Georgia Institute of Technology – College of Computing — Technical Report #GIT-CC-02-06

Gamma System: Continuous Evolution of Software after Deployment[*]

Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton
College of Computing
Georgia Institute of Technology
{orso|dliang|harrold|rjl}@cc.gatech.edu

ABSTRACT

In this paper, we present the Gamma system—a new approach for continuous improvement of software systems after their deployment. The Gamma system facilitates remote monitoring of deployed software using a revolutionary approach that exploits the opportunities presented by a software product being used by many users connected through a network. Gamma splits monitoring tasks across different instances of the software, so that partial information can be collected from different users by means of light-weight instrumentation, and integrated to gather the overall monitoring information. This system enables software producers (1) to perform continuous, minimally intrusive analyses of their software's behavior, and (2) to use the information thus gathered to improve and evolve their software. We describe the Gamma system and its underlying technology in detail, and illustrate the different components of the system. We also present a prototype implementation of the system and show our initial experiences with it.

1. INTRODUCTION

Developing reliable software is difficult because of software's inherent complexity and because of the limited availability of resources. Many analysis and testing techniques have been proposed for improving the quality of software during development. However, because of the limitations of these techniques, time-to-market pressures, and limited development resources, software products are still being released with missing functionalities or errors. Because of the growth of the Internet and the emergence of ubiquitous computing, the situation has worsened in two respects. First, the widespread use of computer systems has caused a dramatic increase in the demand for software. This increased demand has forced many companies to shorten their software development time, and to release software without performing required analysis and testing. Second, many of today's software products are run in complicated environments: a software product may need to interact through a network with software products running on many other computers; a software product may also need to be able to run on computers that are each configured differently. It may be impractical to analyze and test these software products before release under all possible runtime environments and configurations.

The solutions proposed today to limit the number of failures in the field are mostly based on performing alpha and beta testing of software products. *Alpha* testing is performed by a limited number of users, selected within the organization developing the software, who use the product and report problems they find. Although useful for an early assessment of the software, the effectiveness of alpha testing is limited by the small number of users considered. Conversely, *beta* testing is performed by users selected outside the organization developing the software, through a sampling of the intended audience for the product. Although it addresses a larger set of users, beta testing is still unsatisfactory because it relies only on the feedback obtained from the users, which can be inadequate. Users are not necessarily willing to compile a bug report and send it back to the software producer. (Most often, users simply shutdown and restart the application that failed.) Moreover, even "good" users, who actually take the time to prepare and send bug reports, often send either too little or too much information for the producer to be able to conveniently investigate the problem in-house [10, 18].

In short, software is error-prone due to its increasing complexity, decreasing development time, and the intrinsic limitations of current analysis and testing techniques. Therefore, there is a need for techniques that monitor software's behavior during its lifetime, and enable software producers to effectively find and fix problems after the software is

[*]Patent pending.

deployed in the user's execution environment. Such techniques would let us prevent problems or at least efficiently react when they occur.

One existing research project, Perpetual Testing, targets these needs, and aims at providing "analysis and testing as on-going activities to improve quality assurance without pause through several generations of product, in the development environment as well as the deployed environment [4, 15, 17]." This project has produced an approach, called *Residual Testing* [13], to continuously monitor for test coverage. This approach monitors for test obligations that were not fulfilled in the development environment so that they can be fulfilled during actual use. Residual Testing proposes an interesting approach to the problem of continuous monitoring after deployment, but addresses the problem only partially.

A first limitation of Residual Testing is that it does not address the problem of how to reduce the instrumentation overhead cost; the approach, even with its reduced cost, may still require significant instrumentation and therefore be too expensive to be applied to real systems. Moreover, Residual Testing implicitly requires interaction between software users and producers, but no details are provided on how such interaction takes place. In particular, no approach is considered to collect monitoring information in the field and no techniques have been reported that can either use the monitoring information to help uncover problems or modify the software to change the instrumentation or update it. Finally, Residual Testing cannot be applied to generic monitoring tasks; for the approach to be applicable, a considerable part of the monitoring must be completed in-house, before releasing the product.

To address the need for continuous analysis and improvement of software products after deployment in a general way, we developed a new approach to software monitoring—the GAMMA system. The GAMMA system is based on two core technologies:

Software Tomography. (based on sparse sampling and information synthesizing). This technology (1) divides the task of monitoring software and gathering necessary execution information into a set of subtasks that require only minimal instrumentation, (2) assigns the subtasks to different instances of the software, so that none of the instances will experience significant performance degradation due to instrumentation, and (3) integrates the information returned by the different software instances to gather the overall monitoring information.

Onsite code modification/update. This technology enables modification or update of the code on the users' machines. This capability lets software producers dynamically reconfigure the instrumentation to gather different kinds of information (e.g., to further investigate a problem) and to efficiently deliver solutions or new features to users.

The main contributions of the paper are:

1. A new approach to software monitoring that lets software producers (1) perform continuous, minimally intrusive analysis and testing of their software in the field, and (2) use the information thus gathered to respond promptly and effectively to problems and to improve and evolve their software.
2. A prototype system that we developed and that implements our new approach for monitoring.
3. A case study that uses the prototype system to assess of the feasibility of the approach.

The rest of the paper is organized as follows: Section 2 presents an overview of the technology underlying the GAMMA system; Section 3 describes the GAMMA system; Section 4 describes the current implementation of the system and our initial experience with it; Section 5 describes open issues; finally, in Section 6, we draw some conclusions and describe ongoing research.

2. GAMMA CONCEPTS

The goal of the GAMMA system is to let software developers monitor the execution of their software after its deployment. Such monitoring can be used to gather information to detect, investigate, and solve problems that occur when the software is in use. The problems include errors, incompatibility with the running environment, security holes, poor performance, poor usability, or failure of the system to satisfy the users' needs.

