simplest regression testing strategy is to rerun all existing test cases. This is simple to implement, but can be unnecessarily expensive, particularly when changes affect only a small part of the system.

Consequently, an alternative approach, regression test selection (RTS) technique, has been proposed [AHKL93, CRV94, HGS93, HR90, OW88, RH93]. With this approach only a subset of test cases are selected and rerun. Although several empirical studies suggest that RTS techniques are cost-effective, they are rarely used in practice. One reason is that almost none of these studies takes real-world time and resource constraints into consideration. A recent study by PI Porter [KPR00] suggests that RTS techniques perform poorly when systems are undergoing heavy modifications in a time-constrained development environment and, therefore, cannot be used as their inventors intended. Instead, the RTS-selected test cases must be reduced even further so they can be executed under given constraints. This process is called *test case prioritization*.

Some work has been done to study test prioritization. For example, Wong et al. [WHLA97] proposed several techniques: (1) modification-based test selection then block-coverage-preserving minimization and modification-based test selection then prioritization based on the increasing order of additional cost per coverage. They conducted a case study in which their techniques were applied to a 5000 line program with ten faulty versions. They concluded that both techniques could be cost-effective alternatives in constrained environments.

Rothermel et al. [RUCH98] also proposed and evaluated a family of prioritization techniques. Based on several different programs and test suites, their study suggested that their techniques could improve fault detection rate (faults/number of test cases run). It also suggested that more expensive prioritization techniques might not be as cost-effective as other less expensive techniques.

A key limitation of these approaches is that they are done in-house and treat regression testing as a one-time activity, rather than the recurring process it is. Instead, our work will be done using user resources and will be guided by information about each test case's prior performance. Our approach is based on ideas taken from statistical quality control (exp. weighted moving average) and statistical forecasting (exp. smoothing).

1.3 Technical Approach and Research Plan

This section describes the technical approach to our proposed research. Section 1.3.1 describes our overall plan of attacking the problems described in Section 1.1. Section 1.3.2 then presents six scenarios that drive the implementation of our research. Finally, Section 1.3.3 presents the current status of our preliminary empirical evaluations.

1.3.1 Basic Research Approach

Figure 1.3.1 outlines the basic structure of our research approach. The high-level input into our analysis environment

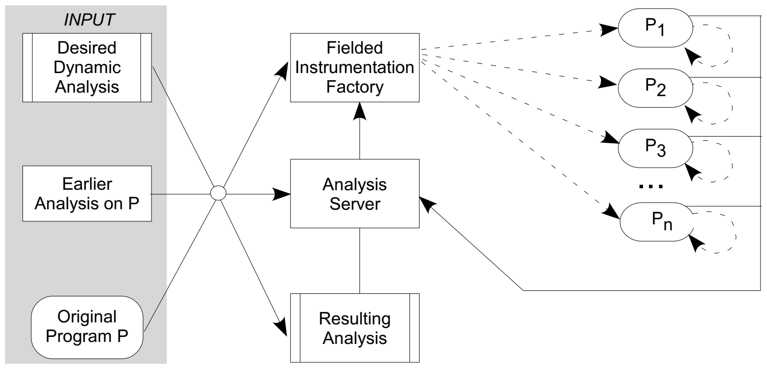

Figure 1: Basic Research Approach

consists of (1) the program to be analyzed, (2) the specific analysis to be applied, and (3) any results from earlier analysis, such as entities already covered by the test suite or a statically extracted call graph. These inputs are fed into an *instrumentation factory* that creates a set of lightly and slightly differently instrumented instances of the program. These instances are then distributed into the field, where they are executed by users. During execution, the instrumentation code provides limited amounts of information to our *analysis server*, which combines the data gathered collectively across the fielded instances along with any earlier analysis performed on the original program. Finally, the analysis server can provide information to the instrumentation factory to re-instrument instances if needed to aid the overall analysis.

To flesh out this approach, our research plan will address the three components described below.

