Proc. of the ACM International Symposium on Software Testing and Analysis (ISSTA'02), ACM Copyright.

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, which facilitates remote monitoring of deployed software using a new 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.

1. INTRODUCTION

Developing reliable software is difficult because of software's inherent complexity and 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.

[{]Patent pending.

In short, software is error-prone due to its increasing complexity, decreasing development time, and the limitations of current analysis and testing techniques. Thus, 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 deployed in the user's environment. 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. Such techniques would let developers prevent problems or at least efficiently react when they occur.

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, which is based on two core technologies:

Software Tomography, which (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, which 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) the definition of a new technology, namely, software tomography, for partial instrumentation; (2) the use of onsite software updating for dynamically updating instrumentation after deployment; (3) a new approach to software monitoring that leverages software tomography and onsite updating to let software producers (a) perform continuous, minimally intrusive analysis and testing of their software in the field, and (b) use the information thus gathered to respond promptly and effectively to problems and to improve and evolve their software; and (4) a prototype system that we developed and that implements our new approach for monitoring.

2. GAMMA CONCEPTS

There are several requirements for the GAMMA system to be acceptable. The users require the execution monitoring be low-impact, so that the monitored software executes without noticeable performance loss; also, the execution monitoring should be minimally intrusive, so that monitoring does not affect users' normal usage of the software. The develop-

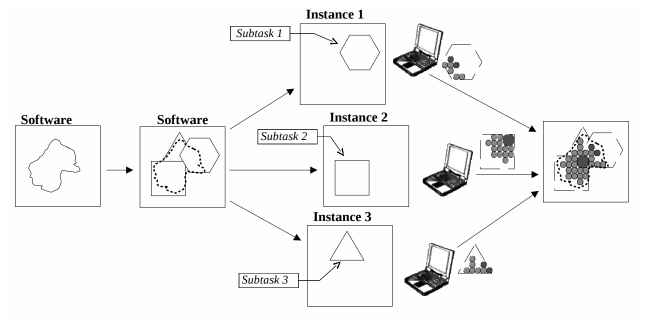

Figure 1: Software tomography.

ers' require the system to be general, flexible, and facilitate easy reconfigurability, so that developers can monitor and collect information for a broad range of tasks.

This section presents the GAMMA concepts, and describes how they meet the users' and developers' requirements.

2.1 Software tomography

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 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 (i.e., to achieve low-impact and minimally-intrusive 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 instances, and obtains the monitoring information required for the original task.

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. Boxes labeled Instance 1, Instance 2, and Instance 3 represent the instances of the software running on different user sites. Each instance performs one subtask, and sends the information back to the producer. The figure also shows how this returned information is integrated to provide the monitoring information for the original task.

To use our new monitoring technique, which we call *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. 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.

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 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. Thus, each subtask may require insertion of probes 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.

Note 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.

Software instances are executed by users at will and different users may use different parts of the software. Therefore, the frequency of execution and the parts of the software exercised may vary across instances and over 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. 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. 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.

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. To increase the efficiency of the approach, we can also use historical data (if available) and static and statistical analysis 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 reinstrument the software instances from time to time, without using feedback from the field. For example, a given subtask may be assigned to a different instance every given time period, using a round-robin-like approach. The subtasks to be reassigned are those 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.

2.1.3 Optimization of number and placement of probes.

Optimizing the number and placement of probes 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. Thus, 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

To use software tomography 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,[1] 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.

The GAMMA system accommodates 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 regular mail, e-mail, or posting 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., [4, 6]). 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: connection type, platform characteristics, software characteristics, and the monitoring task.

Connection type. We distinguish sites based on how often they are on-line. 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. For the connection, 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 sub-

[1]Here we assume that we have a way of identifying "abnormal behaviors" in the software. A discussion on how we may obtain this information is provided in Section 4.

tasks 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 infrequent modifications do not generally constrain the update mechanism used.

3. EXPERIENCE WITH GAMMA

We developed a prototype of the GAMMA system and performed a case study to assess the feasibility of the approach. We provide only a high-level description of the system and of the case study; Reference [8] provides a detailed description.

The GAMMA system is composed of four main components: *Instrumentation Policy Generator (IPG)*, which divides a monitoring task into a set of subtasks and assigns the subtasks to individual instances of the software to be monitored; *Instrumentation Manager (IM)*, which instruments a software instance according to the directives produced by IPG; *Information Collector (IC)*, which collects the information output by the probes when a software instance is executed and sends it back to software producers for analysis; and *Synthesizer Analyzer/Visualizer (SAV)*, which 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.

After the software is deployed, the software producers can use the GAMMA system to start monitoring the software's execution. Based on the information collected, software producers can then decide to (1) continue monitoring the software in the same way, (2) reconfigure the software instances for further investigation, or (3) modify and evolve the software.