There are several requirements for such a system to be accepted, from both the users' and the software developers' viewpoints. From the users' viewpoint, the execution monitoring should be low-impact, so that the monitored software executes with no noticeable performance loss; also, the execution monitoring should be minimally intrusive, so that monitoring does not affect users' normal usage of the software.

From the software developers' viewpoint, the system should be general, flexible, and should facilitate easy reconfigurability, to let the developers monitor and collect information for a broad range of tasks.

To satisfy these requirements, we developed a system based on two main technologies: *software tomography* and *onsite code modification/update.* In the rest of this section, we illustrate these two technologies.

2.1 Software tomography

The GAMMA system performs monitoring of the execution of software. Traditional monitoring approaches are based on instrumentation and insert all the probes that are needed for monitoring into each instance of the software. For most monitoring tasks, this approach requires inserting probes at many points in the software, significantly increasing

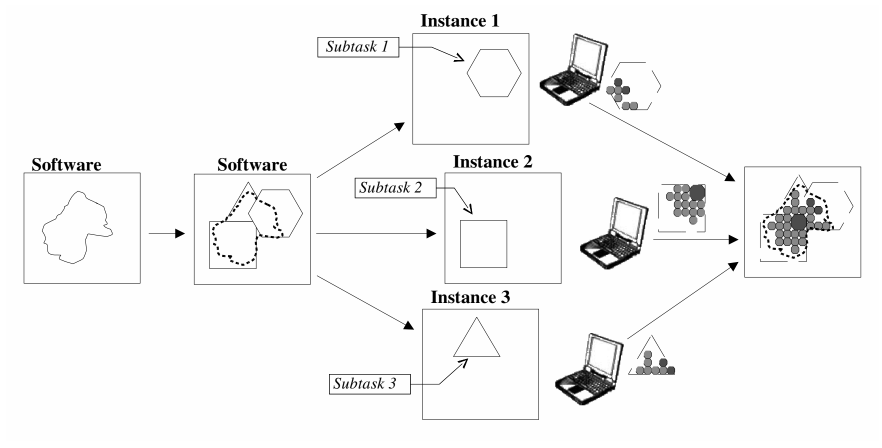

Figure 1: Software tomography.

its size and compromising its performance. Such an approach is unlikely to be accepted by the software users. For example, monitoring for statement coverage may require inserting probes in many basic blocks in the program, which in general results in an overhead unacceptable for the user.

To accommodate users' needs, that is, to achieve low-impact and minimally-intrusive software monitoring, we developed a new technique. Our technique divides the monitoring task into a set of subtasks, each of which involves little instrumentation, and assigns these subtasks to individual software instances for monitoring. The technique then synthesizes the information collected from the different software instances, and obtains the monitoring information required for the original task.

For example, consider monitoring for data-flow coverage; in such a case, the subtasks might monitor coverage for a subset of the definition-use pairs. Every software instance would then be assigned a subtask and instrumented accordingly; by integrating the information reported by the software instances, we could then estimate the overall coverage achieved by executions at the different user sites.

Figure 1 presents, in an intuitive way, this new monitoring technique. In the figure, the leftmost box represents the software to be monitored; the cloud shape within the box indicates the monitoring information we want to collect—the monitoring task. The geometric shapes in the adjacent box on the right represent the monitoring subtasks; the figure shows how the combination of the information provided within such tasks includes the required monitoring information. The boxes labeled Instance 1, Instance 2, and Instance 3 represent the instances of the software running on different user sites. Each instance performs one of the subtasks, and sends the information back to the producer. The figure also shows how the information sent back by the software instances is integrated to provide the monitoring information for the original task.

We call our new monitoring technique *software tomography.* We use this name because our monitoring technique is analogous to *Computer-Aided Tomography* (CAT), which scans an anatomical structure to obtain a set of partial images from different angles or different locations, and then reconstructs the three-dimensional model from the images.

Software tomography distributes the monitoring overhead cost among many instances of the software. To use software tomography for gathering information for a specific monitoring task, we must perform a set of steps. First, we determine what information is needed for the task. Then, we partition the original task into a set of subtasks, determine the instrumentation required for each subtask, and assign the tasks to the different instances. Finally, we instrument the software instances by inserting probes in various program points. To perform these steps, we must address many issues: identification of basic subtasks, assignment of subtasks to instances, and optimization of number and placement of probes.

2.1.1 Identification of basic subtasks.

When performing software tomography for a given task, one important issue is how to partition a task into basic subtasks so that information collected for the individual subtasks can be integrated to produce the information required for the overall task.

Some simple tasks, such as monitoring for statement coverage, can be easily partitioned into minimal subtasks—each of which monitors one statement in the program. In this case, instrumenting for a given subtask simply consists of inserting a single probe for the statement associated with this subtask, and the coverage for the entire program can be assessed through a simple union of the information for each subtask.

More complex monitoring tasks may require that each subtask monitors more than one point in the program. For example, for data-flow coverage, each subtask must monitor for at least one data-flow association, which requires inserting probes for the definition, the use, and all possible intervening definitions. Nevertheless, also for this kind of task, the basic subtasks can still be easily determined. In the case of data-flow coverage, for example, the basic subtasks are the data-flow relationships in the program.

Yet other tasks, such as monitoring for memory-access information, may require even more complicated techniques to identify the basic subtasks. In fact, these tasks usually require recording execution facts that occur at several program points. Therefore, each subtask may require that probes be inserted in several points in the program. Moreover, each subtask may also need to collect a large amount of information at these program points. This kind of tasks may require static program analysis techniques to identify the basic subtasks.