1. Lightweight Instrumentation The goal of this part of our research is to define and implement the functionality of *probes*. Since our probes must not unduly disturb system users, we place strict limits on them. We do this knowing it will result in greater postprocessing costs and reduced information precision. The research challenge is thus to maximize information precision given these rather severe constraints. Initially, we will constrain probes in the following ways:

- *Limited observer role*—Probes can only observe host programs, i.e., they cannot change host program variables.
- *Limited memory allocation*—Probes cannot dynamically allocate storage. A small amount of memory, e.g., 8K, will be statically allocated for use by all probes within a fielded instance.
- *No inter-instance communication*—Probes must be able to pass data from the fielded instances to central servers. To prevent scalability problems as the number of fielded instances grows and to ensure privacy, the number of servers will be small and fielded instances cannot be servers.
- *Limited data transfer*—Throughout the lifetime of the fielded instance we will limit the total amount of data sent to consumers to a maximum of 8K.
- *Limited measurement primitives*—To allow better cost prediction we will allow probes only the following capabilities:
  - *Booleans*. Probes may declare and boolean objects. Boolean support standard relational operators.
  - *Counters*. Probes may declare and manipulate integer and floating point variables. Counters support simple arithmetic.
  - *Timers*. Probes may declare and use timer objects. Timers support start and stop operations.
  - *Host program objects*. Probes can declare objects of types defined by the host program. These objects can be used for copying the contents of host program objects. Probes cannot however change any objects created by the host program.

  - *Send*. Probes send data to consumers using this directive.
  - *Probe language features*. Probe code will allow statement sequencing, decision statements. and bounded loops. Unbounded loops are not allowed.

Naturally, we will recalibrate these constraints as we gain more experience and feedback from our experience.

2. Developing Compositional Analysis Techniques Traditional analysis approaches rely on extensive instrumentation of one or more program instances to collect all necessary data. For example, consider analyses aimed at capturing coverage data or building an operational profile, where a developer instruments all basic blocks (or record all users commands) and then runs the program and collects instrumentation output. In contrast, our approach may not completely instrument fielded instances since we limit the number of probes running at any one time on a given fielded instance. We must therefore decide how we will allocate probes to instances since this will significantly affect the conclusions we can draw from the collected data. Probe allocation raises the following concerns:

- Suppose we want to calculate test coverage. With incomplete profiles it is possible that a basic block is covered by a particular instance, but that the information goes unrecorded because that the necessary instrumentation was not in place. One way to address this issue is to require that multiple instances monitor each basic block.
- Other issues arise if we need to compare the behavior of two or more instances. For example, for calculating procedure profiles we may want to use data from one or more instances to estimate missing data in another instance. To do this we may need to ensure that any two instances have some basic blocks instrumented in common. Moreover, we must do this in a way that does not bias the estimates we want to make later.

These simple examples illustrate that probes cannot be installed haphazardly. How many there are, where we place them, and what data each one collects will change depending on the type of analysis we are doing. Each analysis may therefore have its own constraints. Moreover, when trying to meet these constraints we must carefully explore tradeoffs between maximizing information capture and minimizing the total probe overhead.

To handle these installation problems we will rely heavily on statistical modeling techniques described in Section 1.3.3. At a high level, these techniques will depend on at least the following dimensions:

1. *Desired precision of analysis results*, e.g., exact, approximate, or probabilistic results.
2. *Precision of data requirements*—In some cases, the scope of the needed data is easy to define. In other cases, we must collect more data than we need, filtering it later to answer our original question.
3. *Location of computation*, which is typically split between online (by the fielded instances) and offline (by the analysis server) activities.
4. *Probe adaptation strategy*, i.e., whether and how we change the execution status, location, and function of any probe.

If we do need to dynamically adapt the probes we have several options: (1) *static*, where probes do not change after they have been placed in a fielded instance, (2) *dynamic with internal control*, which can change and where changes are initiated under control of the fielded instance, and (3) *dynamic with external control*, which can change, but where change is initiated under control of an external agent.

3. Management of Fielded Instances After we have defined an analysis technique using the available lightweight instrumentation primitives, we must have technology to insert the correct instrumentation in each program instance. Moreover, this technology must allow instrumentation to be dynamically changed after the instance is fielded. As shown in Figure 1.3.1, we call this facility an *instrumentation factory*.

As a first step in this research we have created the following two prototype instrumentation factories:

1. *White-box factory*—This factory is used for white-box analyses. It takes as input the host program binary, an integer N, indicating the number of instances to create, and a file mapping basic blocks to binary images of probe code. The factory uses Hollingsworth's dynamic runtime instrumentation system, *DynInst* (see Section 1.2.2), to patch the host program binary. *DynInst* facilities also allow the factory to change probe code while the fielded instance is running.
2. *Black-box factory*—This factory is used for black-box analyses. It produces wrappers that optionally give input to the system and monitor external system events.

1.3.2 Research Implementation Strategy

The goals of each type of analysis described in Section 1.3.1 above will affect how we implement them. We therefore divide our research implementation strategy in two tracks: one for white-box analyses and one for black-box analyses. In this section, we present three scenarios for each track that drive the implementation of our research. We describe each scenario to indicate those parts of the scenario supported by our existing infrastructure. Each iteration through the scenarios identifies which parts are currently unsupported, how we propose to enhance the infrastructure, and how we will validate the resulting performance.

