A Multi-Agent Debugging Extension Architecture

Ziad Al-Sharif, Clinton Jeffery
Computer Science Department
University of Idaho
zsharif@ieee.org, jeffery@cs.uidaho.edu

Abstract
The Idaho Debugging Extension Architecture (IDEA) enables dynamic analysis agents, such as automatic debugging and visualization agents, to be loaded on the fly in a source-level debugger. IDEA is an event-driven debugging architecture that provides a simple interface to load API-compliant external dynamic analysis agents during a debugging session. Multiple standalone agents can be loaded and managed under the control of the source-level debugger. Successful agents can be migrated into the source code of the debugger core as permanent features with higher performance.

1. Introduction
A source-level debugger helps programmers locate bugs by stepping through the source code and examining the current state of the execution. Some drawbacks of typical source-level debuggers are: 1) limited information provided about the execution history, 2) lack of automated, analysis-based debugging techniques, and 3) closed architecture that provides little or no cooperation with external debugging and visualization tools.

Reversible and post-mortem (trace-based) debuggers, such as ODB[13], TODI[15], and Whyline [11,12] provide debugging techniques based on the ability to browse forward and backward through the states of an execution. This approach provides outstanding debugging capabilities such as finding where and why some action has happened, but poses formidable scalability problems, and is good at finding some types of bugs and not others. While they provide valuable capabilities, some trace-based debuggers neglect common debugging techniques such as altering the state of the buggy program.

This paper presents the Idaho Debugging Extension Architecture (IDEA). IDEA supports extensions called agents. An agent is an event-driven task-oriented program execution monitor. IDEA’s agents are written and tested as standalone programs, after which they can be loaded and used on the fly from within the conventional source-level debugger, or integrated as permanent features into the debugging core, with almost no source code alteration. Different agents perform different debugging missions such as detecting a suspicious execution behavior, performing an automatic debugging procedure, or executing a dynamic analysis technique.

IDEA’s debugging extension agents monitor the execution of a program for specific run time events; an event is an action during the execution of the program such as a method being called or a major syntax construct being entered. Different agents can be loaded and active, and each agent receives different runtime events based on their own request. Agents are coordinated by a central debugging core. Each agent 1) provides the debugging core with its set of desired events, 2) receives relevant events from the debugging core, 3) performs its debugging mission, which may utilize execution history prior to the current execution state, and 4) sends its analysis results back to the user.

IDEA processes and filters execution events, manages the debugging session, and handles external and internal debugging agents. IDEA allows the debugging core and any number of compatible dynamic analysis agents to assist in locating and finding bugs. Separately-compiled dynamically-loaded external agents receive their information from IDEA’s debugging core, which controls them. All active external/internal agents are suspended whenever a breakpoint or a watchpoint is reached, and they are resumed whenever the user resumes the buggy program. The external debugging agents’ standard inputs and outputs are redirected and coordinated by IDEA’s debugging core.

2. Debugging with Agents
Conventional debuggers allow users to explore their debugging hypotheses using manual investigation. Debugging with agents leverages the conventional debugging process by empowering the user with more tools to inspect the state of the buggy program. IDEA’s agents may retain information beyond the current state of execution and perform automatic debugging and dynamic analysis techniques that could be supported by trace-based debuggers such as ODB [13,15]. However, IDEA’s agents are task-oriented; each agent embodies a lightweight task-specific analysis technique.

IDEA’s agents can be written and tested as standalone programs and then loaded into the debugger to work in concert with each other. Using IDEA, it is easy to define debugging agents that capture specific execution behaviors.
such as: 1) loops that iterate N times, for some N >= 0; 2) variables that are read and never assigned or assigned and never read during a particular execution; 3) expressions such as subscripts that fail silently in a context where failure is not being checked; 4) a variable that may change its type during the course of execution; or 5) a trace of variable states, which allows users to trace backward and see where a specific variable was assigned long before it is involved in a crash. For example, many functions return a specific value when they encounter an error or fail to accomplish their job. An agent can automatically catch any of these failed functions and save the user the time that can be spent during a manual inspection.

Furthermore, IDEA’s agents are employed within the conventional source-level debugging session, which provides a simple interface to load, unload, enable, or disable debugging agents on the fly, and the user can be selective about which agent(s) to use.

3. Design

IDEA features novel properties that distinguish it from other debugging architectures. First, it provides two types of extensions: dynamic extension on the fly during the debugging session (external agents), and formal steps for migrating and adopting standalone agents as permanent debugging features (internal agents). Second, it encourages users to write their own agents and incorporate them into a typical source-level debugging session. Finally, it supports an interactive users interface, where simultaneous agents can be loaded and managed during a debugging session. The user does not need to restart the debugging session whenever a decision is made to incorporate any of the debugging agents in that session. In contrast, common static and dynamic analysis tools and libraries have to be linked in advance into the source code of the buggy program, or initialized at the start of the host debugger.