Finally, other tasks may seem unsuitable for software tomography. Consider the case of mutation analysis [3]. Each mutant contains a deliberately-injected fault, which may result in a failure in the field. Clearly, software instances that contain these faults are unacceptable to the users. However, if we limit the use of mutation analysis to only a part of the software, we may still be able to use software tomography for this task. For example, we could restrict the mutation analysis to a single procedure p. In this case, we could provide a software instance with both the original and the mutated procedure (p_o and p_m, respectively), and with a new version of procedure p that, when invoked, executes both p_o and p_m, compares their results, and reports the results of the comparison.[1] This form of mutation is a type of *weak* mutation, which is performed without requiring complete execution of the program.

It is worth noting that software tomography, if suitably exploited, can enable monitoring tasks that require too much overhead even for their in-house application. For example, we may be able to collect path-coverage information using software tomography on a large enough number of software instances.

2.1.2 *Assignment of subtasks to instances.*

Another important issue involved in software tomography is how to assign subtasks to instances after the subtasks have been identified. Software instances are executed by users at will. Therefore, the frequency of execution may vary from instance to instance and from time to time. To ensure that we gather enough information for each subtask, so that the overall monitoring information is meaningful, we may need to assign a subtask to more than one instance. (Clearly, the higher the number of instances responsible for the same subtask, the more accurate the information that can be gathered for the subtask, but also the greater the number of software instances that are needed to assign all subtasks.) To address this problem, we defined two main approaches.

The first approach uses feedback information from the field to tune the instrumentation process. When the system is instrumented for the first time, the subtasks are evenly distributed among the different instances. Under the assumption that we will have more instances than subtasks, the assignment of the subtasks will result in each instance being assigned only one basic subtask. (The case of the number of instances being less than the number of subtasks is addressed at the end of this section.) When these instances are deployed, we use dynamic data on the usage of the instances to ensure that each subtask is assigned to at least one frequently-used instance. This kind of dynamic information can be easily collected by using a lightweight instrumentation and having the different software instances report basic usage information. For example, each instance could contain a single probe that records how often the instance is executed, and the information could then be sent back to the producer site.

When we have enough historical data about the execution of the single instances for such data to be statistically meaningful, we analyze the data and identify which subtasks have to be reassigned. If $freq(inst_i)$ is the frequency of execution of a given instance $inst_i$, and $assigned(st_j)$ is the set of instances to which subtasks st_j is assigned, we can compute, for each subtask, its *estimated execution frequency* (*EEF*) as:

$EEF(st_j) = \sum_{inst_i \in assigned(st_j)} freq(inst_i)$[2]

We can then normalize the range of EEF and select two set of subtasks that must be reassigned: the set of subtasks whose EEF is below a given threshold, $STLOW$, and the subtasks whose EEF is above a given threshold, $STHIGH$. Through onsite code updating (see Section 2.2) we can update the instrumentation so that instances responsible for subtasks in STSLOW are assigned subtasks in STHIGH, and instances responsible for subtasks in STHIGH are assigned subtasks in STSLOW. For a trivial example, consider the case of a subtask st_a associated with a frequently-executed instance i_1, and a subtask st_b associated to a seldom-executed instance i_2. In such a case, st_a would be reassigned to i_2, and st_3 would be reassigned to i_1.

To increase the efficiency of the approach, we can also use historical data (if available) to optimize the initial assignment of tasks to instances, and then use the dynamically-gathered information to fine-tune the distribution of subtasks.

The second approach is also based on updating the code of the different instances. Unlike the first approach, however, in this case we change the assignment of the subtasks and re-instrument the software instances from time to time, without using any feedback from the field. For example, a given subtask may be assigned to a different instance

[1] In doing so, we must first make sure that calls to p_o and p_m have no side effects. Otherwise, the portion of the state affected by the side effects must be saved and then suitably compared and restored.

[2] Note that this is just an estimated frequency; the fact that an instance assigned to a subtask is executed does not imply that the part of its code related to the subtask is actually exercised.

every given amount of time, using a round-robin-like approach. The subtasks to be reassigned are the subtasks not adequately accomplished.

This latter approach is more suitable for cases in which the different instances are not always accessible, and therefore the collection of information about their frequency of execution may be problematic. An example is the software running on a palm computer that is mostly disconnected and provides only limited space for storing information locally.

For both approaches, if there are fewer software instances than subtasks, then more than one subtask must be assigned to the same instance. In this case, we may instrument the software instances to monitor all the assigned subtasks at the same time. However, if doing so results in unacceptable execution overhead, we may use a *time-sharing* approach: a software instance si responsible for a set of subtasks S is instrumented to monitor only for a subset of subtasks $S' \subset S$ at a time, and the subtasks in S' are updated every given amount of time. Note that the updating of S' is orthogonal to the updating of the subtasks assigned to si.

2.1.3 Optimization of number and placement of probes.

A third issue is how to optimize the number and the placement of probes for monitoring a task. This optimization can further reduce the instrumentation overhead for the software instances. For many tasks, static information extracted from the software can be used to reduce the number of probes needed for a specific subtask. For example, if a data-flow relation is such that the execution of the use implies the execution of the definition, then the coverage information for this data-flow relation can be inferred from the coverage information of the statement containing the use. Therefore, we can monitor this data-flow relation using only one probe at the use statement. For more complicated cases, more sophisticated static-analysis techniques may be needed.

For many tasks, static information can also be used to optimize the placement of the probes. Such optimization can further reduce the number of probes. For example, for path profiling, we use an existing technique by Ball and Larus [2], which can reduce the number of probes needed for the task. The technique instruments a program for profiling by choosing a placement of probes that minimizes run-time overhead. We have adapted this technique so that it determines the placement of probes that monitor only the desired paths instead of all acyclic paths in the procedure or program.

When more than one subtask must be assigned to one software instance, the way subtasks are grouped may affect the number of probes needed. Therefore, in these cases, optimizing subtask grouping may let us reduce the instrumentation cost. For example, in a data-flow coverage task, if two subtasks are monitoring two data-flow relations that occur along the same path, then assigning these two subtasks to the same instance may enable sharing of some of the probes.

