1 First Workshop on Advancing the State of the Art in Run-Time Inspection

Organizers: Robert Filman, Katharina Mehner, Michael Haupt

Modern software development is inconceivable without tools to inspect running programs. Run-time inspection is a crucial factor for both building complex systems and for maintaining legacy systems. Run-time inspection includes not only querying of program state but also controlling its execution. Applications of run-time inspection range from code-level tasks like debugging, profiling, tracing, testing, and monitoring to conceptual activities such as program comprehension, software visualization and reverse engineering. New applications incorporate run-time inspection as a programming concept in the style of event-condition-action rules.

The diversity of programming paradigms such as configurable software, components, aspect-orientation, generative programming, real-time programming, distributed programming, ubiquitous computing, applets and web services emphasizes the need to deal with heterogeneous run-time information and different levels of abstraction. Lacking well-established technologies and models for representing and accessing program dynamics, tools must use ad-hoc mechanisms. This limits reuse and interoperability. De facto standards for run-time inspection such as the Java Platform Debugger Architecture (JPDA) have improved the situation but do not cope with all requirements. Implementers seeking to create debugging environments for ubiquitous computing are faced with even greater difficulties.

The workshop sought to identify best practices and common requirements, to specify conceptual data, control models and implementation approaches for run-time inspection and to discuss practical issues such as standardized APIs and data exchange formats.

The contributions for the workshop covered a wide range of topics. We divided the contributions into two sessions: classical application of run-time inspection and novel architectures made possible by the expanded use of run-time inspection. Classical uses of run-time inspection included debugging and program visualization. The dominant novel use of run-time inspection was as a foundation for dynamic aspect-oriented programming.

1.1 Session on Run-Time Inspection: Methods, Applications, and Special Issues

The first three position papers of the first session presented run-time tools: a debugger and two visualization tools, one of which also allows run-time program manipulation. The last two position papers tackled specific problems in run-time inspection: data collection and scalability.

E. Tanter, P. Ebraert. A Flexible Approach to Interactive Runtime Inspection Tanter and Ebraert propose using *behavioral reflection* as a foundation for run-time inspection. Behavioral reflection allows introspection (querying the parts of a program describing its dynamic behavior), and intercession (controlling its execution). Tanter presented a tool for run-time inspection which is based on a previously developed system called *Reflex*.

Reflex supports partial (behavioral) reflection for Java by allowing the programmer to choose which operations of which classes or objects should be reified during which part of the program lifetime. These reifications of base-level occurrences of operations are accessible statically and at run-time in terms of meta-objects. Reflex is implemented using the Java Platform Debugger Architecture.

This run-time inspection tool provides a graphical interface representing the meta-objects. Through this interface the program can be manipulated at run-time. Tanter pointed out that the particular challenge of such a tool lies in dealing with the visual load and in synchronizing the user interaction with the running program by providing direct feedback. He also envisioned raising the level of abstraction to yet another meta-level by viewing the meta-objects as units of separated concerns and by providing an additional meta layer for monitoring and manipulating these units.

This work is related to the position papers on dynamic AOP discussed in the next section. The proposal of behavioral reflection as a foundation of run-time inspection was picked up in the discussion group on a model for run-time inspection.

B. Lewis. Recording Events to Analyze Programs Lewis presented the *Omniscient Debugger*, a Java debugger which allows making a detailed recording (also called trace) of a running Java program and stepping through this recording, going both backwards and forwards. After each such step, a detailed state of the program is presented. This approach allows an intensive examination of the execution history without ever making a choice where to set a breakpoint. Going backwards is extremely useful in tracing problems to the place where they first occurred—the preeminent question is debugging is “How did that happen?”

Lewis entertained the group with a plush-toy demonstration of the importance of such a debugger: bugs are not “bugs” but different animals, namely snakes and lizards who hide in the grass. Snake and lizard tails often stick out of the grass. Following the bug to its source means grabbing the animal by its tail. If the program produces an incorrect answer you have a tail on which you can pull. If you pull on a snake’s tail long enough you will get to its head; by analogy, this is finding the source of the bug. The Omniscient Debugger facilitates this through its ability to move backwards in time, like moving along the body of the snake. However, if you pull on a lizard’s tail long enough it will eventually break, keeping you from getting to the source of the bug. Lizard-like bugs are endemic to break-point centered debuggers, because you can’t pull the tail much at all—you know the “now,” but not the “then.”