IDEA’s debugging core is comprised of five major components: 1) a console that provides the interface between the user and the debugging facilities, 2) a session that initializes and coordinates the debugging situation, 3) a debugging evaluator that provides the main monitoring loop and event filtering, 4) an agents interface that facilitates and provides the programming interface for external and internal extensions, and 5) a debugging state that maintains and shares the state of the debugger between the rest of the components and the user. See Figure 1.

5. Implementation

IDEA’s implementation is based on two components that make the source-level debugger an event coordinator for the extensions; Internals and Externals. These components are plugged in to the main debugging loop as extra listeners on the runtime events. IDEA manages and coordinates the external agents and forwards received events, from the buggy program into different external debugging agents based on their interest. Extensions to an IDEA-based debugger are coordinated within the agents’ interface of the evaluator component. Extensions are abstracted by objects which serve as Proxies for external agents or as Listeners in the case of internal extensions.

The Alamo monitoring framework provides high level primitives to control the buggy program and to customize the reported events. The next reported event will be one of those specified by a set of event types called an event mask; a detailed event filter for the current set of desired events. For the selected event types, if the event code has a corresponding entry in the hash table named value mask, then only those events that have a matched value is reported. This optimization limits the number of reported events to the inquired ones, and reduces the number of context switches.

IDEA’s debugging core coordinates all of the built-in classical debugging techniques, the internal agents, and the dynamically loaded external agents. For every received event, first it checks whether any classical action is needed such as a breakpoint, or watchpoint. Then it checks for any enabled internal and/or external agent; it forwards events to the enabled agents based on their event mask.

6. Extensions

IDEA supports two types of agents: 1) standalone agents that can be loaded on the fly, from any point, during the debugging session, and 2) built-in debugging agents that are incorporated into the debugging core as permanent debugging features.

The event masks of extension agents (internals and externals) are added to the set of events that are requested by default by the debugging core. On the fly, the debugging core starts asking the buggy program about those extra events. When the debugging core receives an event from the buggy program, it forwards the received event to those extension agents that are enabled and requested this event in

Figure 1. The IDEA Architecture
their event mask. For internal agents, this takes the form of a call to a listener method, while for external agents it takes the form of a thread switch (Unicon threads are called co-expression), which the agent sees as a return from its EvGet() event request. EvGet() is an Alamo primitive that resumes the buggy program until the next available event. See Figure 3.

Different agents can be loaded and active, and each agent receives different runtime events based on their own event mask. An extension agent may change its event mask during the course of execution. A change on any extension agent’s event mask immediately triggers an update of the event mask of the debugging core and alters the set of events received by the debugging core and forwarded to the extension agent.

6.1. Sample Agent

The code provided in Figure 2 shows a prototype of an IDEA-based agent. It is a toy example that captures the number of calls of user-defined functions/methods and native built-in functions, and finds the ratio for each call type. This provides a rough measure of the degree of VM overhead for a particular application. The class Example() contains three types of methods: 1) event handlers, which collect information based on the received events; handler methods start with the prefix “handle_” followed by the name of the event code (handle_E_Pcall()), 2) information analyzers, which analyze the collected information by the event handlers; analyzer methods start with the prefix “analyze_” followed by any name (analyze_info()), and 3) information or result writers, which output the result found by the agent; writer methods start with the prefix “write_” followed by any name (write_info()). Agents that follow this method naming convention can be registered automatically with the library of internal agents. Otherwise, agents can be registered manually. See Section 6.4.

6.2. External Agents

External agents can be written and tested as standalone tools, and subsequently loaded on the fly and used together during a debugging session. IDEA’s external agents are loaded and controlled by its debugging core. Active agents are paused whenever the buggy program is paused and they resume whenever it resumes.

IDEA’s debugging core receives runtime events from the buggy program based on the current debugging context, and the event masks of the external agents. The Externals component multiplexes the received events between different external agents. Events are sent to related active agents. An external agent requests events from the debugging core using the EvGet() primitive, which transfers control from the external agent to the debugging core. EvGet() is the same primitive that transfers control

Figure 2. An IDEA-based agent prototype

Figure 3. IDEA’s general control/events flow

Figure 4. UML of UDB; An IDEA-based debugger
and acquires events from the buggy program when the agent is used in a standalone mode. The Externals component forwards events to any of the external agents using the EvSend() primitive, which is another Alamo primitive that sends the last event received by the debugging core to the external agent. A context switch occurs whenever control transfers between the debugging core and either a buggy program or an external agent. Event forwarding is accomplished without the knowledge of the external agent itself, which means the external agent needs no modification to be loaded and used by IDEA’s core.