2.2 Onsite code modification/update

Software tomography provides a low-intrusive way for the software producer to collect information from the execution of a software product at the user's site. To use such technology for software maintenance and evolution, a software producer may require the capability of gathering different or more detailed monitoring information from time to time. For example, consider monitoring for structural coverage. We may want to start monitoring at the procedure level. If an instance behaves incorrectly,[3] we can then start monitoring at the statement level to get more precise information. In this case, we need to modify the instrumentation of such instances to gather statement coverage, rather than procedure coverage.

Furthermore, to evolve and maintain the software more effectively, the software producer may also require the capability of deploying updates to the users' sites with minimal users' involvement.

We propose to accommodate these requirements using *onsite code modification/update.* Onsite code modification/update can be implemented using a wide range of techniques. At one end, the software producer can send the updates through regular software distribution channels, such as by regular mail, by e-mail, or by posting the updates on ftp sites. These techniques require the users to participate actively, and therefore may not be effective or appropriate for some monitoring tasks. At the other end, the software producer can connect to the user's site and update the software directly. These techniques do not require the users to participate. However, they may require sophisticated infrastructure support.

Existing work has proposed various techniques for updating a component within a software system even when the program is being run (e.g., [6, 8]). To perform onsite code modification/update, we can leverage these existing techniques by selecting and adapting them based on the specific context in which we are operating.

More precisely, the way we perform code modification/update is highly dependent on several dimensions: (1) the type of connection between the platform where the software is deployed and the producer's site, (2) the characteristics of the platform on which the software executes, (3) the characteristics of the software, and (4) the monitoring task we are considering.

[3]Here we assume that we have a way of identifying "abnormal behaviors" in the software. Details on how we can obtain this information are provided in Section 5.

Connection type. We distinguish different kinds of sites based on how often they are connected to the Internet.[4] This dimension defines how much we can rely on onsite updating. For sites that are always on-line, such as desktop computers connected to a network, or mostly on-line, such as cellular phones, we can easily perform direct on-line updates. Sites that are mostly offline, such as palm computers occasionally connected for synchronization, require mechanisms to handle situations in which an update is required, but a connection is not available. These mechanisms include buffering of changes and polling for user's on-line availability, as well as user-driven updates at the time of the connection. As far as the connection is concerned, we must also consider the available bandwidth, and distinguish among connections with different speeds. Obviously, the bandwidth constrains the amount of information that can be sent from the producer to the user. Low-speed connections may require the use of sophisticated techniques to optimize the size of the updates sent to the users, whereas for high-speed connections the size of the updates is in general not an issue.

Platform characteristics. If the connection type permits on-line and onsite updating, an adequate infrastructure must be available on the users sites for the mechanism to be activated. The difficulty involved in the development of such an infrastructure is highly dependent on the characteristics of the platform addressed. Some platforms may provide facilities for code updating at the operating-system level, and require little additional infrastructure. Some other, more primitive platforms, may require a very sophisticated infrastructure for the code updating to be usable. Yet other platforms, such as mobile phones or not-so-powerful palm computers, may have resources so limited that no infrastructure can be provided—in these cases, updating can only be performed off-line, despite the connectivity of the platform.

Software characteristics. When identifying onsite code-updating techniques, we distinguish software systems based on the level of continuity of their execution. Software that is continuously running, such as an operating system, a web server, or a database server, can only be updated using hot-swapping mechanisms. Software that is not continuously running, but may execute for a long period of time, such as a mail client or a web browser, may require either hot-swapping mechanisms or techniques for storing the updates locally, until the application terminates its execution. Finally, software that executes for a short period of time, such as simple console commands, does not impose in general specific requirements on the updating mechanism.

Monitoring task. The first characteristic of the monitoring task that affects the way we perform onsite code modification is the "locality" of the instrumentation required for the subtasks composing the task. Monitoring that involves only "local" instrumentation, such as instrumentation for coverage of an intraprocedural path, can be modified through changes affecting only a small portion of the code; updating information for this kind of monitoring is in general limited in size and therefore suitable for on-line code modification. On the other hand, monitoring tasks that require widespread instrumentation, such as instrumentation for coverage of interprocedural definition-use pairs involving several procedures and several alias relations, require changes in several different places of the code; this kind of updates may be considerably large, and may require efficient techniques to reduce their size before sending them to the users. The second characteristic of the monitoring task that affects the onsite code modification mechanisms used is the estimated frequency of updates. Tasks that involve frequent updates require efficient modification mechanism, such as on-line hot swapping, whereas tasks characterized by unfrequent modifications do not generally constrain the update mechanism used.

In Section 4, where we present our prototype implementation of the GAMMA system, we describe the infrastructure that we developed for onsite code modification/update.

3. THE GAMMA SYSTEM

In this section, we describe our GAMMA Testing and Analysis System (in short, GAMMA system), which uses software tomography and onsite code modification/update to support software evolution and maintenance. Figure 2 shows the four major components of this system and the process of evolving and maintaining software using the system. In the figure, rounded boxes represent the actors (i.e., the software producer and the users), rectangular boxes represent components of the GAMMA system, and ovals represent software development activities. In the following, we first discuss the components of the system and then discuss the process for using the system.

3.1 Components of the GAMMA System

Because the different components can be developed in several different ways, in this section we provide only a high-level description. A more detailed description of the GAMMA components is provided in Section 4.

[4]In general, the connection does not necessarily have to be to the Internet. The only requirement is that the producer site and the users' sites are connected through some kind of network.