Three white-box analysis scenarios. Initially, we will focus on three types of white-box analyses: *test coverage*, *profiling*, and *computing dynamic call graphs*. Although there are many other analyses that we expect to support, we have selected these three since they are important and characterize the broader set of white-box analyses.

1. Test coverage. The first example is intended to improve the test coverage provided by a test suite. Assume that local testing has achieved X% edge coverage and that the program must now be put into beta test. In this case, we would instrument all uncovered edges uniformly across the instances to be fielded during beta test. When any fielded instance executes the uncovered edge, the instrumentation code would indicate this by sending single bit back to the analysis server. In the simplest case, the central test group would know that this edge was omitted from the test suite but was indeed reachable, encouraging them to search for an appropriate test case.

In a more complicated case, the instances detecting execution of the edge might insert additional lightweight instrumentation intended to provide specific information about situations under which the edge can be executed. The overhead of this approach is low, both because the needed instrumentation is lightweight and because the instrumentation is placed on edges that are expected to be executed infrequently. This scenario can be extended to gather more complex information, such as data-flow or condition coverage.

2. Profiling. The second scenario is intended to improve the accuracy of information about the execution profile of a program. In this case, instead of instrumenting just infrequently executed edges, we instrument edges anticipated to be executed frequently. The challenge here is to enable and disable instrumentation in a way that ensures both that the execution is not excessively perturbed and also that a relatively accurate statistical model of execution frequency of the edges is provided (we plan to use a mechanism such as *ephemeral instrumentation* [TSS00] to permit this).

Initially, we will monitor all edges in each instance. We will extend this scenario in two ways:

1. We will explore lowering the cost of instrumentation by monitoring only a subset of the edges in each instance and using overlaps between instances to estimate the missing information.
2. We will use this information to identify distinct categories of user behaviors rather than create a single model of all users. Ultimately, only a small amount of information—such as the execution frequency of several edges—will be sent from each fielded instance to the central server, but the collective information will provide significant profiling guidance for later optimization activities.

3. Dynamic call graphs. The third scenario is intended to enrich a program's static call graph with dynamically-generated call information. Here we first create the static call graph using standard approaches (PI Notkin and colleagues have reported on the empirical effectiveness of tools that embed these approaches [MNGL98]). Being conservative, these approaches identify pairs of procedures that may call each other, which often results in a large number of false positives. We will uniformly distribute instrumentation across the fielded instances with each instance monitoring for calls between a specific set of procedures. If such a call occurs, the instance will report it to the central server. The central server will then instruct the instance to instrument a previously unseen call.

Another strategy would be to instrument only a fixed number of potential call sites and have the server instruct instances to change their instrumentation location after a fixed amount of time if they have not already responded. The resulting data would be used to create a static call graph with each potential call labeled as either a proven call or as an unproven call (with the amount of time over which it has been instrumented). This approach should incur low overhead since over time only infeasible or very rarely executed procedure calls will be instrumented. Although imprecise, we expect this data would still be useful for tasks such as debugging and program understanding (this assumption is consistent with the use of the reflexion model tools [MN97] by the Microsoft engineer (Section 1.5), who did not need complete and accurate information to perform his component extraction).

Three black-box analysis scenarios. Our research effort will also focus on developing the infrastructure and algorithms that support three types of black-box analyses: *regression testing and fault isolation*, *calculate operational profiles*, and *identifying anomalous quality of service behavior*. Although there are many other analyses that we expect to support, we selected these three since they are important and characterize the broader set of black-box analyses.

1. Regression testing and fault isolation. The first example is intended to help isolate faults introduced into a new release. In this example, users first register with the analysis server to provide information about their computing platform. When a user machine becomes available, the factory takes the following steps:

1. It selects a set of "interesting" configuration options and then generates configuration file appropriate for the user's platform and the chosen configuration options

2. It generates necessary test scaffolding
3. It constructs the test suite to be run and
4. It sends these to the user machine, which then downloads any required patches, compiles the system, runs the test suite, and returns the results to the analysis server.

Using this information the analysis server tries to classify the configuration space into three categories: (1) those parts that have a high failure probability, (2) those that have a low probability, and (3) those for which it is unclear. When a new user machine become available the analysis server selects configurations from the "unclear set" with the goal of obtaining a more accurate classification scheme. The net effect is to steer this analysis away from configurations where further testing is not needed towards those where it is.