6.3. Internal Agents

Besides support for whole programs as external agents, IDEA supports insertion of dynamic analyses into the debugging core as a listener agent that implements a set of callback methods. IDEA’s debugging core implements different built-in agents for different classes of bugs. For performance reasons, each agent has its own implementation based on the type and the combination of events that the debugging core must monitor in the buggy program.

The Internals component handles the built-in agents. Internal agents are called from the main debugging loop with a call to the forward() method of the Internals component, where internal agents are registered during initialization. The Internals component checks which agents are active and calls the related underlying method(s) based on the event code that is received by the debugging core.

6.4. Migration from Externals to Internals

External agents allow automatic debugging techniques based on various dynamic analyses to be developed and tested easily in the production environment. Selected external agents may become internal—built-in monitors within the debugging core for improved performance. Internal agents do not pay the (lightweight, but still painful) cost of the context-switch communication between the debugging core and the external agents. IDEA provides smooth migration from external agents to internal. The first issue in migration is to accept a callback-style event listener architecture in place of the more general main() procedure that an external agent uses from a separate thread. IDEA provides an abstract class called Listener, which must be subclassed within the external agent before the external can be used as internal. The Listener class allows the debugging core to acquire the event mask of the migrated internal agents, and to determine which listener methods to use for the various event types.

The agent prototype discussed in Figure 2 can be used as a standalone program or as an external agent under IDEA without any modification. In order to move such an external agent to an internal one, the user must derive this Example class from IDEA’s Listener abstract class and register it in the Internals class. Whenever its own event mask changes, this abstract class helps the Internals class rebuild the event mask for the internal agents and the debugging core by calling the updateEventMask() method in IDEA’s State class to update the debugging core with the new event mask obtained from the internal agent.

An object of the newly migrated internal agent must be instantiated and inserted into the list of clients in the Internals class. This can be done through the method register() from the Internals class. For example, to register the prototype Example agent provided Figure 2 as an internal agent, the programmer has to place a call to the method register() in the constructor of the Internals class where the first parameter associates the agent with a formal name as an ID during the debugging session, and the second parameter is an object of that agent class (register("call_count", Example())). This is the simple automatic registration that applies for agents who follow the sample agent convention shown in Figure 2 and discussed in Section 6.1. To register a complex agent that does not follow this sample convention, the method register() can be called with three extra parameters to register the method handlers, the analyzers, and the writers respectively.

register( "call_count", Example() ,
["handle_E_Pcall", "handle_E_Fcall"],
["analyze_Info"], ["writer_Info"]
)

Furthermore, the new internal agent must be stripped of its main() procedure before compilation and linking into the debugging core. Alamo primitives found in the external agent are no longer needed when it becomes an internal agent. EvInit() is needed once per each buggy program and it is already performed by the debugging core. IDEA’s EvGet() asks the buggy program for the next event, so the extension agent has no need to call this function. IDEA forwards events to relevant enabled internal agents based on their event mask. The mapping of events such as E_Pcall to their listener methods (handle_E_Pcall) is constructed automatically and used by the Internals class report() method.

7. Evaluation

In order to evaluate and refine the IDEA architecture, a debugger called UDB was constructed. UDB is an event-driven source-level debugger for Icon and Unicon programs. It implements the classical debugging features found in a typical debugger such as GDB, and it utilizes IDEA’s agent-based extensions. See Figure 4.

An IDEA-based debugger must use different approaches to implement features found in standard source-level debuggers, and faces potential performance challenges. In compensation, this type of implementation greatly simplifies the process of experimenting with new debugging
techniques that probably would not be undertaken if the implementation was limited to the low-level approaches found in other debuggers.

Nevertheless, one of the biggest considerations in the design of a source-level debugger is the performance. In IDEA, a considerable amount of time is spent on: 1) processing the instrumentation in the buggy program, 2) filtering the received events in the debugging core, and 3) processing the context switches between the debugging core, the buggy program and the external debugging agent. Since IDEA’s debugging core is a mediator between those external agents and the buggy program, each external agent imposes two extra context switches. For events that the debugging core does not itself need, there are two levels of context switches where outside IDEA there would be only one: the first is between the debugging core and the buggy program, and the second is between the debugging core and the external agent. Alamo’s event filtering techniques, provided by the event mask and the value mask, reduce the amount of context switches; a context switch occurs only when there is an event and it is needed by either the debugging core or any of its agents.