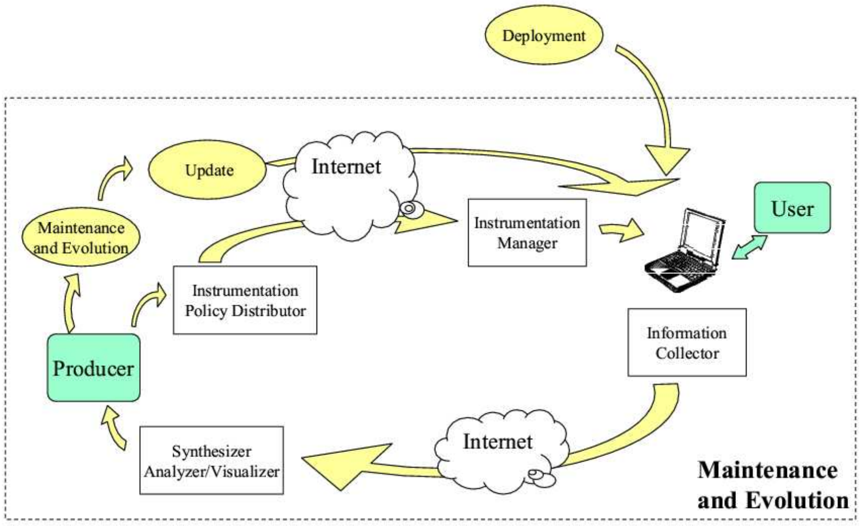
Deployment
Update
Internet
Maintenance and Evolution
Instrumentation Manager
User
Instrumentation Policy Distributor
Producer
Information Collector
Internet
Synthesizer Analyzer/Visualizer
Maintenance and Evolution

Figure 2: The process of using Gamma system for maintenance and evolution.

3.1.1 Instrumentation Policy Generator (IPG)

Given a monitoring task and a set of instances of the software product to monitor, IPG divides the task into a set of subtasks, assigns the subtasks to individual instances, and creates an instrumentation policy for each instance. An *instrumentation policy* describes what instrumentation should be added to an instance for collecting information. For a simple example, consider a task that requires monitoring the coverage for three procedures (p_a, p_b, and p_c) in a software system that has five instances (i_1 through i_5). In such a case, IPG divides the task into three subtasks, each of which monitors one specific procedure. IPG then assigns the monitoring of at least one procedure to each instance (e.g., $p_a \rightarrow \{i_1, i_4\}$, $p_b \rightarrow \{i_2, i_5\}$, and $p_a \rightarrow \{i_3\}$) and creates instrumentation policy for each instance accordingly.

3.1.2 Instrumentation Manager (IM)

This component instruments a software instance according to the instrumentation policy specified for the instance. For the procedures-monitoring example considered above, IM would instrument i_1 by inserting a probe at the entry point of procedure p_a, i_2 by inserting a probe at the entry point of procedure p_b, and so on. IM can also remove previously inserted instrumentation from a software instance. In Figure 2, we show IM as one module that resides on the user's site. This configuration allows IM to add the instrumentation to the software instance or to remove the instrumentation from the software instance quickly. However, in systems in which the resources on the user's site are limited, such as hand-held devices, IM may have to be split in two submodules, IM_p and IM_u: IM_p resides on the producer's site and performs most of the IM's computation, whereas IM$_u$ resides on the user's site and acts as a very light-weight agent for IM_p.

3.1.3 Information Collector (IC)

This module collects the information output by the probes when a software instance is executed and sends it back to software producers for analysis. IC may use different technologies to communicate with the producer's site, depending on user's execution environment (e.g., the operating system, type of network connectivity, available bandwidth). The environment may also affect the frequency at which such communication occurs: for systems that are always connected, the monitoring information may be sent back after every execution or every given amount of time; for systems that are only at times connected, the monitoring information must be stored locally to be sent back the next time the system is on-line.

3.1.4 Synthesizer Analyzer/Visualizer (SAV)

This component synthesizes and analyzes the information sent back from the IC components located at the various user sites, presents the resulting information to software producers, and assists software producers in discovering and investigating problems. Many software analysis and visualization techniques have been developed for using information collected during execution to facilitate tasks such as program understanding and fault localization (e.g., [9]). SAV employs these techniques to help software producers to utilizes the possibly large amount of data that are collected from software instances. SAV may also be split in two submodules, SAV_p and SAV_u: SAV_p resides on the producer's site and SAV_u resides on the user's site. In this case, SAV_u performs some analyses at the user's site, before sending back the data, and may either react promptly to problems that it detects during execution, or send back pre-processed and "filtered" data. This configuration provides a very flexible error handling mechanism.

3.2 Process of Using the GAMMA System

After the software is deployed, the software producers can start monitoring the execution of the software using the GAMMA system. For a given task, they use IPG to identify the subtasks and distribute the corresponding instrumentation policies to the IM components on the different users' machines. Each IM instruments the software instance according to the policy received. After the software is instrumented, when it is executed by the users, execution information is collected by the IC component and sent back to the SAV component, which synthesizes and analyzes the information from the different sources.

SAV presents the information to the software producers through a set of effective visualization tools. Based on the information, software producers can then decide to (1) continue to monitor the software in the same way, (2) reconfigure the software instances for further investigation, or (3) modify and evolve the software.

As shown in Figure 2, the process involves two cycles, *incremental monitoring* (the inner cycle) and *feedback-based evolution* (the outer cycle), which also represent key features of our approach.

Incremental monitoring. Software producers can interact with the software instances to collect more information. Such interaction provides an efficient way of locating problems in the software. Incremental monitoring lets producers investigate problems directly in the field, without having to recreate the user environment in-house, which is often difficult and expensive, and sometimes impossible.

Feedback-based evolution. In the process, software is maintained and evolved based on the results of the continuous monitoring. Because the information gathered through monitoring reflects, and is highly dependent on, the way users use the software, it is more likely for the software evolution to fit users' needs. For example, features reported as most commonly used will be fixed or improved before features reported as rarely used, thus maximizing users' satisfaction.