In more complicated examples, we may need to give different subsets of the test suite to each user to reduce the burden on user machines. We may also need to incorporate domain knowledge in the analysis. For example, we may know that a certain feature is implemented across multiple configurations by the same piece of source code. Such knowledge could sharpen the classification done by the analysis server, further reducing the configuration space to be searched.

2. Calculate operational profiles. This scenario is similar in intent to the white-box profiling scenario described on page 9. The difference here is that we are profiling user input, rather than program behavior. Moreover, in contrast to program profiling, user input events tend to arrive slowly, so we are not overly concerned with instrumentation reducing overhead. However, where this is important (e.g., embedded devices) we will adapt mechanisms such as ephemeral instrumentation to toggle data collection in a way that reduces overhead while allowing for reasonably accurate statistical models. We will extend this scenario to do user segmentation (identify distinct categories of user behaviors rather than create a single model of all users). These analyses will be useful both for directing optimizations and as a fundamental input to reliability modeling.

3. Identifying anomalous quality of service behavior. As discussed in Section 1.1, developers of performance-intensive infrastructure software often make development decisions that work well on developer platforms, but perform much differently in the field. This scenario is intended to help uncover and address these situations.

In this scenario, instances are given input data to run. As they do so, the probe collects externally-visible performance metrics, such as throughput, latency and jitter. These metrics are sent back to the analysis server. For each configuration for which enough data exists, the analysis server compares the current metrics to those taken from earlier versions of the system running the same configuration. After normalizing the data, the analysis server then tries to identify configurations whose performance change is statistically distinguishable from that seen in other configurations.

1.3.3 Current Status of Preliminary Empirical Evaluation

To determine the feasibility of our proposed research plan, we have been conducting preliminary work on prototypes of certain scenarios described above. This section presents the current status of our preliminary empirical evaluations.

Current status: Implementing white-box scenarios. Our fundamental research hypothesis is that analyses using small amounts of data taken from many users will give results that are competitive with traditional analyses that take a large amount of data from a single user. To test our hypothesis, PI Nodkin has conducted two preliminary feasibility studies, which are described below.

• **Initial feasibility study.** Our first study examined this hypothesis in an in-house laboratory. In this study, we focused on a small, well-understood program called `print_tokens`, which is a simple lexical analyzer written in C with 18 procedures. We also have an extensive test suite (over 4,000 test cases) that we used as the test input universe.

The study manipulated three independent variables: (1) the number of procedures monitored (1–18), (2) the number of instances (various settings between 100 and 1000), and (3) the number of test cases each instance ran (250, 500, or 1000). For selected settings of the independent variables we captured procedure invocation counts for all monitored procedures. We then normalized the counts by the number of test cases run, statistically estimated missing counts, and measured the sum of squares error between the estimated counts and the count obtained by instrumenting all procedures, in one instance, running all the test cases.

The most important result of this study was that under certain settings of the independent variables our approach yielded results that were quite similar to those obtained by traditional methods. The study also suggest that tradeoffs exist between the independent variable. For example, we can lower the number of procedures monitored, but only if we can increase the number of instances and/or the length of time over which we are monitoring.

• **Expanded feasibility study.** The goal of the second study was to better understand the infrastructure needs arising from our desire to interact with fielded instances. For this study we used the publicly available UNIX utilities: `gzip`, `less`, and `grep`, which are written in C. The `gzip` utility has 142 subroutines, `less` has 359, and `grep` has 114. Each program has on the order of 5–10K basic blocks.

We instrumented each program to collect basic block coverage and procedure execution frequencies. We created four instances of each program and gave one to each of four users, who used these instrumented versions in place of the one their systems provided. For a period of one week, each time one of these instrumented instances ran we captured the output and timestamped it. At the end of the week we gathered the data and analyzed it.

We are still evaluating all the data, but with respect to test coverage analysis we have made some initial observations. First, we tried static probes. Each instance exclusively monitored one-fourth of the basic blocks. We saw that this strategy captured about 90% of the coverage that a complete profile would have. We also saw that the coverage data stabilized after a relatively short period, i.e., continued program runs identified no previously uncovered blocks. We next used static probes, but allowed each program to instrument 50% of the basic blocks. This approach captured 97% of the the covered blocks on the average.

After this we tried using dynamic probes with internal controls, i.e., each program instrumented 25% of the basic blocks, but the blocks were chosen at random, changing before each run. We repeated this with each program instrumenting 50% of the basic blocks. With 25% of the blocks instrumented we saw that over 99% of the basic blocks were captured on the average. We also saw that coverage data improved the more we ran the programs. Our results showed little difference between the two levels of instrumentation.