The current implementation of GAMMA is targeted to Java code, is written in Java, and provides software-tomography and onsite-code-modification/update capabilities. The IPG module is currently able to define policies for monitoring structural coverage at different levels—statement, method, and class coverage. We split the IM module into 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 using Soot [12] (for method- and class-level instrumentation) and Gretel [10] (for statement-level instrumentation), and sends the instrumented parts of the code to the user-site component. The *user-site component* deploys the updates on the user site using a hot-swapping mechanism [9]. 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. At given time intervals, and when the program terminates, the monitoring information is sent back to the producer site, via TCP sockets. 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 lets the producer analyze such data. The visualization tool shows in real-time the information in the database using a SeeSoft [3, 7] view of the program monitored, and coloring the different entities in the code based on coverage information.

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 task, 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.

We performed three case studies: *residual coverage* [10], in which we used the GAMMA system to reduce the instrumentation by dynamically eliminating probes for those parts of the code already covered; *incremental monitoring*, in which we used the GAMMA system to instrument the subject for class monitoring and, as soon as a class was reported as covered in a given instance, to update the code in that instance to collect method and then statement coverage for the methods in the class; and *feedback-based instrumentation*, in which we optimized the assignment of subtasks to instances using the strategy described in Section *2.1.2*. More details on the case studies are available in Reference [8].

Although preliminary in nature, the results of these studies are encouraging, in that they provide evidence of the feasibility of the approach and motivate further research.

4. OPEN ISSUES

There are issues related to the GAMMA system that we have not addressed or have only marginally addressed.

4.1 Oracles

When collecting coverage information through remote monitoring, we may not have information about the program's behavior—whether it executes correctly. To effectively use coverage information collected in the field for testing, we require information about the program's behavior. To this end, we will combine several approaches. First, 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 will automate part of this process by trapping failures, such as program crashes, and possibly using assertions for sanity checks. Third, we will further investigate the automatic identification of possible failures using mechanisms that compare programs' behavior signatures (e.g., [2, 5, 11]) computed in the field with signatures computed in-house.

4.2 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. In general, 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 will address these issues using program analysis techniques. For example, we are investigating static identification of subsumption relations among program entities that let us reduce the number of probes and optimize subtask assignment.

4.3 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. Nevertheless, we are confident that we will be able to leverage today's available technology to address the aforementioned problems. For example, datamining 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.

4.4 Privacy and Security

Two important issues that we must address are privacy and security [10]. 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. One way to address the problem is to let users verify the information returned to the producer site, but this solution may be unacceptable when monitoring tasks require fast and frequent communication between producer and users. In these cases, users can have two choices: (1) permit the gathering of information (which may still be logged and available for off-line checking) and utilize 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 producers and users, and are related to the possibility of an attacker tampering with the communication between the two parties. To address these issues, we will leverage existing mechanisms, such as private/public key cryptography mechanisms and digital signatures.

5. 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 exploitation of high numbers of users and network connectivity to split the monitoring tasks across different instances of the software. Such splitting, which we call software tomography, facilitates gathering monitoring information through light-weight instrumentation. We described 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 have also presented our current implementation of the Gamma system and described our initial experience with the system. Although preliminary in nature, our study lets 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, 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 and by targeting different kinds of systems, such as embedded software on mobile devices.

Acknowledgments

Partially supported by grants to Georgia Tech from Boeing Aerospace Corp., NSF awards CCR-9988294, CCR-0096321, and EIA-0196145, and the State of Georgia under the Yamacraw Mission. Michal Young provided the tool Gretel, which we used for instrumentation. T. Apiwattanapong, J. Bowring, J. Jones, and A. Rao provided useful ideas and comments, and helped with the system's implementation.

6. 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 *Proc. of the 29th Int'l Symp. on Microarchitecture*, pages 46–57, 1996.

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

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

[5] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and Y. L. An empirical evaluation of the correlation between fault-revealing test behavior and differences in program spectra. *Journal of Software Testing, Verification, and Reliability*, 2000.

[6] M. Hicks and J. Moore. Dynamic software updating. In *Proc. of the ACM SIGPLAN '01 Conf. on Prog. Lang. Design and Implementation*, pages 13–23, jun 2001.

[7] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of Test Information to Assist Fault Localization. In *Proc. of the 24th Int'l Conf. on Softw. Eng.*, 2002.

[8] A. Orso, D. Liang, M. J. Harrold, and R. Lipton. Gamma System: Continuous Evolution of Software after DeploymentD. Technical Report GIT-CC-02-06, Georgia Tech, February 2002.

[9] A. Orso, A. Rao, and M. J. Harrold. Dusc: Dynamic Updating through Swapping of Classes. Technical Report GIT-CC-02-24, Georgia Tech, March 2002.

[10] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st Int'l Conf. on Softw. Eng., 1999*, pages 277–284, May 1999.

[11] 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 *Proc. of the 6th European Softw. Eng. Conf.* , pages 432–449, Sept. 1997.

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