4. EXPERIENCE WITH THE SYSTEM

In this section, we describe the current implementation of the GAMMA system and a case study we performed using the system.

4.1 Current implementation

We have implemented a first prototype of the GAMMA system. The prototype is targeted to Java code, is written in Java, and provides software-tomography and onsite-code-modification/update capabilities. We illustrate how we implemented the different modules described in the previous section.

The IPG module is currently able to define policies for monitoring structural coverage at different levels: statement coverage, method coverage, and class coverage. IPG inputs Java bytecode, information on which task the producer wants to perform, and number of instances involved, and output a description file that contains the policy definition for each instance. The current IPG implementation is still in a preliminary stage and it does not use any additional information from the field (e.g., historical data on the frequency of execution of the different instances) to improve the policy definition. Therefore, it simply divide the task evenly among the different instances, as described in Section 3. To facilitate experimenting with incremental monitoring, the producer can provide additional input to IPG to force the assignment of some subtasks to some specific instances. In this case, IPG first assigns subtasks as specified by the producer, and then assigns the remaining tasks (if any) among the instances in the usual way (i.e., by balancing the number of subtasks assigned to each instance).

We split the IM module in two parts: a producer-site component (IM_p) and a user-site component (IM_u). For each instance, the *producer-site component* inputs the instrumentation policy generated by the IPG module, suitably instruments the instance at the bytecode level, and sends the instrumented parts of the code to the user-site component. The producer-site component instruments the code for class and method coverage using Soot [16]. By using Soot capabilities for bytecode rewriting, we insert probes at the methods' entry point—we instrument all methods of a class for class coverage, and a single method for method coverage. Instrumentation for statement coverage is performed using a slightly modified version of Gretel [12], which facilitates instrumenting only a subset of the statements in the program.

There is a *user-site component* for each instance, residing on the user site. This component is responsible for communicating with the producer-site component, receiving updates, and deploying the updates on the user site. To deploy the updates, the component uses a hot-swapping mechanism based on the use of wrappers that perform forwarding of the messages to and from the swappable objects. The use of this mechanism allows for dynamically substitution of classes and object at run-time (if needed). In case hot-swapping is not necessary, the mechanism can also be used to perform off-line updating. For the sake of brevity, details of the hot-swapping technique are provided elsewhere [11].

We implemented the IC module as an in-memory data structure that is added to the instances by the IM module and gets updated during execution. We followed this approach to avoid writing to the file-system, which may be disallowed in some cases, and because the monitoring information collected for the tasks considered so far requires little space for storage. Every given amount of time, and when the program terminates, the monitoring information is sent back to the producer site, either via TCP sockets using an ad-hoc protocol, or via e-mail (i.e., using the SMTP protocol)—to account for the cases in which the system is behind a firewall that blocks other kinds of network connections).[5] To implement the communication layer, we used the standard Java library.

The SAV component is composed of (1) a network layer, that listens for communications from the IC modules at the different sites, (2) a database, where the monitoring information is synthesized and stored, and (3) a visualization tool that shows to the producer the monitoring information in a graphical fashion and allows the producer to analyze such data.

The network layer, analogous to the network layer of IC, is developed using standard Java libraries and can receive communications that use either the ad-hoc protocol mentioned above or SMTP.

The database gets the data sent through the network layer, synthesizes them by aggregating coverage information from different sites, and stores the information.

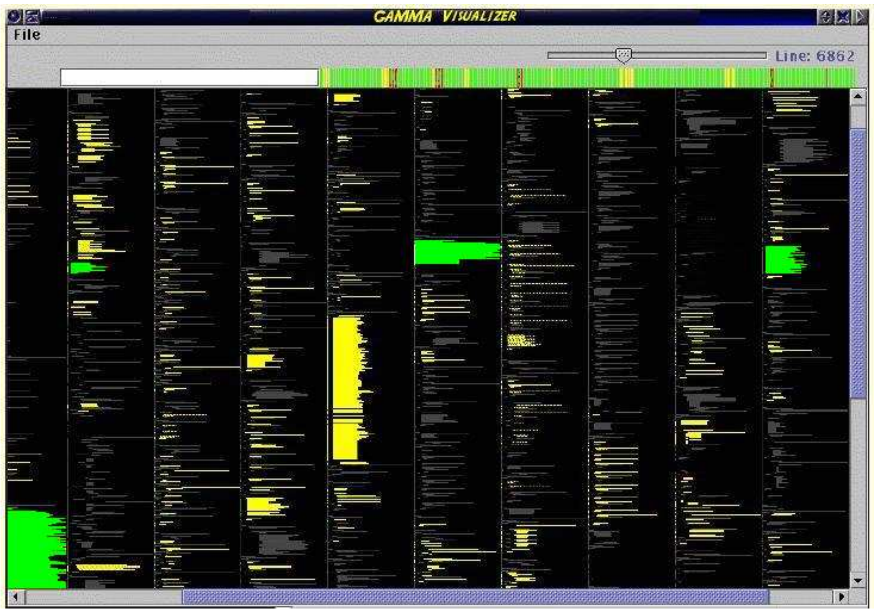

Figure 3: Visualization tool for Gamma.

The visualization tool shows in real-time the information in the database using a SeeSoft [5, 9] view of the program monitored, and coloring the different entities in the code based on coverage information. For example, for statement coverage, lines not covered are colored in gray, lines covered only once are colored in yellow, and lines covered more than once are colored in green (see Figure 3).[6] The visualization tool also lets the producer set guards and alarms, so that a notification is issued, or an action is performed, when the coverage reaches a given threshold. In particular, this mechanism can be used to react promptly to the monitoring information. For example, we may disable instrumentation for a given statement when that statement has been covered, or enable monitoring at the statement level within a given method after the method is reported as covered.