Although the results are still preliminary, they are consistent with our belief that instrumenting fielded instances can in some cases yield program information comparable to that normally gathered in-house, but at considerably reduced costs. Clearly, instrumenting even 25% of a program is more than we intend to do in practice. We now intend to investigate what happens when we scale up to hundreds and then thousands of fielded instances.

Current status: Implementing black-box scenarios. To investigate the feasibility of dynamic analysis techniques, PIs Porter and Schmidt have conducted an initial study and a survey of user community's willingness to participate [SP01]. The feasibility study is based on the regression testing scenario described earlier (see Section 1.3.2). Testing is being done on the ACE+TAO middleware, but we are doing it on a cluster of 60+ workstations and servers distributed throughout WUSTL, UCI, and UMD, rather than on user community machines.

Our hypothesis is that distributed continuous regression testing will prove superior to the ad hoc approach currently used by the ACE+TAO project since it will detect software faults that would not have been found otherwise. It will also isolate faults more quickly than current processes because it pinpoints the class of configurations in which faults manifests themselves.

An important subgoal in the regression testing scenario is to precisely determine the configuration options that cause faults to manifest themselves. At every step we must therefore decide which configuration to test next. We call this the *search strategy*. We are currently considering the following two search strategies: (1) *ad Hoc*, which is the strategy ACE+TAO currently uses and (2) *random*, i.e., pick a set of configuration options on a random basis without replacement. As part of our proposed work we will explore more intelligent search strategies, such as:

• **Classification-based search:** Here we will record the results of previous runs and use statistical classification techniques to model the configuration options and settings that are highly correlated with failing/non-failing test behavior. In choosing the next configuration to test, we want to select one that is likely to help us improve the models, i.e., narrow the set of configurations options in the model without reducing model accuracy.

• **Modeled, classification-based search:** Similar to the previous approach, but incorporates external information that ACE+TAO developers have about the code. For example, some features are implemented across multiple configurations with the same source code, this information can be used to reduce the space of configurations that must be considered when doing classification.

So far our distributed approach has identified faults that had not been found with the current ad hoc ACE+TAO QA process. We are therefore proceeding with the study. We are also designing the infrastructure needed for a longer-range set of experiments. Finally, we have surveyed the ACE+TAO user community and determined that a sufficient number of users participate in a online study in which they:

1. Use a web-based registration form to characterize their testing platforms, i.e., their operating system and compiler/linker characteristics.
2. Read and post to a mailing list to communicate with researchers and ACE+TAO developers.

3. Download and run a portable Perl script to automatically download ACE+TAO from the main CVS repository where it resides at WUSTL.
4. At periodic intervals designated by the study participants, receive a configuration file that is customized for their platform. The configuration options will be generated according to the search strategies described above while ensuring the semantic validity of the configuration file.
5. Upon receipt of the configuration file, compile ACE+TAO run all the tests, and send any problems back to analysis server or link it into a "virtual scoreboard," such as `http://ringil.ece.uci.edu/scoreboard/`, that can be examined later via a Web browser. The ACE+TAO core developers who are responsible for different components of ACE+TAO will have their own view into the problem results or virtual scoreboard so that they know what bugs to fix.

Uses will run two types of tests:

1. Tests to evaluate the syntactic and semantic correctness of ACE+TAO with respect to the generated configuration. Syntactic checks will be performed at compile-time. More advanced semantic checks will be performed by running regression tests.
2. Tests that collect metrics on throughput, latency, scalability, predictability, and static/dynamic footprint to pinpoint when and why ACE+TAO's performance decreases on particular platforms.

Based on the results of the initial study outlined above, we plan to conduct more advanced studies that will use the results and feedback from temporally earlier experiments to focus subsequent tests (and subsequent versions) that are distributed around the world. For example, results of tests run in India can be used to guide the configurations tested in Europe six hours later. We also plan to automate error detection (e.g., via CVS rollback capabilities) and provide tools that will recommend a course of action to human "build czars" if errors cannot be detected automatically.

Current status: Statistical models to support compositional analysis. The compositional analysis techniques that underlie our proposed research depend on state-of-the-art statistical analysis. PI Karr and his group have developed some preliminary Bayesian/network models to support the scenarios for calculating procedure profiles and operational profiles. These models provide a system for modeling a complex multivariate system [CDLS99]. Difficult problems, such as missing or unobserved variables, can be handled readily via this unified framework.