Lewis pointed out that even with the arbitrary reversibility of the omniscient debugger, some bugs are still intractable. Bugs due to the failure to do something cannot be traced back to a mistaken action. If a program fails to produce a correct answer then one does not have an animal tail to grab. However, the Omniscient Debugger can be used to help search for the hidden-in-the-grass tail.

The Omniscient Debugger is implemented by Java byte code instrumentation, and relies heavily on timestamps. An instrumented event causes an average slowdown of two microseconds. At best, a slowdown of factor two can be achieved for an entire program. Due to the efficient implementation and clever recognition of state-preserving routines, the Omniscient Debugger can work with programs generating a ten million events.

A. Cain, J. Schneider, D. Grant, T. Chen. Runtime Data Analysis for Java Programs Cain et al. examined the usefulness of the Java Platform Debugger Architecture (JPDA) for data flow analysis. Data flow analysis requires capturing the definition and the references of all variables in a program. Often, it is desirable that the scope of the analysis can be targeted. Because local variables and array elements are not covered by the JPDA, Cain presented a hybrid solution using source code transformation and run-time analysis through the JPDA.

A. Zaidman, S. Demeyer. Program Comprehension Through Dynamic Analysis Zaidman and Demeyer addressed the problem of the size of trace data. Huge traces are not only a problem of memory but also a cognitive problem for the user. Their solution to reducing trace size is to compress trace data. They propose to cluster method-call events based on the same frequency of occurrence in a program run. This clustering technique is based on the heuristic assumption that methods working together to reach a common goal are be executed about the same number of times. Zaidman suggested that the spectral visualizations of these clusters could also be used to do dissimilarity measures and to search for visual patterns.

H. Leroux, C. Mingins, A. Requile-Romanczuk. JACOT: A UML-Based Tool for the Runtime-Inspection of Concurrent Java Programs Leroux et al. presented a tool called *JACOT* to dynamically visualize a running program by means of animated UML sequence diagrams. A sequence diagram can show object interaction over time and thus can present the history of program execution. JACOT has a special focus on threading and exceptions. It provides an animated state chart for depicting thread state and an activity diagram for depicting exception flow. The tool is implemented using the Java Platform Debugger Architecture.

Session Conclusion The discussion group on visualization noted that the presentations by Lewis and Leroux contrasted different visions of how to present the history of program execution. Lewis's approach provides a detailed, step-by-step history. Leroux presents a quickly understood though less-precisely meaningful visual abstraction. Clearly, the ultimate run-time inspection tool would present both kinds of mechanisms, and allow the user to switch between them as needed.

1.2 Session on Dynamic Aspect-Oriented Programming

The presenters in this session focused on implementation approaches to dynamic Aspect-Oriented Programming (AOP) systems and language design problems. Implementation languages used in the papers were Common LISP, Java, and Smalltalk.

P. Costanza. Dynamically Scoped Functions for Runtime Modification Costanza argued that a function definition is dynamically scoped if its binding is *always* looked up in the current call stack, regardless of which point in program execution is currently being processed. This contrasted with lexically scoped definitions, whose meaning is determined by examining a program's syntactic structure.

Costanza claimed that dynamically scoped functions are useful for modifying an application's behavior, a common task for aspect oriented programming. The paper proposes extending LISP with dynamically scoped functions to be able to decorate functions with additional enclosing behavior.

S. Chiba, Y. Sato, M. Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems Chiba et al. presented *Wool*, an implementation of dynamic aspect weaving for Java utilizing the standard JVM's HotSwap capabilities. HotSwap allows for changing class implementations at run-time via a function that is part of the Java debugger API.

Wool follows a hybrid approach in decorating join points with additional behavior at run-time. At first, all active join points are registered as debugger break points. The system checks each time such a break point is reached to see if any aspects should be invoked. However, there is a large execution costs to this repeated checking. When a break point has been reached often enough (decided by a simple heuristic), the corresponding method is replaced with a modified version that directly calls the aspect.

S. Aussmann, M. Haupt. Axon — Dynamic AOP through Runtime Inspection and Monitoring Haupt and Aussmann presented Axon, an implementation approach to dynamic AOP that relies solely on the JVM's debugger to intercept application execution at join points and branch to aspect functionality.