4.2 Empirical evaluation

To assess the system, we performed a case study using JABA [1], an analysis tool developed by some of the authors, as a subject program. JABA consists of 419 classes and approximately 40KLOC. We monitored for three tasks—statement coverage, method coverage, and class coverage. For each of these tasks, we (1) generated the instrumentation policies, (2) used the IM$_p$ module to build the instrumented version of the instances, based on the instrumentation policies previously defined, and (3) released the different instances to the (nine) students working in our group.

As students used their version of JABA for their own work, the probes reported the corresponding coverage information back to a server that gathered and analyzed the coverage information.

[5]Here we assume that user sites are always connected. If this is not the case, and the software does not run continuously, storing the monitoring information on a persistent media is unavoidable.

[6]If the paper is viewed or printed in black and white, there is a color version of the paper available at `http://www.cc.gatech.edu/aristotle/Publications/`.

We performed three case studies:

Residual coverage. In this study, we used the GAMMA system to address the problem of reducing the instrumentation by eliminating probes for those parts of the code already covered, as described in Reference [13]. Using GAMMA, we have been able to push the idea further: we started with a full instrumentation (split among the different instances using software tomography) and dynamically eliminated instrumentation of the parts of the code that got covered during execution (using onsite updating). We performed the study at the class level. When a class was covered on an instance, we disabled the instrumentation for that class in all instances. In this way, as the students used the software, less and less classes remained instrumented.

Incremental monitoring. In this study, we used the GAMMA system to perform incremental monitoring. Like the previous study, we instrumented the subject for class monitoring and let the students use the system. As soon as a class was reported as covered in a given instance, we updated the code in that instance to collect method coverage for the methods in the class. Analogously, as soon as a method was reported as covered in a given instance, we updated the code in that instance to collect statement coverage within that method. In this way, we have assessed that GAMMA can be effectively used to perform incremental analyses, by starting with a coarse-grained monitoring and refining it along the way.

Feedback-based instrumentation. In this study, we investigated the possibility of implementing a strategy for optimizing the assignment of subtasks to instances. More precisely, we implemented the strategy described in Section *2.1.2*. To this end, we instrumented for method coverage (as usual, by evenly distributing the method monitoring among the instances). We also instrumented the `main` method of all instances to be able to collect information on the frequency of execution of the different instances. By performing these two simple steps, both supported by the GAMMA system, we have been able to put the strategy into action and to improve the distribution of the monitoring subtasks.

Although preliminary in nature, the results of these studies are encouraging, in that they provide evidence of the feasibility of the approach. In fact, the studies show that the GAMMA system, through software tomography and onsite updating, lets us perform light-weight monitoring, at least for the initial set of tasks considered, and therefore motivate further research.

5. OPEN ISSUES

Even though we used a general approach when defining GAMMA and its underlying technology, our experimentation with the system is still preliminary. There are several issues that we are aware of and have not addressed yet or have only marginally addressed. Moreover, we expect several, mostly practical, issues to arise as we continue our experimentation with the system and evolve the technology.

5.1 Oracles

So far, we have not addressed an important issue related to the use of the GAMMA system for testing. When performing "traditional" testing of a program (i.e., testing of the program in-house, before releasing it), we run test cases composed of the input to the program and of an expected output. By comparing expected and actual output, we can assess whether a given test case passes or fails for the program. In this context, coverage information can be used to assess the adequacy of the testing performed: if the coverage level achieved by our test cases meets our coverage goals, and the programs behaves correctly for those test cases, we can considered the program as adequately tested (according to the coverage criterion considered). Moreover, when the program fails, coverage information can also be used to guide the debugging process, by limiting the area in which to search for the fault.

When collecting coverage information through remote monitoring, we have no information about the program's behavior, that is, whether it executes correctly or failures occur. To effectively use coverage information collected in the field for testing, we must be able to gather information about the program's behavior. To this end, we plan to combine several approaches. First, although we do not rely entirely on "good" users, we expect a percentage of the users to actually send bug reports containing information that we can use to relate failures to monitoring information. Second, we plan to automate part of this process by trapping failures, such as program crashes, and possibly using assertions for sanity checks. Third, we plan to further investigate the automatic identification of possible failures by using mechanisms that compare programs' behavior signatures (e.g., program's spectra [2, 7, 14]) computed in the field with signatures computed in-house.

5.2 Information Composability

In our preliminary experience, we have found that it is feasible to perform some monitoring tasks by introducing only a limited number of probes, using software tomography. For the cases considered, the identification of basic subtasks, the assignment of subtasks to instances, and the optimization of number and placement of probes do not involve complex analysis. However, other monitoring tasks may require advanced analysis of the code. For example, we are currently investigating the use of software tomography for acyclic path profiling, using the approach presented in Reference [2]. From what we have found, the number of probes to be assigned to each program and their placement is a non-trivial problem, due to possible dependences among the different paths in the program. More generally, dependences among the entities we are monitoring limit the composability (and thus de-composability) of the monitoring information, and therefore complicate the use of software tomography for these kinds of tasks. We plan to address these issues using program analysis techniques. In particular, we are investigating how to statically identify subsumption relations among different entities in the program that let us reduce the number of probes and optimize the assignment of subtasks.

5.3 Scalability

One of the strengths of the GAMMA system is its ability to split monitoring tasks across different instances of the software, thus leveraging the presence of several users connected through a network. So far, we addressed only a small number of users. Interacting with a possibly high number of users may raise scalability issues unforeseen in our initial experimental setting.

Scalability problems may arise due to the fact that the producer site must handle a high number of user sites, possibly keeping different kinds of information about the sites. Although there may be, for some tasks, worst-case situations, in which the producer site must keep track of a significant amount of information for every user site, in general we do not expect this situation to occur often. In most cases, we expect GAMMA to require that only a minimal amount of information be stored locally, whereas additional information will be stored remotely, on the user sites.