• **Data setup.** Assume that there are N records/sessions, each with some observed characteristics (S) such as the OS and middleware default configuration settings, and some unobserved variables (H), such as some unknown user configuration settings. A usage variable, such as session time (T), is needed to scale the observed counts from each probe location to "normalize" the counts. Also required is a measure (Y) of the performance of the program for a given session (Y). Probes can be placed one of $1, 2, \ldots, M$ locations, each independently with probability p. For each session, the numbers of visits at each of the M locations as reported by the probes are $X_1, X_2, \ldots, X_M$. Some visits will be unobserved if that location was not instrumented in this particular session.

• **The Bayesian model.** The model is a way of specifying the probabilistic dependencies among the variables. This allows us to model the joint distribution of all the variables by just specifying and fitting local conditional distributions and combining them in a systematic manner. Figure 2 illustrates a sample Bayesian network for an example with $M = 4$.

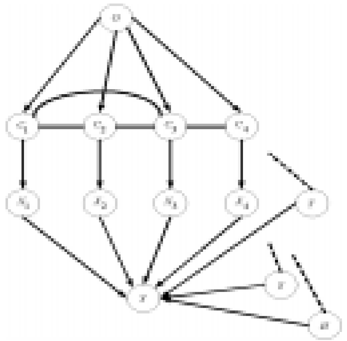

Figure 2: Example Bayesian Network ($M = 4$)

At the top is an unobserved variable G that indicates the user type. The model in Figure 2 assumes we know the number of user types and its distribution. A given user type determines a probability distribution for the data X_i. It will often be easier to model the distribution of X_i via an intermediate layer of unobserved variables (parameters), C_i, that parametrize the distribution of X_i. The interrelations between C_i (the $\{C_i\}$ subgraph) will be specified by domain knowledge. Finally, the X_i and S, H and T determine performance Y, *e.g.*, a simple model like $Y = \sum_i \omega_i X_i + f(S, H, T) + \epsilon$. The model can be made more complex by adding more unobserved variables (e.g., the number of user types) and more edges on the graph.

• **Analysis.** Our principal goal thus far has been to learn the joint multivariate distribution of all variables from the observed data. Techniques we plan to apply in the proposed project include parameter estimation by Maximum Likelihood via the EM algorithm [MK97] and use of Markov chain Monte Carlo [GRS96] to generate samples from the joint distribution. Then, clustering/profiling questions can be addressed from the distribution of $[G|\text{Observed}]$; type-dependent user characteristics can be inferred from the $[\text{User Variables}|G]$ distribution. To evaluate the instrumentation policy (p) we will use a small validation set of fully instrumented programs to examine how diffuse our estimated distribution is (diffuse distributions being less "informative").

1.4 PI Qualifications and Work Breakdown

This project will be conducted by a multidisciplinary research team made up of the following PIs who have extensive expertise in developing, measuring, and optimizing performance-intensive infrastructure software, software testing, source code analysis, program evolution, and statistics.

Alan F. Karr is the director of the National Institute of Statistical Sciences (NISS), an independent research institute whose mission is to identify, catalyze and foster high-impact cross-disciplinary research involving the statistical sciences. He has conducted extensive research at the interface of information technology and the statistical sciences. PI Karr will lead the efforts on developing statistically sound compositional analysis techniques and will assist in all aspects of empirical evaluation.

David Notkin is the Boeing Professor and Chair of the Department of Computer Science & Engineering at the University of Washington. His research and educational expertise is in software engineering, with a particular emphasis on reducing the costs and difficulties of software evolution; he has also worked in software model checking, programming environments, and parallel and distributed systems. PI Notkin will lead the white-box analysis scenario aspects of the project and assist in lightweight instrumentation insfrastructure.

Adam Porter is an Associate Professor of Computer Science at the University of Maryland and the University of Maryland Institute for Advanced Computer Science. His general research interests include empirical methods for identifying and eliminating bottlenecks in industrial-scale software development processes. Specific recent projects regression testing, software inspection, and tools to support distributed software development. PI Porter will lead the efforts on black-box regression scenarios and assist in overall empirical evaluation.

Douglas C. Schmidt is an Associate Professor of Electrical & Computer Engineering at the University of California, Irvine. He served as a Program Manager and Deputy Director of the Information Technology Office (ITO) at DARPA from 6/00 to 6/02. His research focuses on patterns, design and optimization techniques, and empirical analysis of high-performance, real-time distributed object computing middleware. He is also the chief architect for the ACE+TAO software, which are the open-source testbeds for our proposed research. PI Schmidt will lead the efforts on lightweight instrumentation and general performance evaluation. He will assist in the black-box scenarios.