Under UDB, eight different debugging agents were loaded and tested as external agents, and then migrated to become part of the UDB’s library of internal agents. The slowdown imposed by the external agents was at most 3 times slower than the standalone agent mode, and the slowdown imposed by the migrated internal agents, was at most 2 times slower than the standalone agents. This suite of debugging agents imposes at most 20 times slowdown on the execution of the buggy program over an uninstrumented execution mode, but in the general case, the slowdown depends on the algorithms used by the dynamic analysis technique implemented by the debugging agent. To place this in perspective, a debugger such as valgrind [14] imposes a 20 to 50 times slowdown, and it does not provide the interactive debugging environment that IDEA and its debugging tools provide, where the user can be selective about which and where to enable/disable agents from within a breakpoint based debugging session.

8. Related Work

Standard source-level debuggers such as GDB [16] and its graphical front end DDD [20] provide convenient debugging and tracing facilities. But using a conventional source-level debugger is still time consuming; it is largely based on forming a hypothesis and guessing where to place breakpoints. One way to reduce the debugging time is to automate the debugging process. Automated debugging is a challenging problem that goes back to the 60’s [5] and mid 70’s [10]. Its most challenging part is the reasoning about the information supplied by the debugging tool, which still heavily depends on the human factor.

Trace-based debuggers such as ODB [13, 15], provide debugging facilities by tracing the complete program history of states. In ODB, the program must be traced first before the debugging techniques can be used, but it provides the ability to investigate backward in the execution history. IBM’s JInsight [3] combines tracing and visualization techniques, but it generates a huge amount of traced data in a short period of time. In general, the most common problem of trace-based debuggers is the huge amount of traced data that limits the scalability and raises the level of complexity. An example of a recent tool is JDLab [1], which applies the graph theoretical algorithms to reduce the amount of traced events.

A debugging architecture such as JPDA, with its lowest level JVM TI [6], provides an event-based debugging infrastructure and enough events for conventional debugging, profiling, and visualization. JVM TI supports about thirty five kinds of events, whereas IDEA incorporates more than one hundred kinds of events. IDEA users use events, event-sequences, and event-patterns to write their own debugging agents that detect specific execution behaviors— some of which are suspicious behaviors and others are defined bugs.

Both IDEA and JVM TI, provide techniques to inspect the state and to control the buggy program running in the VM. JVM TI agents must be loaded and initialized at the start of the JVM, whereas IDEA’s extension agents can be loaded on the fly from any point during the debugging session. IDEA’s debugging agents can be designed and tested as standalone programs, then dynamically linked into debugging sessions, and finally incorporated in the debugger source code as permanent extension.

Valgrind [14] provides a formal mechanism for custom extensions, but it does not provide dynamic extensions or interactive debugging. IDEA provides a debugging architecture that maintains the interactive debugging found in conventional debuggers such as GDB, and it provides enough events to maintain lightweight task-oriented custom trace-based debugging agents.

9. Future Work

Our future work aims at using IDEA to develop a wide range of automatic debugging agents. Performance can be further improved by buffering related events and avoiding extra context switches; this will help for both internal and external extension agents. Another potential improvement is to offload the cost of external agents onto additional processor cores. The simplest speedup approach initially will be to add the most useful agents of automatic debugging techniques as internals, which will reduce the context switches on external agents.

Subsets of the Alamo framework used by IDEA for Unicon debugging have been implemented for monitoring C and Python [9, 19]. Future work may extend IDEA’s debugging facilities to these languages, or port IDEA to run on other debugging platforms such as JPDA. Another potential future work is to use an instrumentation
framework, such as ASM, PIN, and Atom, as a substitute for Alamo.

10. Conclusion

Different programmers and bugs require different debugging techniques. It is impossible to provide every desirable debugging technique in one tool. IDEA provides a compromise solution by allowing programmers to run a chosen suite of dynamic analysis agents from within a normal interactive debugging session. The IDEA architecture: 1) combines the capabilities of classical and trace-based debuggers, 2) lets users write their own high level standalone debugging agents and loads them from any point into an interactive debugging session, and 3) provides simple mechanism to incorporated standalone programs as internal permanent features.

Compared with the slowdown of many automatic debugging techniques, the performance of IDEA is very good. However, the true test of IDEA’s performance will be whether it enables debugging agents that justify their time cost by the value they provide to programmers, as valgrind often does.

11. Acknowledgments

This research was supported in part by an appointment to the National Library of Medicine Research Participation Program. This program is administered by the Oak Ridge Institute for Science and Education for the National Library of Medicine.

12. References