Other scalability issues may arise, that are related to the communication between user sites and the producer site, when the number of sites grows too large. In particular, we may expect problems related to having too much monitoring information flowing from the users to the producer, and too many simultaneous updates from the producer to the users. Also in this case, we are confident that scalability issues will be manageable. Intuitively, because there is limited monitoring information that can be collected for a given task, we expect the amount of information exchanged per-instance to decrease with the increasing of the number of instances. For example, the more the number of instances, the less the number of monitoring subtasks assigned to each instance (and therefore, the less the amount of information reported by each instance).

Furthermore, even in the case where a worst-case scenario occurs, we will leverage today's available technology to address the aforementioned problems. For example, data-mining and database technologies can be used to address data-management problems, and the use of a distributed architecture can address issues related to network bottlenecks.

5.4 Privacy and Security

As also stated in Reference [13], two very important issues that we must address are privacy and security. Privacy concerns arise in the information collection phase, in which sensitive and confidential information may be sent from the user site to the production site. This is not an issue for the monitoring tasks considered so far, but may become so for different kinds of monitoring. One way to address the problem is to allow the users to verify which information is sent back to the producer site, but this solution may be unacceptable in case of monitoring tasks requiring a fast and frequent communication between producer and users. In these cases, users may be provided with two choices: (1) allow for the gathering of information (which may still be logged and available for off-line checking) and take advantage of the presence of the GAMMA system (e.g., in terms of prompt and customized updates); and (2) forbid the sending of information and therefore be excluded from the monitoring.

Security concerns involve both producer and users. On the user's side, an attacker may tamper with the information sent from the producer to the user (*man-in-the-middle* attack), so to interfere with the modification/update of the code or even deploy malicious code on the user site. An attacker may also *eavesdrop* on the communication between users and producer, to collect possibly relevant information. On the developer site, an attacker may perform man-in-the-middle attacks to send rogue information to the producer, so to affect the monitoring results and the consequent producer's reactions. Another risk for the producer are *denial-of-service* (DOS) attacks, in which the producer site may be flooded with spurious information to exhaust its resources. To address these issues, we plan to apply existing mechanisms, such as private/public key cryptography mechanisms and digital signatures.

6. CONCLUSION

We have presented the GAMMA system, which supports continuous evolution and maintenance of software products based on monitoring after deployment. Although previous approaches exist with similar goals, the GAMMA system is unique in its idea of exploiting high number of users and network connectivity to split the monitoring tasks across different instances of the software. Such splitting, which we call software tomography, allows for gathering monitoring information through light-weight instrumentation. We showed how GAMMA, by combining software tomography and onsite code modification/update, enables the software producer to perform different monitoring tasks and collect useful information on the software from the field.

We also presented our current implementation of the GAMMA system and described our initial experience with the system. Although preliminary in nature, our study let us assess the practical feasibility of the approach for the set of monitoring tasks considered, and motivates further research. The GAMMA system has the potential to change the way we create, test, debug, and evolve software. Moreover, because of the way monitoring-based software evolution can be managed, users will perceive that the software they are using automatically evolves to better serve their needs.

Our ongoing and future work follows three main directions: (1) extending the set of monitoring tasks considered, by selecting tasks with increasing levels of complexity; (2) investigating effective ways of identifying software instances that are behaving incorrectly; and (3) further evaluating the system, by extending the base of users (first through simulation, and then with real users), and by analyzing the performances of the monitored applications for different monitoring tasks and in different conditions.

Acknowledgments

This work has been partially supported by grants to Georgia Tech from Boeing Aerospace Corporation, by National Science Foundation awards CCR-9988294, CCR-0096321, and EIA-0196145, and by the State of Georgia under the Yamacraw Mission. Michal Young provided the tool Gretel, which we used for statement-level conditional instrumentation. Taweesup Apiwattanapong, James F. Bowring, Jim Jones, and Anup Rao provided useful ideas and comments and helped with the implementation of the system.

7. REFERENCES

[1] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `http://www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 46–57. ACM Press, 1996.

[3] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. *Principles of Programming Languages*, 1980:220–233, 1980.

[4] L. Clarke and L. Osterweil. Perpetual Testing Project. `http://laser.cs.umass.edu/perpetualtesting.html`.

[5] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[6] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Transactions on Software Engineering*, 22(2):120–131, Feb. 1996.

[7] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 83–90, 1998.

[8] M. Hicks and J. Moore. Dynamic software updating. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 13–23, N.Y., June 20–22 2001. ACMPress.

[9] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proceedings of the 24th International Conference on Software Engineering*, 2002, (to appear).

[10] Mozilla web site: The gecko bugathon. `http://www.mozilla.org/newlayout/bugathon.html`.

[11] A. Orso, A. Rao, and M. J. Harrold. DUSC: Dynamic Updating through Swapping of Classes. Technical report, Georgia Institute of Technology, January 2002.

[12] Pacemaker Project. GRETEL: An Open Source Residual Test Coverage Tool. `http://www.cs.uoregon.edu/research/perpetual/dasada/Software/Gretel/`.

[13] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proceedings of the 21st International Conference on Software Engineering, 1999*, pages 277–284, May 1999.

[14] T. Reps, T. Ball, M. Das, and J. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European Software Engineering Conference (ESEC/FSE 97)*, pages 432–449. Lecture Notes in Computer Science Nr. 1013, Springer–Verlag, Sept. 1997.

[15] D. Richardson. Perpetual Testing Project. `http://www1.ics.uci.edu/~djr/edcs/PerpTest.html`.

[16] Sable Group. SOOT: a Java Optimization Framework. `http://www.sable.mcgill.ca/soot/`.

[17] M. Young. Perpetual Testing Project. `http://www.cs.purdue.edu/AnnualReports/96/research/perpetual.html`.

[18] A. Zeller. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 2002, (to appear).