We plan to organize our research around the challenge problems exposed by the six scenarios described in Section 1.3.2. Our research implementation plan will be iterative. During each iteration we will add the tool infrastructure and algorithms needed to further support the scenarios. Consistent with our proposed work in Section 1.3, we will decompose our effort as follows:

- *Task 1.* Develop tool and analysis infrastructure. This is a task assignment shared by all PIs, which is already underway, as described in Section 1.3.3.
- *Task 2.* Refine evaluation infrastructure, including preparing subject programs and designing underlying experiments (PIs Karr and Porter are leads).
- *Task 3.* Perform and empirically evaluate the white-box analysis scenarios (PIs Nodkin and Porter are leads).
- *Task 4.* Perform and empirically evaluate the black-box analysis scenarios (PIs Schmidt and Porter are leads).

We anticipate that the first iteration of these efforts will comprise the first 18 months the grant. At that point, we will assess our results to (1) refine and augment our tools and analysis techniques, evaluation infrastructures, and white-box and black-box scenarios and (2) reiterate through the tasks outlined above to improve the fidelity of our analyses, methods, and tools. Given the early nature of the work, it would be hard to make an effective prediction of a schedule for the out-years.

Over the past six months, the PIs have worked closely together on the planning and feasibility studies for the proposed project. We have made progress through the use of extensive email interactions, weekly phone calls (including several of the students already involved in the project), and with numerous face-to-face meetings in Washington DC and at various conferences and workshops. To facilitate interactions between the research groups run by each PI we will perform the following activities during the period of the grant:

1. Conduct monthly joint group research planning meetings via Netmeeting to discuss technical objectives and present interim results.
2. Create technical working groups consisting of small groups of PIs and their students who are addressing complementary technologies in order to determine shared representations, tools, and external interfaces. These working groups will collaborate via email and Netmeetings and will meet annually at ICSE to plan out-year activities.

1.5 Results from Prior NSF Support

As discussed in Section 1.2.1, each PI has held NSF grants in the past five years that have strongly motivated the current proposal. In this section, we describe the results from our earlier NSF support.

Karr: Code Decay in Legacy Software Systems SBR–9529926 and Pilot Projects to Explore Large Data Sets DMS 97–11365. The Code Decay Project, conducted together with Lucent Technologies and PI Porter, quantified, measured, and predicted *code decay*, i.e., the increasing difficulty to change large software systems over time without negative consequences. Contributions include characterization and quantification of code decay [EGK+01a], predictions of software fault incidence [GKMS00], tools for visualization of large-scale data [EGK+01b, EK01] and *Live Documents* that define new modes of interaction between readers of scientific documents and the underlying data [EGKM97]. The Large Data Sets project, conducted with AT&T Research, involved computer scientists, network engineers, and statisticians. It developed methods for detection of network-based intrusion into computer systems [SDJ+01] and analysis of customer behavior for Internet service providers [RBK+00]. Relevant results include techniques for inference from relational databases, scalable algorithms for dealing with data whose massive scale precludes analysis even if all data could be retained, and techniques for detection of data anomalies.

Notkin: Software Reflexion Models CCR–9506779. The reflexion model approach aids software engineers in assessing, planning and performing specific software engineering activities, primarily on existing software systems. Our key results to date include: solidifying and reporting on the techniques and formalisms underlying reflexion models; developing several implementations of reflexion model toolset; exploring the utility of the approach in the context of a variety of application areas; some initial results on the tradeoffs between parsing and scanning to extract information from source code; a lexical approach for extracting particular kinds of information from source code; and consideration of a number of variants on the basic reflexion model concept intended to address complex mappings between models and implementations.

Development of human resources arising from this grant to date include: Kevin Sullivan's appointment to the faculty at the University of Virginia (recently becoming a tenured Associate Professor), Gail Murphy receiving a Ph.D. (now an Assistant Professor at the University of British Columbia); the receipt of a Ph.D. by Michael VanHilst (now at HP Labs); the involvement of an undergraduate (Paul Bugni) in the research; and the (limited) use of the tools in our undergraduate compiler course. The following publications acknowledge this NSF award [MN97, MNL96, MN96, MNGL98].