Axon's underlying model is strongly influenced by the concept of event-condition-action (ECA) rules found in active databases. An ECA rule comprises an event describing a (complex) situation in the database, a condition that is evaluated when the event is signaled and an action that is executed when the condition evaluates to true.

Haupt argued that ECA rules and aspects are related in that join points and pointcuts, marking that a certain point in execution has been reached, form an event algebra and advice correspond to actions. From the observation of this relationship, Axon's aspect model was developed that strongly decouples the different parts of an aspect from each other.

S. Hanenberg, R. Hirschfeld, R. Unland, K. Kawamura. Aspect Weaving: Using the Base Language's Introspective Facilities to Determine Join Points Using a sample implementation of a generic observer in AspectJ, Hanenberg et al. first showed that most existing AOP languages suffer from not being thoroughly equipped with reflective capabilities. An observer ensures that any change to a subject's instance variables that is done through the subject's interface is reported to the observers. However, this is not true if the instance variables are changed by some other means, e. g., if they are instances of reference types and a modification of their own data is done.

Having presented this problem, the authors demonstrated a solution based on the using the reflective capabilities of Smalltalk and the facilities of the AspectS AOP extension to Smalltalk. In their solution, reflection is used to traverse an observed object structure and to decorate all necessary places—i. e., all places where data belonging to the structure may be changed—with notification calls, thereby ensuring that *any* change in the subject is reported to the observers.

Session Conclusion The two concrete implementation approaches presented by Chiba et al. and Aussmann/Haupt show that implementing run-time AOP systems is an active area. Both approaches utilize an interception-based strategy, illustrating the relationship between run-time AOP and event-based systems. Using dynamically scoped functions to implement dynamic AOP contributes to a better understanding of the semantics of such systems.

The work of Hanenberg et al. shows that purely lexical join points are not enough to precisely express the interactions of aspects with the applications they decorate. There is a need for mechanisms that are able to capture and express logical requirements of application structures.

1.3 Discussion topics of break-out groups

The workshop broke into three groups to discuss particular topics in greater detail.

A model for Run-time Inspection The aim of this group was to describe a model capturing the essence of run-time inspection.

The group argued that the term "inspection" is a source of misunderstanding. It was coined by the workshop organizers in analogy to the term guided inspection, sometimes also called code inspection, which refers to a testing practice using code walkthroughs. In analogy to guided inspection, run-time inspection can be seen as being purely observational, perhaps also including the possibility of pausing program execution. However, this definition does not capture the ability to change the program's execution. Some of the participants considered this to be an essential part of run-time inspection. Reflection was therefore proposed as a foundation for run-time inspection because it includes the possibility of making changes to the program and its execution.

The model for run-time inspection proposed by this discussion group is built not only on reflection but also draws on existing models from the related area of software visualization. Models for software visualization typically distinguish the phases data collection, data transformation, and data presentation. The following model aims at clearly identifying the different phases in the process of run-time inspection:

- Running the program
- Collecting data
- Transforming data
- Presenting data
- Effecting changes based on presentation of data
- Propagate changes to the program

This is a cyclic process. The changes become effective in the running program.

Visualization This group was considered the visual presentation of information about running programs. The group noted that the first step in visualization is always defining what information is to be visualized. The group then addressed the problem of dealing with high volumes of information gathered at run-time. Suggestions to shape a call graph to one line per function were contrasted with animation of a class diagram by highlighting fields and members involved in method invocation during (re)play. Information compression was addressed through clustering of method calls based on the coupling, i. e., the number of calls between two methods. Also, different diagrammatic notations were discussed such as tree-like structures for hierarchical information and spiral and kiviat (spider) structures.

Applications This group was looking at applications of run-time inspection with a primary focus on applications for dynamic aspect-oriented programming. Client-reside code and application servers were identified as promising application areas for technologies that can make changes to running code. Examples of uses of such technologies in these domains include:

- Bug fixes
- Usage monitoring
- Integration with other applications

The group raised the important issue of the safety of run-time changes. Safety concerns arise both for security and for system integrity—it is easy to imagine making dynamic changes that corrupt a system's logical state. After all, even in tightly release-controlled environments, a high percentage of bug fixes themselves introduce new bugs. This issue reflects similar concerns in AOP, which has the hope of modularizing changes in aspects but has long recognized the potential problem of conflicting aspects.