Porter: Scalable Program-Analysis-Based Testing and Maintenance: Infrastructure and Experimentation CCR–9707792 and Empirical Studies of Large-Scale Regression Testing CCR–0098158. Involved three overall efforts: construction of an extensible experimental infrastructure including a software system and repository of subject systems; development of program-analysis techniques that support testing and maintenance of large- scale systems; and rigorous empirical evaluation of the infrastructure, and of the program-analysis-based testing and maintenance techniques. The principal results of this work are a system for performing large-scale program analysis, empirical results helping to explain when and why these analyses do and don't work, a publicly available infrastructure that supports tool development for program-analysis-based testing and maintenance techniques. Development of human resources arising from this grant to date include: Jung-Min Kim who is expected to receive a Ph.D. in Dec. 2001, The following publications acknowledge this NSF award [GHK+98, KPR00, GHKR01].

Schmidt: Compilation and Automatic Optimization of Network Protocol Implementations NCR–96–28218 and Monitoring, Visualization, and Control of High Speed Networks NCR–97–14698. In these projects, PI Schmidt conducted research on optimizing protocols and middleware for high-performance, real-time object request brokers (ORBs). The focus was on optimizing the efficiency and predictability of the TAO ORB, which is a real-time ORB endsystem designed to meet end-to-end application QoS requirements by vertically integrating distributed object computing middleware with OS I/O subsystems, communication protocols, and network interfaces. TAO was the first real-time ORB endsystem to support end-to-end QoS guarantees over high-speed networks, such as ATM, and embedded system interconnects, such as VME and Fibrechannel. Work in this area has resulted in the Ph.D. dissertations of Aniruddha Gokhale, Chris Gill, Irfan Pyarali, and Carlos O'Ryan. The following is a synopsis of the key research contributions and publications stemming from these projects:

1. An ORB Core that supports deterministic real-time scheduling and dispatching strategies. TAO's ORB Core concurrency models minimize context switching, synchronization, dynamic memory allocation, and data movement. TAO is the first ORB with these capabilities. Papers published on this topic that acknowledge NSF support include [SMFGG01, Sch98].

2. An active demultiplexing strategy that associates client requests with target objects in constant time, regardless of the number of objects and operations. TAO is also the first ORB with these capabilities. Papers published on this topic that acknowledge NSF support include [POS$^+$00, GS98].

3. A highly-optimized CORBA IIOP protocol engine and an IDL compiler that generates compiled and/or interpreted stubs and skeletons, which enables applications to make fine-grained time/space tradeoffs. Papers published on this topic that acknowledge NSF support include [AOS$^+$00, GS99a, GS99b].

4. A Real-time Event Service and Scheduling Service that integrates the capabilities of TAO described above. These real-time services form the basis for next-generation real-time applications for many research and commercial projects, including Boeing, CERN, Cisco, Lockheed Martin, Raytheon, SAIC, Siemens, and SLAC. Papers published on this topic that acknowledge NSF support include [OSN01, GLS01, SLM98, HLS97].

1.6 Broader Impacts

We plan to leverage our proposal research effort to explore ideas in *educational laboratories*. Educational laboratory exercises are a standard part of physical science education. These "labs" require students to learn and apply the scientific method, and examine physical principles. While conducting a lab a student monitors a physical process, gathers and analyzes data about the process, and uses the data to test hypotheses-which often challenge his or her intuition.

Other researchers, e.g., Tichy [Tic98], claim that validation and experimentation in software engineering is generally poor. To rectify this situation, we propose to increase the training that researchers received in performing high-quality experiments. In particular, we propose to integrate experimental methods into the computer science graduate and undergraduate curriculum.

We plan to accomplish this goal by creating a short, e.g., four-week, teaching module in which students learn key technology, perform experiments, collect and analyze data, and test hypotheses as part of their software engineering courses. This particular module will include educational materials for teaching the foundations of software testing and performance evaluation. Of course, this approach can be used for teaching other subjects as well.

These teaching modules support four primary objectives:

1. Teach the fundamental technical issues underpinning testing and performance evaluation
2. Show how experiments can be used to evaluate hypotheses concerning open research issues
3. Teach students to design and conduct experiments to evaluate their own research and
4. Teach basic statistical procedures for collecting and analyzing data from their own experiments.

These modules will take the form of interactive educational laboratories. To construct these labs we will package our empirical studies into lab manuals, each containing the training materials for lectures, reference articles, sample specifications, data collection forms, a description of the experimental procedures, and a post-experiment survey and take-home assignment. After the lab has been performed the instructor collates the data, recoding it to ensure the anonymity of the participants. Next, the hypotheses behind the experiment are fully explained and the students are taught the statistical rationale for the experimental design, and learn statistical procedures for data analysis and hypothesis testing. The final step involves a take-home assignment in which the students are required to propose an experiment to evaluate some hypotheses in which they are interested. We will prepare lab manuals for each of our experiments using the Live Documents technology invented in PI Karr's Code Decay project [EGKM97].